Lego-DLC: batching module for commit-carrying SNARK under Pedersen Engines

Byeongjun Jang†, Gweonho Jeong*, Hyuktae Kwon*, Hyunok Oh*, and Jihye Kim†

***Abstract*—The synergy of commitments and zk-SNARKs is widely used in various applications, particularly in fields like blockchain, to ensure data privacy and integrity without revealing secret information. However, proving multiple commitments in a batch imposes a large overhead on a zk-SNARK system. One solution to alleviate the burden is the use of commit-and-prove SNARK (CP-SNARK) approach. LegoSNARK defines a new notion called commit-carrying SNARK (cc-SNARK), a specialized form of CP-SNARK, and introduces a compiler to build commit-carrying SNARKs into commit-and-prove SNARKs. Using this compiler, the paper shows a commit-and-prove version of Groth16 that improves the proving time (about 5,000×). However, proving l-multiple commitments simultaneously with this compiler faces a performance issue, as the linking system in LegoSNARK requires $O(l)$ pairings on the verifier side.**

To enhance efficiency, we propose a new batching module called Lego-DLC, designed for handling multiple commitments. This module is built by combining a Σ-protocol with commitment-carrying SNARKs under Pedersen engines in which our module can support all commit-carrying SNARKs under Pedersen engines. In this paper, we provide the concrete instantiations for Groth16 and Plonk. In the performance comparison, for 2^{16} commitments, with a verification time of just 0.064s—over 30x faster than LegoSNARK's 1.972s—our approach shows remarkable efficiency. The slightly longer prover time of 1.413s (compared to LegoSNARK's 0.177s), around 8x is a small trade-off for this performance gain.

***Index Terms*—ZKP, zk-SNARK, CP-SNARK, Sigma protocol, Pedersen commitment**

I. Introduction

Zero-knowledge proofs, originating from the work in [1], have profoundly influenced modern cryptography by enabling a prover to prove that a statement is valid without revealing any details beyond its correctness. These proofs have evolved, transitioning from theoretical constructs to essential components in practical applications, such as succinct non-interactive arguments of knowledge (zk-SNARKs). Such proof systems reduce both the proof size and the computational effort required by the verifier to sub-linear levels relative to the statement. Generic zero-knowledge proof systems using a single homogeneous representation tend to incur the large computational cost on the prover side as the circuit grows. Simply, imagine proving many commitments within a single circuit: it would increase the circuit size, leading to impractically large proving time and CRS size. This inefficiency can be mitigated through the commit-and-prove SNARK (CP-SNARK) approach, which effectively links different proof systems using commitments. Thus CP proof systems enhance performance due to their modularity and choice-free adaptability for the nuances of a computation.

In the CP-SNARK literature, LegoSNARK [2] defines a "lifting" compiler to convert commit-carrying SNARK (cc-SNARK) into CP-SNARK. This tool requires a linking proof system to prove that multiple commitments open to the same value. For example, using QA-NIZK [3] requires $O(l)$ pairing operations and demands $O(l)$ key space on the verifier's side, where l is the number of commitments. Similarly, the compressed Σ-protocol in Eclipse [4] generates a proof of size $O(\log l)$[1] and still requires the verifier to maintain linear key space. Certain environments like smart contract restrict the practicality of these proof systems, as these intensive system resource demands directly translate to higher user costs. Specifically, on the Ethereum Virtual Machine, a pairing operation requires 45,000 gas, each group exponentiation consumes 6,000 gas, and storing 32 bytes cost 20,000 gas. Although proving multiple commitments in a batch is naturally more advantageous than proving them individually, to the best of our knowledge, no practical method currently exists that efficiently proves multiple commitments while considering both the prover and verifier.

In this paper, we provide a verifier-friendly batching module for cc-SNARK under Pedersen engines, which can demonstrate the equivalence between a committed message and its committed chunks efficiently.

A. Applications

Commitment can be employed to validate confidential data while preserving privacy. However, if numerous commitments are involved, verifying each one can be inefficient. Motivated by this, we propose a batching scheme that enables simultaneous verification of multiple commitments. To emphasize the importance of our scheme, we present high-level use cases of batching functionality in the following applications.

1) Proof of solvency: It is notable in financial applications to ensure that institutions can meet their liabilities. In such scenarios, each customer's balance must remain confidential, and the individual can check their own balance against the

† Byeongjun Jang is with Kookmin University, Seoul, Korea (email: byeongjunjang98@gmail.com).
* Gweonho Jeong is with Hanyang University, Seoul, Korea (email: kwonhojeong@hanyang.ac.kr).
* Hyuktae Kwon is with Hanyang University, Seoul, Korea (email: kwonhyuktae00@gmail.com).
* Hyunok Oh is with Hanyang University, Seoul, Korea (email: hoh@hanyang.ac.kr).
† Jihye Kim is with Kookmin University, Seoul, Korea (email: jihyek@kookmin.ac.kr).

[1]We omit the each committed vector size d for legibility.

institution's reported totals. To protect individual privacy, commitments related to accounts are published on the blockchain, enabling verification that each commitment has been properly formed. The batching technique enables proving of each commitment through a single proof. This functionality is essential for institutions that manage a large number of individual customer balances, providing a robust and privacy-preserving solution in the financial sector.

2) Digital credentials: In scenarios where individuals or members of organizations need to maintain anonymity while proving ownership of credentials—such as digital certificates issued by authorities. Each user's credentials are committed and stored by a third-party service, enhancing privacy and reducing data storage burdens on the individual. For instance, in settings where a service provider frequently validates their users' credentials against public commitments, the batching technique simplifies the process. Instead of generating a separate proof for each user, the service provider can accumulate multiple requests and generate a single proof that collectively validates all of them. This batching approach not only ensures privacy but also significantly reduces the computational and time costs associated with proof generation.

B. Technical Overview

In this section we briefly give a high level overview of our technique. We leverage the additive homomorphic property of Pedersen commitments to perform randomized aggregation of multiple commitments, and prove them using a Σ-protocol with cc-SNARK. In the Prove algorithm, the prover $\mathcal{P}$ takes an instance $\boldsymbol{x}$ and a witness $\boldsymbol{w}$ as inputs in which $\boldsymbol{w}$ can be split into a committed witness ($\boldsymbol{u}$) and a non-committed witness ($\boldsymbol{\omega}$). Then the prover computes a proof π and a proof-dependent commitment $\widetilde{\mathsf{cm}}$, where the commitment is uniquely generated and based on the commitment key ck. If a cc-SNARK is constructed under Pedersen engines, the commitment can be regarded as algebraic commitment (i.e. Pedersen commitment). LegoSNARK [2] leverages this point to efficiently prove Pedersen vector commitment. Proof systems such as Plonk [5], which operate under the algebraic group model assumption and use polynomial commitment (PC) scheme (e.g., KZG10 [6]), can output a commitment for the committed witness $\boldsymbol{u}$ as Pedersen commitment by adding committed witness-encoded polynomial when proving the relation $\mathcal{R}(\boldsymbol{x};(\boldsymbol{u},\boldsymbol{\omega}))$.

Aggregation of multiple Pedersen commitments. We start from defining a relation $\mathcal{R}$ as proving knowledge of multiple commitments under the committer key ck. The relation $\mathcal{R}$ can be shortly expressed as, where Com is denoted by a commitment scheme

$$\mathcal{R}(\boldsymbol{x};\boldsymbol{w}) = \left\{ \begin{matrix} (\mathsf{ck}, \{\mathsf{cm}_i\}_{i\in[l]}); \\ (\{m_i\}_{i\in[l]}, \{o_i\}_{i\in[l]}) \end{matrix} \quad : \quad \forall (\mathsf{cm}_i, o_i) = \mathsf{Com}(\mathsf{ck}, m_i) \right\}$$

However if attempting to efficiently prove the above relation using only cc-SNARK, generating l independent proofs in a naive manner would result in inefficient performance. This implies that when multiple commitments $\{\mathsf{cm}_i\}_{i=1}^{l}$ are present, which will be referred as $\mathsf{List}_{\mathsf{cm}}$, it requires an additional proof system for proving linkage between the proof-dependent commitment $\widetilde{\mathsf{cm}}$ of cc-SNARK and $\mathsf{List}_{\mathsf{cm}}$.

To improve this inefficiency, we propose generating each commitment cm_i based on a portion ck_1 of the commitment key ck from cc-SNARK. We then aggregate l-multiple commitments into a single commitment $\mathsf{cm}_{\mathsf{agg}}$, thus proving the correlation between $\mathsf{List}_{\mathsf{cm}}$ and $\mathsf{cm}_{\mathsf{agg}}$ without requiring an additional proof system. For example, assuming the form of each commitment cm_i is Pedersen commitment under the same commitment key ck_1 of cc-SNARK, we can employ the additively-homomorphic property to simply linear encode the knowledge of cm_i such as $\mathsf{cm}_{\mathsf{agg}} = \prod_{i=1}^{l} \mathsf{cm}_i$. However, since it does not ensure that the knowledge for each commitment resides in an independent space, we cannot extract each knowledge of cm_i from $\mathsf{cm}_{\mathsf{agg}}$. Therefore, we compute $\mathsf{cm}_{\mathsf{agg}}$ by employing linear combination with a randomness, blending it with arbitrary value τ. We can briefly ensure independence to each commitment by adding a random coefficient using powers of τ, such that

$$\mathsf{cm}_{\mathsf{agg}} = \prod_{i=1}^{l} (\mathsf{cm}_i)^{\tau^i} \tag{1}$$

Combine the cc-SNARK with Σ-protocol. Since we aim to prove $\mathsf{List}_{\mathsf{cm}}$ with a single commit-carrying SNARK proof π_{cc}, we must assume that each commitment cm_i already exists, rather than being generated during the Prove algorithm. However, if the prover arbitrarily selects the randomness τ, they could potentially generate the simulated commitment $\mathsf{cm}'_{\mathsf{agg}}$, which is different from the original $\mathsf{cm}_{\mathsf{agg}}$. For example, if the aggregator function is defined as $\mathsf{Agg}(\{m_i\}_{i\in[l]}) := \sum_{i=1}^{l} \tau^i \cdot m_i$, there are numerous possible solutions (m'_i) that can simulate a valid agg_m, such that:

$$\mathsf{agg}_m = \mathsf{Agg}(\{m_i\}_{i\in[l]}) = \mathsf{Agg}(\{m'_i\}_{i\in[l]})$$

Thus we need a robust method to prevent this issue without an additional proof system. To achieve that, we utilize a Σ-protocol. The prover computes a commitment that encodes the knowledge of $\mathsf{List}_{\mathsf{cm}}$ ahead of time. Once a verifier $\mathcal{V}$ chooses a random challenge, the prover $\mathcal{P}$ computes the aggregated knowledge (agg_m and agg_o) of the batched commitment $\mathsf{cm}_{\mathsf{agg}}$ by an aggregator function Agg. Its relation can be expressed as defined below:

$$\mathcal{R}_{\mathsf{Batch}}(\boldsymbol{x};(\boldsymbol{u},\boldsymbol{\omega})) = \left\{ \begin{matrix} \tau; \\ \begin{pmatrix} \mathsf{agg}_m, \mathsf{agg}_o, \\ \{m_i, o_i\}_{i\in[l]} \end{pmatrix} \end{matrix} \quad : \quad \begin{matrix} \mathsf{agg}_m = \mathsf{Agg}(\{m_i\}_{i\in[l]}) \\ \wedge\ \mathsf{agg}_o = \mathsf{Agg}(\{o_i\}_{i\in[l]}) \end{matrix} \right\}$$

The prover does not provide the batched commitment $\mathsf{cm}_{\mathsf{agg}}$. Instead, the verifier reconstructs $\mathsf{cm}_{\mathsf{agg}}$ using $\mathsf{List}_{\mathsf{cm}}$ and the challenge.

C. Our Contributions

A new batching module for cc-SNARK: **Lego-DLC**. We propose a new module by applying Σ-protocol to commit-carrying SNARK that efficiently proves multiple Pedersen commitments with a single proof, significantly reducing the computational overhead compared to traditional approach (i.e.

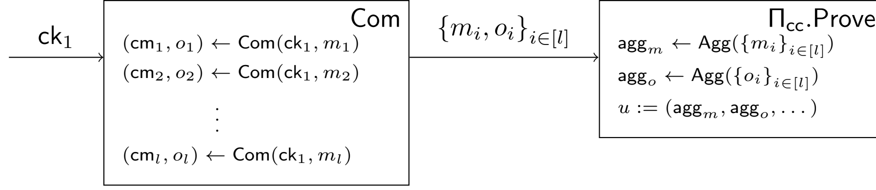


Fig. 1: Overview of our approach for aggregation of multiple commitments based on cc-SNARK. ck_1 denotes the portion of commitment key ck from cc-SNARK.

in-the-circuit). Additionally, akin to the transition method from cc-SNARK to CP-SNARK as introduced by LegoSNARK, this module can operate similarly through a linkable protocol. By aggregating commitments prior to proving linkage, it significantly improves efficiency over the method originally proposed in LegoSNARK.

Implementation and Evaluations. We have implemented and empirically tested our scheme, demonstrating its practical efficiency and scalability in handling large batches of commitments, providing a substantial improvement over existing approaches. Using our module with commit-carrying Groth16, we achieve meaningful performance improvements: for 2^{16} Pedersen commitments, it takes only about 1.413s, which is a substantial enhancement compared to the in-the-circuit approach. For 2^{10} commitments, our system can generate proofs in just 59 ms, whereas the in-the-circuit method takes approximately 57.203 seconds, showing a performance difference of about 970x. In comparison experiments conducted under the same environment with the related research, LegoSNARK, at 2^{16}, LegoSNARK exhibits a prover time of about 0.177s, while our system shows 1.413s, making our system about 8 times slower. However, for the verifier time, our system takes 0.064s compared to 1.972s for LegoSNARK, indicating a notable performance advantage. From an application perspective, performance metrics on the blockchain demonstrate that our system can verify 2^{10} commitments at about 5.2 transactions per second (TPS), offering more practical utility compared to LegoSNARK's 0.5 TPS.

D. Related work

The approach on integrating different proof systems has progressed with the goal of getting the computational efficiency, as evidenced by several studies [7, 2, 4, 8]. One important work of these studies, Chase et al. [7], provides a method that combines algebraic-based proofs, such as Σ-protocols, with garbled circuit proofs. This technique efficiently computes algebraic operations through algebraic-based proofs and handles non-algebraic operations using garbled circuit proofs. However, since this approach leverages a private garbling scheme from JKO13 [9], the necessity for private garbled circuits imposes limitations on the applicability to proof systems that do not employ such circuits. LegoSNARK [2] introduces a generic framework for constructing composite system from different proof systems by linking different systems using a generic compiler to build the generic integration of proof systems. In the paper, it shows a high-performance commit-and-prove proof system for proving Pedersen commitment, instantiated in a modular manner. This approach is more efficient than traditional methods (i.e. the commitment is encoded in the circuit). Eclipse [4] has crafted a compiler that transforms proof system based on algebraic holographic proof into commit-and-prove SNARK. Using compressed Σ-protocols, the proof systems such as Plonk [5], Sonic [10], and Marlin [11] can be instantiated into commit-and-prove SNARK with logarithmic proof size. In the recent study detailed in [8], the authors provide techniques for offloading non-native arithmetic operations from zero-knowledge circuits. By employing Σ-protocols for proving algebraic operations and SNARK for non-algebraic parts, the paper reduces the computational burden typically associated with embedding complex arithmetic in zero-knowledge circuits.

II. Preliminaries

A. Notations

We use $\boldsymbol{a}$ or $\{a_i\}$ for the list of elements, which is equivalent to a vector. We denote by λ a security parameter and by $\epsilon(\cdot)$ as a negligible function. Let $\mathbb{F}$ denote a finite field and $\mathbb{G}$ denote a group. A bilinear group generator $\mathcal{BG}$ takes a security parameter as input in unary and returns a bilinear group $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ consisting of cyclic groups $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$ of prime order p and a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Given a security parameter 1^λ, a relation generator $\mathcal{RG}$ returns a polynomial time decidable relation $\mathcal{R} \leftarrow \mathcal{RG}(1^\lambda)$. For $(\boldsymbol{x}, \boldsymbol{w}) \in \mathcal{R}$ we say $\boldsymbol{w}$ is a witness to the instance $\boldsymbol{x}$ being in the relation. We use the bracket for any bilinear group such as $[a]_s \equiv a \cdot g_s \in \mathbb{G}_s$.

B. Pedersen vector commitment

Pedersen vector commitment for vector $\boldsymbol{w}$ of size n can be expressed succinctly with the following algorithms:

- $\mathsf{Ped.Setup}(1^\lambda)$: chooses $g \xleftarrow{\$} \mathbb{G}$, $\boldsymbol{h} \xleftarrow{\$} \mathbb{G}^n$ from a domain $\mathcal{D}$. It outputs a commit key $\mathsf{ck} := (g, \boldsymbol{h})$.
- $\mathsf{Ped.Commit}(\mathsf{ck}, \boldsymbol{m}; o)$: returns $\mathsf{cm} := (o, \boldsymbol{m})^\top \cdot \mathsf{ck}$.
- $\mathsf{Ped.VerCom}(\mathsf{ck}, \mathsf{cm}, \boldsymbol{m}, o)$: returns true if $\mathsf{cm} = (o, \boldsymbol{m})^\top \cdot \mathsf{ck}$. Otherwise, false.

Lemma 1. The Pedersen vector commitment is perfectly hiding and computationally binding if the discrete logarithm assumption holds.

C. Succinct Non-interactive arguments of knowledge

Definition 1. A succinct non-interactive arguments of knowledge $(SNARK)$ for $\mathcal{R}$ is a tuple of algorithms $\boldsymbol{\Pi}_{\mathsf{SNARK}}$ = (SetUp, Prove, Verify) working as follows:

- $\mathsf{crs} := (\mathsf{ek}, \mathsf{vk}) \leftarrow \mathsf{SetUp}(\mathcal{R})$: takes a relation $\mathcal{R} \leftarrow \mathcal{RG}(1^\lambda)$ as input and returns a common reference string crs consisting of an evaluation key ek and a verification key vk.
- $\pi \leftarrow \mathsf{Prove}(\mathsf{ek}, \boldsymbol{x}, \boldsymbol{w})$: takes an evaluation key ek, a statement $\boldsymbol{x}$, and a witness $\boldsymbol{w}$ as inputs, and returns a proof π.
- $\mathsf{true}/\mathsf{false} \leftarrow \mathsf{Verify}(\mathsf{vk}, \boldsymbol{x}, \pi)$: takes a verification key vk, a statement $\boldsymbol{x}$, and a proof π as inputs and returns false $(reject)$ or true $(accept)$.

It satisfies completeness, knowledge soundness, and succinctness described as below:

Completeness. Given a true statement $\boldsymbol{x}$, for all relation $\mathcal{R}$ and for all $(\boldsymbol{x}; \boldsymbol{w}) \in \mathcal{R}$,

$$\Pr\left[\begin{array}{c} \mathsf{crs} \leftarrow \mathsf{SetUp}(\mathcal{R}), \\ \pi \leftarrow \mathsf{Prove}(\mathsf{ek}, \boldsymbol{x}, \boldsymbol{w}) \end{array} : \mathsf{Verify}(\mathsf{crs}, \boldsymbol{x}, \pi) = 1 \right] = 1$$

Knowledge Soundness. Knowledge soundness states that a prover must know a witness and such knowledge can be efficiently extracted from π by a knowledge extractor $\mathcal{E}$. Formally, the following is $negligible$ for any PPT adversary $\mathcal{A}$.

$$\Pr\left[\begin{array}{lcl} \mathsf{crs} \leftarrow \mathsf{SetUp}(\mathcal{R}), & & \mathsf{Verify}(\mathsf{crs}, \boldsymbol{x}^*, \pi^*) = \mathsf{true} \\ (\boldsymbol{x}^*, \pi^*) \leftarrow \mathcal{A}(\mathsf{crs}), & : & \wedge \\ \boldsymbol{w} \leftarrow \mathcal{E}_{\mathcal{A}}(trans_{\mathcal{A}}), & & (\boldsymbol{x}^*; \boldsymbol{w}) \notin \mathcal{R} \end{array}\right]$$

Succinctness. Succinctness states that the argument generates the proof of polynomial size in the security parameter, and the verifier's computation time is polynomial in the security parameter and in statement size.

Remark. A SNARK may also satisfy *zero-knowledge*. It states that the system does not leak any information besides the truth of the statement. This is modelled by a simulator that does not know the witness(but has some trapdoor information that enables it to simulate proofs). We refer to it as a zk-SNARK in this scenario.

Commit-carrying SNARK. There exists a variant of commit-and-prove SNARK ($\mathsf{SNARK}_{\mathsf{cp}}$), referred to as a commit-carrying SNARK ($\mathsf{SNARK}_{\mathsf{cc}}$), which is a SNARK whose proof includes a commitment to the portion of witnesses. It also satisfies completeness, succinctness, knowledge soundness, and binding. A commit-carrying SNARK consists of a set of algorithms, represented as tuple Π_{cc}.

- $\mathsf{crs} := (\mathsf{ck}, \mathsf{ek}, \mathsf{vk}) \leftarrow \mathsf{SetUp}(\mathcal{R})$: takes a relation $\mathcal{R}$ as input, and outputs a common reference string which includes a commitment key ck, an evaluation key ek, and a verification key vk.
- $(\widetilde{\mathsf{cm}}, \pi, \widetilde{o}) \leftarrow \mathsf{Prove}(\mathsf{ek}, \boldsymbol{x}; \boldsymbol{w})$: takes an evaluation key ek, a statement $\boldsymbol{x}$ and a witness $\boldsymbol{w} := (\boldsymbol{u}, \boldsymbol{\omega})$ such that the relation $\mathcal{R}$ holds as inputs, and outputs a proof π, a proof-dependent commitment $\widetilde{\mathsf{cm}}$ and an opening $\widetilde{o}$ such that $\mathsf{VerCom}(\mathsf{ck}, \widetilde{\mathsf{cm}}, \boldsymbol{u}, \widetilde{o}) = \mathsf{true}$.
- $\mathsf{true}/\mathsf{false} \leftarrow \mathsf{Verify}(\mathsf{vk}, \boldsymbol{x}, \widetilde{\mathsf{cm}}, \pi)$: takes a verification key vk, a statement $\boldsymbol{x}$, a proof-dependent commitment $\widetilde{\mathsf{cm}}$, a proof π as inputs, and outputs true if $(\boldsymbol{x}, \widetilde{\mathsf{cm}}, \pi) \in \mathcal{R}$, or false otherwise.

The commit-carrying SNARK satisfies the properties of completeness, succinctness, knowledge soundness, zero-knowledge, and binding.

D. Σ-protocols

With an arbitrary relation $\mathcal{R}(\boldsymbol{x}, \boldsymbol{w})$, we briefly recapitulate Σ-protocols. A Σ-protocol for the relation $\mathcal{R}$ is a three-round interactive proof system between a prover (with $\boldsymbol{x}$ and $\boldsymbol{w}$) and a verifier (with $\boldsymbol{x}$). $\boldsymbol{\Pi}_\Sigma$ consists of a tuple of efficient algorithms (Com, Chl, Res) run as follows:

- $\mathcal{P}$ runs $\mathsf{Com}(\boldsymbol{x}, \boldsymbol{w}) \rightarrow a$: sends a commitment a
- $\mathcal{V}$ runs $\mathsf{Chl}() \rightarrow c$: chooses a challenge c is distributed uniformly at random and sends c to $\mathcal{P}$.
- $\mathcal{P}$ runs $\mathsf{Res}(\boldsymbol{x}, \boldsymbol{w}, c) \rightarrow z$: returns some response value z.
- $\mathcal{V}$ runs $\mathsf{Verify}(\boldsymbol{x},\ (a, c, z))$ returns a bit $b \in \{0, 1\}$. If $b = 1$, the verifier accepts the proof, otherwise rejects.

where (a, c, z) is called *transcript*. A Σ-protocol satisfies *completeness, special soundness, (honest verifier) zero-knowledge*.

Completeness. δ-completeness is satisfied if honestly-generated transcripts always verify, unless the prover aborts, which occurs with a probabilty of δ. Formally, $(\boldsymbol{x}, \boldsymbol{w}) \in \mathcal{R}$ we have that, for all honestly generated transcripts (a, c, z)

$$\Pr\left[\ \mathsf{Verify}(\boldsymbol{x}, a, c, z) = 1 \mid z \neq \bot\right] = 1, \text{ and } \Pr\left[z = \bot\right] = \delta$$

Special soundness. Special soundness is satisfied if there exists an efficient extractor $\mathcal{E}$ that, for any PPT adversary $\mathcal{A}$, returns a statement $\boldsymbol{x}$ and two distinct accepting transcripts $(a, c_0, z_0), (a, c_1, z_1)$ where $c_0 \neq c_1$ such that $\mathcal{E}(\boldsymbol{x}, (a, c_0, z_0), (a, c_1, z_1))$ extracts a valid witness $\boldsymbol{w}$ with an exception probability ϵ, known as the *knowledge soundness error*.

Honest verifier zero-knowledge. Honest verifier zero-knowledge is satisfied if there exists a simulator $\mathcal{S}$ such that for all $(\boldsymbol{x},\ \boldsymbol{w}) \in \mathcal{R}$ the following distributions are *indistinguishable*.

$$\{(a, z) | c \leftarrow \mathsf{Chl}(); a, z \leftarrow \mathcal{S}(\boldsymbol{x}, c)\}$$

$$\{(a, z) | c \leftarrow \mathsf{Chl}(); a \leftarrow \mathsf{Com}(\boldsymbol{x}, \boldsymbol{w}); z \leftarrow \mathsf{Res}(\boldsymbol{x}, \boldsymbol{w}, c)\}$$

III. A NEW BATCHING MODULE FOR CC-SNARK: **Lego-DLC**

A. Bifurcate commitment key

In the setup phase of cc-SNARK, a common reference string $\mathsf{crs} := (\mathsf{ck}, \mathsf{ek}, \mathsf{vk})$ is generated. Similar to commit-and-prove approach, where commitments are pre-computed and proved, we posit that the commitment key ck can likewise be used to compute each commitment cm_i within cc-SNARK. Then we split the commitment key ck into two parts, denoted as

$$\mathsf{ck} := (\mathsf{ck}_1, \mathsf{ck}_2)$$

Specifically ck_1 serves as the commitment key enabling the prover to generate multiple commitments that the prover aims to prove. The secondary commitment key, ck_2, acts as a bridge by encapsulating the knowledge of these commitments (e.g., m_i, o_i) as committed witnesses within the commit-carrying SNARK. We define $\mathsf{List}_{\mathsf{cm}}$ as a commitment list where each cm_i and o_i is generated by commitment scheme Com with commitment key ck_1 and values of m_i, for $i \in [l]$. Each commitment cm_i is constructed as:

$$(\mathsf{cm}_i, o_i) = \mathsf{Com}(\mathsf{ck}_1, m_i)$$

At this point, the committed witness $\boldsymbol{u}$ consists of pairs (m_i, o_i) indexed by $i \in [l]$, which is used to compute a proof-dependent commitment $\widetilde{\mathsf{cm}}$ under a commitment key ck_2:

$$\boldsymbol{u} = \{m_i, o_i\}_{i\in[l]}, (\widetilde{\mathsf{cm}}, \widetilde{o}) = \mathsf{Com}(\mathsf{ck}_2, \boldsymbol{u})$$

The prover's claim is that each commitment cm_i is committed to m_i, and the proof-dependent commitment $\widetilde{\mathsf{cm}}$ is committed to the committed witness $\boldsymbol{u}$. It can be expressed as:

$$\{\mathsf{VerCom}(\mathsf{ck}_1, \mathsf{cm}_i, m_i, o_i)\}_{i\in[l]} \wedge \mathsf{VerCom}(\mathsf{ck}_2, \widetilde{\mathsf{cm}}, \boldsymbol{u}, \widetilde{o})$$

B. Batched commitment with Σ-protocol

Recall that the proof-dependent commitment $\widetilde{\mathsf{cm}}$ in cc-SNARK, each of pairs (m_i, o_i) can be viewed as in a committed witness $\boldsymbol{u}$. However, $\widetilde{\mathsf{cm}}$ cannot be considered as the proof-dependent commitment for the multiple commitment relation $\mathcal{R}$, since we must prove the knowledge of each of commitment based on the *identical* commitment key ck_1. Thus, since the naive existing cc-SNARK cannot prove the multiple commitments at once, the linking proof system must be required. To handle this limitation and facilitate the aggregation of multiple commitments into a single commitment, we use a randomness τ to apply unique encoding to each message and opening such as

$$\mathsf{agg}_m = \sum_{i=1}^{l} \tau^i \cdot m_i \qquad \mathsf{agg}_o = \sum_{i=1}^{l} \tau^i \cdot o_i.$$

By attaching a unique identifier through the linear combination with the randomness, each element is independently encoded. Then we add aggregated values $(\mathsf{agg}_m, \mathsf{agg}_o)$ into committed witness $\boldsymbol{u}$. At this point, we can prove that accumulated values $(\mathsf{agg}_m, \mathsf{agg}_o)$ are correctly derived from the pairs (m_i, o_i) within the circuit, which requires the prover to engage in only $O(l)$ field operations to evaluate agg_m and agg_o. Specifically the witness $\boldsymbol{w}$ can be expressed as follows.

$$\boldsymbol{w} = (\boldsymbol{u}, \boldsymbol{\omega}) = \left(\left\{\mathsf{agg}_m, \mathsf{agg}_o, \{m_i, o_i\}_{i\in[l]}\right\}, \boldsymbol{\omega}\right)$$

The prover sends a proof-dependent commitment $\widetilde{\mathsf{cm}}$ along with the proof π to the verifier, who then checks the validity of π using the commitments cm_i. However, there remains an issue to consider in our protocol: while it is possible to combine the knowledge of each commitment into a single value using randomness to ensure knowledge integrity, the prover can compute a simulated proof-dependent commitment $\widetilde{\mathsf{cm}}$. Our verifier knows the commitments cm_i but does not know the underlying knowledge for each commitment. This means that if the prover does not fix the proof-dependent commitment, it would be impossible to extract the knowledge of each commitment. To prevent this, we employ a Σ-protocol. Rather than concurrently transmitting the proof-dependent commitment $\widetilde{\mathsf{cm}}$ and the proof, we first bind the knowledge within $\widetilde{\mathsf{cm}}$ and send it ahead of the proof. Subsequently, the verifier sends a challenge τ to the prover in the Chl phase. This procedure ensures that the prover cannot generate the knowledge of $\mathsf{cm}_{\mathsf{agg}}$ before knowing the challenge. Upon receiving τ, the prover constructs $\mathsf{cm}_{\mathsf{agg}}$, and then sends the proof π for the cc-SNARK, excluding the proof-dependent commitment in the Res phase. The verifier can verify the proof with $\mathsf{List}_{\mathsf{cm}}$.

C. Putting Together

Our protocol is also a three-move protocol between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ with a triple of algorithms: $\mathsf{Com}, \mathsf{Chl}, \mathsf{Res}$. In the literature on Σ-protocols, it is common to present the interactive form of protocol, as it can straightforwardly be converted to a non-interactive version by applying the Fiat-Shamir transform, ensuring security in the random oracle model (ROM). Additionally, our protocol leverages commit-carrying SNARK ($\mathsf{SNARK}_{\mathsf{cc}}$) under Pedersen engines, which means that a proof is constructed as group linear encoding. Note that the commitment key ck of $\mathsf{SNARK}_{\mathsf{cc}}$ consists of (ck_1, ck_2).

Protocol. By $\Pi_{\mathsf{cc}}.\mathsf{SetUp}$ algorithm, the prover and verifier have a common reference string $\mathsf{crs} := (\mathsf{ck}, \mathsf{ek}, \mathsf{vk})$ for the following relation $\mathcal{R}_{\mathsf{Batch}}$.

$$\mathcal{R}_{\mathsf{Batch}}(\boldsymbol{x}; \boldsymbol{u}) = \left\{ \begin{array}{c} \tau; \\ (\mathsf{agg}_m, \mathsf{agg}_o), \\ (\{m_i\}_{i\in[l]}, \{o_i\}_{i\in[l]}) \end{array} : \begin{array}{c} \mathsf{agg}_m = \sum_{i=1}^{l} \tau^i \cdot m_i \\ \mathsf{agg}_o = \sum_{i=1}^{l} \tau^i \cdot o_i \end{array} \right\}$$

In our protocol, the prover's input is each of pairs (m_i, o_i) and a commitment list $\mathsf{List}_{\mathsf{cm}} := \{\mathsf{cm}_i\}_{i\in[l]}$ committing to values m_i with the opening o_i under the partial commitment key ck_1. The verifier's input to the protocol is a commitment list $\mathsf{List}_{\mathsf{cm}}$. In the committing phase, the prover $\mathcal{P}$ computes a proof-dependent commitment $\widetilde{\mathsf{cm}}$ as $\mathsf{Ped.Commit}(\mathsf{ck}_2, \{m_i, o_i\}_{i\in[l]}; \widetilde{o})$ where $\widetilde{o}$ is a random chosen in $\mathsf{SNARK}_{\mathsf{cc}}$. Then $\mathcal{P}$ sends a message $\widetilde{\mathsf{cm}}$ to the verifier $\mathcal{V}$. Given the proof-dependent commitment $\widetilde{\mathsf{cm}}$, the verifier chooses a challenge $\tau \xleftarrow{\$} \mathbb{F}$ and sends it to the prover $\mathcal{P}$. The prover runs $\boldsymbol{\Pi}_{\mathsf{cc}}.\mathsf{Prove}(\mathsf{ek}, \boldsymbol{x}, \boldsymbol{w})$ to generate a proof π_{cc}. The prover returns the proof π_{cc} as a response to the verifier $\mathcal{V}$. The verifier, given the commitment list $\mathsf{List}_{\mathsf{cm}}$ and the challenge τ, verifies the proof. Formally we describe the protocol as an interactive Σ-protocol in Figure 2.

D. Commit-and-prove SNARK for our protocol

In this section, we extend our protocol to commit-and-prove SNARK for multiple Pedersen commitments. Initially, we assume that there exists an external commitment scheme, denoted by Com_{ext}, which satisfies the properties of binding and hiding. The algorithms of Com_{ext} consists of two algorithms

Initialization: A trusted party $\mathcal{T}$, given a relation $\mathcal{R}_{\mathsf{Batch}}$ and a security parameter 1^λ, generates $(\mathsf{ck}, \mathsf{ek}, \mathsf{vk})$ through $\Pi_{\mathsf{cc}}.\mathsf{SetUp}$. The commmitment key ck can be split into $(\mathsf{ck}_1, \mathsf{ck}_2)$. We denote l-Pedersen commitments by $\mathsf{List}_{\mathsf{cm}} := \{\mathsf{cm}_i\}_{i \in [l]}$, each based on the batching key ck_1.

Prover and Verifier: A prover $\mathcal{P}$ knows the message m_i and the opening o_i corresponding to each commitment cm_i, as well as an auxiliary instance $\boldsymbol{x}$ and a witness $\boldsymbol{w}$. $\mathcal{P}$ then follows the subsequent procedure to generate a zk-SNARK proof π. The prover and verifier run the following Σ-protocol.

1) $\mathcal{P}$ chooses a random $\widetilde{o} \xleftarrow{\$} \mathbb{F}$. Then $\mathcal{P}$ computes a proof-dependent commitment $\widetilde{\mathsf{cm}}$ as follows, where m_i and o_i refer to the message and opening of each commitment:
$$\widetilde{\mathsf{cm}} := \mathsf{Ped.Commit}(\mathsf{ck}_2, \{m_i, o_i\}_{i \in [l]}\, ; \widetilde{o})$$
2) $\mathcal{P}$ sends a proof-dependent commitment $\widetilde{\mathsf{cm}}$ to $\mathcal{V}$.
3) $\mathcal{V}$ chooses a random $\tau \xleftarrow{\$} \mathbb{F}$, and returns it to $\mathcal{P}$.
4) $\mathcal{P}$ computes the aggregated message agg_m and opening agg_o through the powers of τ as follows:
$$\mathsf{agg}_m := \sum_{i=1}^{l} \tau^i \cdot m_i, \quad \mathsf{agg}_o := \sum_{i=1}^{l} \tau^i \cdot o_i$$
5) $\mathcal{P}$ runs $\Pi_{\mathsf{cc}}.\mathsf{Prove}(\mathsf{ek}, \boldsymbol{x}, \boldsymbol{w})$, and generates a proof π_{cc} where a witness $\boldsymbol{w} := (\boldsymbol{u}, \boldsymbol{\omega})$ and $\boldsymbol{u}$ constructed as
$$\boldsymbol{u} := \left\{\mathsf{agg}_m, \mathsf{agg}_o, \{m_i, o_i\}_{i \in [l]}\right\}$$
6) Finally, $\mathcal{P}$ returns π_{cc} to $\mathcal{V}$.
7) $\mathcal{V}$ runs the following algorithm to verify the proof π_{cc}:
   a) $\mathcal{V}$ computes $\mathsf{cm}_{\mathsf{agg}}$ using the powers of τ.
   b) Lastly $\mathcal{V}$ runs
   $$\Pi_{\mathsf{cc}}.\mathsf{Verify}(\mathsf{vk}, \boldsymbol{x}, \widetilde{\mathsf{cm}} \cdot \mathsf{cm}_{\mathsf{agg}}, \pi_{\mathsf{cc}})$$
   where $\boldsymbol{x} := \tau$

Fig. 2: Our protocol applying Σ-protocol for proving multiple Pedersen commitments

(SetUp, Com). Simply, SetUp is a function that produces a commitment key ck_{ext} suitably. The committing algorithm Com outputs a commitment $\hat{\mathsf{cm}}$ upon receiving an input message m and opening o. For practical implementations, we utilize the Pedersen vector commitment scheme described as our external commitment scheme Com_{ext}.

We aim to validate multiple pre-generated l-commitments $\mathsf{List}_{\hat{\mathsf{cm}}}$ by Com_{ext} using our protocol through a commit-and-prove approach. Typically, a conventional commit-and-prove framework requires that all l-commitments be included in the relation. Since we can prove multiple commitments within a single proof by applying our protocol, we leverage this advantage when we require commit-and-prove approach. More specifically, if our protocol verifies correctly, the verifier becomes aware of the value for $\mathsf{cm}_{\mathsf{agg}}$. If the messages $\hat{m}_i$ of l-commitments $\mathsf{List}_{\hat{\mathsf{cm}}}$ generated by $\mathsf{Com}_{\mathsf{ext}}$ and the messages m_i committed in our protocol are same, we prove their coherence not by individually proving each commitment corresponds to the same message, but by employing the same challenge used in our protocol to generate $\mathsf{cm}_{\mathsf{agg}}$. Thus, we can effectively design a commit-and-prove SNARK system using a condensed form, represented as $\hat{\mathsf{cm}}_{\mathsf{agg}}$ and $\mathsf{cm}_{\mathsf{agg}}$. This approach allows us to employ a Non-Interactive Zero-Knowledge Argument of Knowledge (NIZKAoK) to prove that two aggregated commitments under different keys correctly encode the same underlying data. The relation $\mathcal{R}_{\mathsf{Eq}}^{\mathsf{Batch}}$can be expressed as follows

$$\mathcal{R}_{\mathsf{Eq}}^{\mathsf{Batch}}(\boldsymbol{x}; \boldsymbol{w}) = \left\{ \begin{array}{c} (\hat{\mathsf{ck}}, \mathsf{ck}), \\ (\hat{\mathsf{cm}}_{\mathsf{agg}}, \mathsf{cm}_{\mathsf{agg}}); \\ (\mathsf{agg}_m, \hat{\mathsf{agg}}_o, \mathsf{agg}_o) \end{array} : \begin{array}{c} \hat{\mathsf{cm}}_{\mathsf{agg}} = \mathsf{Com}_{ext}(\hat{\mathsf{ck}}, \hat{\mathsf{agg}}_m, \hat{\mathsf{agg}}_o) \\ \mathsf{cm}_{\mathsf{agg}} = \mathsf{Com}(\mathsf{ck}, \mathsf{agg}_m, \mathsf{agg}_o) \end{array} \right\}$$

We can prove the above relation $\mathcal{R}_{\mathsf{Eq}}^{\mathsf{Batch}}$ using simple Σ-protocol, but like LegoSNARK [2] we can also use several schemes such as QA-NIZK [3], compressed-Σ protocol.

IV. Security Proof

Theorem 1. Our protocol for the relation $\mathcal{R}_{\mathsf{Batch}}$ satisfies completeness, computational special soundness, and honestly verifier zero-knowledge

Proof. We focus on special soundness and zero-knowledge in priority since completeness is relatively straightforward.

Completeness. It reduces to the completeness of commit-carrying SNARK. Therefore, the verification equation is always satisfied.

Special soundness. It reduces to the knowledge soundness of commit-carrying SNARK and the binding property of Pedersen commitments.

We define a knowledge extractor $\mathcal{E}$ that on input $\mathsf{List}_{\mathsf{cm}} \in \mathbb{G}_1^l$, and two accepting transcripts $\mathsf{Tr}_0 := (\widetilde{\mathsf{cm}}, \tau^*, \pi^*_{\mathsf{cc}})$ and $\mathsf{Tr}_1 := (\widetilde{\mathsf{cm}}, \tau', \pi'_{\mathsf{cc}})$ we must recover $\{m_i, o_i\}_{i \in [l]}$ such that

$$\{\mathsf{cm}_i = \mathsf{Ped.Commit}(\mathsf{ck}_1, m_i; o_i) \mid i \in [l]\}$$

If given the proofs are valid, by leveraging knowledge extractor for the commit-carrying SNARK proofs we can extract the following with all but ϵ_{cc} which is the extractor failed error

$$\left\{\mathsf{agg}^*_m, \mathsf{agg}^*_o, \{m^*_i, o^*_i\}_{i \in [l]}, \widetilde{o}^*\right\}, \left\{\mathsf{agg}'_m, \mathsf{agg}'_o, \{m'_i, o'_i\}_{i \in [l]}, \widetilde{o}'\right\}$$

such that

$$\begin{aligned} \widetilde{\mathsf{cm}} &= \mathsf{Ped.Commit}(\mathsf{ck}_2, \{m^*_i, o^*_i\}_{i \in [l]}\, ; \widetilde{o}^*) \\ &= \mathsf{Ped.Commit}(\mathsf{ck}_2, \{m'_i, o'_i\}_{i \in [l]}\, ; \widetilde{o}') \end{aligned}$$

However, by the binding of **Lemma** 1, the probability of having two pairs of a Pedersen commitment is *negligible*.

This means that $\{m_i^*, o_i^*\} = \{m_i', o_i'\}$ with all but negligible probability $\widetilde{\epsilon}_{\mathsf{binding}}$. Therefore we have that

$$\mathsf{agg}_m^* = \sum_{i\in[l]} \tau^{*^i} \cdot m_i^*, \qquad \mathsf{agg}_o^* = \sum_{i\in[l]} \tau^{*^i} \cdot o_i^*, \quad (2)$$

$$\mathsf{agg}_m' = \sum_{i\in[l]} \tau'^i \cdot m_i^*, \qquad \mathsf{agg}_o' = \sum_{i\in[l]} \tau'^i \cdot o_i^*. \quad (3)$$

From the verification equation, we can compute a batched commitment $\mathsf{cm}_{\mathsf{agg}}$ using the randomized aggregation for $\mathsf{List}_{\mathsf{cm}}$ and the two challenges τ^*, τ' under the commitment key ck_1. For the legibility we denote these elements by $\overline{\mathsf{cm}}_{\mathsf{agg}}^*$ and $\overline{\mathsf{cm}}_{\mathsf{agg}}'$ respectively. We can express these batched commitments as

$$\overline{\mathsf{cm}}_{\mathsf{agg}}^* = \mathsf{Ped.Commit}(\mathsf{ck}_1, \overline{\mathsf{agg}}_m^*, \overline{\mathsf{agg}}_o^*)$$
$$\overline{\mathsf{cm}}_{\mathsf{agg}}' = \mathsf{Ped.Commit}(\mathsf{ck}_1, \overline{\mathsf{agg}}_m', \overline{\mathsf{agg}}_o').$$

where

$$\overline{\mathsf{agg}}_m^* = \sum_{i\in[l]} \tau^{*^i} \cdot m_i, \qquad \overline{\mathsf{agg}}_o^* = \sum_{i\in[l]} \tau^{*^i} \cdot o_i, \quad (4)$$

$$\overline{\mathsf{agg}}_m' = \sum_{i\in[l]} \tau'^i \cdot m_i, \qquad \overline{\mathsf{agg}}_o' = \sum_{i\in[l]} \tau'^i \cdot o_i. \quad (5)$$

Since the proofs are verified, we can say that with all but $\overline{\epsilon}_{\mathsf{binding}}$,

$$\mathsf{agg}_m^* = \overline{\mathsf{agg}}_m^*, \quad \mathsf{agg}_o^* = \overline{\mathsf{agg}}_o^*, \quad \mathsf{agg}_m' = \overline{\mathsf{agg}}_m', \quad \mathsf{agg}_o' = \overline{\mathsf{agg}}_o'$$

Combining equations (2) to (5), we obtain the following result:

$$\sum_{i\in[l]} \left(\tau^{*^i} - \tau'^i\right)(m_i^* - m_i) = 0,$$
$$\sum_{i\in[l]} \left(\tau^{*^i} - \tau'^i\right)(o_i^* - o_i) = 0$$

Since the challenges (τ^*, τ') are distinct, $\left(\tau^{*^i} - \tau'^i\right)$ terms cannot be 0, and for all $i \in [l]$

$$m_i^* = m_i' = m_i, \qquad o_i^* = o_i' = o_i.$$

Therefore $\mathcal{E}$ extracts a valid witness for the commitment list $(\mathsf{List}_{\mathsf{cm}})$ with error $\epsilon = \epsilon_{\mathsf{cc}} + \widetilde{\epsilon}_{\mathsf{binding}} + \overline{\epsilon}_{\mathsf{binding}}$. □

Zero-knowledge. Informally we show that it is hard for an adversary $\mathcal{A}$ to distinguish simulated transcripts from real transcripts generated by an honest prover via a hybrid argument on the distribution of prover transcripts. Note that we denote a simulator as $\mathcal{S}$, which can choose $\widetilde{\mathsf{cm}}, \tau$ and simulate π_{cc} for the statement $\mathsf{List}_{\mathsf{cm}}$.

- Game_0: An honestly-generated prover transcript is tuple $(\widetilde{\mathsf{cm}}, \tau, \pi_{\mathsf{cc}})$.
- Game_1: π_{cc}^* is computed using $\mathcal{S}$ for $(\mathsf{List}_{\mathsf{cm}}, \widetilde{\mathsf{cm}}, \tau)$. The two distributions are indistinguishable by zero-knowledge of π_{cc}.

Since the simulated transcript is indistinguishable from real transcript via hybrid argument, our protocol satisfies honestly verifier zero-knowledge.

<u>$\mathsf{NILP.SetUp}(\mathcal{R}) \to \boldsymbol{\sigma}$</u>

$\alpha, \beta, \gamma, \delta, \boxed{\eta}, x \xleftarrow{\$} \mathbb{F}$, and define $y_i(x) := \beta a_i(x) + \alpha b_i(x) + c_i(x)$

$$\boldsymbol{\sigma}_1 \leftarrow \left(\begin{matrix} 1, \alpha, \beta, \delta, \{x^i\}_{i=1}^{d-1}, \\ \left\{\frac{y_i(x)}{\gamma}\right\}_{i=1}^{l}, \left\{\frac{y_i(x)}{\delta}\right\}_{i=l+1}^{n}, \left\{\frac{x^i t(x)}{\delta}\right\}_{i=0}^{d-2}, \boxed{\frac{\eta}{\gamma}, \frac{\eta}{\delta}} \end{matrix} \right)$$

$$\boldsymbol{\sigma}_2 \leftarrow \left(1, \beta, \gamma, \delta, \left\{x^i\right\}_{i=1}^{d-1}\right)$$

return $\boldsymbol{\sigma} := (\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2) \in \mathbb{F}^{(m+2d+6)} \times \mathbb{F}^{d+4}$

<u>$\mathsf{NILP.PrfMtx}(\mathcal{R}, \boldsymbol{\sigma}, \boldsymbol{w}) \to (\boldsymbol{\Pi}_1, \boldsymbol{\Pi}_2)$</u>

parse $\boldsymbol{w}$ as $(\boldsymbol{u}, \boldsymbol{\omega}), \boldsymbol{\sigma}$ as $(\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2)$, and $r, s, \boxed{v} \xleftarrow{\$} \mathbb{F}$

$\boldsymbol{\Pi}_1 \in \mathbb{F}^{3\times(m+2d+6)}, \boldsymbol{\Pi}_2 \in \mathbb{F}^{1\times(d+4)}$

s.t. $(A, C, D)^\top = \boldsymbol{\Pi}_1 \cdot \boldsymbol{\sigma}_1, B = \boldsymbol{\Pi}_2 \cdot \boldsymbol{\sigma}_2$

$$A \leftarrow \alpha + \sum_{i=0}^{n} w_i a_i + r\delta \; ; \; B \leftarrow \beta + \sum_{i=0}^{n} w_i b_i(x) + s\delta$$

$$C \leftarrow \sum_{i=l+1}^{n} w_i \frac{y_i(x)}{\delta} + \sum_{i=0}^{d-2} \frac{h_i x^i t(x)}{\delta} + As + Br - rs\delta - \boxed{\frac{v\eta}{\delta}}$$

$$\boxed{D \leftarrow \sum_{i=0}^{l} \frac{w_i y_i(x)}{\gamma} + \frac{v\eta}{\gamma}}$$

<u>$\mathsf{NILP.Test}(\mathcal{R}) \to \mathsf{true/false}$</u>

check $A \cdot B = \alpha \cdot \beta + C \cdot \delta + D \cdot \gamma$

Fig. 3: A NILP tailored for an augmented QAP relation, underpinning the ccGro16 [2]. The boxed elements indicate terms introduced in the modification from Gro16 [12] to the construction of ccGro16.

V. Instatiation based on ccGro16 [12, 2]

A. Overview

LegoSNARK [2] introduces the commit-carrying SNARK version (ccGro16) based on the SNARK of Gro16. The scheme is constructed from the Non-Interactive Linear Proof (NILP), a cryptograhpic primitive of Gro16.

Non-Interactive Linear Proof (NILP). A NILP comprises a tuple of algorithms (SetUp, PrfMtx, Test) operating in the following manner:

- SetUp: takes a relation $\mathcal{R}$ (e.g., QAP) as input, and returns vectors $\boldsymbol{\sigma} := (\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2) \in \mathbb{F}^{\kappa_1} \times \mathbb{F}^{\kappa_2}$.
- PrfMtx: given a relation $\mathcal{R}$ and a pair $(\boldsymbol{x}, \boldsymbol{w})$, outputs two matrices $(\boldsymbol{\Pi}_1, \boldsymbol{\Pi}_2) \in \mathbb{F}^{m_1\times\kappa_1} \times \mathbb{F}^{m_2\times\kappa_2}$. This facilitates the computation of a proof $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$ as $(\boldsymbol{\Pi}_1 \cdot \boldsymbol{\sigma}_1, \boldsymbol{\Pi}_2 \cdot \boldsymbol{\sigma}_2)$.
- Test: upon receiving a relation $\mathcal{R}$ and a statement $\boldsymbol{x}$, yields a set of matrices $T_1, \ldots, T_\mu \in F^{(m_1+\kappa_1)\times(m_2+\kappa_2)}$, with the acceptance condition for a proof $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$ being $(\boldsymbol{\sigma}_1^\top, \boldsymbol{\pi}_1^\top) \cdot T_i \cdot (\boldsymbol{\sigma}_2^\top, \boldsymbol{\pi}_2^\top) = 0$ for all $i = \{1, \ldots, \mu\}$.

Also, NILP satisfies completeness, statistical knowledge soundness, and zero-knowledge properties.

Generic SNARK compiler based on NILP. A compiler for constructing a generic SNARK based on NILP is depicted in Figure 4.

Here is a brief overview of how commit-carrying SNARK based on Groth16 work, as described in figure 3. This

$\underline{\mathsf{SetUp}(\mathcal{R}) \to \boldsymbol{\sigma}}$

$(\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2) \xleftarrow{\$} \mathsf{NILP.SetUp}(\mathcal{R})$
return $\boldsymbol{\sigma} := ([\boldsymbol{\sigma}_1]_1, [\boldsymbol{\sigma}_2]_2)$

$\underline{\mathsf{Prove}(\mathcal{R}, \boldsymbol{\sigma}, \boldsymbol{x}, \boldsymbol{w}) \to \pi}$

$(\boldsymbol{\Pi}_1, \boldsymbol{\Pi}_2) \xleftarrow{\$} \mathsf{NILP.PrfMtx}(\mathcal{R}, \boldsymbol{x}, \boldsymbol{w})$
$[\boldsymbol{\pi}_1]_1 \leftarrow \boldsymbol{\Pi}_1 \cdot [\boldsymbol{\sigma}_1]_1$
$[\boldsymbol{\pi}_2]_2 \leftarrow \boldsymbol{\Pi}_2 \cdot [\boldsymbol{\sigma}_1]_2$
return $\pi := ([\boldsymbol{\pi}_1]_1, [\boldsymbol{\pi}_2]_2)$

$\underline{\mathsf{Verify}(\boldsymbol{\sigma}, \boldsymbol{x}, \boldsymbol{w}) \to \mathsf{true/false}}$

$T_1, .., T_\mu \xleftarrow{\$} \mathsf{NILP.Test}(\mathcal{R}, \boldsymbol{x}), L = ([\boldsymbol{\sigma}_1]_1, [\boldsymbol{\pi}_1]_1)^\top, R = ([\boldsymbol{\sigma}_2]_2, [\boldsymbol{\pi}_2]_2)^\top$
return $\forall i : [0]_{T_i} \stackrel{?}{=} L \cdot T_i \cdot R$

Fig. 4: A compiler a split NILP to SNARK in asymmetric groups

construction aims to design a commit-carrying SNARK that provide double binding when proving the satisfiability of QAP relations s.t. $\mathcal{R}(\boldsymbol{x}, (\boldsymbol{u}, \boldsymbol{w}))$. This scheme includes a binding commitment to a portion $\boldsymbol{u}$ of the witness, with the public input being void (i.e. $\boldsymbol{x} = \perp$). LegoSNARK leverages the fact that witness encoded polynomials are linearly independent, and its structure can be seen as linear group encoding (e.g., Pedersen commitment). Therefore, LegoSNARK adds a blinding factor, reconstructs the common reference string crs, and generates a new term $[D]_1$, which is a proof-dependent commitment. The term $[D]_1$ is structurally similar to a Pedersen commitment and is verified through the following verification equation as

$$\mathsf{Ped.VerCom}(\mathsf{ck}, [D]_1, \boldsymbol{u}, v) \stackrel{?}{=} \mathsf{true/false}$$

B. Our instantiation: $\mathsf{Gro16}_{\mathsf{cc}}^{\mathbf{Lego\text{-}DLC}}$

We introduce $\mathsf{Gro16}_{\mathsf{cc}}^{\mathbf{Lego\text{-}DLC}}$ described in figure 5, which is designed from ccGro16 by applying our Σ-protocol. The commitment key ck generated during SetUp can be viewed as $\left\{\frac{y_i(x)}{\gamma}, \frac{\eta}{\gamma}\right\}$. Without loss of generality, assume the public input consists solely of τ and the non-committed witness is empty. We denote the starting indices for ck_1 and ck_2 as pfx_1 and pfx_2, respectively. Thus ck_1 and ck_2 are represented as follows

$$\mathsf{ck}_1 := \left\{\frac{y_{\mathsf{pfx}_1+i}(x)}{\gamma}\right\}, \mathsf{ck}_2 := \frac{\eta}{\gamma}, \left\{\frac{y_{\mathsf{pfx}_2+i}(x)}{\gamma}\right\}$$

Each commitment cm_i in $\mathsf{List}_{\mathsf{cm}}$ is computed under ck_1. The proof-dependent commitment $\widetilde{\mathsf{cm}}$ is computed as $\widetilde{\mathsf{cm}} = \mathsf{Ped.Commit}\ (\mathsf{ck}_2, \{m_i, o_i\}; v)$ where v is an opening chosen by ccGro16. In the verification, the final proof-dependent commitment $[D]_1$ can be computed by computing $\mathsf{cm}_{\mathsf{agg}}$ and performing a group addition of three group elements: $[\mathsf{cm}_{\mathsf{agg}}]_1$, $[\widetilde{\mathsf{cm}}]_1$, and $[PI]_1$, where $[PI]_1$ represents the result of the linear encoding of the public input.

Initialization: A trusted party $\mathcal{T}$, given an arbitrary relation $\mathcal{R}_{\mathsf{Batch}}$ and a security parameter 1^λ, generates $(\mathsf{ck}, \mathsf{ek}, \mathsf{vk})$ through $\boldsymbol{\Pi}_{\mathsf{ccGro16}}.\mathsf{SetUp}$ where ck consists of $(\mathsf{ck}_1, \mathsf{ck}_2)$.

Prover and Verifier: The prover and verifier execute the Σ-protocol described in Figure 2, applying the specific procedures outlined below, with all other remaining processes the same.

- In committing phase, $\mathcal{P}$ computes a proof-dependent commitment $\widetilde{\mathsf{cm}}$ by mapping each commitment's message $\boldsymbol{u}$ and opening o as follows:
$$\widetilde{\mathsf{cm}} := v \cdot \frac{\eta}{\gamma} + \sum_{i=0}^{l-1} \left(u_i \cdot \frac{y_{\mathsf{pfx}+2i}(x)}{\gamma} + o_i \cdot \frac{y_{\mathsf{pfx}+2i+1}(x)}{\gamma} \right)$$
where pfx refers to the prefix index related to the committed witness for each of cm.
- In proving phase, $\mathcal{P}$ runs $\boldsymbol{\Pi}_{\mathsf{ccGro16}}.\mathsf{Prove}(\mathsf{ek}, \boldsymbol{x}; \boldsymbol{w})$, and generates a proof $\pi_{\mathsf{cc}} := ([A]_1, [B]_2, [C]_1)$ in which $\mathcal{P}$ uses the chosen opening v in committing phase.
- In verification phase, the verifier $\mathcal{V}$ computes $\mathsf{cm}_{\mathsf{agg}}$ using binary encoding technique, and then $\mathcal{V}$ generates a $[D]_1$ such that
$$[D]_1 := [\mathsf{cm}_{\mathsf{agg}}]_1 + [\widetilde{\mathsf{cm}}]_1 + [PI]_1$$
Lastly, $\mathcal{V}$ runs the following verification as
$$e([A]_1, [B]_2) \stackrel{?}{=} e([a]_1, [b]_2) \cdot ([D]_1, [d]_2) \cdot ([C]_1, [c]_2)$$
where $[a]_1, [b]_2, [c]_1, [d]_1$ denote the group elements within the verification key vk of Gro16.

Fig. 5: $\mathsf{Gro16}_{\mathsf{cc}}^{\mathbf{Lego\text{-}DLC}}$: Our construction from based on ccGro16

VI. Instatiation based on Plonk [5]

A. Overview

Plonk [5] is a universal SNARK, which use polynomial commitment scheme to prove knowledge of any arbitrary relation $\mathcal{R}$. Polynomial commitment scheme (PC) enables a prover to generate a commitment for a polynomial, valid at any given point of evaluation. Subsequently, the prover sends an opening proof for the verifier's evaluation. If the used PC is under group linear encoding for the public parameters such as KZG commitment [6], our batch commit-carrying SNARK scheme can also be applied in Plonk.

In the Plonk protocol, the prover proves knowledge of fan-in 2 and fan-out 1 gate values for each of the N gates. The constraint verification within Plonk is divided into *gate constraints* and *copy constraints*. PC is used to prove the validity of two constraints. We briefly describe Plonk [5] protocol.

Constraint system. Plonk designs its own constraint system that requires satisfying the following equation through the use of *selector vectors* $(\boldsymbol{q}_l, \boldsymbol{q}_r, \boldsymbol{q}_o, \boldsymbol{q}_m, \boldsymbol{q}_c)$ and *wire vectors* $(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$

$$q_{l,i} \cdot a_i + q_{r,i} \cdot b_i + q_{o,i} \cdot c_i + q_{m,i} \cdot (a_i b_i) + q_{c,i} = 0$$

where we denote the left, right, and output wire as $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$.

Lagrange basis. Given a characteristic q of $\mathbb{F}$ and n satisfying $q \equiv 1 \mod n$, the multiplicative group of $\mathbb{F}^*$ reduces a subgroup $\mathbb{H} = \{\zeta, \zeta^2, \ldots, \zeta^n\}$ generated by an n-th primitive root of unity $\zeta \in \mathbb{F}^*$. By $\mathbb{H}$, we can construct a zero-polynomial $z_{\mathbb{H}}(X) = X^n - 1$, which can be expressed as $X^n - 1 = \prod_{i=1}^{n}(X - \zeta^i)$. There exists a Lagrange basis $Ł_i(X)$ for each $i \in [n]$ such that:

$$L_i(X) = \frac{\zeta^i(X^n - 1)}{n(X - \zeta^i)}$$

where

$$L_i(\zeta^i) = 1 \wedge L_i(\zeta^j) = 0 \ (i \neq j)$$

Copy constraint. Let multiple polynomials be $\boldsymbol{f} = (f_1, f_2, \ldots, f_\ell) \in \mathbb{F}[X]_\ell$ and $\sigma : \ell n \to \ell n$ be a permutation. For $\boldsymbol{g} = (g_1, g_2, \ldots, g_\ell) \in \mathbb{F}[X]_\ell$, we say that $\boldsymbol{f} = \sigma(\boldsymbol{g})$ if for each $i \in [n], j \in [\ell]$ the following holds,

$$f_{((j-1)\cdot n+i)} := f(\zeta^i)_j \quad g_{((j-1)\cdot n+i)} := g_j(\zeta^i), \quad \forall l \in [\ell n] : g_l = f_{\sigma(l)}$$

Initialization: A trusted party $\mathcal{T}$, given a security parameter 1^λ, generates a commitment key ck through $\mathbf{\Pi}_{PC}.\mathsf{SetUp}$ where ck consists of $(\mathsf{ck}_1, \mathsf{ck}_2)$.

Prover and Verifier: The prover and verifier execute the Σ-protocol described in Figure 2, applying the specific procedures outlined below, with all other remaining processes the same.

- In committing phase, $\mathcal{P}$ computes a proof-dependent commitment $\widetilde{\mathsf{cm}}$ by mapping each commitment's message $\boldsymbol{u}$ and opening v_1, v_2 as follows:

$$\widetilde{\mathsf{cm}} := (v_1 X + v_2) z_{\mathbb{H}}(X) + \sum_{i=0}^{l-1} \left(u_i \cdot L_{\mathsf{pfx}_{2,2i}}(X) + o_i \cdot L_{\mathsf{pfx}_{2,2i+1}}(X) \right)$$

  where pfx refers to the position related to the committed witness for each of cm.
- During the proving phase, the prover $\mathcal{P}$ runs $\mathbf{\Pi}_{\mathsf{Plonk}}.\mathsf{Prove}(\mathsf{ek}, \boldsymbol{x}, \boldsymbol{w})$ to generate a Plonk proof π using a commitment and an evaluation to the polynomial defined as

$$f_u(X) := (v_1 X + v_2) z_{\mathbb{H}}(X) + \sum_{i \in [l]} -u_i L_i(X)$$

  Here, $\mathcal{P}$ utitlizes the chosen opening v during the committing phase.
- In verification phase, the verifier $\mathcal{V}$ computes a batched commitment $[u_{\mathsf{agg}}]_1$ using binary encoding technique, and then $\mathcal{V}$ generates a $[u]_1$ such that

$$[u]_1 := [u_{\mathsf{agg}}]_1 + [\widetilde{\mathsf{cm}}]_1$$

  Lastly, $\mathcal{V}$ verifies the proof with $[u]_1$.

Fig. 6: $\mathsf{Plonk}_{\mathsf{cc}}^{\mathbf{Lego\text{-}DLC}}$: Our construction based on Plonk

B. *Our instantiation:* $\mathsf{Plonk}_{\mathsf{cc}}^{\mathbf{Lego\text{-}DLC}}$

Plonk leverage a polynomial commitment scheme (PC). If the PC employs the [6] scheme, which uses a discrete log-based group encoding of polynomial, a polynomial $p(X)$ can be represented as follows, with a commitment key $\mathsf{ck} := \left\{g^{x^i}\right\}_{i \in [n]}$:

$$\mathsf{PC.Com}(\mathsf{ck}, p(X)) := \prod_{i=0}^{n-1} g^{p_i \cdot x^i}$$

In the Plonk protocol, there exist the wire polynomials, which can be shortly expressed as follows, with random blinding scalars $(v_1, v_2, \ldots, v_6) \in \mathbb{F}$

$$f_L(X) = (v_1 X + v_2) z_{\mathbb{H}}(X) + \sum_{i \in [n]} w_i L_i(X),$$

$$f_R(X) = (v_3 X + v_4) z_{\mathbb{H}}(X) + \sum_{i \in [n]} w_{n+i} L_i(X),$$

$$f_O(X) = (v_5 X + v_6) z_{\mathbb{H}}(X) + \sum_{i \in [n]} w_{2i+i} L_i(X)$$

For the public input polynomial $f_{pi}(X)$, we can separate the public input $\boldsymbol{pi} \in \mathbb{F}^l$ into two parts $(\boldsymbol{x}, \boldsymbol{u})$, where we regard $\boldsymbol{u}$ as committed witness. As in previous descriptions, assume that the public input consists solely of τ, and the non-committed witness is empty. We denote the starting indices for ck_1 and ck_2 as pfx_1 and pfx_2, respectively. By integrating blinding factors (v_1, v_2) into the committed witness encoded polynomial $f_u(X)$ to ensure zero-knowledge, its form aligns with that of wire polynomials such as

$$f_u(X) = (v_1 X + v_2) z_{\mathbb{H}}(X) + \sum_{i \in [l]} -u_i L_i(X)$$

If we employ the KZG10 scheme, the polynomial commitment of $f_u(X)$ can be recast as a Pedersen vector commitment using a specific commitment key ck as

$$\left\{[z_{\mathbb{H}}(X)], [z_{\mathbb{H}}(X)X], [(L_i(X))_{i \in [n]}]\right\}$$

We can split the commitment key into (ck_1, ck_2) as

$$\mathsf{ck}_1 := [L_{\mathsf{pfx}_{1,i}}(X)], \mathsf{ck}_2 := [z_{\mathbb{H}}(X)], [z_{\mathbb{H}}(X)X], \left\{[L_{\mathsf{pfx}_{2,i}}(X)]\right\}$$

Hence we can construct a proof-dependent commitment for the committed witness u, as $\widetilde{\mathsf{cm}} := \mathsf{Ped.Commit}(\mathsf{ck}_2, \{m_i, o_i\}; \widetilde{o})$ where the opening $\widetilde{o}$ consists of (s_1, s_2).

VII. Experiment

We implement $\mathsf{SNARK}_{\mathsf{cc}}^{\mathsf{Batch}}$ based on top of Rust `Arkworks` library[2], which provides the useful cryptographic primitives such as finite field and elliptic curve. We adopted BN254 and BLS12-381, which are pairing-friendly elliptic curve offering 128 bits of security. BN254 is used to compare linking proof systems on a single computing platform, while BLS12-381 is utilized for comparing proving and verifying times using two distinct computing platforms. Specifications for these devices are presented in table I. We implement and

[2]https://github.com/arkworks-rs

evaluate our scheme based on Groth16 [12] and Plonk [5], which we denote as $\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$ and $\mathsf{Plonk}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$, respectively.

TABLE I: Device specification

Device	Specification
Device 1: Mac Pro (2021)	OS: macOS 14.4.1 CPU: M1 Pro RAM: 32 GB
Device 2: iMac (2019)	OS: macOS 14.4.1 CPU: i9-9900K RAM: 64 GB

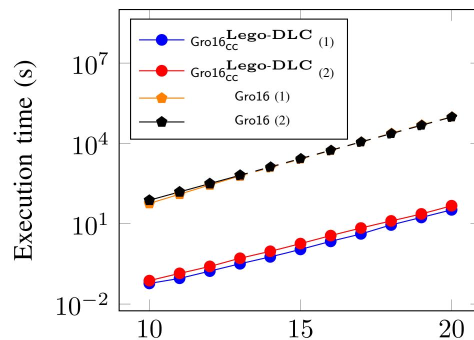
A. Microbenchmark

Execution time. Figure 7 illustrates the performance for varying batch size, showing an increase in execution time as the batch size expands. To more explicitly demonstrate our performance, we also add results measured using a naive approach (i.e., in-the-circuit) within Groth16 [12] and Plonk [5]. In the scheme, although the number of constraints grows linearly with each exponential increment in batch size, the operations (e.g., group scalar exponentiation) in the naive approach are more expensive than the constraints required in our system. Specifically, for Groth16, we can prove commitments for 2^{20} batches in approximately 33.024s, whereas the estimated time for the naive approach, which can be not measured in our device, would be around $98,000$s showing a high performance improvement where the dashed line is estimated. In Plonk, similar improvements are observed; for a batch size of 2^{10}, the performance difference is about 120x, with time of 1.492s compared to 182.078s respectively.

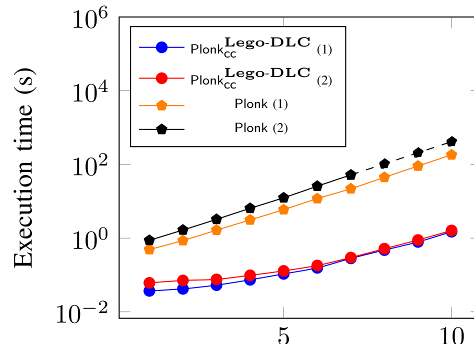
B. Comparison

To better demonstrate the practicality of our system, we have conducted a comparative analysis with the widely recognized LegoSNARK in table III to VI. For implementing LegoSNARK, denoted by **Lego16**, we utilize ccGro16 as the commit-carrying SNARK. **Lego16** is further divided into $\textbf{Lego16}_{\mathsf{QA}}$ and $\textbf{Lego16}_{\mathsf{Comp}}$, corresponding to QA-NIZK [3] and Compressed-Σ protocol for the linking proof system, respectively.

Asymptotic performance. We give an asymptotic performance in table II. Both **Lego16** and $\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$ are based on ccGro16. Considering $\mathcal{P}$'s work, **Lego16** requires more $2l + 2\ E_1$ for QA-NIZK and $8l + 4\log l - 9\ E_1$ for Compressed-Σ, while $\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$ has additional $3l + 3$ gates and $5l + 4$ wires (i.e., additional $11l + 10\ E_1, 3l + 3\ E_2$ needed). The proof size in $\textbf{Lego16}_{\mathsf{QA}}$ and $\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$ are constant, with only $1\ \mathbb{G}_1$ increased for QA-NIZK. In contrast, $\textbf{Lego16}_{\mathsf{Comp}}$ increases by $4\ \mathbb{F}$ and $4\log l + 2\ \mathbb{G}_1$ for Compressed-Σ protocol. However, There is significant improvement in $\mathcal{V}$'s work. To verify the proof, $\textbf{Lego16}_{\mathsf{QA}}$ performs $l + 2$ pairings for linking and $4l + 4\log l$ times $\mathbb{G}_1$ exponentiation for $\textbf{Lego16}_{\mathsf{Comp}}$, while our scheme requires only $l\ E_1$ for aggregation.



(a) Prover time for Groth16 [13] and our scheme ($\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$)



(b) Prover time for Plonk [5] and our scheme ($\mathsf{Plonk}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$)

Fig. 7: Prover time for varying the number of batch size, where the x-axis represents the batch size in $\log$ and the dashed line means the estimated value. Note that (1) and (2) refer to devices in the table I, respectively.

TABLE II: Comparison of **Lego16** and $\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$. $\mathcal{P}$ and $\mathcal{V}$ represent additional operations on ccGro16. $|\pi|$ represents the proof size of each scheme added to ccGro16. We denote by l the number of commitments, where E refers to group exponentiation and P refers to pairing.

	$\mathcal{P}$	$\mathcal{V}$	$\lvert\pi\rvert$
$\textbf{Lego16}_{\mathsf{QA}}$	$O(l)\ E_1$	$l + 2\ P$	$1\ \mathbb{G}_1$
$\textbf{Lego16}_{\mathsf{Comp}}$	$O(l)\ E_1$	$3l + 7\log l\ E_1$	$4\log l + 2\ \mathbb{G}1, 4\ \mathbb{F}$
$\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$	$O(l)\ E_1,\ O(l)\ E_2$	$l + 1\ E_1$	-

Prover and Verifier time. When comparing prover times in the table III, $\textbf{Lego16}_{\mathsf{QA}}$ outperforms our scheme due to the absence of additional operations required to generate aggregated elements for a batched commitment in our approach. Interestingly, in $\mathcal{P}$'s time, $\textbf{Lego16}_{\mathsf{Comp}}$ has asymptotically fewer computation than our scheme, but it is slower. This is because the aggregation of commitments can be computed more quickly using multi-scalar exponentiation. Specifically, based on 2^{16}, our prover time is 1.413 seconds, but $\textbf{Lego16}_{\mathsf{QA}}$ shows performance of about 0.177 seconds while $\textbf{Lego16}_{\mathsf{Comp}}$ performs about 2.476 seconds. However, due to the reliance on a linkable proof system, our scheme exhibits superior performance in verifier time in the table IV.

Key size. Examining the key sizes from the table V and VI, the proving key sizes generated by Gro16 are $4\times$ to $6\times$ bigger in our scheme, which necessitates additional keys due

TABLE III: Comparison of prover times between **Lego16** and $\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$ for varying batch sizes in log scale.

Batch size (log)	$\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$ (s)	$\textbf{Lego16}_{\mathsf{QA}}(s)$	$\textbf{Lego16}_{\mathsf{Comp}}(s)$
7	0.01	0.002	0.02
8	0.014	0.002	0.028
9	0.023	0.005	0.041
10	0.038	0.005	0.063
11	0.063	0.007	0.113
12	0.117	0.013	0.209
13	0.223	0.022	0.344
14	0.395	0.044	0.652
15	0.746	0.084	1.28
16	1.413	0.177	2.476

TABLE IV: Comparison of verifier times between **Lego16** and $\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$ for varying batch sizes in log scale.

Batch size (log)	$\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}(s)$	$\textbf{Lego16}_{\mathsf{QA}}(s)$	$\textbf{Lego16}_{\mathsf{Comp}}(s)$
7	0.002	0.006	0.012
8	0.003	0.009	0.017
9	0.003	0.017	0.026
10	0.004	0.032	0.043
11	0.005	0.062	0.077
12	0.007	0.122	0.145
13	0.011	0.246	0.27
14	0.021	0.49	0.513
15	0.036	0.98	1.028
16	0.064	1.972	2.017

to the nature of aggregating commitments in the circuit. The verifying key size in our scheme could be reduced to constant $296KB$ if there is no need to maintain the committing key. In contrast, the sizes of both $\textbf{Lego16}_{\mathsf{QA}}$ and $\textbf{Lego16}_{\mathsf{Comp}}$ scale linearly with the batch size in the linking proof system

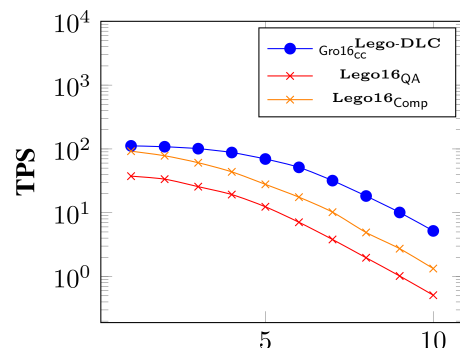
TABLE V: Comparison of proving key sizes for varying batch sizes

Batch Size (log)	$\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$ pk (KB)	$\textbf{Lego16}_{\mathsf{QA}}$ pk (KB)	$\textbf{Lego16}_{\mathsf{Comp}}$ pk (KB)
7	130	34	21
8	259	66	42
9	517	132	83
10	1,033	264	164
11	2,065	526	328
12	4,129	1,050	656
13	8,258	2,099	1,311
14	16,516	4,196	2,622
15	33,031	8,390	5,244
16	66,061	16,779	10,486

TABLE VI: Comparison of verifying key sizes for varying batch sizes

Batch Size	$\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$		$\textbf{Lego16}_{\mathsf{QA}}$		$\textbf{Lego16}_{\mathsf{Comp}}$
(log)	vk (B)	ck (KB)	vk (KB)	ck (KB)	vk (KB)
7	296	8	9	4	5
8		17	17	8	9
9		33	33	16	17
10		66	66	33	33
11		131	131	66	66
12		262	263	131	131
13		524	525	262	263
14		1,049	1,049	524	525
15		2,097	2,098	1,049	1,049
16		4,194	4,195	2,097	2,098

Application. We provide detailed performance metrics in applications such as verifying proofs on blockchain platforms. We consider a scenario within smart contracts on the blockchain where users' commitments are stored, and a prover (e.g., bank, authority, etc.) must demonstrate the validity of these commitments by including proofs in transactions. This scenario aligns with simplified versions of applications such as proof of solvency or digital credentials. To compare performance, we have measured the transactions per second (TPS) and gas costs for each system. For our experimental measurements, we utilize the Hardhat testnet. Specifically we generate $1,000$ transactions and measure the time taken for these transactions to be confirmed on the blockchain network. Additionally, in this experiment, we utilize the BN254 curve, which is particularly advantageous as it is supported by precompiled functions in smart contract. TPS is computed by dividing the total number of transactions by the total time taken for their processing.



(a) Transaction per second for **Lego16** and $\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$

Batch size (log)	$\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$	$\textbf{Lego16}_{\mathsf{QA}}$	$\textbf{Lego16}_{\mathsf{Comp}}$
1	294K	488K	425K
2	309K	583K	538K
3	340K	773K	726K
4	403K	1,161K	1,067K
5	527K	1,918K	1,713K
6	777K	3,440K	2,966K
7	1,283K	6,485K	5,441K
8	2,282K	12,579K	10,350K
9	4,308K	24,779K	20,152K
10	8,323K	49,214K	39,752K

(b) Gas costs for **Lego16** and $\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$

Fig. 8: Performance for varying the number of batch size.

Figure 8 shows the performance comparison between **Lego16** and our scheme ($\mathsf{Gro16}_{\mathsf{cc}}^{\textbf{Lego-DLC}}$). As batch sizes increase, the difference in TPS becomes more pronounced. For instance, at a batch size of 2^{10}, $\textbf{Lego16}_{\mathsf{QA}}$ and $\textbf{Lego16}_{\mathsf{Comp}}$ can handle approximately 0.51 and 1.334 transactions per second respectively, whereas our scheme can process about 5.19 transactions per second, which indicates that we can verify about 5,300 commitments per second. This notable performance discrepancy is due to the computational overhead. $\textbf{Lego16}_{\mathsf{QA}}$ needs $O(l)$ pairings and $\textbf{Lego16}_{\mathsf{Comp}}$ needs $O(l)$ group exponentiations to rescale commitment keys, whereas in our scheme, $O(l)$ group exponentiations with a smaller constant factor is performed to aggregate commitments.

Additionally, concerning verification key (vk), $\textbf{Lego16}_{\mathsf{QA}}$

and $\mathbf{Lego16}_{\mathsf{Comp}}$ require the number of $\mathbb{G}_2$ and $\mathbb{G}_1$ elements that scales linearly with l.

VIII. Conclusion

Our paper proposes a batching module, **Lego-DLC**, which can efficiently prove and verify multiple commitments. As batch sizes increase, our performance surpasses that of other works in terms of verifier efficiency (i.e., time, key size) and proof size, although the proving time is slightly longer than in other works. Our work holds significant potential for applications that demand efficient proving and verification, particularly when dealing with numerous commitments. It offers a far more efficient approach compared to the traditional method of verifying each commitment individually. Consequently, our module proves to be highly effective in applications that heavily rely on the use of commitments, such as distributed ledgers.

References

[1] A. M. Odlyzko, Ed., *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, ser. Lecture Notes in Computer Science, vol. 263. Springer, 1987.

[2] M. Campanelli, D. Fiore, and A. Querol, "Legosnark: Modular design and composition of succinct zero-knowledge proofs," Cryptology ePrint Archive, Report 2019/142, 2019, https://eprint.iacr.org/2019/142.

[3] E. Kiltz and H. Wee, "Quasi-adaptive nizk for linear subspaces revisited," in *Advances in Cryptology - EUROCRYPT 2015*, E. Oswald and M. Fischlin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 101–128.

[4] D. F. Aranha, E. M. Bennedsen, M. Campanelli, C. Ganesh, C. Orlandi, and A. Takahashi, "Eclipse: Enhanced compiling method for pedersen-committed zksnark engines." Springer-Verlag, 2022.

[5] A. Gabizon, Z. J. Williamson, and O.-M. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 953, 2019.

[6] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *Advances in Cryptology - ASIACRYPT 2010*, M. Abe, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 177–194.

[7] M. Chase, C. Ganesh, and P. Mohassel, "Efficient zero-knowledge proof of algebraic and non-algebraic statements with applications to privacy preserving credentials," in *Advances in Cryptology–CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III 36*. Springer, 2016, pp. 499–530.

[8] M. Orrù, G. Kadianakis, M. Maller, and G. Zaverucha, "Beyond the circuit: How to minimize foreign arithmetic in zkp circuits," Cryptology ePrint Archive, Paper 2024/265, 2024, https://eprint.iacr.org/2024/265. [Online]. Available: https://eprint.iacr.org/2024/265

[9] M. Jawurek, F. Kerschbaum, and C. Orlandi, "Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 955–966. [Online]. Available: https://doi.org/10.1145/2508859.2516662

[10] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn, "Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings," ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2111–2128. [Online]. Available: https://doi.org/10.1145/3319535.3339817

[11] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward, "Marlin: Preprocessing zksnarks with universal and updatable srs," in *Advances in Cryptology – EUROCRYPT 2020*, A. Canteaut and Y. Ishai, Eds. Cham: Springer International Publishing, 2020, pp. 738–768.

[12] J. Groth, "On the size of Pairing-Based non-interactive arguments," in *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, 2016, pp. 305–326.

[13] ——, "Short pairing-based non-interactive zero-knowledge arguments." in *Asiacrypt*, vol. 6477. Springer, 2010, pp. 321–340.

[14] T. Attema and R. Cramer, "Compressed Σ-protocol theory and practical application to plug & play secure algorithmics," in *Advances in Cryptology – CRYPTO 2020*, D. Micciancio and T. Ristenpart, Eds. Cham: Springer International Publishing, 2020, pp. 513–543.

[15] T. Attema, R. Cramer, and S. Fehr, "Compressing proofs of k-out-of-n partial knowledge," in *Advances in Cryptology – CRYPTO 2021*, T. Malkin and C. Peikert, Eds. Cham: Springer International Publishing, 2021, pp. 65–91.

Appendix

A. Quasi-Adaptive NIZK arguments for linear spaces

Intuitively, a QA-NIZK argument, as defined by Jutla and Roy, allow proving the membership of an instance $\boldsymbol{x}$ with a witness $\boldsymbol{w}$ in a language $\mathcal{L}$, defined by a relation $\mathcal{R}(\boldsymbol{x}, \boldsymbol{w})$. The QA-NIZK arguments consist of a set of PPT algorithms $\mathbf{\Pi}_{\cdot\mathsf{QA\text{-}NIZK}} = (\mathsf{KeyGen}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Sim})$.

Kiltz and Wee [3] introduce constructions for QA-NIZK arguments for linear spaces. The linear space language $\mathcal{L}_{LS}$ can be represented as,

$$\mathcal{L}_{LS} = \{[\boldsymbol{x}]_1 \in \mathbb{G}_1^n : \exists \boldsymbol{w} \in \mathbb{Z}_p \text{ s.t. } \boldsymbol{x} = M \cdot \boldsymbol{w}\}$$

, where the relation $\mathcal{R}$ is defined as

$$\mathcal{R}_M(\boldsymbol{x}; \boldsymbol{w}) = \left\{(\boldsymbol{x}; \boldsymbol{w}) \in \mathbb{G}_1^n \times \mathbb{Z}_p^m : \boldsymbol{x} = M \cdot \boldsymbol{w}\right\}$$

We provide the construction of the Kiltz-Wee's QA-NIZK arguments for linear subspaces in the CRS model, described in figure 9.

$\underline{\mathsf{KeyGen}([M]_1 \in \mathbb{G}_1^{n \times m}) \to \mathsf{crs}, \mathsf{td}}$

$\boldsymbol{K} \xleftarrow{\$} \mathbb{Z}_p^{n \times \hat{k}}, a \xleftarrow{\$} \mathbb{Z}_p, \boldsymbol{C} \leftarrow \boldsymbol{K} \cdot a, \boldsymbol{P} \leftarrow [M]_1^\top \cdot \boldsymbol{K} \in \mathbb{G}_1^{n \times \hat{k}}$

$\mathsf{ek} := \boldsymbol{P}, \mathsf{vk} := ([\boldsymbol{C}]_2, [a]_2), \mathsf{td} := \boldsymbol{K}$

$\mathbf{return}\ \mathsf{crs} := (\mathsf{ek}, \mathsf{vk}), \mathsf{td}$

$\underline{\mathsf{Prove}(\mathsf{ek}, \boldsymbol{x}, \boldsymbol{w}) \to [\pi]_1}$

$\pi_1 \leftarrow \boldsymbol{w}^\top \cdot P \in \mathbb{G}_1$

$\mathbf{return}\ [\pi]_1 \in \mathbb{G}_1^{\hat{k}}$

$\underline{\mathsf{Verify}(\mathsf{vk}, \boldsymbol{x}, [\pi]_1) \to (\mathsf{true}/\mathsf{false})}$

Check that $[\boldsymbol{x}]_1^\top \odot [\boldsymbol{C}]_2 \stackrel{?}{=} [\pi]_1 \odot [a]_2$

$\underline{\mathsf{Sim}([M]_1, \mathsf{td}, \boldsymbol{x}) \to [\pi]_1}$

$\pi_1 \leftarrow \boldsymbol{K}^\top \cdot [\boldsymbol{x}]_1 \in \mathbb{G}_1^{\hat{k}}$

Fig. 9: KW15 [3] QA-NIZK $\mathbf{\Pi}_{\mathsf{QA\text{-}NIZK}}$

Similar to the approach in LegoSNARK, we set $\hat{k} = 1$. In LegoSNARK, it is demonstrated that when $\hat{k} = 1$, knowledge soundness is achieved under the discrete logarithm assumption within the algebraic group model (AGM). A comparable proof for the application of this scheme in a non-falsifiable setting is also provided in KW15 [3]. We recall the proof form LegoSNARK and describe it simply as follows:

Theorem 2. Assuming that $\mathcal{D}$ is a witness-sampleable matrix distribution, under the discrete logarithm assumption in AGM, the QA-NIZK $\mathbf{\Pi}_{\mathsf{QA\text{-}NIZK}}$ from KW15 [3] (with $\hat{k} = 1$) is a knowledge-sound SNARK for the relation $\mathcal{R}_{LS}$ with matrices from $\mathcal{D}$.

Proof. Let $\mathcal{A}$ be an algebraic adversary against the knowledge soundness of $\mathbf{\Pi}_{\mathsf{QA\text{-}NIZK}}$. The adversary takes the matrix M,

the CRS (i.e., $\boldsymbol{P}$, $\boldsymbol{C}$), and the auxiliary input (aux) as inputs (i.e. $\mathcal{A}([M, \boldsymbol{P}]_1, [a, \boldsymbol{C}])$. Consider $[\boldsymbol{z}]_1$, a vector comprising M, elements from aux in the group $\mathbb{G}_1$, and the generator of $\mathbb{G}_1$. Then the adversary $\mathcal{A}$ outputs a pair $([\boldsymbol{x}]_1, [\pi]_1)$ and coefficients $\boldsymbol{w}$ that express these elements as linear combinations of its input in $\mathbb{G}_1$. We denote the coefficients for $\boldsymbol{x}$ and π by $(\boldsymbol{X}_0, \boldsymbol{X}_1)$ and $(\boldsymbol{\pi}_0, \boldsymbol{\pi}_1)$ respectively,

$$[\boldsymbol{x}]_1 = \boldsymbol{X}_0[M^\top \boldsymbol{K}]_1 + \boldsymbol{X}_1[\boldsymbol{z}]_1$$
$$[\pi]_1 = \boldsymbol{\pi}_0^\top [M^\top \boldsymbol{K}]_1 + \boldsymbol{\pi}_1^\top [\boldsymbol{z}]_1$$

Now, we define the extractor $\mathcal{E}$ that extracts the witness $\boldsymbol{\pi}_0$. Then we prove that the following probability is negligible:

$$\Pr\left[\mathsf{Verify}(\mathsf{vk}, [\boldsymbol{x}]_1, [\pi]_1) = \mathsf{true} \land [\boldsymbol{x}]_1 \neq [M] \cdot \boldsymbol{w}\right]$$

If $\mathcal{A}$ returns such a tuple with non-negligible probability, we construct an algorithm $\mathcal{B}$ that, on input $([\boldsymbol{K}]_1, [\boldsymbol{K}]_2)$, outputs the elements $(\boldsymbol{a}, \boldsymbol{b}, c)$ such that:

$$\boldsymbol{K}^\top \cdot \boldsymbol{a} \cdot \boldsymbol{K} + \boldsymbol{K}^\top \cdot \boldsymbol{b} + c = 0$$

The algorithm $\mathcal{B}$ proceeds as follows,

1) it uses $\mathcal{D}$ to sample ($[M]_1$, aux) along with its witness over $\mathbb{G}_1$, which is a vector $\boldsymbol{z}$ where each element of $\boldsymbol{z}$ is an entry from $\mathbb{Z}_p$.
2) it samples $a \xleftarrow{\$} \mathbb{Z}_p$ and runs $\mathcal{A}([\boldsymbol{z}, \boldsymbol{P}]_1, [a, a \cdot \boldsymbol{K}]_2)$.
3) Upon receiving the output from $\mathcal{A}$, $\mathcal{B}$ sets:

$$\boldsymbol{a} := \boldsymbol{X}_0 \cdot M^\top,\ \boldsymbol{b} = \boldsymbol{X}_1 \boldsymbol{z} - M \cdot \boldsymbol{\pi}_0,\ \boldsymbol{c} = -\boldsymbol{\pi}_1^\top \cdot \boldsymbol{z}$$

At least one of $\boldsymbol{a}$, $\boldsymbol{b}$, or $\boldsymbol{c}$ must be nonzero. If all are zero, then $\boldsymbol{X}_1 \boldsymbol{z} - M\boldsymbol{\pi}_0 = 0$, which implies $\boldsymbol{x} = M \cdot \boldsymbol{\pi}_0$ since $\boldsymbol{X}_0 \cdot M^\top = 0$, contradicting our assumption about $\mathcal{A}$'s output.

Using algorithm $\mathcal{B}$, we construct an algorithm $\mathcal{B}'$ that deals with the discrete logarithm problem. On input $([y]_1, [y]_2)$, the algorithm $\mathcal{B}'$ chooses $\boldsymbol{r}, \boldsymbol{s} \in \mathbb{Z}_p^n$ and sets $\boldsymbol{K} := y \cdot \boldsymbol{r} + \boldsymbol{s}$. It can be shown that $([\boldsymbol{K}]_1, [\boldsymbol{K}]_2)$ can be simulated with a distribution identical to the one expected by $\mathcal{B}$. Given a solution $(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$, one can find (a_0, b_0, c_0) such that:

$$\begin{aligned} 0 &= (y\boldsymbol{r} + \boldsymbol{s})^\top \cdot \boldsymbol{a} \cdot (y \cdot \boldsymbol{r} + \boldsymbol{s}) + (y \cdot \boldsymbol{r} + \boldsymbol{s})^\top \cdot \boldsymbol{b} + \boldsymbol{c} \\ &= a_0 \cdot y^2 + b_0 \cdot y + c_0 \end{aligned}$$

With high probability, $c_0 \neq 0$. From this $\mathcal{B}'$ can extract y.

B. Compressed-Σ protocol

Compressed-Σ protocols are interactive protocols that maintain the same functionality and remain honest-verifier zero-knowledge proofs of knowledge for a given relation $\mathcal{R}$. These protocols achieve succinct communication complexity, reducing from linear to logarithmic size.

In this section, we introduce a protocol for proving the equality of committed vectors. By the protocols proposed in [14] and [15], the proposed protocol can serve the same role as CP_{link}, providing a proof for N Pedersen commitments with a size of $O(\log N)$. Referencing the relation described in Eclipse [4], the relation $\mathcal{R}_{\mathsf{Eq}}^{\mathsf{Batch}}$ that we aim to prove can be described as follows:

$$\mathcal{R}_{\mathsf{Eq}}^{\mathsf{Batch}}(\boldsymbol{x}; \boldsymbol{w}) = \left\{ \begin{array}{c} (\boldsymbol{g}, \boldsymbol{h}, \tilde{\boldsymbol{g}}, \tilde{\boldsymbol{h}}, n, d, d_1, d_2), \\ (C, \{D_i\}_{i\in[n]}); \\ (\boldsymbol{m}, \boldsymbol{o}, \{\boldsymbol{o}_i\}_{i\in n}) \end{array} : \begin{array}{c} C = \boldsymbol{g}^{\boldsymbol{m}} \cdot \boldsymbol{h}^{\boldsymbol{o}}, D_i = \tilde{\boldsymbol{g}}^{\boldsymbol{m}_i} \cdot \tilde{\boldsymbol{h}}^{\tilde{\boldsymbol{o}}_i}, \\ \boldsymbol{g} \in \mathbb{Z}_q^{nd}, \tilde{\boldsymbol{g}} \in \mathbb{Z}_q^{d}, \\ \boldsymbol{h} \in \mathbb{Z}_q^{d_1}, \tilde{\boldsymbol{h}} \in \mathbb{Z}_q^{d_2}, \\ \boldsymbol{m} = \{\boldsymbol{m}_i\}_{i\in[n]}, \\ \boldsymbol{o} \in \mathbb{Z}_q^{d_1}, \boldsymbol{o}_i \in \mathbb{Z}_q^{d_2} \end{array} \right\}$$

The compressed version of Σ-protocol for the above relation $\mathcal{R}_{\mathsf{Eq}}^{\mathsf{Batch}}$ is described as follows.

1) The verifier $\mathcal{V}$ samples a random challenge $\delta \in \mathbb{Z}_q$, and sends it to the prover $\mathcal{P}$. Then both parties scale out $\tilde{\boldsymbol{g}}$ as follows:

$$\tilde{\boldsymbol{g}} := \left\{ \tilde{\boldsymbol{g}}^{\delta^i} \right\}_{i=0}^{n-1} \in \mathbb{G}^{nd}$$

2) The prover $\mathcal{P}$ chooses random $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma} \in \mathbb{Z}_q^{nd\times d_1 \times d_2}$, and sends the following elements to the verifier $\mathcal{V}$

$$X = \boldsymbol{g}^{\boldsymbol{\alpha}} \cdot \boldsymbol{h}^{\boldsymbol{\beta}}, \qquad \tilde{X} = \tilde{\boldsymbol{g}}^{\boldsymbol{\alpha}} \cdot \tilde{\boldsymbol{h}}^{\boldsymbol{\gamma}}$$

3) The verifier samples a challenge $e \in \mathbb{Z}_q$ and sends it to the prover $\mathcal{P}$.
4) The prover $\mathcal{P}$ computes

$$\boldsymbol{z} = \boldsymbol{\alpha} + e \cdot \boldsymbol{m}, \quad \boldsymbol{k} = \boldsymbol{\beta} + e \cdot \boldsymbol{o}, \quad \boldsymbol{\omega} = \boldsymbol{\gamma} + e \cdot \sum_{i=1}^{n} \boldsymbol{o}_i \cdot \delta^{i-1}$$

5) Let

$$\boldsymbol{g} = \boldsymbol{g}_L \| \boldsymbol{g}_R, \qquad \tilde{\boldsymbol{g}} = \tilde{\boldsymbol{g}}_L \| \tilde{\boldsymbol{g}}_R, \qquad \boldsymbol{z} = \boldsymbol{z}_L \| \boldsymbol{z}_R$$

and

$$Y = X \cdot C^e \cdot \boldsymbol{h}^{-\boldsymbol{k}}, \qquad \tilde{Y} = \tilde{X} \cdot (\prod_{i=1}^{n} D_i^{\delta^{i-1}})^e \cdot \tilde{\boldsymbol{h}}^{-\boldsymbol{\omega}}$$

6) The prover $\mathcal{P}$ sends

$$L = \boldsymbol{g}_R^{\boldsymbol{z}_L}, \qquad R = \boldsymbol{g}_L^{\boldsymbol{z}_R}$$
$$\hat{L} = \tilde{\boldsymbol{g}}_R^{\boldsymbol{z}_L}, \qquad \tilde{R} = \tilde{\boldsymbol{g}}_L^{\boldsymbol{z}_R}$$

7) The verifier $\mathcal{V}$ sends a challenge $c \in \mathbb{Z}_q$
8) The prover $\mathcal{P}$ computes

$$\boldsymbol{z}' = \boldsymbol{z}_L + c \cdot \boldsymbol{z}_R$$

and both parties compute

$$Y' = L \cdot Y^c \cdot R^{c^2}, \qquad \tilde{Y}' = \tilde{L} \cdot \tilde{Y}^c \cdot \tilde{R}^{c^2}$$
$$\boldsymbol{g}' = \boldsymbol{g}_L^c \odot \boldsymbol{g}_R, \qquad \tilde{\boldsymbol{g}}' = \tilde{\boldsymbol{g}}_L^c \odot \tilde{\boldsymbol{g}}_R$$

where $\odot$ is element-wise product.
9) If $n > 2$, then both parties execute the above step (5)-(8) with

$$((\boldsymbol{g}, \tilde{\boldsymbol{g}}', n/2), (Y', \tilde{Y}'), \boldsymbol{z}')$$

Otherwise, the verifier $\mathcal{V}$ checks

$$\boldsymbol{g}'^{\boldsymbol{z}'} \stackrel{?}{=} Y', \qquad \tilde{\boldsymbol{g}}^{\boldsymbol{z}'} \stackrel{?}{=} \tilde{Y}'$$

Fig. 10: Compressed Σ version for the relation $\mathcal{R}_{\mathsf{Eq}}^{\mathsf{Batch}}$

Theorem 3. The protocol described in Fig.10 is a (2κ + 4) protocol for the relation $\mathcal{R}_{\mathsf{Eq}}^{\mathsf{Batch}}$ where $\kappa = \lceil \log nd \rceil$ - 1. It satisfies completeness, computationally (n, 2, $\{t_i\}_{i\in[\kappa]}$)-special sound if finding discrete-logarithm, and special honest verifier zero-knowledge where $t_i = 3$ for all $i \in [\kappa]$.

Proof. Since completeness is straightforward, we omit the description.

$(n, 2, \{t_i\}_{i\in[\kappa]})$**-special soundness**. To simplify, we assume a single recursive step execution. Specifically, we analyze

the 4-move protocol, where the prover sends the response $\boldsymbol{z}'$ irrespective of it dimension, and proves that this protocol is 4-special sound. Then t_i-special soundness can then be derived through an inductive argument, the details of which are omitted here (i.e. omit j).

First of all, we denote the transcript as Tr, which consists of $(L, R, \tilde{L}, \tilde{R}, Y, \tilde{Y}', c_i, \boldsymbol{z}_i)$. Given three accepting transcripts $(\mathsf{Tr}_0, \mathsf{Tr}_1,\ \mathsf{Tr}_2)$ for the same challenge δ but the distinct challenge $c_i \in \{0,1,2\}$, we can show that there exists an efficient algorithm χ that outputs a valid witness. Given these transcripts, Since $\prod_{0\le i<k\le 2}(c_k - c_i) \neq 0$, we define (v_0, v_1, v_2) such that

$$\sum_{i=0}^{2} v_i = 0, \quad \sum_{i=0}^{2} v_i \cdot c_i = 1, \quad \sum_{i=0}^{2} v_i \cdot c_i^2 = 0$$

Define $\bar{\boldsymbol{z}}_i = (v_i c_i \boldsymbol{z}_i \| v_i \boldsymbol{z}_i)$. Then let $\boldsymbol{w} = \sum_{i=0}^{2} \bar{\boldsymbol{z}}_i$ be the extracted value. We show the correctness of extracted value as follows

$$\begin{aligned}
\boldsymbol{g}^{\boldsymbol{w}} &= \boldsymbol{g}^{(\sum_{i=0}^{2} v_i c_i \boldsymbol{z}_i) \| (\sum_{i=0}^{2} v_i \boldsymbol{z}_i)} \\
&= \boldsymbol{g}_L^{v_0 c_0 \boldsymbol{z}_0} \cdot \boldsymbol{g}_L^{v_1 c_1 \boldsymbol{z}_1} \cdot \boldsymbol{g}_L^{v_2 c_2 \boldsymbol{z}_2} \cdot \boldsymbol{g}_R^{v_0 \boldsymbol{z}_0} \cdot \boldsymbol{g}_R^{v_1 \boldsymbol{z}_1} \cdot \boldsymbol{g}_R^{v_2 \boldsymbol{z}_2} \\
&= \prod_{i=0}^{2} \left((\boldsymbol{g}_L^{c_i} \odot \boldsymbol{g}_R)^{\bar{\boldsymbol{z}}_{i,L} + c_i \bar{\boldsymbol{z}}_{i,R}} \right)^{v_i} \\
&= \prod_{i=0}^{2} (\boldsymbol{g}_L^{c_i \bar{\boldsymbol{z}}_{i,L}} \cdot \boldsymbol{g}_L^{c_i^2 \bar{\boldsymbol{z}}_{i,R}} \cdot \boldsymbol{g}_R^{\bar{\boldsymbol{z}}_{i,L}} \cdot \boldsymbol{g}_R^{c_i \bar{\boldsymbol{z}}_{i,R}})^{v_i} \\
&= \prod_{i=0}^{2} \left((\boldsymbol{g}^{\bar{\boldsymbol{z}}_i})^{c_i} \cdot \boldsymbol{g}_R^{\bar{\boldsymbol{z}}_{i,L}} \cdot (\boldsymbol{g}_L^{\bar{\boldsymbol{z}}_{i,R}})^{c_i^2} \right)^{v_i} \\
&= \prod_{i=0}^{2} (Y^{c_i} \cdot L \cdot R^{c_i^2})^{v_i} \\
&= Y
\end{aligned}$$

where $\odot$ denotes the element-wise product. In a similar vein, extraction can also be performed for $\tilde{\boldsymbol{g}}$.

Special honest verifier zero-knowledge. With the challenge x and e provided, the simulator randomly samples $\boldsymbol{z}$, $\boldsymbol{k}$, and $\boldsymbol{\omega}$, subsequently using these to perfectly simulate the remaining messages as follows:

$$X := \boldsymbol{g}^{\boldsymbol{z}} \cdot \boldsymbol{h}^{\boldsymbol{k}} \cdot C^{-e}, \quad \tilde{X} := \tilde{\boldsymbol{g}}^{\boldsymbol{z}} \cdot \tilde{\boldsymbol{h}}^{\boldsymbol{k}} \cdot (\prod_{i=1}^{l} D_i^{x^{i-1}})^{-e}$$