VERI-ZEXE: Decentralized Private Computation with Universal Setup

Alex Luoyuan Xiong, Binyi Chen, Zhenfei Zhang, Benedikt Bünz, Ben Fisch, Fernando Krell, and Philippe Camacho

Espresso Systems
{alex,binyi,zhenfei,benedikt,ben,fernando,philippe}@espressosys.com

Abstract. Traditional blockchain systems execute program state transitions *on-chain*, requiring each network node participating in state-machine replication to re-compute every step of the program when validating transactions. This limits both scalability and privacy. Recently, Bowe *et al.* introduced a primitive called *decentralized private computation* (DPC) and provided an instantiation called ZEXE, which allows users to execute arbitrary computations *off-chain* without revealing the program logic to the network. Moreover, transaction validation takes only constant time, independent of the off-chain computation. However, ZEXE required a separate trusted setup for each application, which is highly impractical. Prior attempts to remove this per-application setup incurred significant performance loss.
We propose a new DPC instantiation VERIZEXE that is highly efficient and requires only a single universal setup to support an arbitrary number of applications. Our benchmark improves the state-of-the-art by 9x in transaction generation time and by 2.6x in memory usage. Along the way, we also design efficient gadgets for variable-base multi-scalar multiplication and modular arithmetic within the PLONK constraint system, leading to a Plonk verifier gadget using only $\sim 21k$ PLONK constraints.

Table of Contents

1 Introduction

Distributed ledgers are replicated state machines maintained by a network of potentially faulty nodes via a distributed consensus algorithm. The state machine might range from a specialized accounting system, as in Bitcoin [Nak08], to a Turing complete virtual machine, as in Ethereum [W$^+$14], where any user can instantiate a stateful program called a *smart contract*. These platforms are resilient to failures or even malicious behavior by a subset of the network nodes. This resilience enables a new class of applications in cryptocurrencies, governance, digital collectibles, and more. Unfortunately, privacy, which is paramount for many applications, is disregarded in ledger systems like Bitcoin and Ethereum.

There is a rich literature of work attempting to improve privacy guarantees on distributed ledgers [BSCG$^+$14,NVV18,KMS$^+$16,CZJ$^+$17,BAZB20,CZK$^+$19].The Zerocash protocol [BSCG$^+$14] is a privacy-preserving payment system that achieves user anonymity and amount confidentiality in transactions. Hawk [KMS$^+$16] proposes a smart-contract framework that preserves program data privacy. Zether [BAZB20] enables confidential transactions among publicly known smart contracts, and hides the identities of transacting parties within a small anonymity set. All of these prior designs, however, are either limited to a fixed functionality (e.g., payments), or lack function-privacy, i.e. transactions do not hide which smart-contract is being executed. Zexe [BCG$^+$20] addresses this by proposing a new cryptographic primitive called *decentralized private computation* (DPC) scheme that achieves both data privacy and function privacy for arbitrary user-defined programs. The scheme hides from the network nodes both the states and the logic of the programs being called in each transaction. Users in DPC schemes execute computations and update the ledger by sending a transaction with a publicly verifiable cryptographic proof attached, attesting to the correctness of the computation.

One of the core building blocks in a DPC construction is a SNARK system (see Appx. 2.1 for more background). A SNARK system for a binary relation $\mathcal{R}$ provides a prover algorithm $\mathcal{P}(x, w)$ that on any valid input $(x, w) \in \mathcal{R}$ outputs a valid proof π, and a verifier algorithm $\mathcal{V}(x, \pi)$ that always accepts valid proofs and rejects invalid proofs with overwhelming probability. A zk-SNARK proof leaks no information about the *witness* w. We will generally refer to $\mathcal{R}$ as a "circuit", which outputs 1 on input (x, w) if and only if $(x, w) \in \mathcal{R}$, however alternative representations, i.e. *constraint systems*, may be utilized. The SNARK system may also require a trusted setup procedure to generate a structured reference string (SRS), which is an input to both $\mathcal{P}$ and $\mathcal{V}$. A SNARK system is *universal* if it has a single setup to generate a single SRS that can be reused for all circuits, and *non-universal* if it requires a new setup per circuit. The original ZEXE [BCG$^+$20] system uses a non-universal scheme [Gro16,GM17], thus requiring a trusted setup for every application. As this is extraordinarily inconvenient in practice, the authors also suggest an alternative instantiation from universal SNARKs [CHM$^+$20], which requires a one-time setup to support all future applications up to a maximum circuit complexity. However, the performance of this alternative instantiation[1] is significantly worse than the original protocol due to the higher complexity of the universal SNARK verification logic and the fact that ZEXE requires producing a SNARK proof for a circuit that encodes the SNARK verification logic. Specifically, its transaction generation speed is an order-of-magnitude slower than in the original ZEXE. Hence we ask the following question:

Problem 1. Can we obtain DPC with universal setup without sacrificing transaction generation speed?

[1] https://github.com/AleoHQ/snarkVM/tree/testnet1

Implementation	**Universal Setup**	Execute^L	**Memory**	Verify^L	**Proof Size**
Original ZEXE [BCG$^+$20]	✗	14.3 s	6.56 GB	15 ms	0.482 KB
SnarkVM testnet-2	✓	151.4 s	16.59 GB	15 ms	0.482 KB
VERIZEXE (this work)	✓	16.9 s	6.49 GB	18 ms	4.138 KB

Table 1: Comparison of three DPC implementations for 2-input-2-output transaction. Details in § 4.

1.1 Our Contributions

We answer the above question in the affirmative. The contributions of this paper are:

- VERIZEXE, a DPC scheme instantiation that supports both one-time universal system setup and efficient transaction generation comparable to ZEXE (see Table 1).
- Constraint designs for efficient variable-base MSM and modular arithmetics, leading to a Plonk verifier gadget taking only $\sim 21k$ Plonk constraints which are of independent interests.
- Implementation (open-sourced[2], written in Rust) and evaluation of VERIZEXE showing its practicality and most notably its 9x improvement on transaction generation time and 2.6x improvement on memory usage over the prior state-of-the-art.

1.2 Our Techniques

In ZEXE, a user creates a transaction that nullifies some existing records and mints some new records, representing a state transition for the ledger. Each record has an associated *birth* and *death* predicate. The state transition is valid only if all death predicates of nullified records and birth predicates of minted records are satisfied. The user submitting the transaction creates a SNARK proof that these predicates are satisfied, called the *inner proof.* To achieve *function privacy*, the user additionally creates an *outer proof*, which is a zero-knowledge SNARK proving the existence of a correct inner proof. Only the outer proof is included in the transaction and hides the predicates involved.

ZEXE [BCG$^+$20] instantiates both the outer proof and the inner predicate proofs with SNARK schemes that require predicate-specific trusted setups. Instead, we propose the use of SNARKs with universal setup parameters that can be reused for all predicates. Normally, universal SNARKs are built on information-theoretic Polynomial Interactive Oracle Proofs (PIOP)[BFS20] which are then cryptographically compiled using Polynomial Commitment Schemes and Fiat-Shamir Transform to turn the transcripts of IOPs into succinct NARKs. To make VERIZEXE efficient we need to overcome several obstacles when encoding the verifier logic of a universal SNARK inside a circuit:

- **Pairing checks:** SNARKs that utilize pairing-based PCS [KZG10,CHM$^+$20] require some pairing operations in their verification logic, which is very expensive and requires a large number of constraints in circuit. Note that this is not a unique problem for universal SNARKs, as many non-universal counterparts [Gro16,GM17] also need pairing checks.
- **Multi-Scalar Multiplications:** There are more variable-based MSM operations in the verification steps of universal SNARKs than their non-universal counterparts, which results in high circuit complexity with naïve implementation.

[2] https://github.com/EspressoSystems/veri-zexe

- **Polynomial evaluations over non-native field:** The predicate (inner) proofs and final outer proof are generated in different circuits over different finite fields, thus polynomial evaluations over the inner fields will be simulated in an outer circuit with a different field, which involves high overheads.
- **Fiat-Shamir transform:** Unrolling all the challenges generated by FS transform requires applying a hash function for many times. However, commonly used hash functions are not SNARK friendly and results in high circuit cost.

We now give an overview of our techniques that drastically reduce the outer circuit complexity whose proof generation dominates the cost of DPC transaction generations. The set of techniques can be split into two categories where we start with generic ones that are agnostic to the choice of constraint systems. Later we present the other techniques that particularly fit into Plonk-based constraint systems.

Lightweight Verifier Circuit from Accumulation Scheme. Inspired by Halo [BGH19], we move out the expensive pairing check from the SNARK verifier circuit and delay the final proof verification step to ledger validators. Intuitively, the verification logic of universal SNARKs with pairing-based PCS culminates in producing $2\,\mathbb{G}_1$ points for the final bilinear pairing check. Instead of carrying out the full proof verification in circuit, we output the $2\,\mathbb{G}_1$ points as public inputs and attach to the transaction validity proof. To ensure these two points reveal no information about the underlying predicates, we further mask them by simultaneously applying a blinding factor on both points so that the masked points preserve the pairing check result. The actual pairing check will be executed by ledger maintainer who receive the transaction validity proof and the two masked points. This technique has been formalized and generalized as a new primitive called *accumulation scheme*[3] by Bünz *et al.* [BCMS20] whose results are originally targeting incrementally verifiable computation (IVC). We explain in §3.1 how to cast the DPC transaction creation task as a simplified two-step IVC, and naturally rely on theorems from the paper for correctness and security.

Instance Merging. As briefly explained, the outer circuit needs to verify $m + n$ universal SNARK proofs for m death predicates and n birth predicates in an m-input-n-output transaction (W.L.O.G. we assume $m = n$). We halve the number of proofs the outer circuit needs to verify (from $2m$ to m) by merging each pair of death predicate and birth predicate into a single larger predicate. The critical precondition for this technique to have positive net saving is that: verifying one proof for a merged statement twice as large requires significantly fewer constraints than verifying two proofs for two statements; which holds true for SNARKs such as Plonk [GWC19].

Assume that the original circuit size bound for birth/death predicates is N, the merging technique simply left/right pad another N dummy gates to birth/death predicate circuits respectively before arithmetizing them into polynomials that constitutes the proving keys. The key observation is that with additive homomorphic polynomial commitment schemes (such as [KZG10]), the commitment to the addition of two polynomials is simply the addition of their polynomial commitments, namely $\mathsf{Commit}(p_1+p_2) = \mathsf{Commit}(p_1)+\mathsf{Commit}(p_2)$ where $p_1, p_2 \in \mathbb{F}[X]$. Therefore, by adding a pair of verification keys of any padded birth predicate and any padded death predicate, the verifier can obtain the verification key of the merged predicate and thus be able to verify the proof for the merged predicate.

Notice that theoretically one can merge more than two predicates, but in the DPC context, merging a pair of birth and death predicates hits the sweet spot of flexibility

[3] not to be confused with cryptographic accumulator concept for set membership.

and efficiency improvement. This is because circuit/predicate key preprocessing happens beforehand in the offline phase, and instance merging-then-proving happens later in the online phase. For example, if we merge 3 predicates, then during circuit key generation phase, we will have to decide which N-out-of-$3N$ slots (in the merged circuit) should a particular predicate be assigned to, which restrict it from merging with other predicates that occupy the same N slots. In contrast, our merging of a death and a birth predicate requires easy slot allocation and allows for arbitrary assembly of death/birth predicates in a transaction.

Proof Batching. Instead of generating and verifying m proofs separately, we exploit proof batching technique to achieve a lower amortized cost. We leverage the fact that universal SNARKs are cryptographically compiled from a PIOP using a PCS and many choices of PCS support batch opening which reduces opening proofs size and amortizes verification cost. Thus, we present a generic compiler in § 3.3 to transform a PIOP-based SNARK into a batched prover and verifier for a list of NP relations. In the case of KZG-Plonk, batching ℓ relations reduces the total proof size by $5(\ell - 1)\,\mathbb{G}_1$ elements by sharing the same quotient polynomial and the same opening polynomials; reduces the number of MSM operation by $7(\ell - 1)$ and the number of pairing operation by $(\ell - 1)$. Batching TurboPlonk with more selector polynomials and wire polynomials leads to even greater savings. Since both MSM gadget and pairing gadget are expensive, we would noticeably reduce outer circuit complexity by batch verifying m merged predicate.

Next we present techniques that are tailored for (customized) Plonk-based constraint systems that support lookup gates.

Variable-base MSM via Online Lookup Table. Instead of naïvely enforce variable-base MSM computation, we design a Pippenger-base MSM gadget and further reduce its complexity by relying on a special variant of lookup argument called *online lookup table argument*. Recall that Pippenger algorithm [Pip80] reduces a b-bit MSM into b/c instances of c-bit MSMs ($c < b$) and finally sums them together. When computing a c-bit MSM, instead of unrolling the exact Pippenger algorithm in the circuit which is very expensive, we utilize a lookup table containing all resulting points from the scalar multiplications between the base point and all $2^c - 1$ possible scalar values. With such lookup table, any c-bit scalar multiplication on this specific base point becomes a table query rather than elliptic curve group operation. Given that these bases are unfixed, the lookup table cannot be pre-processed – table values are only known during online proving phase. Such online lookup tables are already possible with [GW20] although whose presentation is limited to preprocessed query table whose values are known ahead of time. We provide detailed gadget descriptions and circuit size breakdown in § 3.4.

Polynomial Evaluation over Non-native Field. To facilitate polynomial evaluation over non-native field, we device efficient modular multiplication and modular addition gadgets by leveraging range check via lookup argument [GW20]. Compared to other modular arithmetic gadget designs, ours take advantage of (a) clearer UltraPlonk constraint system design to do range-check with little to no additional circuit cost; (b) specialized use case of two-chain curves for depth-2 proof recursion (instead of cycling curves for deeper proof recursions), which allows us to safely assume finer-grained requirements on the sizes of two fields to make our circuit simpler. We provide detailed gadget descriptions in § 3.5.

SNARK-friendly Symmetric Primitives. To reduce the number of non-algebraic operations in the circuit, we instantiate symmetric primitives used such as commitment schemes, PRF, and CRH with SNARK-friendly candidates which are designed to work natively with finite field involving mostly algebraic operations. We specify our concrete implementations in § 4.1, most of which are based on Rescue hash functions [AABS+20]. More importantly, we carefully design customized gates in our TURBO-PLONK (Def. 2) to optimize for these rescue operations. Our Fiat-Shamir transcript uses Sponge-based hash from Rescue permutation so that verifier challenge derivation is much cheaper in circuit. We further designed an optimized predicate commitment gadget in § 3.6 to ensure two circuits over different fields are committing to the same list of predicates. Particularly, the number of non-native hash operations in our gadget does not grow with the number of predicates committed.

1.3 Related Works

We refer readers to Section 1.2 in [BCG+20] for a comprehensive literature review on privacy-preserving computation on ledgers before ZEXE.

Private Smart Contracts. Since the exciting work of ZEXE [BCG+20], there has been more works on privacy-preserving smart contracts. ZKay [SBG+19] observes the difficulty of expressing programs in low-level circuits correctly, and designs a high level language to annotate private data with explicit ownership. Zkay provides a compiler transforming zkay contracts into Solidity contracts on Ethereum that leverages encryption for privacy and NIZK proofs for correctness. Unfortunately, transactions in Zkay cannot operate on "foreign values" (values owned by parties other than the caller), a limitation addressed by ZeeStar [SBBV22] which uses additive homomorphic encryption to allow simple addition on foreign values. Zkay and ZeeStar lower the barrier for non-cryptographer to write contracts in higher level language but have many restrictions and limited expressiveness.

Meanwhile zkHawk [BCT21] extends Hawk [KMS+16] by replacing the minimal trusted manager who will learn the private inputs from users with an MPC protocol. To avoid running a SNARK prover in MPC which is prohibitively expensive, they simplify the Hawk framework by assuming a "freeze-compute-finalize" three-phase process for program execution. To enforce correct payout of the original deposits from "freeze" phase during "finalize" phase or contract closure, zkHawk uses sigma protocols and homomorphic commitments, similar to techniques used in confidential transaction. While this model is perfect for applications like sealed bid auction, it is arguably restricted since many applications run forever without a clear closure yet require frequent intermediate on-chain state commitments.

SmartFHE [SA21] is the first to use fully-homomorphic encryption in the blockchain model and allow multi-user computation on-chain with hidden function inputs and outputs. Additionally, to mitigate concurrency issue, SmartFHE introduces account locking to freeze account balance or states from unintended update when some private transactions are still in the mempool. Inevitably, the biggest obstacle is still the staggering cost of FHE for arbitrary computation.

Kachina [KKK21] provides a unified universal composable (UC) model on private smart contracts which claims to be an overarching framework to capture Zexe, Hawk, Zether and more while preserving their original privacy guarantees. One of the novelties of Kachina is introducing the concept of *state oracle transcript* and model read/write of private and public states as query/response from the local and public oracles. Transactions are further allowed to declare inter-dependencies, which together with the public oracle transcripts are supposed to help with the concurrency issue. However, it remains to be demonstrated how to achieve flexible composability of contracts and complex interactions among contracts in Kachina. Particularly, [KKK21] did not offer concrete construction that improve ZEXE.

In short, ZEXE remains the only concrete private smart contract construction to date that offers both data privacy and function privacy with rich expressiveness.

Universal SNARKs. Zero-knowledge proof [GMR85] allows a prover to convince a verifier an NP statement without revealing any extra information. Subsequent works lead to *non-interactive* proofs [BFM88] in the common reference string model and arguments with sublinear communication [Kil92,Mic00] where malicious provers are computationally bounded. In the recent decade, a long line of work [Gro10,BCCT12,BCI+12,GGPR13,Gro16,GM17] has focused on *succinct non-interactive argument of knowledge* (SNARK) with succinct proofs, sometimes of constant size, and fast verification. These SNARKs usually rely on some heavy offline pre-processing to generate a circuit-specific *structured reference string* (SRS) to facilitate faster online verification. Even though some constructions like Groth16 is highly efficient and widely deployed, sampling of the SRS would require a trusted setup for each circuit, instantiated using a secure multi-party ceremony [BGG18] that takes months in practice, which is highly unsustainable. One way around is using argument system with transparent setup depending on only uniformly random reference string without any toxic trapdoor; however, they usually results in larger proof size [BBHR18] or non-succinct (linear) verification cost [BBB+18]. Another alternative is using an universal and updatable model [GKM+18] where a circuit-independent SRS is generated when system boots up, and any party can update the SRS in a verifiable way; the trapdoor is unknown to all parties as long as at least one contributor is honest. Sonic [MBKM19] presents the first efficient universal SNARK construction, followed up by Marlin [CHM+20], Plonk [GWC19], and Lunar [CFF+21] to further improve the efficiency of the proof system.

Universal SNARKs strike a good balance between efficiency and acceptable trust assumption. We choose variants of Plonk for our implementations primarily due to its excellent performance, customizable gates and importantly its support for lookup argument [BCG+18,GW20] that some of our optimization techniques depend on.

2 Preliminaries

We denote $[n]$ as the set $\{1, \dots, n\} \subseteq \mathbb{N}$, $\lambda \in \mathbb{N}$ as the security parameter, $\mathtt{negl}(\lambda)$ as a *negligible* function in λ if it vanishes faster than the inverse of any polynomial in λ. A probability is overwhelming if it is $1 - \mathtt{negl}(\lambda)$ for a negligible function $\mathtt{negl}(\lambda)$. Further, we use *efficient algorithms* to refer to probabilistic polynomial time algorithms in λ.

We provide formal definitions and security properties of *Commitment Scheme*, *Polynomial Commitment Scheme* (PCS), *indexed relation*, *(preprocessing) non-interactive argument of knowledge* (NARK), and *incrementally verifiable computation* (IVC) in Appx. A.

2.1 Decentralized Private Computation (DPC)

Decentralized Private Computation (DPC) scheme is a cryptographic primitive introduced in [BCG+20] to capture a decentralized computational model where parties execute state transitions offline, and then update an online, append-only ledger with the updated new states via publicly verifiable transactions which serve as attestations to the correctness of the offline computation. As a generalization to prior work on Decentralized Anonymous Payments in Zcash protocol [BSCG+14], DPC allows arbitrary bounded computation (i.e. applications with bounded complexity) to be carried out in a privacy-preserving manner. Specifically, achieving both *data privacy* where application data are hidden from ledger maintainers, and *function privacy* where the target function/application being invoked in each transaction is also private, is the a major novelty of DPC. In addition to privacy, DPC schemes also attain succinctness where the verification of transactions is fast and independent of the cost of offline computation, thus achieving scalability compared to a standard decentralized ledger (a.k.a. blockchain) design where all computation are conducted online and re-executed by every ledger maintainer.

In a DPC scheme, the core data structures are *records*, *transactions*, and the *ledger*. Note that many notations and part of the descriptions below are verbatim from [BCG+20] with a few modifications (marked in cyan).

Records. A *record*, denoted by rec, is the generalization of an *unspent transaction output (UTXO)* and the core unit of data our scheme is centered around. Similar to UTXOs in Bitcoin [Nak08], records are consumed and created inside a transaction to reflect state updates in the system.

Concretely, a record is a tuple

$$\mathsf{rec} := (\mathsf{apk}, \mathsf{payload}, \Phi_b, \Phi_d, \rho; r_{\mathsf{cm}})$$

that consists of (a) a *address public key* apk, which specifies the record's owner; (b) a *data payload* $\mathsf{payload}$, which contains arbitrary application states;[4] (c) a *birth predicate* Φ_b that must be satisfied when rec is created; (d) a *death predicate* Φ_d that must be satisfied when rec is consumed; (e) a *record nonce* ρ which is a deterministically computed, unique nonce; (f) a *record randomizer* r_{cm} used to hide record attributes in its commitment cm.

Similar to Zcash, records are never published on the ledger in plaintext. Only commitments of these records are publicly stored. A record is created when its hiding *record commitment* cm is declared as one of the outputs of a transaction (details later); and it can be consumed or destroyed when its respective *nullifier* nf[5] is included as one of the inputs of a transaction.

Concretely, a *record commitment* is defined as:

$$\mathsf{cm} = \mathsf{COM.Commit}(\mathsf{pp}_{\mathsf{COM}}, \mathsf{apk} \| \mathsf{payload} \| \mathsf{pid}_b \| \mathsf{pid}_d \| \rho; r_{\mathsf{cm}})$$

where birth predicate identifier $\mathsf{pid}_b = \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, \mathsf{vk}_b)$ is the hash of the verifying key of Φ_b (similarly for pid_d);[6] record nonce $\rho = \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, j \| \mathsf{nf}_0^{in})$[7] is (w.h.p.) a unique nonce deterministically computed inside the transaction that created it – since the first input nullifier nf_0^{in} is unique across the entire ledger (see justification below) and the record's position j among all output records is unique across this transaction. We note that record commitment cm does not reveal anything about the actual record openings.

[4] $\mathsf{payload}$ contains a designated subfield $\mathsf{isDummy}$ indicating whether rec is dummy or not.

[5] Nullifier is equivalent to the *serial number* in ZEXE paper, but the former is more descriptive and becoming more common in the literature.

[6] Whereas in ZEXE paper, they commit to Φ_b, Φ_d directly, whose representation can be huge. Admittedly in practice, one might employ the same techniques, basically we just choose to make them explicit from the syntax.

[7] Whereas in ZEXE paper, the record nonce is hashing over all input nullifiers of the transaction that generates it $\rho = \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, j \| \mathsf{nf}_0^{in}, \dots, \mathsf{nf}_m^{in})$, which we think it's unnecessary and more expensive to enforce in circuit.

To consume rec, one needs to prove integrity of its record commitment and its nullifier. The former requires proof of knowledge of the entire record openings used to generate cm; the latter requires proof of knowledge of the address private key ask corresponding to the owner address $\mathsf{rec.apk}$.

Intuitively, the ledger prevents "double-spending" by disallowing a same nullifier to appear twice in the ledger. By construction, each record has a deterministically computable, unique nullifier. Additionally, non-owners cannot generate a valid nf to consume a record without the knowledge of owner's ask, even if they know the entire record openings. Finally, it is computationally infeasible to come up with two distinct records with a same cm and different nfs.

Transactions. A transaction, denoted by tx, represents a state change to the system by consumption of input records and creation of output records. A transaction usually contains a list of record nullifiers as inputs and record commitments as outputs – while former are considered to be consumed or died, thus whose death predicate Φ_d must be satisfied, and latter are created or born, thus whose birth predicate Φ_b must be satisfied. Dummy records, with $\mathsf{isDummy} = 1$ and mostly used as inputs to create non-dummy records, can be created freely, but consuming them still requires satisfaction of their death predicates.

Concretely it is a tuple of $\mathsf{tx} := ([\mathsf{nf}]_1^m, [\mathsf{cm}]_1^n, \mathsf{memo}, \mathsf{cm}_\Phi, \mathsf{cm}_{\mathsf{ldata}}, \pi_e, \mathsf{st}_{\mathbf{L}})$[8] for an m-input-n-output transaction, where memo is some memorandum or arbitrary data one can attach to the transaction; cm_Φ is a hiding commitment to all predicates checked inside this transaction;[9] $\mathsf{cm}_{\mathsf{ldata}}$ is a hiding commitment to all *local data* ldata (defined later in this section) used; π_e is a cryptographic proof attesting to the valid execution of the state transition (namely destruction and creation of records); $\mathsf{st}_{\mathbf{L}}$ is the historical ledger digest the validity of tx is proved against. The transaction only reveals nullifiers of old records, commitments of new records and the fact that death predicates of all old records and birth predicates of all new records are satisfied. Note that the input/output size (m, n) is also public but the number of dummy records are completely hidden.

Informally, a valid transaction satisfies a list of predicates including: Φ_d of all its input records, Φ_b of all its output records, and an extended UTXO predicate $\Phi^{\mathbf{L}}_{\mathsf{utxo}}$, and it contains a $\mathsf{st}_{\mathbf{L}}$ that indeed exists in the canonical history of the ledger state digest. The *extended UTXO predicate* $\Phi^{\mathbf{L}}_{\mathsf{utxo}}$ ensures the following for a transaction: (a) all input records are well-formed whose respective cm were accumulated before, whose nullifiers have not shown in the ledger before; (b) all output records are well-formed with correctly computed cm.

During predicate checking, predicates in a transaction are given access to a common input called *local data*, denoted by ldata. Local data includes (a) every record's contents; (b) shared, publicly revealed field: *transaction memorandum* memo; (c) shared, hidden secret fields: *auxiliary input* aux; and more:

$$\mathsf{ldata} := ([\mathsf{rec}_i]_1^m, [\mathsf{nf}_i]_1^m, [\mathsf{rec}_j]_1^n, \mathsf{memo}, \mathsf{aux})$$

Ledger. A publicly-accessible, append-only ledger, denoted by $\mathbf{L}$, stores all published valid transactions and keep track of output commitments through a Merkle tree

[8] Usage of cm_Φ and $\mathsf{cm}_{\mathsf{ldata}}$ is not included in original syntax but is implicit after applying solutions mentioned in Section 7 of [BCG+20].

[9] We use cm_Φ to reduce circuit complexity of the outer (second layer) proof statement. Roughly, we need to prove correctness of a list of inner proofs for birth/death predicates correspond to the matching predicates of records involved instead of some arbitrary predicates. We enforce that through checking the commitment over all relevant pid, namely cm_Φ. Note that, even if we can use $\mathsf{cm}_{\mathsf{ldata}}$ to ensure the same condition since ldata contains all relevant pid, it is much cheaper to check correct commitment of cm_Φ with smaller input message size.

accumulator. We describe the ledger's basic functionalities as follows (notations are mostly verbatim to that in Zexe):

- $\mathbf{L}.\mathsf{Len}$: returns the number of transactions currently on the ledger.
- $\mathbf{L}.\mathsf{Append}(\mathsf{tx})$: append a verified, valid transaction tx to the ledger.
- $\mathbf{L}.\mathsf{Digest} \to \mathsf{st}_{\mathbf{L}}$: returns a short digest of the current state of the ledger.
- $\mathbf{L}.\mathsf{ValidateDigest}(\mathsf{st}_{\mathbf{L}}) \to b$: checks if $\mathsf{st}_{\mathbf{L}}$ is a valid historic ledger state.
- $\mathbf{L}.\mathsf{Contains}(\mathsf{nf}) \to b$: checks if the nullifier nf has been published on the ledger.
- $\mathbf{L}.\mathsf{Prove}(\mathsf{cm}) \to \pi_{\mathsf{mt},\mathbf{L}}$: if a record commitment cm appears on the ledger, then returns a proof of membership $\pi_{\mathsf{mt},\mathbf{L}}$; else returns $\perp$.
- $\mathbf{L}.\mathsf{Verify}(\mathsf{st}_{\mathbf{L}}, \mathsf{cm}, \pi_{\mathsf{mt},\mathbf{L}}) \to b$: check if the membership proof for cm is consistent with current ledger state $\mathsf{st}_{\mathbf{L}}$.

DPC Syntax: Algorithms.

A *Decentralized Private Computation* scheme $\mathsf{DPC} = (\mathsf{Setup}, \mathsf{GenAddress}, \mathsf{Execute}^{\mathsf{L}}, \mathsf{Verify}^{\mathsf{L}})$ is a tuple of efficient algorithms:

- $\mathsf{DPC}.\mathsf{Setup}(\lambda) \to \mathsf{pp}$: Given the security parameter λ, a trusted party setup the system once and output a public parameter pp for the system, and the trusted party is no longer needed.
- $\mathsf{DPC}.\mathsf{GenAddress}(\mathsf{pp}) \to (\mathsf{apk}, \mathsf{ask})$: Given the system wise public parameter pp, any user can create a new address key pair with address public key apk, and corresponding secret key ask.
- Any user can make consumed old records and create new ones (thus making state transitions) offline:

$$\mathsf{DPC}.\mathsf{Execute}^{\mathsf{L}} \begin{pmatrix} \text{public parameters} & \mathsf{pp} \\ \text{old records} & [\mathsf{rec}_i]_1^m \\ \text{old secret keys} & [\mathsf{ask}_i]_i^m \\ \text{new public keys} & [\mathsf{apk}_j]_1^n \\ \text{new record payloads} & [\mathsf{payload}_j]_1^n \\ \text{new birth predicates} & [\Phi_{b,j}]_1^n \\ \text{new death predicates} & [\Phi_{d,j}]_1^n \\ \text{aux. predicate input} & \mathsf{aux} \\ \text{transaction memo} & \mathsf{memo} \end{pmatrix} \to \begin{pmatrix} \text{new record } [\mathsf{rec}_j]_1^n \\ \text{transaction } \mathsf{tx} \end{pmatrix}$$

- $\mathsf{DPC}.\mathsf{Verify}^{\mathsf{L}}(\mathsf{pp}, \mathsf{tx}) \to b$: Given the public parameter pp and a transaction tx, an online ledger maintainer can verify transaction validity and output a bit b indicating acceptance or rejection.

  We refer definitions of all security properties to Section 3.3 in [BCG+20].

3 Verizexe: Practical Zexe with Universal SNARKs

To tackle the challenges of efficiently instantiating the DPC scheme with universal SNARKs described in Sec. 1.2, we propose numerous optimization techniques many of which can be applied to a wide range of protocols beyond DPC. With all optimizations applied, we expect to bring VERIZEXE to the realm of practicality. Detailed benchmark is reported in Sec. 4.

3.1 Lightweight Verifier Circuit from Accumulation Scheme

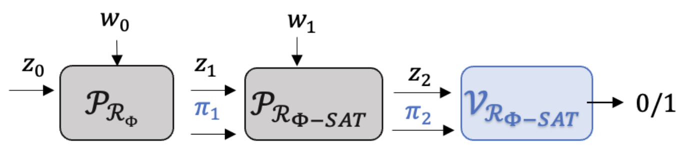
We apply a technique called *Accumulation Scheme* (AS), originally introduced in [BGH19] and later generalized in [BCMS20], to move the expensive pairing check out of the SNARK verifier circuit. While the technique is not new, we try to cast part of $\mathsf{DPC.Execute}^{\mathsf{L}}$ procedure into an *incrementally verifiable computation* (Appx. A.5) and show explicitly how accumulation schemes can improve the performance of ZEXE. The core observations are:[10]

1. Proving satisfiability of user-defined predicates during $\mathsf{DPC.Execute}^{\mathsf{L}}$ can be modeled as a two-step IVC.
2. Original ZEXE instantiated this IVC using SNARK composition.
3. We instead construct this IVC using a SNARK with accumulation scheme which has a more lightweight IVC prover. The key observation is that many known universal SNARKs have succinct verification apart from the polynomial opening check, which in turn can be handled by an accumulation scheme for the specific PCS used in the SNARK.

Modeling DPC executions as IVCs. In $\mathsf{DPC.Execute}^{\mathsf{L}}$, a user creates a transaction that nullifies some existing records and mints some new records, representing a state transition for the ledger. The validity of those state transitions (equivalently transition functions allowable) are governed by $\mathcal{R}_e$ which specifies constraints among old and new states that such transitions must satisfy. For efficiency reason, ZEXE has proposed to split $\mathcal{R}_e$ into two relations – one is the extended UTXO relation $\mathcal{R}_{\mathsf{utxo}}$ (Appx. 2.1), the other is a predicate satisfiability relation $\mathcal{R}_{\Phi}$ which is the target for our optimization here. As shown in Fig. 1, this process of proving $\mathcal{R}_{\Phi}$ can be modeled as a two-step IVC. In the first step, users produce SNARK proofs certifying all relevant predicates are satisfied over some local data ldata of that transaction. To achieve *function privacy*, SNARK proofs for predicates-SAT are not directly posted on the ledger. Instead, an outer proof π_{Φ} is generated in the second step attesting to the correctness of these predicate proofs, by taking predicate proofs and their verification keys as secret witness and run SNARK verifier inside the outer circuit. Finally, ledger maintainers run the IVC verifier to verify the outer proof which reveals nothing about the actual predicates involved (*i.e.* functions invoked) in the transaction. To ensure consistency of records used in $\mathcal{R}_{\mathsf{utxo}}$ and $\mathcal{R}_{\Phi}$, commitments to the local data $\mathsf{cm}_{\mathsf{ldata}}$ and list of predicates cm_{Φ} involved are returned as public outputs. Note that when applying proof batching technique (see § 3.3), the IVC proof from the first step will be a single batched proof denoted as $\pi_{\circledast}$ instead of a list of predicate proofs.

Remark 1. In Fig. 1, we only produce a single proof π_1 for proving satisfiability of $m+n$ user-defined predicates: $\forall i \in [m], j \in [n], \Phi_{d,i}(\mathsf{ldata}) = 1 \wedge \Phi_{b,j}(\mathsf{ldata}) = 1$. The reason

[10] Point (c) are Theorem 1 and 2 in [BCMS20] and concrete constructions for accumulation schemes for PCS are also provided in Section 8 of the same paper. We skipped elaborating on why these generic constructions are secure or property preserving and refer readers to the Bünz *et al.* [BCMS20] above for details. We provide a concrete instantiation of this technique on ZEXE with PLONK in Appx. D.


w_0
w_1
z_0
$\mathcal{P}_{\mathcal{R}_\Phi}$
z_1
π_1
$\mathcal{P}_{\mathcal{R}_{\Phi-SAT}}$
z_2
π_2
$\mathcal{V}_{\mathcal{R}_{\Phi-SAT}}$
0/1

$$z_0 := \begin{pmatrix}\text{local data comm.} & \mathsf{cm}_{\mathsf{ldata}}\end{pmatrix}, z_1 := z_0, z_2 := \begin{pmatrix}\text{pred. comm.} & \mathsf{cm}_\Phi \\ \text{local data comm.} & \mathsf{cm}_{\mathsf{ldata}}\end{pmatrix},$$

$$w_0 := \mathsf{ldata} = \begin{pmatrix}\text{input records} & [\boldsymbol{r}_i]_1^m \\ \text{input nullifiers} & [\mathsf{nf}_i]_1^m \\ \text{output records} & [\boldsymbol{r}_j]_1^n \\ \text{transaction memo} & \mathsf{memo} \\ \text{aux. pred. input} & \mathsf{aux}\end{pmatrix} w_1 := \begin{pmatrix}\text{input death pred. ver. key} & [\mathsf{ivk}_{d,i}]_1^m \\ \text{output birth pred. ver. key} & [\mathsf{ivk}_{b,i}]_1^n \\ \text{pred. comm. randomness} & r_\Phi\end{pmatrix}$$

$$\pi_1 := \pi_{\circledast}, \pi_2 := \pi_\Phi$$

Fig. 1: Casting $\mathcal{R}_\Phi$ proving into a two-step IVC (with different step function at each step).

why we can do this is through technique introduced in § 3.3 where we batch prove and verify a list of instances. We also borrow the notation $\pi_{\circledast}$ to indicate a batched proof. Obviously, the accumulation scheme is not dependent on such proof batching, one could cast the $\mathcal{R}_\Phi$ proving process into a *proof carrying data* (PCD) which is a generalization of IVC where an IVC can be viewed as PCD for a path in a directed acyclic graph (DAG). Fortunately, theorems and compilers in [BCMS20] are applicable to both IVC and PCD, thus so does the technique discussed in this section. Casting $\mathcal{R}_\Phi$ proving to an IVC makes cleaner presentation.

Remark 2. When casting $\mathcal{R}_\Phi$ into an IVC, we are not strictly using the standard definition, but rather a weaker notion of the IVC verifier where it will only validate the final state and proof instead of any intermediate outputs. We emphasize that this simplification won't affect any security property – IVC prover incrementally compute on correct previous state and proof will produce correct new state and proof (completeness); convincing final state and proof implies a knowledge extractor for all the witnesses (knowledge soundness). This modification makes the instantiation of the IVC verifier simpler since it no longer has to incorporate verification logic for every step functions. Concretely, in ZEXE, the IVC verifier corresponds to $\mathsf{DPC.Verify}^{\mathsf{L}}$ run by validators maintaining the ledger, who won't have to deal with inner predicate proof(s) π_1, but only final outer proof π_2. Thus there is no need to include verification logic for the first step function – making our verifier predicate independent.

Zexe: IVC from SNARK composition. Next, we explain how the original ZEXE instantiate this IVC using SNARK composition (see left half of Fig. 2). For a general IVC, at each step, the prover will receive the state z and an IVC proof π from the last computation step, compute the next state by applying the step function F to get the new state z', and create another IVC proof π' for the statement "$F(z) = z' \wedge V(z, \pi) = 1$" where V is a SNARK verifier. The fact that we have to embed the entire SNARK verifier logic inside the IVC prover's circuit is where the complexity comes from. For example, many PIOP-based universal SNARK are instantiated with pairing-based PCS like KZG that contains a pairing check, which incurs a high prover circuit complexity and slows down the proof generation significantly.

Verizexe: IVC from accumulation schemes. Finally, we give a high level intuition on leveraging an accumulation scheme for SNARK to defer the heavy-lifting during the SNARK verification to the IVC verifier, thus liberating the IVC prover from a complex

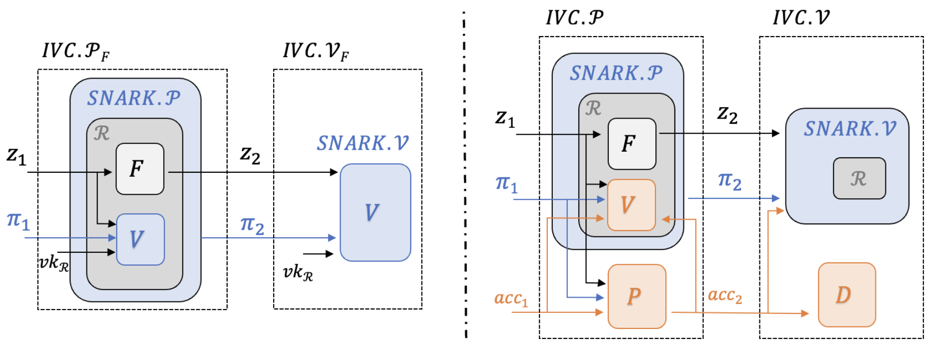


Fig. 2: IVC from SNARK compositions (left) v.s. IVC from accumulation schemes (right). Blue boxes are SNARK prover $\mathcal{P}$ and verifier $\mathcal{V}$ for the relation $\mathcal{R}$, and orange boxes are accumulation prover P, verifier V (more lightweight than a SNARK verifier) and decider D. z_1, z_2, π_1, π_2 in both schemes are the same as those in Fig. 1 where F is the step function that calculates the predicate commitment cm_Φ using Pedersen Commitment over the hashes of the predicate verifying keys. There are a few inputs dropped from the diagram for visual clarity, *e.g.* witness w_1 as an input to the IVC prover in both diagrams; SNARK verifying key $vk_\mathcal{R}$ for the SNARK verifier inside the IVC verifier on the right diagram.

circuit (see right half of Fig. 2).[11] At each step, the IVC prover receives an additional *accumulator* acc_i (think of the tuple (acc_i, π_i) as the new IVC proof), and eventually the accumulator will be validated by a *decider* algorithm as part of the IVC verifier logic. The core idea is: at the second step, the IVC prover will receive the predicate proof $\pi_\circledast$ and an empty accumulator $\mathsf{acc}_1 = \perp$; then instead of verifying the predicate proof entirely, we partially verify it (*e.g.* compute everything except the pairing check in case of PIOP+KZG SNARKs); the expensive steps in verification are delayed to the IVC verifier via the accumulator (*e.g.* acc_2 would contain the final two $\mathbb{G}_1$ elements used in KZG opening proof check). Informally, our accumulation prover will compute the group elements for PCS opening proof check, our accumulation verifier will ensure correct accumulations (*i.e.* correct derivation of the two $\mathbb{G}_1$ elements in KZG), our IVC prover only embeds the accumulation verifier's logic in its circuit which is much more lightweight than a SNARK verifier, and finally our IVC verifier ($\mathsf{DPC.Verify}^\mathsf{L}$ in ZEXE) will run a SNARK verifier for $\pi_2 := \pi_\Phi$ and a decider algorithm which completes the PCS opening proof check (*e.g.* the final pairing check in KZG).

Remark 3. We emphasize that the accumulation must be zero-knowledge – the accumulator acc_2 and accumulation proof π_V shouldn't reveal anything about the predicates being accumulated. In the context of an AS for PLONK with KZG, this means the two $\mathbb{G}_1$ elements for pairing must be masked/randomized and the randomizer is an additional secret witness for the accumulation verifier. Note that authors of [BCMS20] have already showed how to make AS for inner-product-argument-based and pairing-based PCSs zero knowledge in their Appendix A and Section 8.

3.2 Instance Merging

Recall that in $\mathsf{DPC.Execute}^\mathsf{L}$, a user needs to generate predicate (inner) proofs for all death predicates of input records and birth predicates of output records. We describe a method to merge two proving instances (*e.g.* a birth predicate and a death predicate) into one by exploiting the algebraic nature of preprocessing in a NARK (Appx. A.4) and the homomorphism of polynomial commitment schemes (Appx. A.2), thus halving the number of proofs the outer circuit needs to verify.

[11] In [BCMS20], the authors are motivated by achieving IVC using NARKs that don't have a succinct (polylog) verifier, such as [BBB$^+$18], whereas we are concerned about some heavy (circuit-unfriendly) computation of NARK verifiers even if they are succinct.

Technique. In a NARK based on polynomial IOP (such as *Algebraic Holographic Proof* (AHP) in MARLIN, and *idealized low-degree protocol* in PLONK), the preprocessing of circuit (i.e. $\mathsf{NARK}.\mathcal{I}^{\mathsf{srs}}$) involves an *arithmetization* process where constraints in an algebraic circuit (or equivalent computational models) are being transformed into constraints about polynomials. The resulting proving key ipk usually contains these index polynomials and the verifying key ivk contains the commitments to these index polynomials. During arithemtization, for a birth predicate circuit $\mathscr{C}_1$ of size n, we pad the circuit to size of $2n$, with $\mathscr{C}_1$ being right padded (last n gates are dummy), and compute the proving key and verification key as usual; for a death predicate circuit $\mathscr{C}_2$ of size n, we perform similar operations but left pad the circuit (first n gates are dummy). Subsequently, whenever we want to merge $\mathscr{C}_1$ and $\mathscr{C}_2$, we can construct a *merged* circuit of size $2n$ just by adding the two padded circuits while maintaining overall circuit satisfiability. The *merged* proving key can be easily obtained via addition of two polynomials of the same degree, and the *merged* verification key (i.e. the commitments) can be similarly derived thanks to additive homomorphism of PCS (such as KZG10 and its variants).

Syntax. We proceed to propose a slightly modified syntax for NARKs that supports instance merging. A *k-Mergeable NARK* scheme

$$\mathsf{NARK}_{\oplus}^{\mathsf{k}} = (\mathcal{G}, \mathcal{I}, \mathcal{M}_{\mathsf{ipk}}, \mathcal{M}_{\mathsf{ivk}}, \mathcal{M}_{\mathbb{w}}, \mathcal{P}, \mathcal{V})$$

supports merging k *slotted* instances into one single merged instance, where a slotted instance is labeled with a $\mathsf{slot} \in [k]$, and only a batch of non-overlapping instance $\{\mathsf{slot}_i\}$ where $\mathsf{slot}_i \neq \mathsf{slot}_j$ for any $i \neq j, i, j \in [k]$ can be merged together. For simplicity, we present the variant we will use to improve ZEXE with $k = 2$ which allows for merging of a death and a birth predicates into one.

- $\mathsf{srs} \leftarrow \mathsf{NARK}_{\oplus}.\mathcal{G}(\lambda, N)$: same as $\mathsf{NARK}.\mathcal{G}$ except $N = 2n$ where n is the size bound for each instance.
- $(\mathsf{ipk}_b, \mathsf{ivk}_b) \leftarrow \mathsf{NARK}_{\oplus}.\mathcal{I}^{\mathsf{srs}}(\Phi_b, b)$: Given circuit description Φ_b, slot number $b \in [2]$, and oracle access to SRS srs, it deterministically outputs the slotted proving key and verifying key $(\mathsf{ipk}_b, \mathsf{ivk}_b)$. The relation for the merged instance is $\mathcal{R}_{\oplus} := \{(\mathbb{x}_0 || \mathbb{x}_1, \mathbb{w}_0 || \mathbb{w}_1) : \phi_0(\mathbb{x}_0, \mathbb{w}_0) = 1 \wedge \phi_1(\mathbb{x}_1, \mathbb{w}_1) = 1\}$.
- $\mathsf{ipk} \leftarrow \mathsf{NARK}_{\oplus}.\mathcal{M}_{\mathsf{ipk}}(\mathsf{ipk}_0, \mathsf{ipk}_1)$: Given any two complementarily slotted proving keys $\mathsf{ipk}_0, \mathsf{ipk}_1$, it outputs a merged proving key ipk.
- $\mathsf{ivk} \leftarrow \mathsf{NARK}_{\oplus}.\mathcal{M}_{\mathsf{ivk}}(\mathsf{ivk}_0, \mathsf{ivk}_1)$: Given any two complementarily slotted verifying keys $\mathsf{ipk}_0, \mathsf{ipk}_1$, it outputs a merged verifying key ivk.
- $\mathbb{w} \leftarrow \mathsf{NARK}_{\oplus}.\mathcal{M}_{\mathbb{w}}(\mathbb{w}_0, \mathbb{w}_1)$: Given any two witnesses $\mathbb{w}_0, \mathbb{w}_1$ corresponding to relations $\mathcal{R}_{\Phi_0}, \mathcal{R}_{\Phi_1}$, it outputs a merged witness $\mathbb{w}$ for $\mathcal{R}_{\oplus}$.
- $\pi \leftarrow \mathsf{NARK}_{\oplus}.\mathcal{P}(\mathsf{ipk}, \mathbb{x}, \mathbb{w})$: same as $\mathsf{NARK}.\mathcal{P}$ except $N = 2n$.
- $b \leftarrow \mathsf{NARK}_{\oplus}.\mathcal{V}(\mathsf{ivk}, \mathbb{x}, \pi)$: same as $\mathsf{NARK}.\mathcal{V}$ except $N = 2n$.

We present a concrete construction of such technique for PLONK in Appx. D.

Analysis. A clear trade-off we make here is halving the number of proving instances by doubling the circuit size of each instance. Concretely in ZEXE's context, given an m-input-m-output transaction, we have $2m$ predicate proofs (m death and m birth) to be verified in the outer circuit, which is over a larger field with more expensive computation within it. Now by merging each pair of $(\Phi_{b,i}, \Phi_{d,i})_{i=1}^{m} \mapsto [\phi'_i]_{i=1}^{m}$, we reduce the number of inner predicate proofs to m, potentially lowering the outer circuit complexity. The concrete net saving is dependent on the choice of NARK proof system for predicate circuits. Assume a circuit of size n, the proof for the circuit satisfiability can be checked by a verifier gadget using $\mathscr{N}_n$ constraints; while a verifier gadget for circuit of size $2n$ takes $\mathscr{N}_{2n} = \mathscr{N}_n + \delta$ constraints. Our instance merging techniques effectively reduce the outer circuit complexity from roughly $2m \cdot \mathscr{N}_n$ to $m \cdot \mathscr{N}_{2n}$, which is a significant saving as long as $\delta \ll \mathscr{N}_n$. In the case of a Plonk verifier gadget, δ is very small and attributed to a few additional modular arithmetic constraints from computing the polynomial evaluations that are dependent on the doubled evaluation domain size; whereas $\mathscr{N}_n$ is orders of magnitude larger. Meanwhile, inevitably there is additional cost associated with a larger circuit per inner instance. The only noticeable cost boils down to running polynomial interpolations using FFT over a domain size of $2n$ instead of n during inner proof generation – effectively 2 FFT of the size n v.s. 1 FFT over the size of $2n$. Given that the running time of FFT is $O(n \cdot \log(n))$, the increased cost is really negligible compared to the efficiency gain from a simpler outer circuit.

3.3 Proof Batching

We describe a generic compiler that transforms a public-coin non-interactive argument that proves *a single* relation into an argument that batch proves a list of relations, while preserving all security properties. Notice that one could trivially run multiple instances of the argument protocol independently in parallel. Our compiler below is non-trivial as it reduces the total communication complexity (thus the final proof size) and the total verification computation, which in turn ultimately reduces the overall verifier circuit complexity in ZEXE compared to verifying them individually.

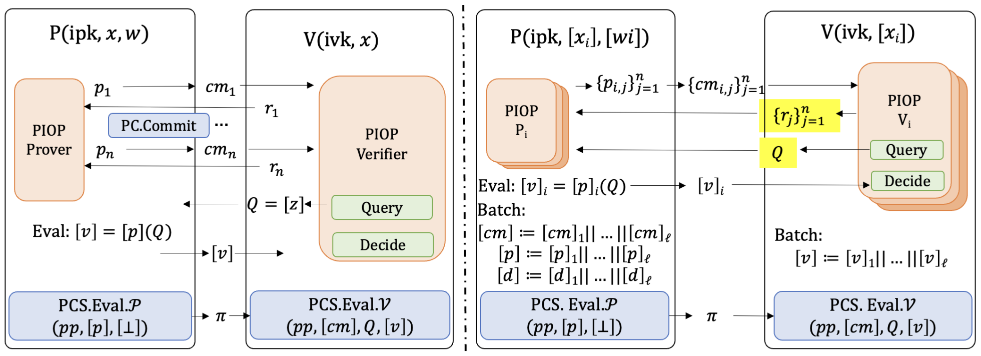
Syntax. A NARK that supports proof batching shares most of the syntax from NARK except that the proving and verification algorithm now accepts a list of instances, witnesses and proofs instead of one:

- $\pi_{\circledast} \leftarrow \mathsf{NARK}.\mathcal{P}_{\circledast}([\mathsf{ipk}_i]_{i=1}^{\ell}, [\mathbb{x}_i]_{i=1}^{\ell}, [\mathbb{w}_i]_{i=1}^{\ell})$: Given a list of ℓ proving keys, instances and witnesses, it proves them in batch and outputs a proof $\pi_{\circledast}$.
- $b \leftarrow \mathsf{NARK}.\mathcal{V}_{\circledast}([\mathsf{ivk}_i]_{i=1}^{\ell}, [\mathbb{x}_i]_{i=1}^{\ell}, \pi_{\circledast})$: Given a list of ℓ verifying keys, instances and an aggregated proof, it outputs a success bit b.

We explain the high level techniques below and present a concrete construction in Appx. D.

Technique. MARLIN presents a compiler that combines any public-coin AHP/PIOP for a relation $\mathcal{R}$ and an extractable polynomial commitment scheme to obtain a public-coin pre-processing argument with universal SRS for the same relation (see Theorem 1 in [CHM+20]). The universal SNARKs we use also fit into this construction paradigm, and we summarize it schematically in Fig. 3. To extend the above paradigm and support batching, the core idea is to leverage the batch opening of PCS, which reduces opening proofs size and amortizes verification cost. We observe that many existing PCSs have a *linear combination scheme*, and thus support batch opening of multiple polynomials at multiple points (proven in Theorem 3 of [BDFG21] on private aggregation scheme).

Next we summarize the general paradigm and its batching extension in Fig. 3. On the left side of Fig. 3 is an interactive argument between a Prover $\mathcal{P}$ and a Verifier $\mathcal{V}$ both of whom are running an information-theoretic PIOP as a sub-protocol. The prover starts by running the PIOP prover with given instance $\mathbb{x}$ and witness $\mathbb{w}$, where in each round it produces a polynomial p_i to be committed into cm_i and sent over to the verifier. Meanwhile the verifier $\mathcal{V}$ who internally runs the PIOP verifier randomly samples a coin r_i in each round, and at the end of n-th round, outputs a query set Q containing algebraic queries such as "evaluate $\{p_i\}$ at point r_j" or some polynomial identity testing. Upon receiving the queries, $\mathcal{P}$ calculates the replies as a list of evaluated values $[v]$ and return to $\mathcal{V}$ who will decide whether the replied values are acceptable. Additionally $\mathcal{P}$ has to prove that the replies to algebraic queries are consistent with committed polynomials by running $\mathsf{PCS.Eval}.\mathcal{P}$ as a sub-procedure whose opening proof will be verified by $\mathcal{V}$ who runs $\mathsf{PCS.Eval}.\mathcal{V}$.



P(ipk, x, w)
V(ivk, x)
PIOP Prover
p_1
cm_1
r_1
PC.Commit
...
p_n
cm_n
r_n
PIOP Verifier
$Q = [z]$
Query
Decide
Eval: $[v] = [p](Q)$
$[v]$
PCS.Eval.$\mathcal{P}$ $(pp, [p], [\perp])$
π
PCS.Eval.$\mathcal{V}$ $(pp, [cm], Q, [v])$
P(ipk, $[x_i]$, $[wi]$)
V(ivk, $[x_i]$)
PIOP $\mathrm{P_i}$
$\{p_{i,j}\}_{j=1}^n$
$\{cm_{i,j}\}_{j=1}^n$
$\{r_j\}_{j=1}^n$
Q
PIOP $\mathrm{V_i}$
Query
Decide
Eval: $[v]_i = [p]_i(Q)$
$[v]_i$
Batch:
$[cm] \coloneqq [cm]_1 || \dots || [cm]_\ell$
$[p] \coloneqq [p]_1 || \dots || [p]_\ell$
$[d] \coloneqq [d]_1 || \dots || [d]_\ell$
Batch:
$[v] \coloneqq [v]_1 || \dots || [v]_\ell$
PCS. Eval.$\mathcal{P}$ $(pp, [p], [\perp])$
π
PCS. Eval.$\mathcal{V}$ $(pp, [cm], Q, [v])$

Fig. 3: Generic compiler for batching PIOP-based NARKs.

On the right side of Fig. 3 is an interactive argument, compiled from the one on the left, for a list of relations $\{\mathcal{R}_i\}_{i=1}^{\ell}$ with the same size bound. In j-th round ($j \in [n]$), the i-th PIOP prover ($i \in [\ell]$), sends over the committed polynomial for that round $\{p_{i,j}\}$ and the PIOP verifier would replied with a random coin r_j *after it receives all polynomials from ℓ PIOP provers*. After n rounds of polynomial commitments and coin flips, the PIOP verifier outputs a *single query set* Q for all ℓ relations and the size of this set should be the same as that of a single PIOP run. Finally, $\mathcal{P}$ and $\mathcal{V}$ run batch opening of PCS over all polynomials at those query points.

Note that a strawman (yet non-trivial) compiler would run ℓ PIOP instances in parallel, where the verifier produces ℓ random challenges $\{r_{i,j}\}_{j=0}^{n}$ (in total $\ell \cdot n$ challenges) and ℓ query set Q_i. Subsequently the $\mathsf{PCS.Open}$ will proceed to prove opening of polynomials $\left\{\{p_{i,j}\}_{j=1}^{n}\right\}_{i=1}^{\ell}$ at different subsets in $Q := \bigcup\{Q_i\}$. In contrast, our compiler utilizes the same random challenges (in total n) and the same query set Q, independent of the number of batched relations ℓ, so that the batched opening of PCS is even simpler. Intuitively our compiler preserves security since these random challenges are only sent *after* receiving the committed polynomials (for that round) from all of the ℓ PIOP provers, and the query set is constructed *after* finishing the n rounds of *all* PIOPs, thus there won't be any knowledge soundness compromise (although the knowledge extractor requires slight modifications).

Analysis. The efficiency improvement of our generic compiler is dependent on the concrete choice of PCS and PIOP. For batching ℓ vanilla relations in PLONK[12] that combines an idealized low-degree protocol with KZG polynomial commitment:

- the prover benefits from sharing the same quotient polynomial (which is further split into 3 polynomials) and the two opening polynomials. The total proof size is reduced by $5(\ell - 1)$ $\mathbb{G}_1$ elements.

[12] Other variants such as TURBO-PLONK (PLONK with customized gates) and ULTRA-PLONK (with lookup table [GW20] support), the concrete efficiency will also be different and the improvement will mostly be more significant.

- the verifier requires only a single pairing check during batch checking; it also benefits from fewer and smaller MSMs thanks to the shared quotient and opening polynomials. The total number of base points in MSMs is reduced by $7(\ell - 1)$, and the total number of pairing operation is reduced by $(\ell - 1)$.

3.4 Variable-base Multi-Scalar Multiplication via Online Lookup Table

We generalize the lookup table argument in [GW20] by enabling a variant we call *online lookup table* to constrain MSM in circuit more efficiently.

Motivation. Recall that a b-bit multi-scalar multiplication (MSM) problem of size $n \in \mathbb{N}$ is to compute $Q = \Sigma_{i\in[n]}(s_i \cdot P_i)$ where $s_i \in [0, 2^b)$ are scalars, $P_i \in \mathbb{G}$ are bases, $\cdot$ is scalar multiplication, and $+$ is group addition. When all bases are fixed and known in advance, we call such instance a fixed-base MSM (fMSM); otherwise variable-base MSM (vMSM).

During verification of inner proofs in the outer circuit in ZEXE, there are some vMSM computation where the bases are commitments to witness polynomials inside proofs or commitments to preprocessed circuit descriptions inside verifying keys (note that verifying keys for user-defined predicates are dynamic). On a high level, we employ Pippenger-like [Pip80][13] strategy by reducing a b-bit MSM into b/c instances of c-bit MSMs ($c < b$), and finally summing them together. Particularly, when computing a c-bit MSM, instead of unrolling the exact Pippenger algorithm in the circuit which is very expensive,[14] we utilize a lookup table containing all resulting points from the scalar multiplications between the base point and all $2^c - 1$ possible scalar values. With such lookup table, any c-bit scalar multiplication on this specific base point becomes a table query rather than elliptic curve group operation. Given that these bases are unfixed, the lookup table cannot be pre-processed – table values are only known during online proving phase which give rise to our following technique.

Pre-processed v.s. Online Lookup Table. PLONKUP [GW20] presents a polynomial IOP (PIOP) protocol for checking values of a *query table* $\boldsymbol{f} := (f_1, \dots, f_n) \in \mathbb{F}^n$ are contained in the values of a *lookup table* $\boldsymbol{t} := (t_1, \dots, t_d) \in \mathbb{F}^d$. They further generalize the protocol to support vector lookup where each entry in the query table and lookup table is a vector (*i.e.* $f_i, t_i \in \mathbb{F}^w$); and to support multiple tables by adding an additional column for table index and concatenating multiple tables into one. However, the presentation in [GW20] only considers *pre-processed lookup tables* where values in the lookup tables are predefined and fixed. The key observation is that the PIOP protocol for lookup relations works regardless of how the query table and the lookup table are constructed – whether those values are known in advance or determined during online phase of the protocol run. Intuitively, the PIOP for online lookup tables still preserve soundness because online columns constructed by the prover are committed first (sent to the verifier for oracle access), before any verifier-initiated checks are carried out.

[13] We only use a special case of a simplified Pippenger algorithm which is sometimes refers to as "the bucket method". For detailed literature review and comparisons among different variants of Pippenger's predecessors, please see [Ber02].

[14] For each c-bit MSM, there are $2^c - 1$ buckets each representing a possible non-zero scalar. We need to compute the "bucket sum" $\{S_1, \dots, S_{2^c-1} \in \mathbb{G}\}$ by adding all base points that are supposed to multiply with that the scalar (*e.g.* S_i is computed by adding all bases that multiply with i, with $i \in [1, 2^c - 1]$), then finally the MSM result is computed as $\sum_{i\in[2^c-1]} i \cdot S_i$. We note that the main circuit complexity does not come from point additions, but maintaining $2^c - 1$ bucket sums and selectively update the correct bucket sum for each base and its scalar – which is trivial outside the circuit, but expensive to enforce inside the circuit.

Optimized MSM Circuit. With the online lookup table in our toolbox, we proceed to present an optimized circuit for MSM.

Inputs: Base point variables: $[P_1, \ldots, P_n]$, scalar variables: $[s_1, \ldots, s_n]$ where scalar values $\in [0, 2^b)$.
Outputs: A point variable $Q = \sum_{i\in[n]} s_i \cdot P_i$.
Circuit: We break b-bit MSM into $m := b/c$ instances of c-bit MSM and finally summing over m points.

1. For $i \in \{1 \ldots n\}$:
   (a) Compute $(2 \cdot P_i, \ldots, (2^c - 1) \cdot P_i)$ using repeated point addition from P_i.
   (b) Create online lookup table: $\mathcal{T}_i = [(0, 0_{\mathbb{G}}), (1, P_i), (2, 2 \cdot P_i), \ldots, (2^c - 1, (2^c - 1) \cdot P_i)]$.
   (c) Decompose s_i into m chunks of c-bit value $[s_{i,0}, \ldots, s_{i,m-1}]$, such that $s_i = \sum_{j=0}^{m-1} s_{i,j} \cdot 2^{cj}$ (we don't need to further range-check $s_{i,j}$, as it is implicitly constrained later in lookup gates).
2. For $j = \{0 \ldots m-1\}$:
   (a) For $i = \{0 \ldots n\}$:
      i. Create a point variable $Q_{i,j}$ for the value $s_{i,j} \cdot P_i$.
      ii. Add an entry to query table $(s_{i,j}, Q_{i,j})$ (lookup argument will check if $(s_{i,j}, Q_{i,j}) \in \mathcal{T}_i$).
   (b) Compute window sum: $\mathsf{wsum}_j = \sum_{i\in[n]} Q_{i,j}$.
3. Compute $Q = \sum_{j=0}^{m-1} \mathsf{wsum}_j \cdot 2^{cj}$.

Fig. 4: Optimized variable-base MSM using online lookup tables.

We denote an elliptic curve point addition gadget $\odot_{\mathsf{add}}$, point doubling gadget $\odot_{\mathsf{double}}$, linear combination gadget $\odot_{\mathsf{lc}}$ for k terms, lookup gadget (for either filling entry in query or lookup table) $\odot_{\mathsf{lookup}}$. Then our overall circuit size (number of gates) is dominated by:

$$n \cdot \left(\underbrace{(2^c - 2) \odot_{\mathsf{add}}}_{\text{step 1a}} + \underbrace{2^c \odot_{\mathsf{lookup}}}_{\text{step 1b}} + \underbrace{(m-1)/k \odot_{\mathsf{lc}}}_{\text{step 1c}} \right) + m \cdot \left(\underbrace{n \odot_{\mathsf{lookup}}}_{\text{step 2(a)ii}} + \underbrace{n \odot_{\mathsf{add}}}_{\text{step 2b}} \right) + \underbrace{m \odot_{\mathsf{add}} + b \odot_{\mathsf{double}}}_{\text{step 3}}$$

As a point of reference, with the TURBO-PLONK circuit used to generate benchmark number in § 4, which supports linear combination of $k = 4$ terms using 1 gate, elliptic curve point addition and doubling using 2 gate, a lookup entry or query using 1 gate, a 256-bit vMSM of size 128 takes only around 34,516 gates with chosen chunk size $c = \log(n) \approx 5$. In contrast with the naïve circuit implementation, the expected number of gates required is around 230,000[15] – our optimized circuit is more than 6.5 factor smaller.

Remark 4. Step 1b and 2(a)ii in Fig. 4 involves creating multiple online lookup tables and later querying from one of them. To achieve this, we implicitly apply the multiple table techniques presented in [GW20] by adding an extra domain separator column both in the merged lookup table and the merged query table. Furthermore, we note that in a TURBO-PLONK constraint system, with 3 (preprocessed) selector polynomials (2 domain separator polynomials and 1 polynomial for the fixed scalar value in the online lookup table) and 5 wire polynomials (2 for point variables in lookup table, 3 for the key-value tuple in query table), we can do an entry creation for both lookup table and query table in a single gate, thus reducing the total number of constraints required. (at the cost of longer proving and verifying keys). Specifically, instead of $(n \cdot 2^c + m \cdot n) \odot_{\mathsf{lookup}}$, we could just use $\max(n \cdot 2^c, m \cdot n) \odot_{\mathsf{lookup}}$.

[15] Roughly, a naïve variable-base MSM can be done by decomposing the scalars to binary representation, then perform conditional addition based on each bit, then finally adding all points together. The decomposing scalars takes $nb/k \odot_{\mathsf{lc}}$; the multiplication takes $6bn \odot_{\mathsf{add}}$ and final combining takes $n \odot_{\mathsf{add}}$ which adds up to 229,632.

3.5 Polynomial Evaluation over Non-native Field

Inner proofs for predicate satisfiability and outer proofs for inner proofs correctness are generated by circuits over different finite fields. Therefore, when running the inner proof verifier in the outer circuit, any polynomial evaluations would require modular arithmetics over a non-native field. In this section, we present efficient gadgets for two main building blocks: modular multiplications for evaluating each monomial and modular additions for summing over evaluations of all monomials. The stepping stone of our modular arithmetic gadgets is a range proof gadget that uses lookup table introduced in [GW20].

Let p, q be the sizes of two fields where $p^2 > q > p$, we want to show how to emulate modular arithmetics over $\mathbb{F}_p$ in a circuit over field $\mathbb{F}_q$. The common theme behind our design is enforcing: (a) an equivalent equation *over integers* expressing the congruence relation of the modular equation and (b) both sides of the equation won't overflow or underflow the native field size q at any intermediate step. For example, to constrain modular operation $z' \equiv x \cdot y \pmod p$, we ensure there exist witnesses w such that (i) $z' + pw = xy$ over integers, and (ii) arithmetic operations that simulate computations of $z' + pw, xy$ never exceeds the range $[0, q)$.

Assume that we already have a linear combination gadget $\odot_{\mathsf{lc}}$ for k_{lc} terms, and a preprocessed range table (with size $K := 2^k$) that enables us to constrain a variable x to be in the range $[0, K)$. We start by constructing a more general range-check circuit, and then build the modular addition/multiplication gadgets on top of it.

Range proofs. We present a range proof gadget in Fig. 5 with the circuit size: $n_{\mathsf{range}}(\ell) := \lceil \frac{\ell-1}{k_{\mathsf{lc}}} \rceil \odot_{\mathsf{lc}} + \ell \odot_{\mathsf{rg}}$.

Modular multiplications. Since we assume $p^2 > q$, we can't directly multiply $x, y \in \mathbb{F}_p$ in circuits over $\mathbb{F}_q$. Instead we choose to break each $\mathbb{F}_p$ element into two limbs with a splitting parameter m such that $2^{2m} \geq p$, so that we can represent any $x \in \mathbb{F}_p$ as $(x_0, x_1) \in [0, 2^m)^2$ such that $x = x_0 + 2^m x_1$. With the range proof gadget for the range $[0, K^\ell)$ in mind (where $K = 2^k$), we recommend fixing m by finding the minimum $\ell \in \mathbb{N}$ such that $2^{2\ell k} \geq p$ (namely we denote $m := \ell k$).

The intuition for proving $x \cdot y = z \pmod p$ is to find a witness $w \in \mathbb{F}_p$ such that $x \cdot y = z + p \cdot w$ holds over integers and that both sides won't overflow $\mathbb{F}_q$. Specifically:

$$(x_0 + 2^m \cdot x_1) \cdot (y_0 + 2^m \cdot y_1) = z_0 + 2^m \cdot z_1 + (w_0 + 2^m \cdot w_1) \cdot (p_0 + 2^m \cdot p_1)$$
$$\Updownarrow$$
$$z_0 + w_0 \cdot p_0 - x_0 \cdot y_0 + 2^m \cdot (z_1 + w_0 \cdot p_1 + w_1 \cdot p_0 - x_0 \cdot y_1 - x_1 \cdot y_0) + 2^{2m} \cdot (w_1 \cdot p_1 - x_1 \cdot y_1) = 0$$
$$\Updownarrow$$
$$\begin{cases} z_0 + w_0 \cdot p_0 - x_0 \cdot y_0 - 2^m \cdot c'_0 = 0 \\ z_1 + w_0 \cdot p_1 + w_1 \cdot p_0 - x_0 \cdot y_1 - x_1 \cdot y_0 + c'_0 - 2^m \cdot c'_1 = 0 \\ w_1 \cdot p_1 - x_1 \cdot y_1 + c'_1 = 0 \end{cases}$$

for some c'_0, c'_1 carriers bounded by $-2^m \leq c'_0 < 2^{m+1}$ and $-2^{m+1} \leq c'_1 < 2^{m+2}$.[16] We present the modular multiplication gadget in Fig. 6 with following notes:

[16] Since $z_0 + w_0 \cdot p_0 \in [0, 2^m + 2^{2m}), x_0 \cdot y_0 \in [0, 2^{2m})$, we know $\frac{0 - 2^{2m}}{2^m} = -2^m \leq c'_0 < \frac{2^m + 2^{2m} - 0}{2^m} = 2^m + 1 < 2^{m+1}$.
Similarly since $z_1 + w_0 \cdot p_1 + w_1 \cdot p_0 + c'_0 \in [-2^m, 2^m + 2^{2m+1} + 2^{m+1}), x_0 \cdot y_1 + x_1 \cdot y_0 \in [0, 2^{2m+1})$, we know $\frac{-2^m - 2^{2m+1}}{2^m} = -1 - 2^{m+1} < -2^{m+1} < c'_1 < \frac{2^m + 2^{2m+1} + 2^{m+1} - 0}{2^m} < 2^{m+2}$.

- To optimize gadget circuit size, we assume that the limbs of input x, y are already in range $[0, 2^m)$ without further checking.
- We shift the actual carriers c'_0, c'_1 to c_0, c_1 in order to have a positive range and upper-bounded by a power of K to utilize our range proof gadget.
- Witness $w \in \mathbb{F}_p$ must exist since we assume both $x, y \in \mathbb{F}_p$ even though we only constrain them to range $[0, 2^{2m})$ which is bigger than $[0, p)$.
- The prover must set witness z to be in range $[0, p)$ in order to continue feeding z as an input to the next modular multiplication gadget, even though another representation such as $z + p$ might still satisfy the current gadget. This is because our modular multiplication gate is only composable when the inputs are strictly within $[0, p)$ bound to guarantee existence of witness $w \in \mathbb{F}_p$.[17]

Public Parameters: $K \in [0, q), \ell \in \mathbb{N}$ where $K^\ell < q$
Input: $x \in \mathbb{F}_q$
Relation: $x \in [0, K^\ell)$
Circuit:

1. Create variables $x_0, \ldots, x_{\ell-1}$ and constrain $x = x_0 + K \cdot x_1 + K^2 \cdot x_2 + \ldots + K^{\ell-1} \cdot x_{\ell-1}$.
2. Range check variables $x_0, \ldots, x_{\ell-1} \in [0, K)$.

Fig. 5: Range proof gadget.

Public Parameters:

- predefined field sizes: $p^2 > q > p$.
- range of $\odot_{\text{rg}}$: $K = 2^k \in [0, q)$
- splitting parameter m such that $2^{2m} = 2^{2\ell k} \geq p$ for a minimum $\ell \in \mathbb{N}$
- limbs of prime (p_0, p_1) such that $p = p_0 + 2^m \cdot p_1$
- additional requirement: $k \geq 3 \ \wedge \ q > 2^{2m+k+1}$

Input: $(x_0, x_1), (y_0, y_1) \in [0, 2^m)^2$ such that $x = x_0 + 2^m \cdot x_1 \in \mathbb{F}_p, y = y_0 + 2^m \cdot y_1 \in \mathbb{F}_p$
Witness: $(w_0, w_1), (z_0, z_1)$
Relation: $(x_0 + 2^m \cdot x_1) \cdot (y_0 + 2^m \cdot y_1) = z_0 + 2^m \cdot z_1 + (w_0 + 2^m \cdot w_1) \cdot (p_0 + 2^m \cdot p_1)$ over integers
Circuit:

1. Range check $w_0, w_1, z_0, z_1 \in [0, 2^m)$
2. Compute carrier c'_0 and $c_0 = c'_0 + 2^m$,
   range check $c_0 \in [0, 2^{m+k})$ and constrain $z_0 + w_0 \cdot p_0 = x_0 \cdot y_0 + 2^m \cdot (c_0 - 2^m)$
3. Compute carrier c'_1 and $c_1 = c'_1 + 2^{m+1}$,
   range check $c_1 \in [0, 2^{m+k})$ and constrain
   $z_1 + w_0 \cdot p_1 + w_1 \cdot p_0 + (c_0 - 2^m) = x_0 \cdot y_1 + x_1 \cdot y_0 + 2^m \cdot (c_1 - 2^{m+1})$
4. Constrain $w_1 \cdot p_1 + (c_1 - 2^{m+1}) = x_1 \cdot y_1$

Fig. 6: Modular multiplication gadget. In circuit description, blue texts are actual circuit constraints whereas black normal text are computation outside the circuit.

Proposition 1. *The modular multiplication gadget in Fig. 6 satisfies*

- ***Completeness**: Given public parameters, for any inputs and their valid witnesses, the circuit for the relation should always be satisfied.*

[17] Our circuit constrains $z, w \in [0, 2^{2m})$, hence $z + p \cdot w$ should be in range $[0, 2^{2m} + 2^{2m} \cdot p)$ with p a fixed public parameter. Suppose inputs $x, y \in [0, 2^{2m})$ exceed p and $x \cdot y$ exceeds $2^{2m} + 2^{2m} \cdot p$ (which is possible as $2^{2m} > p$), then it's impossible to find a proper witness w such that $x \cdot y = z + p \cdot w$.

- ***Soundness:*** *Given public parameters, for any inputs and invalid witnesses, the circuit should never be satisfied.*

See proofs in Appx. E.

Using the ULTRA-PLONK constraint system specified in Def. 3, the circuit size of modular multiplication gadget is: $5+4\cdot n_{\mathsf{range}}(\ell)+2\cdot n_{\mathsf{range}}(\ell+1)$ UltraPlonk constraints. With $k=15, k_{\mathsf{lc}}=4$, range-check of $[0,K)$ for free, and $\mathbb{F}_q, \mathbb{F}_p$ be the base field and scalar field of BLS12-377 curve, our gadget uses only 23 constraints.

Remark 5 (On circuit complexity of range checks). In an ULTRA-PLONK constraint system, by adding a dedicated input wire to each gate (an additional wire polynomial), we can piggyback the range-checking of a variable on any existing gate instead of requiring a dedicated new gate. As long as the number of range checks are fewer than total number of gates used for the rest of the proof relation, we don't have to increase circuit size, in which case we effectively support range checks "for free".

Remark 6. While there are alternative designs for emulating non-native modular multiplication such as [Gab], those designs usually are more general and applicable for any p, q even when $q<p$. In contrast, we have a specialized use case of two-chain curves for depth-2 proof recursion (instead of cycling curves for deeper proof recursions) in mind, therefore we can safely assume finer-grained requirements on p, q to make our circuit more efficient.

Modular additions. The intuition for proving $y = x_1 + \ldots + x_N \pmod p$ is to find a witness $w \in \mathbb{F}_p$ such that $y + p\cdot w = x_1 + \ldots + x_N$ *over integers* and that both sides won't overflow $\mathbb{F}_q$. Assume we take the splitting parameter m from foregoing modular multiplication gadget, we present the modular addition gadget in Fig. 7.

Public Parameters:

- predefined field sizes: $p^2 > q > p$.
- range of $\odot_{\mathsf{rg}}$: $K = 2^k \in [0, q)$
- splitting parameter m such that $c \cdot p \geq 2^{2m} \geq p$ for a minimum $c \in \mathbb{N}$
- maximal number of summards allowed: $N < \frac{K-1}{c} + 1$
- additional requirement: $\frac{q}{p} > c + K$

Input: $x_1, \ldots, x_N \in [0, 2^{2m})$
Witness: w, y
Relation: $y + p\cdot w = x_1 + \ldots + x_N$ over integers
Circuit:

1. Range check $y \in [0, 2^{2m}), w \in [0, K)$
2. Constrain $y + p \cdot w = x_1 + \ldots + x_N$

Fig. 7: Modular addition gadget.

Proposition 2. *The modular addition gadget in Fig. 7 satisfies* ***completeness*** *and* ***soundness****.*

See proofs in Appx. E.

The circuit size of our modular addition gadget is: $n_{\mathsf{addmod}} = \lceil \frac{N}{k_{\mathsf{lc}}} \rceil \ \odot_{\mathsf{lc}} + 1 \ \odot_{\mathsf{rg}} + n_{\mathsf{range}}(2m)$. With $k=15, k_{\mathsf{lc}}=4$, range-check of $[0,K)$ for free, and $\mathbb{F}_q, \mathbb{F}_p$ be the base field and scalar field of BLS12-377 curve, our gadget uses only $\lceil \frac{N}{4} \rceil + 6$ constraints for simulating an addition of N terms.

3.6 SNARK-friendly Symmetric Primitives

Recall that the circuit for relation $\mathcal{R}_e$, which governs the rules of a valid state transition in $\mathsf{DPC.Execute}^{\mathsf{L}}$, requires constraining some symmetric cryptographic primitives such as commitment schemes, pseudo-random functions (PRF), and collision-resistant hashes (CRH). However, some standard implementations of these primitives involves many non-algebraic operations (*e.g.* bit-wise XOR, rotate in SHA256 and s-box bytes substitution in AES) which takes a lot of gates to constrain in an algebraic circuit. There are two main ways to constraining these primitives inside circuit more efficiently:

1. precompute a lookup table containing legitimate (input, output) tuples[18] and the prover argues the witnesses (input and output of intermediate, non-algebraic steps) belongs to the table [GW20].
2. use SNARK-friendly primitives specifically designed to work natively with finite field elements by using mostly algebraic computations and removing non-algebraic steps (notably new hash functions: Rescue and Vision [AABS+20], Poseidon [GKR+21], MiMC [AGR+16]).

Generally, the latter approach produces smaller circuits at the cost of reliance on newer, less time-tested designs which are often much slower outside the circuit due to lack of hardware acceleration. The former approach may allow better candidates with better security bound or relies on weaker cryptographic assumptions for the required security properties.

Fiat-Shamir Transcript. Many universal SNARKs are made non-interactive by applying Fiat-Shamir transformation [FS87] on a public-coin interactive argument where random challenges sent from the verifier are deterministically simulated by hashing all previous transcripts between the prover and the verifier. The heuristic security of these SNARKs assume these hash functions as random oracles. In practice, these random oracles are instantiated using Blake2s or the keccak permutation in SHA3,[19] all of which incurs high circuit complexity as their internals entail many non-algebraic operations. Since the verifier logic includes deriving the verifier challenges in the transcript which should be constrained in the outer circuit, we are motivated to use one of the techniques above to reduce its circuit complexity.

As a point of reference, Halo2 [Ele] designed a highly optimized circuit for SHA256 using lookup table with an overall cost of 2099 TURBO-PLONK constraints; whereas CAP project [KCCX21] designed a circuit for a CRH from using Rescue permutation in a sponge construction with only 148 TURBO-PLONK constraints. Granted that the arithemtizations in those two TURBO-PLONK designs are slightly different, and such numbers can only be used for informal comparison. We decide to use Rescue-based hash when constraining verifier challenges derivation from the transcripts for its better circuit efficiency, knowing that it is still a philosophical question whether these SNARK-friendly hashes suffice as random oracles.

Predicate Commitments. To ensure that death/birth predicates involved in $\mathcal{R}_{\mathsf{utxo}}$ and $\mathcal{R}_{\Phi}$ are consistent, [BCG+20] proposes to make the hiding commitment cm_{Φ} to the predicates in a transaction as a public input for both circuits so that the verifier

[18] For instance, to assist bit-wise XOR between any two 8-bit integers, we can build a table of size 2^{16} of all possible two integer inputs and their XOR outputs – namely each entry is a tuple $(x, y, x \oplus y)$.

[19] For instance, Aztec 2.0 (`https://hackmd.io/@aztec-network/ByzgNxBfd`) and Halo2 (`https://github.com/zcash/halo2`) uses Blake2s; Merlin library (`https://github.com/dalek-cryptography/merlin`) uses keccak permutation.

can check their equality. Concretely, the original ZEXE instantiate CRH with Pedersen hash, COM with Blake2s hash where the message is appended with a randomizer for the hiding property. The primary circuit cost comes from constraining non-algebraic Blake2s hash on a message size of $m + n + 1$ for an m-input-n-output transaction.

We emphasize that directly switching Blake2s to a SNARK-friendly hash is not immediately more advantageous, since we need to constrain this hash function in two different fields (over $\mathbb{F}_r$ for $\mathcal{R}_{\mathsf{utxo}}$ and over $\mathbb{F}_p$ for $\mathcal{R}_{\Phi}$), and constraining algebraic hashes over non-native fields is probably more expensive as it requires many range checks and modular arithmetics. Worse, the number of non-native operations grows linearly with the message size since longer messages require more invocations of the hash function.

In Appx. F, we propose an efficient solution whose non-native operations does not grow regardless of the number of predicates committed, by utilizing rescue-hash-based commitment schemes.

4 Implementation and Evaluation

In this section, we provide an overview of our VERIZEXE implementation with most of our optimization techniques applied and report our experimental evaluation.

4.1 System Implementation

We implemented the delegable DPC scheme and applied all optimizations (§ 3) except the predicate commitment technique (Fig. 14). The resulting system is a ZEXE that only requires a one-time universal setup to produce the system parameter required for all future user-defined predicates which we affectionately call VERIZEXE. Our code base, written in Rust, follows the stack shown in Fig. 8: we utilized **arkworks** library [ac22] as the underlying algebra backend for finite field, elliptic curve, and polynomial operations; necessary cryptographic primitives including zkSNARKs and their circuit constraints are built on top; finally a VERIZEXE library that instantiates the DPC scheme using all building blocks below.

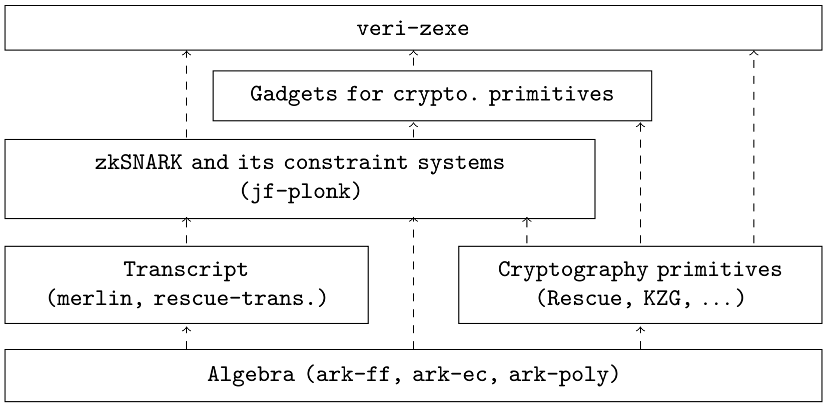

veri-zexe
Gadgets for crypto. primitives
zkSNARK and its constraint systems (jf-plonk)
Transcript (merlin, rescue-trans.)
Cryptography primitives (Rescue, KZG, ...)
Algebra (ark-ff, ark-ec, ark-poly)

Fig. 8: Stack of libraries comprising VERIZEXE.

We break down our concrete instantiations of cryptographic building blocks used to generate benchmark below:

Elliptic Curves. We use two pairing-friendly elliptic curves $E_{\mathsf{BLS}}, E_{\mathsf{BW}}$ in a similar fashion as [BCG⁺20] to support one layer of proof composition, and one twisted edwards curve $E_{\mathsf{Ed/BLS}}$ whose base field matches the prime order subgroup of E_{BLS}. Inner proofs are generated over the BLS12-377 curve (inner circuits are over its scalar fields), and outer proofs are generated over the BW6-761 curve [HG20] (outer circuits are over its scalar field which matches the base field of BLS12-377). Additionally, for some cryptographic primitives that requires a DLP-hard group (*e.g.* transaction signing in delegable DPC), we use the twisted edwards curve whose base field is the scalar field of the BLS12-377 curve.

Pseudorandom Permutation. Many of our following primitives are built from an algebraic pesudorandom permutation using *Rescue* algorithm [AABS⁺20].

The Rescue PRP is defined by a square matrix **MDS** of size $\mathtt{w}\times\mathtt{w}$ (in our instantiation $\mathtt{w} = 4$), an initial constants vector **IC**, and a key-scheduling constant vector **C** and a key-scheduling matrix **K**. We set the number of rounds $\mathtt{n_r} = 12$. For the S-box parameter α, we set $\alpha = 11$ for BLS12-377's scalar field (used by the inner circuit) and $\alpha = 5$ for BLS12-377's base field (used by the outer circuit). Note that during key scheduling, the key injection vectors can be preprocessed yielding a much faster generation of round keys. Formally, our Rescue instance works over a field $\mathbb{F}$, with keys and inputs of size 4 field elements: $m' \leftarrow \mathsf{PRP}(k, m)$ where $k, m, m' \in \mathbb{F}^4$.

For our PRF and hash function below, we need a fixed-key permutation as a building block rather than the full Rescue PRP. We build this by setting the key to the 0 vector: $m' \leftarrow \mathsf{FixedKeyPRP}(m) = \mathsf{PRP}([0, 0, 0, 0], m)$ where $m, m' \in \mathbb{F}^4$.

Pseudorandom Function. We build a sponge-based PRF from the fixed-key Rescue permutation. The construction follows the Full-State Keyed Sponge (FKS) paradigm (see Algorithm 1 in [MRV15]) but here is simplified to output a single field element. The PRF takes a secret key k of one field element, a message $\boldsymbol{m}$ of fixed length: $y \leftarrow \mathsf{PRF}_n(k, x)$ where $k, y \in \mathbb{F}, x \in \mathbb{F}^n$.

The Full-State Keyed Sponge construction works as follows: it set the initial state with zeroes and the key in the last slot. Then it divides the input in chunks of Rescue's state size, and absorb them sequentially by 1) adding the chunk to the state, and 2) calling the Rescue permutation to produce a new state. After the input has been absorbed, it outputs the first element of the state[20].

CRH. We build our collision-resistant hash (CRH) using the Sponge construction [BDPVA07] on top of our Rescue fixed-key permutation. In our instantiation of Rescue, the permutation state is of width 4: 3 slots for the *rate* and 1 for the *capacity* of the sponge construction. We provide two instantiations of the sponge-based CRH. The first one assumes the input length is multiple of the rate. The second one applies the following simple padding before calling the Sponge CRH: append the field element 1 to the input, then append zeroes as necessary until length is multiple of the rate. In short, we have a family of CRH that supports $H : \mathbb{F}^* \mapsto \mathbb{F}$ for arbitrary number of field elements as the pre-image.

Merkle Tree. The append-only ledger L is instantiated using a Merkle tree to accumulate all published record commitments and to generate membership proofs for old input records inside a transaction. Specifically, we implemented a *ternary* Merkle tree (branch factor is 3) using our Rescue-based CRH introduced above. Notice that the permutation in our hash function takes in 4 field elements, out of which the last one is reserved for padding to avoid prefixing attacks. Thus, a ternary Merkle tree is tailored for our hash function in terms of circuit constraints. Our Merkle tree is of fixed height, a parameter initialized during system setup; and it is incremental meaning it is possible to dynamically insert new leaves and update the Merkle root in time $O(\log M)$ where M is the maximum number of leaves allowed. For details on domain separation for different types of nodes in prevention of prefixing attacks and other formal security proofs, please refer to Section 4.1.8 in [KCCX21].

Commitments. We build a resuce-based commitment scheme that takes in a message $\boldsymbol{m} \in \mathbb{F}^n$ of some fixed length n and a randomly sampled blinding factor s, outputs a hiding commitment $c \leftarrow \mathsf{Commit}_n(s, \boldsymbol{m}) = \mathsf{CRH}(s\|\boldsymbol{m}\|\mathbf{0})$ where $\mathbf{0}$ are padded zeros so that the total input to CRH is of multiples of its rate (*i.e.* 3), CRH is the first instantiation of rescue-based CRH introduced above, and $c \in \mathbb{F}$. Intuitively, the binding

[20] For arbitrary length output, the squeeze phase proceeds as in a sponge construction: the rate part of the state is outputed, then the permutation is applied to the state to produce more output chunks until desired output length is achieved

and hiding property of our commitment scheme is derived from the collision resistance and one-wayness of the rescue permutation respectively. We further note that we can safely pad zeros to the fixed-length input messages because any message input of mismatching length should be rejected.

NARK. We instantiate the NARKs using KZG-based PLONK [GWC19]. Concretely, our predicate circuit and circuit for the relation $\mathcal{R}_{\mathsf{utxo}}$ uses our TURBO-PLONK constraint system over E_{BLS} (see Def. 2) with customized gates optimized for rescue-based statements; while our outer circuit for the relation $\mathcal{R}_{\Phi}$ uses our ULTRA-PLONK constraint system over E_{BW} (see Def. 3) with lookup table for efficient range proofs and variable-base MSM gadgets. We further extend the normal capability of a zkSNARK to support instance merging, proof batching and lightweight verifier gadget for our outer SNARK as illustrated in Appx. D. Note that our inner circuits don't need to be zero-knowledge, only the outer circuit require zero-knowledge to reveal nothing about the predicate verification keys used in order to achieve the function privacy of a DPC scheme.

4.2 Experimental Evaluation

We now provide experimental evaluations of our VERIZEXE system with optimization techniques described in § 3, and concrete implementation details in § 4.1.

Metrics and evaluation methodology. As an instantiation of the DPC scheme, our measurements focus on the resources required (including time, memory usage, storage) during the execution of the four algorithms of a DPC scheme (namely $\mathsf{Setup}, \mathsf{GenAddress}, \mathsf{Execute}^{\mathsf{L}}, \mathsf{Verify}^{\mathsf{L}}$). Particularly, the primary target of our optimization has been the circuit complexity of the NP relation $\mathcal{R}_{\Phi}$ (namely the outer circuit) whose SNARK proof generation dominates the cost of $\mathsf{Execute}^{\mathsf{L}}$ (namely transaction generation procedure) – which directly affect the usability and practicality of the final private computation system. To wit, we also provide microbenchmarks on the circuit costs of important cryptographic building blocks used. Note that we do not provide evaluations on dimensions or parts that our optimizations has mild or no effect on, such as the $\mathsf{GenAddress}$ algorithm or the transaction size besides its validity proof size.

All our reported data are measured on an AWS EC2 instance running Ubuntu 20.04. The server has 64 cores (AMD EPYC 7R13 at 2.65 GHz) and 128 GB of RAM.

General benchmark. We first compare our system against other DPC implementations on important dimensions. We manage to find only a handful of public DPC repositories, some are outdated and no longer maintained.[21]. To the best of our knowledge, the most relevant and actively maintained implementation is *snarkVM* by the Aleo team many of whom are the co-authors of the Zexe paper. While there are a few versions of DPC instantiations inside snarkVM, we focus on its testnet-1 and testnet-2 versions. [22]

Here we outlines the technical difference between our system and snarkVM's. First, snarkVM chooses to verify SNARK proof for $\mathcal{R}_{\mathsf{utxo}}$ together with predicate SNARK

[21] *e.g.* `arkworks` has a DPC code base extracted from the original `libzexe` used in submission of [BCG+20], but it is unmaintained, so is Celo's fork of this repository.

[22] We note that snarkVM had shifted away from the original DPC design by removing the notion of death/birth predicates altogether since their testnet-3 and even some late iterations of testnet-2, therefore we only use their earlier testnet-2 version when it still faithfully instantiates the DPC model in the original paper. The design and performance of this new DPC model is outside the scope of this paper.

	Setup		Execute$^{\mathsf{L}}$			
			$\mathcal{R}_{\Phi}$(outer circuit)			
Tx. Dim.	**Time** (s)	**SRS size** (MB)	**Constraints**	**Prover** (s)	**Time** (s)	**Memory** (GB)
snarkVM			R1CS			
2×2	176.8	5,254.2	4,235,068	138.5	151.4	16.6
3×3	246.0	7,056.6	6,330,496	202.7	223.0	20.5
4×4	370.1	10,454.9	8,447,588	293.2	321.1	27.1
verizexe			UltraPlonk			
2×2	11.8	33.1	87,176	13.1	16.9	6.5
3×3	18.4	66.2	126,076	24.7	29.2	8.5
4×4	19.1	66.2	141,492	24.8	32.4	9.1

Table 2: Performance comparison against the state-of-the-art DPC implementation across different transaction dimensions (*e.g.* 2×2 means 2-input-2-output transaction). Under Execute$^{\mathsf{L}}$ category, "Prover" column refers to prover time for outer circuit whereas "Time" column refers to the overall transaction generation time. snarkVM refers to its testnet-2 version where both $\mathcal{R}_{\mathsf{utxo}}$ and $\mathcal{R}_{\Phi}$ are proven using Groth16, birth/death predicates are proven using Marlin; whereas VERIZEXE uses TurboPlonk for both $\mathcal{R}_{\mathsf{utxo}}$ and birth/death predicates, $\mathcal{R}_{\Phi}$ is proven using UltraPlonk. Notice that structured reference string (SRS) size for snarkVM contains the universal SRS of Marlin and preprocessed proving keys of the inner and outer circuits; whereas that for VERIZEXE only contains two universal SRS, one for $\mathcal{R}_{\mathsf{utxo}}$ and predicate circuits, the other for the outer circuit. Further note that constraints number reported for snarkVM are referring to R1CS constraints whereas the number for VERIZEXE are Plonk constraints. All death and birth predicates requires 2^{15} constraints in their respective constraint systems.

Gadgets	**Field of Operation**	**# Constraints**
Rescue Permutation	native over BLS	$n_r = 388$
	native over BW	$n_p = 148$
	non-native over BW	$n_{\mathsf{nn}} = 23,760^*$
CRH	native over BLS	$(\lceil \frac{\ell}{3} \rceil + k - 1) \cdot n_r + 4$
(input: $\mathbb{F}^{\ell}$, output: $\mathbb{F}^{k}$)	native over BW	$(\lceil \frac{\ell}{3} \rceil + k - 1) \cdot n_p + 4$
Commitment	native over BLS	$\lceil \frac{\ell+1}{3} \rceil \cdot n_r + 4$
(input: $\mathbb{F}^{\ell}$)	non-native over BW6	$\lceil \frac{\ell+1}{3} \rceil \cdot n_{\mathsf{nn}} + 4^*$
PRF (input: $\mathbb{F}^{\ell}$)	native over BLS	$\lceil \frac{\ell}{4} \rceil \cdot n_r + 4$
Merkle Path (depth: ℓ)	native over BLS	$(5 + n_r) \cdot \ell + n_r$
ECC Add	native over both	2
Mod Add (input: $\mathbb{F}_r^{\ell}$)	non-native over BW	$\lceil \frac{\ell}{4} \rceil + 6^*$
Mod Mul (input: $\mathbb{F}_r^{2}$)	non-native over BW	23^*
Plonk Verifier		
1 proof	native over BW	$20,232^*$
2 proofs	native over BW	$31,407^*$
3 proofs	native over BW	$42,407^*$
4 proofs	native over BW	$53,735^*$

Table 3: Number of Plonk constraints for major cryptographic building blocks and algebraic operations. These numbers are specific to the TurboPlonk design (see Def. 2), those annotated with $*$ refers to the number of UltraPlonk constraints (see Def. 3). Furthermore, we denote the scalar field of BLS12-377 as $\mathbb{F}_r$, and the scalar field of BW6-761 as $\mathbb{F}_p$.

proofs inside its outer circuit, thus producing only a single outer proof instead of the two proofs per transaction as described in Section 7 of the Zexe paper. To ensure a fair comparison, we have modified their code to accurately reflect the original paper as our VERIZEXE does. SnarkVM testnet-1 (the same implementation used to generate experimental data in Section 9 of [BCG+20]) uses [GM17] for birth/death predicates each of which requires a trusted setup. SnarkVM testnet-2 uses universal SNARK Marlin [CHM+20] for predicates and this will serve as the primary benchmark to gauge the improvements gained from our optimizations.

As shown in Table 2, we achieve a $10.6 \sim 11.8$x improvement on outer proof generation, and a $9 \sim 10$x on overall transaction generation speed; the latter is the most important bottleneck and the determining factor of the usability of a DPC system. Notwithstanding the impossible task of directly comparing numbers of R1CS constraints to numbers of Plonk constraints, it is evident that our optimizations have kept the outer circuit complexity relatively low which results in faster proof generations. We also observe a $2.6 \sim 3.0$x improvement on memory usage during transaction generation, this helps alleviates the hardware requirements for users.

Astute readers may notice the non-linear slowdown in VERIZEXE's performance from 2×2 to 3×3. This is caused by the large number of range checks invoked by non-native rescue permutation pushing the evaluation domain size for FFT to a higher power-of-two, thus effectively increase the cost across the board from universal SRS generation to proving key indexing to proving[23].

For the Setup algorithm, our VERIZEXE is also notably faster. We note that it is a one-time, universal setup for both candidates, thus it is arguably less important in practice. We do want to highlight another significant differences in SRS size – snarkVM has much larger SRS since it requires to store preprocessed proving keys of the $\mathcal{R}_{\mathsf{utxo}}$ and $\mathcal{R}_{\Phi}$ (outer) circuits (Groth16's trusted setups are circuit-dependent), whereas VERIZEXE only contains two universal SRS. We stress that SRS size matters in practice as they are the (partial) size of the system parameter a user needs to download from ledger maintainers when he first join the system.

[23] We could further reduce the number of non-native operations when we implements the optimized predicate commitment in § 3.6.

Microbenchmarks. Since most of our techniques are attempts to reduce the outer circuit complexity, we now provide microbenchmark on concrete circuit costs for major components in Table 3. Among them, one of the highlights is our Plonk verifier gadget only taking roughly 21k UltraPlonk constraints for verifying a single TurboPlonk proof. This is made possible primarily thanks to highly efficient modular arithmetic gates (see § 3.5) for polynomial evaluation over non-native field and compact variable-based MSM gadget (see § 3.4). To illustrates the improvement attributed to our Pippenger-based vMSM gadget relying on online lookup table technique, we provide a benchmark against a naïve implementation in Fig. 9.

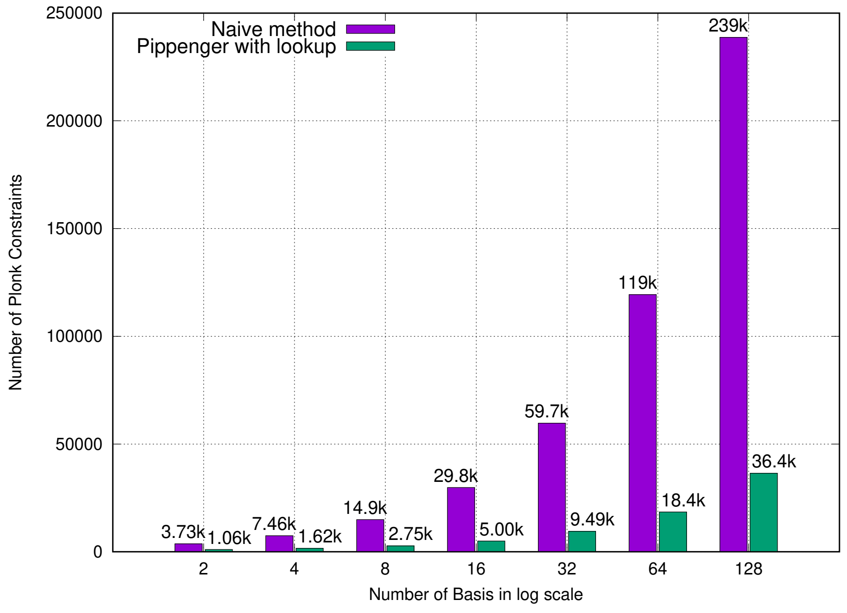


Fig. 9: Circuit complexity for variable-based MSM.

References

AABS+20. Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Design of symmetric-key primitives for advanced cryptographic protocols. *IACR Trans. Symmetric Cryptol.*, 2020:1–45, 2020. 7, 23, 25

ac22. arkworks contributors. `arkworks` zksnark ecosystem. https://arkworks.rs, 2022. 25

AGR+16. Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, 2016. 23

BAZB20. Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security*, pages 423–443, Cham, 2020. 3, 3

BBB+18. Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short proofs for confidential transactions and more. *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334, 2018. 8, 14

BBHR18. Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, 2018. 8

BCCT12. Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12. Association for Computing Machinery, 2012. 8

BCG+18. Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune Kristian Jakobsen, and Mary Maller. Nearly linear-time zero-knowledge proofs for correct program execution. In *IACR Cryptol. ePrint Arch.*, 2018. 8

BCG+20. Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 947–964, 2020. 3, 3, 4, 4, 7, 7, 8, 9, 10, 11, 23, 25, 27, 29, 48, 48

BCI+12. Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In *TCC*, 2012. 8

BCMS20. Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data from accumulation schemes. *IACR Cryptol. ePrint Arch.*, 2020:499, 2020. 5, 12, 12, 12, 13, 14, 14

BCT21. Aritra Banerjee, Michael Clear, and Hitesh Tewari. zkhawk: Practical private smart contracts from mpc-based hawk. *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 245–248, 2021. 7

BDFG21. Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Halo infinite: Proof-carrying data from additive polynomial commitments. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021*, pages 649–680, Cham, 2021. Springer International Publishing. 16

BDPVA07. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *ECRYPT hash workshop*. Citeseer, 2007. 26

Ber02. Daniel Bernstein. Pippenger's exponentiation algorithm, 01 2002. https://cr.yp.to/papers/pippenger-20020118-retypeset20220327.pdf. 18

BFM88. Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88. Association for Computing Machinery, 1988. 8

BFS20. Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent snarks from dark compilers. In *39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings*, volume 12105 of *Lecture Notes in Computer Science*. Springer, 2020. 4

BGG18. Sean Bowe, Ariel Gabizon, and Matthew D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. In *Financial Cryptography and Data Security - FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers*, Lecture Notes in Computer Science, 2018. 8

BGH19. Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021, 2019. https://ia.cr/2019/1021. 5, 12

BSCG+14. Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014. 3, 3, 9

CFF+21. Matteo Campanelli, Antonio Faonio, Dario Fiore, Anaïs Querol, and Hadrián Rodríguez. Lunar: A toolbox for more efficient universal and updatable zksnarks and commit-and-prove extensions. In *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part III*, 2021. 8

CHM+20. Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 738–768. Springer, Cham, 2020. 3, 4, 8, 16, 29

CZJ+17. Ethan Cecchetti, Fan Zhang, Yan Ji, Ahmed E. Kosba, Ari Juels, and Elaine Shi. Solidus: Confidential distributed ledger transactions via pvorm. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017. 3

CZK+19. Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew K. Miller, and Dawn Xiaodong Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200, 2019. 3

Ele. The halo2 book. https://zcash.github.io/halo2/index.html. Accessed: 2022-04-26. 23

FS87. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg. 23

Gab. Ariel Gabizon. Aztec emulated field and group operations. https://hackmd.io/LoEG5nRHQe-PvstVaD51Yw. Accessed: 2022-04-26. 22

GGPR13. Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, Lecture Notes in Computer Science, 2013. 8

GKM+18. Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-snarks. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 698–728, Cham, 2018. Springer International Publishing. 8, 36

GKR+21. Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In *USENIX Security Symposium*, 2021. 23

GM17. Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 581–612, Cham, 2017. Springer International Publishing. 3, 4, 8, 29

GMR85. S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, New York, NY, USA, 1985. Association for Computing Machinery. 8

Gro10. Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *ASIACRYPT*, 2010. 8

Gro16. Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 305–326. Springer, 2016. 3, 4, 8

GW20. Ariel Gabizon and Zachary J Williamson. plookup: A simplified polynomial protocol for lookup tables. *IACR Cryptol. ePrint Arch.*, 2020:315, 2020. 6, 6, 8, 17, 18, 18, 18, 19, 20, 23, 38, 39, 41

GWC19. Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, 2019:953, 2019. 5, 8, 27, 39, 44, 44, 45, 46

HG20. Youssef El Housni and Aurore Guillevic. Optimized and secure pairing-friendly elliptic curves suitable for one layer proof composition. *IACR Cryptol. ePrint Arch.*, 2020:351, 2020. 25

KCCX21. Fernando Krell, Binyi Chen, Philippe Camacho, and Alex Xiong. Configurable asset privacy: Specification. https://raw.githubusercontent.com/EspressoSystems/cap/master/cap-specification.pdf, 2021. Accessed: 2022-04-25. 23, 26

Kil92. Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '92. Association for Computing Machinery, 1992. 8

KKK21. Thomas Kerber, Aggelos Kiayias, and Markulf Kohlweiss. Kachina – foundations of private smart contracts. *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–16, 2021. 7, 8

KMS$^+$16. Ahmed E. Kosba, Andrew K. Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. *2016 IEEE Symposium on Security and Privacy (SP)*, pages 839–858, 2016. 3, 3, 7

KZG10. Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 177–194, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. 4, 5, 34, 35, 40

MBKM19. Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2111–2128, New York, NY, USA, 2019. Association for Computing Machinery. 8

Mic00. Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 2000. 8

MRV15. Bart Mennink, Reza Reyhanitabar, and Damian Vizár. Security of full-state keyed sponge and duplex: Applications to authenticated encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 465–489. Springer, 2015. 26

Nak08. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, Dec 2008. Accessed: 2015-07-01. 3, 9

NVV18. Neha Narula, Willy Vasquez, and Madars Virza. zkledger: Privacy-preserving auditing for distributed ledgers. In *IACR Cryptol. ePrint Arch.*, 2018. 3

PFM$^+$22. Luke Pearson, Joshua Fitzgerald, Héctor Masip, Marta Bellés-Muñoz, and Jose Luis Muñoz-Tapia. Plonkup: Reconciling plonk with plookup. Cryptology ePrint Archive, Report 2022/086, 2022. https://ia.cr/2022/086. 41

Pip80. Nicholas Pippenger. On the evaluation of powers and monomials. *SIAM Journal on Computing*, 9(2):230–250, 1980. 6, 18

SA21. Ravital Solomon and Ghada Almashaqbeh. smartfhe: Privacy-preserving smart contracts from fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2021:133, 2021. 7

SBBV22. Samuel Steffen, Benjamin Bichsel, Roger Baumgartner, and Martin Vechev. Zeestar: Private smart contracts by homomorphic encryption and zero-knowledge proofs. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1543–1543. IEEE Computer Society, 2022. 7

SBG$^+$19. Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin T. Vechev. zkay: Specifying and enforcing data privacy in smart contracts. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019. 7

Val08. Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Proceedings of the 5th Conference on Theory of Cryptography*, TCC'08, page 1–18, Berlin, Heidelberg, 2008. Springer-Verlag. 36

W$^+$14. Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014. 3

A Cryptographic Primitives: Definitions and Security Properties

A.1 Commitment Scheme

A commitment scheme $\mathsf{COM} = (\mathsf{COM.Setup}, \mathsf{COM.Commit}, \mathsf{COM.Open})$ is a triple of efficient algorithms where:

- $\mathsf{pp}_{\mathsf{COM}} \xleftarrow{\$} \mathsf{COM.Setup}(1^\lambda)$ generates a public parameter given the security parameter;
- $\mathsf{cm} \leftarrow \mathsf{COM.Commit}(\mathsf{pp}_{\mathsf{COM}}, m; r)$ produces a commitment cm given the message from a message space to be committed ($m \in \mathcal{M}_{\mathsf{pp}_{\mathsf{COM}}}$), and an explicit randomness $r \xleftarrow{\$} \mathcal{R}_{\mathsf{pp}_{\mathsf{COM}}}$ from the randomness space;
- $b \leftarrow \mathsf{COM.Open}(\mathsf{pp}_{\mathsf{COM}}, \mathsf{cm}, m, r)$ checks whether (m, r) is an *opening* of the commitment cm, and outputs a bit $b \in \{0, 1\}$ representing accept if $b = 1$, and reject otherwise.

Informally, a commitment scheme is called **binding** if once a message is committed, it is infeasible to later open to a different message; and it is called **hiding** if the commitments of any two messages are indistinguishable from one another.

Formally, COM is:

- **Computationally Binding** if for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that:

$$\Pr\left[\begin{array}{c|l} b_0 = b_1 \neq 0 & \mathsf{pp_{COM}} \leftarrow \mathsf{COM.Setup}(1^\lambda) \\ \wedge\, x_0 \neq x_1 & (\mathsf{cm}, x_0, x_1, r_0, r_1) \leftarrow \mathcal{A}(\mathsf{pp_{COM}}) \\ & b_0 \leftarrow \mathsf{COM.Open}(\mathsf{pp_{COM}}, \mathsf{cm}, x_0, r_0) \\ & b_1 \leftarrow \mathsf{COM.Open}(\mathsf{pp_{COM}}, \mathsf{cm}, x_1, r_1) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

  if $\mathsf{negl}(\lambda) = 0$, then we say the scheme is perfectly binding.
- **Statistically Hiding** if for all unbounded adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that:

$$\left|\Pr\left[\begin{array}{c|c} & \mathsf{pp_{COM}} \leftarrow \mathsf{COM.Setup}(1^\lambda); \\ & b \xleftarrow{\$} \{0,1\}, r \xleftarrow{\$} \mathcal{R}_{\mathsf{pp_{COM}}}, \\ b = \hat{b} & (x_0, x_1) \leftarrow \mathcal{A}(\mathsf{pp_{COM}}) \\ & \mathsf{cm} \leftarrow \mathsf{COM.Commit}(\mathsf{pp_{COM}}, x_b; r) \\ & \hat{b} \leftarrow \mathcal{A}(\mathsf{pp_{COM}}, \mathsf{cm}) \end{array}\right] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$$

  if $\mathsf{negl}(\lambda) = 0$, then we say the scheme is perfectly hiding.

A.2 Polynomial Commitment Scheme

Introduced in [KZG10], *Polynomial Commitment Schemes* (PCS) enables a prover to commit to a polynomial $f \in \mathbb{F}[X]$, and later open the commitment c at any point $z \in \mathbb{F}$ by producing an *evaluation proof* π attesting that "the opened value is consistent with committed polynomial and $f(z) = y$". A polynomial commitment scheme is a tuple of algorithms $\mathsf{PCS} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Eval})$ where $(\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open})$ is a binding commitment scheme for a message space $\mathbb{F}[X]$ of polynomials over a finite field $\mathbb{F}$, and:

- $(\perp, b) \leftarrow \mathsf{PCS.Eval}(\mathcal{P}(\mathsf{pp_{PCS}}, \boldsymbol{f}, \boldsymbol{r}), \mathcal{V}(\mathsf{pp_{PCS}}, \mathbf{cm}, \boldsymbol{z}, \boldsymbol{y}))$ is a public-coin interactive protocol between the prover $\mathcal{P}$ who has a list of polynomials and opening hints $\{f_i, r_i\}_{i=1}^n$, where $f_i \in \mathbb{F}^{<d}[X]$; and the verifier $\mathcal{V}$ who has the common input $\mathsf{pp_{PCS}}$ and a list of commitments, evaluation points and their evaluations $\{\mathsf{cm}_i, z_i, y_i\}_{i=1}^n$ where $(\mathsf{cm}_i, z_i, y_i) \in \mathbb{G} \times \mathbb{F}^2$. The verifier outputs $b \in \{0,1\}$ and the prover has no output. The purpose of the protocol is to convince the verifier that for $\forall i \in [n]$, $f_i(z_i) = y_i$ and $\deg(f_i) < d$.

  A PCS is **correct** if for all degree bound $d \in \mathbb{N}$ and efficient adversaries $\mathcal{A}$:

$$\Pr\left[\begin{array}{c|l} & \mathsf{pp_{PCS}} \leftarrow \mathsf{PCS.Setup}(1^\lambda, d) \\ & (d, \boldsymbol{f}, \boldsymbol{r}, \boldsymbol{z}) \leftarrow \mathcal{A}(\mathsf{pp_{PCS}}) \\ & \text{For } i \in [n]: \\ \{b_{1,i}\}_{i=1}^n & \quad \mathsf{cm}_i \leftarrow \mathsf{PCS.Commit}(\mathsf{pp_{PCS}}, f_i; r_i) \\ \wedge\, b_2 = 1 & \quad b_{1,i} \leftarrow \mathsf{PCS.Open}(\mathsf{pp_{PCS}}, \mathsf{cm}_i, f_i, r_i) \\ & \quad y_i \leftarrow f_i(z_i) \\ & (\perp, b_2) \leftarrow \mathsf{PCS.Eval}\begin{pmatrix} \mathcal{P}(\mathsf{pp_{PCS}}, \boldsymbol{f}, \boldsymbol{r}), \\ \mathcal{V}(\mathsf{pp_{PCS}}, \mathbf{cm}, \boldsymbol{z}, \boldsymbol{y}) \end{pmatrix} \end{array}\right] = 1$$

A PCS has **knowledge soundness** if $\mathsf{PCS.Eval}$ has knowledge soundness as an interactive argument for $\mathcal{R}_{\mathsf{Eval}}(\mathsf{pp}_{\mathsf{PCS}})$:

$$\mathcal{R}_{\mathsf{Eval}}(\mathsf{pp}_{\mathsf{PCS}}) = \left\{ \begin{array}{l} \quad (\mathbb{x} = (\mathbf{cm}, \boldsymbol{z}, \boldsymbol{y}, d), \mathbb{w} = (\boldsymbol{f}, \boldsymbol{r})) : \\ \text{For } i \in [n] : \\ \quad f_i \in \mathbb{F}[x] \ \wedge \ \deg(f_i) < d \\ \quad \wedge \ f_i(z_i) = y_i \\ \quad \wedge \ \mathsf{PCS.Open}(\mathsf{pp}_{\mathsf{PCS}}, \mathsf{cm}_i, f_i, r_i) = 1 \end{array} \right\}$$

Linearly Additive Homomorphism. A PCS is *linearly additively homomorphic* if it holds the following property: let $[C_i]_{i=1}^n$ commit to $[f_i]_{i=1}^n$, then $\sum_{i=1}^n a_i \circ C_i$ commits to $\sum_{i=1}^n a_i \cdot f_i$ for any $a_i \in \mathbb{F}$. Here, arithmetics operations for f_i are over $\mathbb{F}[X]$; and $\circ$ is the addition over the commitment space (*e.g.* it is the group addition in [KZG10]).

A.3 Indexed Relation

We define an *indexed relation* $\mathcal{R}$ as a set of $(\mathbb{i}, \mathbb{x}, \mathbb{w})$, where $\mathbb{i}$ is the index that describes the circuit; $\mathbb{x}$ consists of the (public) instances that holds the assignments to a subset of wires; and $\mathbb{w}$ is the witness that holds the assignments to the remaining wires in the circuit. The corresponding *indexed language* is defined as: $\mathcal{L}(\mathcal{R}) := \{(\mathbb{i}, \mathbb{x}) : \exists \mathbb{w} \text{ s.t. } (\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}\}$. We further denote $\mathcal{R}_N$ for a relation with an upper-bounded circuit $|\mathbb{i}| < N$ where $N \in \mathbb{N}$ is the size bound. When there is no ambiguity, we use $\mathbb{i} = \Phi$ to represent the indexing of circuit for the relation: $\mathcal{R}_\Phi := \{(\mathbb{x}, \mathbb{w}) : \Phi(\mathbb{x}, \mathbb{w}) = 1\}$; and refer to Φ as a *predicate*.

A.4 Pre-processing SNARK with Universal SRS

A (pre-processing) *non-interactive argument of knowledge* (NARK) is a tuple of efficient algorithms $\mathsf{NARK} = (\mathcal{G}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ where:

- $\mathsf{srs} \leftarrow \mathsf{NARK}.\mathcal{G}(\lambda, N)$ is a probabilistic algorithm that generates a structured reference string srs from the security parameter λ and a size bound N for the circuit.
- $(\mathsf{ipk}, \mathsf{ivk}) \leftarrow \mathsf{NARK}.\mathcal{I}^{\mathsf{srs}}(\mathbb{i})$ is a *deterministic* algorithm that, given a circuit description $\mathbb{i}$ and oracle access to srs, generates an index proving key ipk and index verifying key ivk for this particular circuit.
- $\pi \leftarrow \mathsf{NARK}.\mathcal{P}(\mathsf{ipk}, \mathbb{x}, \mathbb{w})$ is a probabilistic prover algorithm that given an index proving key ipk corresponding to some relation $\mathcal{R}_\Phi$, an instance $\mathbb{x}$, and a witness $\mathbb{w}$, returns a NARK proof π.
- $b \leftarrow \mathsf{NARK}.\mathcal{V}(\mathsf{ivk}, \mathbb{x}, \pi)$ is a verifier algorithm that given the index verifying key ivk, the instance $\mathbb{x}$, and the proof π, outputs a bit b where $b = 1$ indicates successful verification, $b = 0$ otherwise.

A NARK scheme $\mathsf{NARK} = (\mathcal{G}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ for relation $\mathcal{R}_\Phi$ needs to the following properties to hold:

- **Completeness.** For all size bound $N \in \mathbb{N}$, all adversaries $\mathcal{A}$:

$$\Pr\left[\begin{array}{c} (\mathbb{x}, \mathbb{w}) \notin \mathcal{R}_\Phi \\ \vee \\ \mathsf{NARK}.\mathcal{V}(\mathsf{ivk}, \mathbb{x}, \pi) = 1 \end{array} \middle| \begin{array}{r} \mathsf{srs} \xleftarrow{\$} \mathsf{NARK}.\mathcal{G}(\lambda, N) \\ (\Phi, \mathbb{x}, \mathbb{w}) \leftarrow \mathcal{A}(\mathsf{srs}) \\ (\mathsf{ivk}, \mathsf{ipk}) \xleftarrow{\$} \mathsf{NARK}.\mathcal{I}^{\mathsf{srs}}(\Phi) \\ \pi \xleftarrow{\$} \mathsf{NARK}.\mathcal{P}(\mathsf{ipk}, \mathbb{x}, \mathbb{w}) \end{array} \right] = 1$$

- **Adaptive Knowledge Soundness.** For all $N \in \mathbb{N}$, all efficient adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ with state st, there exists a knowledge extractor $\mathcal{E}^{\mathcal{A}}$ with oracle access to $\mathcal{A}$, such that:

$$\Pr\left[\begin{array}{c|c} & \mathsf{srs} \xleftarrow{\$} \mathsf{NARK}.\mathcal{G}(\lambda, N) \\ (\mathbb{x}, \mathbb{w}) \notin \mathcal{R}_{\varPhi} & (\varPhi, \mathbb{x}, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{srs}) \\ \wedge & (\mathsf{ivk}, \mathsf{ipk}) \xleftarrow{\$} \mathsf{NARK}.\mathcal{I}^{\mathsf{srs}}(\varPhi) \\ \mathsf{NARK}.\mathcal{V}(\mathsf{ivk}, \mathbb{x}, \pi) = 1 & \pi \leftarrow \mathcal{A}_2(\mathsf{st}) \\ & \mathbb{w} \leftarrow \mathcal{E}^{\mathcal{A}}(\mathsf{srs}, \varPhi, \mathbb{x}, \mathsf{ipk}, \mathsf{ivk}) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

A NARK scheme can additionally satisfy the following optional properties:

- **Statistical Zero-knowledge.** For all $N \in \mathbb{N}$, all *unbounded* adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists an efficient simulator[24] $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$:

$$\left|\begin{array}{c} \Pr\left[\begin{array}{c|c} (\mathbb{x}, \mathbb{w}) \in \mathcal{R}_{\varPhi} & \mathsf{srs} \xleftarrow{\$} \mathsf{NARK}.\mathcal{G}(1^\lambda, N) \\ \wedge & (\varPhi, \mathbb{x}, \mathbb{w}, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{srs}) \\ \mathcal{A}_2(\mathsf{st}, \pi) = 1 & (\mathsf{ivk}, \mathsf{ipk}) \xleftarrow{\$} \mathsf{NARK}.\mathcal{I}^{\mathsf{srs}}(\varPhi) \\ & \pi \leftarrow \mathsf{NARK}.\mathcal{P}(\mathsf{ipk}, \mathbb{x}, \mathbb{w}) \end{array}\right] \\ - \Pr\left[\begin{array}{c|c} (\mathbb{x}, \mathbb{w}) \in \mathcal{R}_{\varPhi} & (\mathsf{srs}, \tau) \xleftarrow{\$} \mathcal{S}_1(1^\lambda, N) \\ \wedge & (\varPhi, \mathbb{x}, \mathbb{w}, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{srs}) \\ \mathcal{A}_2(\mathsf{st}, \pi) = 1 & \pi \leftarrow \mathcal{S}_2(\tau, \varPhi, \mathbb{x}) \end{array}\right] \end{array}\right| \leq \mathsf{negl}(\lambda)$$

- **Succinctness.** A NARK scheme is said *succinct* (and thus denoted as SNARK) if there exists a universal polynomial poly (independent of relation $\mathcal{R}_{\varPhi}$) such that:
  - The indexer algorithm $\mathsf{NARK}.\mathcal{I}^{\mathsf{srs}}$ runs in $\mathsf{poly}_\lambda(|\varPhi|)$ time, namely, it is polynomial in the circuit size and independent from the public parameter srs size.
  - The proof size $|\pi|$ is bounded by $\mathsf{poly}(\lambda)$.
  - The verifier $\mathsf{NARK}.\mathcal{V}$ runs in $\mathsf{poly}(\lambda + |\mathbb{x}|)$ time. Particularly if is independent from the size of the predicate $\varPhi$ (equivalently the circuit $\mathbb{i}$).

Furthermore, when the structured referenced string srs is independent from all subsequent relations $\mathcal{R}_N$, we refer to these NARK as *universal* since their SRS can be universally applied to all relations of a bounded size N. Universal (S)NARKs are of tremendous interest because they obviate the requirement of a "circuit-specific" srs and thus a dedicated setup ceremony for each relation. Usually SNARK with universal srs are also *updateable* [GKM+18], allowing anyone to efficiently update the SRS thus reducing the trust assumption on the setup ceremony during $\mathsf{NARK}.\mathcal{G}$.

A.5 Incrementally Verifiable Computation (IVC)

The notion of *incrementally verifiable computation* (IVC), introduced by Valiant [Val08], describes a machine that outputs the updated state in each step of computation, along with a proof attesting the correctness of all historical computation steps. As shown Fig. 10, an IVC starts with an initial state z_0 and takes t steps to compute the function $F_0 \circ F_1 \circ \ldots \circ F_{t-1}$ and outputs the final state z_t. In each step, an IVC prover takes in the state z_i, some optional witness w_i, and integrity proof π_i from the last step, apply the computation F_i and outputs the new state z_{i+1} together with a new proof attesting correctness of π_i *and* correctness of state transitions $z_{i+1} = F_i(z_i)$. An IVC verifier can "jump in" at any step $i \in [t]$, verify the (z_i, π_i), and be convinced that the state is correct since all historical computation are incrementally verified.

[24] We change a bit the syntax of the $\mathsf{NARK}.\mathcal{G}$ algorithm in order to return the trapdoor τ.

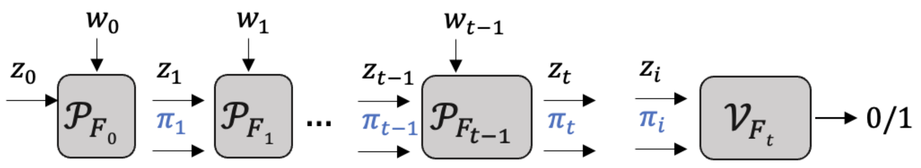


Fig. 10: Incrementally Verifiable Computation.

For simpler presentation and W.L.O.G., we assume that $F = F_0 = \ldots = F_{t-1}$. An IVC for a step function $F : \mathcal{X} \mapsto \mathcal{X}$ is a tuple of efficient algorithms $\mathsf{IVC} = (\mathcal{P}_F, \mathcal{V}_F)$ that follows these properties:

- **Completeness.** For all inputs $z \in \mathcal{X}$, witnesses w and proofs π:
$$\Pr\left[\, z' = F(z, w) \wedge \mathcal{V}_F(z', \pi') = 1 \,\middle|\, (z', \pi') \leftarrow \mathcal{P}_F(z, w, \pi) \,\right] = 1$$
- **Knowledge Soundness.** For all efficient adversaries $\mathcal{A}$, there exists an efficient knowledge extractor $\mathcal{E}_\mathcal{A}$, such that:
$$\Pr\left[\begin{array}{c} \mathcal{V}_F(z_t, \pi) = 0 \\ \vee \\ \forall i \in [t-1], z_{i+1} = F(z_i, w_i) \end{array} \middle| \begin{array}{c} (z_t, \pi) \leftarrow \mathcal{A} \\ \{z_i, w_i\}_{i=1}^{t-1} \leftarrow \mathcal{E}_\mathcal{A} \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

B Plonk, TurboPlonk, UltraPlonk Constraint Systems

A PLONK (and its variants) constraint system over a finite field $\mathbb{F}$ consists of many *gates*, each of which has a predefined number of *wires* where each wire is to be assigned with a value in the *witness vector*. Each gate implies an algebraic relation among all wire values and the exact relation is configurable via some *selectors* to collectively select the exact function applied, and a *public input wire* to be assigned with values of the public input of an NP relation. Value assignments for all wires are described using an *index vector* that connects each wire with a specific value in the witness vector. For an NP relation expressed in this constraint system, the index vector, the selectors and the field $\mathbb{F}$ constitute the circuit description. Such constraint system is satisfied if and only if some "local constraints" (*i.e.* algebraic functions at each gate) are fulfilled *and* some "regional/global constraints" across different/all gates (*e.g.* all wire values respect the index vector connection) are fulfilled.

We denote n, m, ℓ the number of gates, length of witness vector, and number of public inputs respectively. We adapt definitions of PLONK and its variants to indexed relations (defined in Appx. A.3) below.

Definition 1 (Plonk indexed relation). *The indexed relation $\mathcal{R}_{\mathsf{plonk}}$ is the set of all triples:*
$$\left(\mathbb{i} = (\mathbb{F}, n, m, \ell, \boldsymbol{a}, \mathcal{Q}), \mathbb{x} = (w_j)_{j\in[\ell]}, \mathbb{w} = (w_j)_{j\in[\ell+1,m]}\right)$$
where the index vector $\boldsymbol{a} \in [m]^{3n}$, selectors are $\mathcal{Q} := (\boldsymbol{q}_L, \boldsymbol{q}_R, \boldsymbol{q}_O, \boldsymbol{q}_M, \boldsymbol{q}_C) \in (\mathbb{F}^n)^5$, such that $\forall i \in [n]$,
$$(\boldsymbol{q}_L)_i \cdot w_{\boldsymbol{a}_i} + (\boldsymbol{q}_R)_i \cdot w_{\boldsymbol{a}_{n+i}} + (\boldsymbol{q}_M)_i \cdot w_{\boldsymbol{a}_i} w_{\boldsymbol{a}_{n+i}} + (\boldsymbol{q}_C)_i + \mathsf{PI}_i = (\boldsymbol{q}_O)_i \cdot w_{\boldsymbol{a}_{2n+i}}$$
where $\mathsf{PI}_i = w_i$ for $i \in [\ell]$ and $\mathsf{PI}_i = 0$ for $i \in [\ell+1, n]$.

Next, we propose a TURBO-PLONK constraint system that allows for customized gates beyond just addition and multiplication gate. However, we note that TURBO-PLONK proof system has higher per-gate cost for proof generation and higher fan-in resulting in slightly larger proof size and more polynomial to interpolate during proving. Fortunately, our design is extremely efficient for NP relations that involve heavy Rescue computation (*e.g.* Merkle proof verification in a Merkle tree instantiated with Rescue hash) and elliptic curve operations, since the total constraints required is significantly reduced, the overall efficiency will be improved.

Definition 2 (TurboPlonk indexed relation). *The indexed relation $\mathcal{R}_{\mathsf{tplonk}}$ is the set of all triples:*

$$\Big(\mathbb{i} = (\mathbb{F}, n, m, \ell, \boldsymbol{a}, \mathcal{Q}), \mathbb{x} = (w_j)_{j\in[\ell]}, \mathbb{w} = (w_j)_{j\in[\ell+1,m]}\Big)$$

where the index vector $\boldsymbol{a} \in [m]^{5n}$, selectors are $\mathcal{Q} := (\boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3, \boldsymbol{q}_4, \boldsymbol{q}_{M_{1,2}}, \boldsymbol{q}_{M_{3,4}}, \boldsymbol{q}_O, \boldsymbol{q}_C, \boldsymbol{q}_{H_1}, \boldsymbol{q}_{H_2}, \boldsymbol{q}_{H_3}, \boldsymbol{q}_{H_4}, \boldsymbol{q}_{ecc}) \in \mathbb{F}^{n\times 13}$, such that $\forall i \in [n]$,

$$\begin{aligned}
(\boldsymbol{q}_O)_i \cdot w_{\boldsymbol{a}_{4n+i}} = {} & (\boldsymbol{q}_1)_i \cdot w_{\boldsymbol{a}_i} + (\boldsymbol{q}_2)_i \cdot w_{\boldsymbol{a}_{n+i}} \\
& + (\boldsymbol{q}_3)_i \cdot w_{\boldsymbol{a}_{2n+i}} + (\boldsymbol{q}_4)_i \cdot w_{\boldsymbol{a}_{3n+i}} \\
& + (\boldsymbol{q}_{M_{1,2}})_i \cdot w_{\boldsymbol{a}_i} w_{\boldsymbol{a}_{n+i}} + (\boldsymbol{q}_{M_{3,4}})_i \cdot w_{\boldsymbol{a}_{2n+i}} w_{\boldsymbol{a}_{3n+i}} \\
& + (\boldsymbol{q}_{H_1})_i \cdot w^5_{\boldsymbol{a}_i} + (\boldsymbol{q}_{H_2})_i \cdot w^5_{\boldsymbol{a}_{n+i}} \\
& + (\boldsymbol{q}_{H_3})_i \cdot w^5_{\boldsymbol{a}_{2n+i}} + (\boldsymbol{q}_{H_4})_i \cdot w^5_{\boldsymbol{a}_{3n+i}} \\
& + (\boldsymbol{q}_{ecc})_i \cdot w_{\boldsymbol{a}_i} w_{\boldsymbol{a}_{n+i}} w_{\boldsymbol{a}_{2n+i}} w_{\boldsymbol{a}_{3n+i}} w_{\boldsymbol{a}_{4n+i}} \\
& + (\boldsymbol{q}_C)_i + \mathsf{PI}_i
\end{aligned}$$

where $\mathsf{PI}_i = w_i$ for $i \in [\ell]$ and $\mathsf{PI}_i = 0$ for $i \in [\ell + 1, n]$.

Furthermore, to minimize the number of gates used for range proofs and multi-scalar multiplications, we integrate the techniques from Plookup [GW20] with the previous TURBO-PLONK constraint system and propose a customized *UltraPlonk constraint system*. The system is mainly used for outer-layer circuits, where we need to simulate non-native field arithmetics (whose circuit is dominated by range proofs), as well as the Pippenger-based multi-scalar multiplications (which requires lookup over online key-value tables). The UltraPlonk constraint system extends TurboPlonk by further introducing the following:

- To enable efficient range proofs, it introduces a preprocessed range table $\mathcal{T}_{\mathsf{rg}} \in \mathbb{F}^n$, an additional wire to each gate, and an index vector $\boldsymbol{a}_{\mathsf{rg}} \in [m]^n$, such that for each $i \in [n]$, the witness value $w_{(\boldsymbol{a}_{\mathsf{rg}})_i}$ is in the range table $\mathcal{T}_{\mathsf{rg}}$.
- To support multiple online lookup tables[25], each containing key-value tuples where the "keys" are scalars and "values" are affine point variables (*i.e.* two variables for the x and y coordinates)[26], it introduces the following:
    1. A merged, preprocessed table $\mathcal{T}_{key} \in \mathbb{F}^n$ containing predefined "keys" in the key-value entries across all sub-tables.
    2. A lookup selector $\boldsymbol{q}_K \in \mathbb{F}^n$ to indicate whether a gate is performing online table entry insertion and query table insertion.
    3. Two domain separator selectors $\boldsymbol{q}_{lt}, \boldsymbol{q}_{qt}$ for indicating the exact lookup sub-table and query sub-table an entry in the final merged table belongs to.

[25] We merged multiple sub-tables into a single one by adding additional column for table index in both the online lookup sub-tables and online query sub-tables, which results in two additional "domain separator" selectors.

[26] The "values" type here is a pair of variables, but we can easily support "value" type of a single variable by filling the other one with zero variable.

More precisely, the i-th entry in our merged online lookup table is a key-value tuple $\mathcal{T}_i := (\boldsymbol{q}_K)_i \cdot [(\boldsymbol{q}_{lt})_i, (\mathcal{T}_{key})_i, w_{\boldsymbol{a}_{3n+i}}, w_{\boldsymbol{a}_{4n+i}}]$; the i-th entry in our merged online query table is a key-value tuple $\mathcal{Q}_i := (\boldsymbol{q}_K)_i \cdot \left[(\boldsymbol{q}_{qt})_i, w_{\boldsymbol{a}_i}, w_{\boldsymbol{a}_{n+i}}, w_{\boldsymbol{a}_{2n+i}}\right]$. The witness vector and index vector should satisfy that $\forall i \in [n], \mathcal{Q}_i \in \mathcal{T} := (\mathcal{T}_j)_{j\in[n]}$.

Definition 3 (UltraPlonk indexed relation). *The indexed relation* $\mathcal{R}_{\mathsf{uplonk}}$ *is the set of all triples* $(\mathbb{i}, \mathbb{x}, \mathbb{w})$ *where*

$$\mathbb{i} = (\mathbb{F}, n, m, \ell, \boldsymbol{a}, \boldsymbol{a}_{\mathsf{rg}}, \mathcal{Q}, \mathcal{T}_{\mathsf{rg}}, \mathcal{T}_{key}), \quad \mathbb{x} = (w_j)_{j\in[\ell]}, \quad \mathbb{w} = (w_j)_{j\in[\ell+1,m]}$$

where the TurboPlonk index vector $\boldsymbol{a} \in [m]^{5n}$*, the index vector for the range wire:* $\boldsymbol{a}_{\mathsf{rg}} \in [m]^n$*, selectors are:* $\mathcal{Q} := (\boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3, \boldsymbol{q}_4, \boldsymbol{q}_{M_{1,2}}, \boldsymbol{q}_{M_{3,4}}, \boldsymbol{q}_O, \boldsymbol{q}_C, \boldsymbol{q}_{H_1}, \boldsymbol{q}_{H_2}, \boldsymbol{q}_{H_3}, \boldsymbol{q}_{H_4}, \boldsymbol{q}_{ecc}, \boldsymbol{q}_K, \boldsymbol{q}_{lt}, \boldsymbol{q}_{qt}) \in \mathbb{F}^{n\times 16}$*, such that:*

1. $((\mathbb{F}, n, m, \ell, \boldsymbol{a}, \mathcal{Q}), \mathbb{x}, \mathbb{w}) \in \mathcal{R}_{\mathsf{tplonk}}$.
2. $\forall i \in [n]$, $w_{(\boldsymbol{a}_{\mathsf{rg}})_i} \in \mathcal{T}_{\mathsf{rg}}$.
3. $\forall i \in [n]$*, the query key-value tuple*

$$\mathcal{Q}_i := (\boldsymbol{q}_K)_i \cdot \left[(\boldsymbol{q}_{qt})_i, w_{\boldsymbol{a}_i}, w_{\boldsymbol{a}_{n+i}}, w_{\boldsymbol{a}_{2n+i}}\right]$$

is in the lookup table

$$\mathcal{T} := \left\{\mathcal{T}_j = (\boldsymbol{q}_K)_j \cdot \left[(\boldsymbol{q}_{lt})_j, (\mathcal{T}_{key})_j, w_{\boldsymbol{a}_{3n+j}}, w_{\boldsymbol{a}_{4n+j}}\right]\right\}_{j\in[n]}$$

Here $a \cdot \boldsymbol{b}$ *denotes the element-wise multiplications between scalar* a *and vector* $\boldsymbol{b}$*.*

C UltraPlonk Proof Systems

We present a Polynomial IOP for the UltraPlonk indexed relation $\mathcal{R}_{\mathsf{uplonk}}$ (Def. 3) in Fig. 11 and 12 and follow notations similar to [GWC19] and [GW20].

Let $\mathbb{F}$ be a finite field of prime order p, $H \subset \mathbb{F}^*$ the multiplicative subgroup containing the n-th roots of unity where ω is the generator of the subgroup, namely $H := \{1, \omega, \omega^2, \ldots, \omega^{n-1}\}$. The Lagrange polynomial $\mathcal{L}_i(X) \in \mathbb{F}^{<n}[X]$ over H is defined as $\mathcal{L}_i(X) = \frac{\omega^i(X^n-1)}{n(X-\omega^i)}$ so that $\mathcal{L}_i(x) = 1$ when $x = \omega^i$ and $\mathcal{L}_i(x) = 0$ elsewhere. The vanishing polynomial $Z_H(X)$ over H is defined as $Z_H(X) = (X-1)(X-\omega)\ldots(X-\omega^{n-1}) = X^n - 1$ so that $\forall x \in H, Z_H(x) = 0$.

C.1 Witness transformation

Given an UltraPlonk indexed relation $\mathcal{R}_{\mathsf{uplonk}} := \left(\mathbb{i}, \mathbb{x} = (w_j)_{j\in[\ell]}, \mathbb{w} = (w_j)_{j\in[\ell+1,m]}\right)$ with $\mathbb{i} = (\mathbb{F}, n, m, \ell, \boldsymbol{a}, \boldsymbol{a}_{\mathsf{rg}}, \mathcal{Q}, \mathcal{T}_{\mathsf{rg}}, \mathcal{T}_{key})$, we show how to transform the index vector $\boldsymbol{a'} = (\boldsymbol{a}, \boldsymbol{a}_{\mathsf{rg}}) \in [m]^{5n} \times [m]^n$ into a permutation: $\sigma : [6n] \to [6n]$ and transform $\mathcal{R}_{\mathsf{uplonk}}$ into an equivalent relation: $\mathcal{R}'_{\mathsf{uplonk}} := \left(\mathbb{i}, \mathbb{x} = (w'_i)_{i\in[\ell]}, \mathbb{w} = (w'_i)_{i\in[\ell+1,6n]}\right)$ with $\mathbb{i} = (\mathbb{F}, n, m, \ell, \sigma, \mathcal{Q}, \mathcal{T}_{\mathsf{rg}}, \mathcal{T}_{key})$.

1. Define a partition $\mathcal{P}_1, \ldots, \mathcal{P}_m$ corresponding to m values in the witness vector, such that for each $j \in [m]$: $\mathcal{P}_j = \{i \in [6n] : \boldsymbol{a'_i} = w_j\}$. Intuitively, $\mathcal{P}_i$ is the set of gate wire identifiers whose assignments maps to the same witness value w_j.
2. Define a permutation $\sigma : [6n] \to [6n]$ such that for each $j \in [m]$, the restriction of σ on input $\mathcal{P}_j$ forms a cycle going over all elements in $\mathcal{P}_j$.
3. Define a new instance-witness vector $(w'_i)_{i\in[6n]}$ such that for each $j \in [m]$ and each $i \in \mathcal{P}_j$, we have $w'_i = w_j$. It is easy to see that this *copy constraint* holds if and only if $\forall i \in [6n]$, $w'_i = w'_{\sigma(i)}$.

C.2 Polynomial Interpolation

During arithmetization, we turn relations among some vectors indexed by gates into relations among some polynomials indexed by elements in a multiplicative subgroup H – a process that involves polynomial interpolations. For vectors $v = (v_i)_{i\in[n]}$ of size n, the same as the subgroup order, we interpolate using Lagrange polynomial and expressed as $p_v(X) = \sum_{i\in[n]} \mathcal{L}_i(X) \cdot v_i \in \mathbb{F}^{<n}[X]$. In practice, one should use (Inverse) Fast Fourier Transform (IFFT/FFT) to efficiently compute the coefficients of $p_v(X)$ from the data points $(v_i)_{i\in[n]}$. For vectors of size kn for some $k \in \mathbb{N}$, we use k polynomials over k non-overlapping cosets of H to interpolate all data points. In our case, we need to find $\{k_i\}_{i\in[6]}$ such that $k_1H, \dots, k_6H$ are disjoint cosets of H in order to interpolate polynomials from a vector of size $6n$. We choose the cosets using the following algorithm: let $N \geq |H|$ be a global constant that is a multiple of $|H|$. We pick $k_1 = 1$ and pick random $k_2, \dots k_6$ such that $(k_j^{-1} \cdot k_i)^N \neq 1$ for every $i, j \in [6], i \neq j$ because every elements $x \in H$ satisfies $x^N = 1$, and that: $aH = bH$ if and only if $a^{-1} \cdot b \in H$.

D Plonk with Merging, Batching and Accumulation

We present a scheme Plonk' based on the vanilla PLONK in Def. 1 and that incorporates instance merging (§ 3.2), proof batching (§ 3.3), and lightweight verifier via accumulation scheme technique (§ 3.1). The presentation can be easily adapt to TURBO-PLONK which is what we use for inner predicate circuits in ZEXE since it minimize the circuit complexity of the outer proof. Details of $\mathsf{Plonk}' = (\mathsf{Setup}, \mathsf{Index}, \mathsf{MergePK}, \mathsf{MergeVK}, \mathsf{MergeWit}, \mathsf{BatchProve}, \mathsf{BatchPartialVfy}, \mathsf{Decide})$ are shown below.

For generating a n_{pf}-input-n_{pf}-output transaction via $\mathsf{DPC.Execute}^{\mathsf{L}}$, the user will take n_{pf} pair of input death predicate and output birth predicate (Φ_d, Φ_b) and preprocess them via $\mathsf{Plonk}'.\mathsf{Index}$ to get proving keys and verification keys. The user then pass n_{pf} pair of proving keys, verification keys and witnesses into $\mathsf{MergePK}, \mathsf{MergeVK}, \mathsf{MergeWit}$ to get n_{pf} number of merged keys/witnesses. Subsequently, the user generate a batched inner predicate proof for all $2n_{\mathsf{pf}}$ predicates via $\mathsf{BatchProve}$ whose correctness will be checked inside the outer circuit that embeds the logic in $\mathsf{BatchPartialVfy}$. Finally, a ledger maintainer will check the outer proof and run Decide on the accumulator outputted by $\mathsf{BatchPartialVfy}$ to determine validity of the transaction.

$\underline{\mathsf{Setup}(\lambda, N) \to \mathsf{srs}}$

- *inputs:* security parameter λ, and upper bound for merged-instance circuit $N = 2n$ (where n is the size bound for a single instance)
- *outputs:* public parameter srs

1. Run Setup of the [KZG10] PCS to get $\mathsf{srs} := ([1]_1, [x]_1, \dots, [x^{N+5}], [1]_2, [x]_2)$.

$\underline{\mathsf{Index}(\mathsf{srs}, \Phi, b) \to (\mathsf{ipk}, \mathsf{ivk})}$

- *inputs:*
  - public parameter srs
  - predicate Φ (whose circuit size is bounded by n)
  - bit b indicating whether Φ is a birth or death predicate
- *outputs:* circuit proving key ipk and verification key ivk

1. Compute all selectors $\mathcal{Q} \in (\mathbb{F}^n)^5$ and the wire permutation $\sigma : [3n] \mapsto [3n]$ from predicate Φ (similar to the process in Appx. C.1).

Preprocessing Phase:
The indexer $\mathcal{I}$ takes as input an indexed instance $\mathbb{i} = (\mathbb{F}, n, m, \ell, \sigma, \mathcal{Q}, \mathcal{T}_{\mathsf{rg}}, \mathcal{T}_{key})$ and outputs the following polynomial oracles:

- The selector polynomials that interpolates selector vectors in $\mathcal{Q}$:

$$\boldsymbol{q}_1(X) = \sum_{i\in[n]} \mathcal{L}_i(X) \cdot (\boldsymbol{q}_1)_i \,, \ldots, \boldsymbol{q}_{qt}(X) = \sum_{i\in[n]} \mathcal{L}_i(X) \cdot (\boldsymbol{q}_{qt})_i \,.$$

- Define $\sigma^* := \sigma \circ f_\sigma : [6n] \mapsto k_1 H \cup \ldots \cup k_6 H$, where $f_\sigma := i \mapsto k_1 \cdot \omega^i, \ldots, 5n+i \mapsto k_6 \cdot \omega^i \, \forall i \in [n]$. The identity polynomials $S_{\mathsf{ID}j}(X) = k_j X$ for each $j \in [6]$, and the permutation polynomials $S_{\sigma j}(X) = \sum_{i\in[n]} \sigma^*((j-1)n+i) \cdot \mathcal{L}_i(X)$ that encodes σ^* for each $j \in [6]$.
- The preprocessed table polynomials

$$\mathcal{T}_{\mathsf{rg}}(X) = \sum_{i\in[n]} \mathcal{L}_i(X) \cdot (\mathcal{T}_{\mathsf{rg}})_i \,,\ \mathcal{T}_{key}(X) = \sum_{i\in[n]} \mathcal{L}_i(X) \cdot (\mathcal{T}_{key})_i \,.$$

Prover Inputs: The prover $\mathcal{P}$ takes as input the indexed relation $\mathbb{i}$, the online public instance $\mathbb{x} = (w_i)_{i\in[\ell]}$, and the online witness $\mathbb{w} = (w_i)_{i\in[\ell+1,6n]}$. (WLOG we assume $w_{6n} = 0, (\boldsymbol{q}_K)_n = 0$.)
Verifier Inputs: The verifier $\mathcal{V}$ takes as input $\mathbb{F}$, the online public instance $\mathbb{x} = (w_i)_{i\in[\ell]}$, and the oracle access to polynomials the indexer generates.
Online Phase:
Round 1

1. $\mathcal{P}$ computes the public input polynomial $\mathsf{PI}(X) = \sum_{i\in[\ell]} \mathcal{L}_i(X) \cdot (w)_i$.
2. $\mathcal{P}$ computes and sends $\mathcal{V}$ polynomials $f_1(X), \ldots, f_6(X)$ such that for every $i \in [6]$, $f_i(X)$ interpolates $(w_{(i-1)n+1}, \ldots, w_{in})$ over H.

Round 2

1. $\mathcal{V}$ sends $\mathcal{P}$ a random challenge $\tau \in \mathbb{F}$.
2. $\mathcal{P}$ computes the merged query polynomial:

$$\mathcal{Q}^*(X) := f_6(X) + \tau \cdot \boldsymbol{q}_K(X) \cdot (\boldsymbol{q}_{qt}(X) + \tau \cdot f_1(X) + \tau^2 \cdot f_2(X) + \tau^3 \cdot f_3(X)) \,,$$

   and the merged lookup table polynomial:

$$\mathcal{T}^*(X) := \mathcal{T}_{\mathsf{rg}}(X) + \tau \cdot \boldsymbol{q}_K(X) \cdot (\boldsymbol{q}_{lt}(X) + \tau \cdot \mathcal{T}_{key}(X) + \tau^2 \cdot f_4(X) + \tau^3 \cdot f_5(X)) \,.$$

   ($\boldsymbol{q^*} = (\mathcal{Q}^*(\omega^i))_{i\in[n]}$ is the merged query vector, $\boldsymbol{t^*} = (\mathcal{T}_*(\omega^i))_{i\in[n]}$ is the lookup table vector)
3. Let $\boldsymbol{s}$ be the vector $(\boldsymbol{q^*}, \boldsymbol{t^*})$ sorted by $\boldsymbol{t^*}$. We represent $\boldsymbol{s}$ by the vectors $\boldsymbol{h_1}, \boldsymbol{h_2} \in \mathbb{F}^n$ as follows[a]:

$$\boldsymbol{h_1} = (s_1, s_3, \ldots, s_{2n-1}), \quad \boldsymbol{h_2} = (s_2, s_4, \ldots, s_{2n})$$

   $\mathcal{P}$ computes the sorted polynomial $h_1(X), h_2(X) \in \mathbb{F}^{<n}[X]$ and sends them to $\mathcal{V}$:

$$h_1(X) = \sum_{i\in[n]} s_{2i-1} \cdot \mathcal{L}_i(X), \quad h_2(X) = \sum_{i\in[n]} s_{2i} \cdot \mathcal{L}_i(X)$$

Round 3

1. $\mathcal{V}$ sends random challenges $\beta, \gamma \in \mathbb{F}$.
2. $\mathcal{P}$ computes Plonk permutation polynomial $z_1(X)$ and Plookup permutation polynomial $z_2(X)$ and sends to $\mathcal{V}$:

$$z_1(X) = \mathcal{L}_1(X) + \sum_{i=1}^{n-1} \left(\mathcal{L}_{i+1}(X) \cdot \prod_{j=1}^{i} \prod_{\ell=1}^{6} \frac{w_{(\ell-1)n+j} + \beta k_\ell w^{j-1} + \gamma}{w_{(\ell-1)n+j} + \beta\sigma^*((\ell-1)n+j) + \gamma} \right),$$

$$z_2(X) = \mathcal{L}_1(X) + \sum_{i=1}^{n-1} \left(\mathcal{L}_{i+1}(X) \cdot \prod_{j=1}^{i} \frac{(1+\beta)(\gamma + \boldsymbol{q^*}_j)(\gamma(1+\beta) + \boldsymbol{t^*}_j + \beta \boldsymbol{t^*}_{j+1})}{(\gamma(1+\beta) + \boldsymbol{s}_{2j-1} + \beta \boldsymbol{s}_{2j})(\gamma(1+\beta) + \boldsymbol{s}_{2j} + \beta \boldsymbol{s}_{2j+1})} \right).$$

[a] We borrow the alternating method from [PFM+22] to split $\boldsymbol{s}$ into even and odd halves instead of lower and upper halves as in the original [GW20] to save one polynomial identity check used to ensure $\boldsymbol{h_1}$ and $\boldsymbol{h_2}$ are overlapped.

Fig. 11: A Polynomial IOP for the UltraPlonk constraint system: Part I.

(Online Phase) Round 4:

1. $\mathcal{V}$ computes a random challenge $\alpha \in \mathbb{F}$.
2. $\mathcal{P}$ computes some intermediate polynomials:

$$\begin{aligned}
F_{gate}(X) &= \boldsymbol{q}_1(X)f_1(X) + \boldsymbol{q}_2(X)f_2(X) + \boldsymbol{q}_3(X)f_3(X) + \boldsymbol{q}_4(X)f_x(X) \\
&\quad + \boldsymbol{q}_{M_{1,2}}(X)f_1(X)f_2(X) + \boldsymbol{q}_{M_{3,4}}(X)f_3(X)f_4(X) \\
&\quad + \boldsymbol{q}_{H_1}(X)f_1(X)^5 + \boldsymbol{q}_{H_2}(X)f_2(X)^5 + \boldsymbol{q}_{H_3}(X)f_3(X)^5 + \boldsymbol{q}_{H_4}(X)f_4(X)^5 \\
&\quad + \boldsymbol{q}_{ecc}(X)f_1(X)f_2(X)f_3(X)f_4(X)f_5(X) \\
&\quad + \boldsymbol{q}_C(X) + \mathsf{PI}(X) - \boldsymbol{q}_O(x)f_5(X) \\
F_{z_1,1} &= \mathcal{L}_1(X) \cdot (z_1(X) - 1) \\
F_{z_1,2} &= z_1(X) \cdot \left(\prod_{i=1}^{6} f_i(X) + \beta S_{\mathsf{ID}i}(X) + \gamma \right) - z_1(\omega X) \cdot \left(\prod_{i=1}^{6} f_i(X) + \beta S_{\sigma i}(X) + \gamma \right) \\
F_{z_2,1} &= \mathcal{L}_1(X) \cdot (z_2(X) - 1) \\
F_{z_2,2} &= z_2(X) \cdot (1+\beta)(\gamma + \mathcal{Q}^*(X))(\gamma(1+\beta) + \mathcal{T}^*(X) + \beta\mathcal{T}^*(X \cdot \omega)) \\
&\quad - z_2(\omega X) \cdot (\gamma(1+\beta) + h_1(X) + \beta h_2(X))(\gamma(1+\beta) + h_2(X) + \beta h_1(X \cdot \omega))
\end{aligned}$$

3. $\mathcal{P}$ computes polynomial

$$F(X) := F_{gate}(X) + \alpha F_{z_1,1}(X) + \alpha^2 F_{z_1,2}(X) + \alpha^3 F_{z_2,1}(X) + \alpha^4 F_{z_2,2}(X)$$

and sends $\mathcal{V}$ the quotient polynomial $t(X) := \frac{F(X)}{Z_H(X)}$.

__**Verification Phase:**__

1. $\mathcal{V}$ computes public input polynomial $\mathsf{PI}(X)$ defined in online phase round 1.
2. $\mathcal{V}$ checks the polynomial identity

$$F(X) \overset{?}{=} t(X)Z_H(X)$$

where $F(X)$ is defined above in online phase round 4, and can be evaluated given oracle access to the indexer's preprocessed polynomials, polynomials sent by $\mathcal{P}$, and the public input polynomial $\mathsf{PI}(X)$.
We note that the check is equivalent to checking $F(X) = 0, \forall x \in H$.

Fig. 12: A Polynomial IOP for the UltraPlonk constraint system: Part II.

2. Find coset representatives $\{k_1, k_2\}$ such that $H, k_1H, k_2H \subset \mathbb{F}^*$ are non-overlapping cosets (similar to the process in Appx. C.2). Note $H = \{1, \omega, \omega^2, \ldots, \omega^N\}$ is a multiplicative subgroup of size N.
   Derive σ^* as:

$$\overbrace{\underbrace{[3n] \overset{\sigma}{\mapsto} [3n] \mapsto [6n]}_{\sigma'} \overset{f_{\sigma'}}{\mapsto} H \cup k_1H \cup k_2H}^{\sigma^*}$$

$$\sigma'(i) := \begin{cases} \sigma(i) & 0 < \sigma(i) \le n \\ \sigma(i) + n & n < \sigma(i) \le 2n \\ \sigma(i) + 2n & 2n < \sigma(i) \le 3n \end{cases}$$

$$f_{\sigma'}(i) := \begin{cases} \omega^i & 0 < \sigma'(i) \le N \\ k_1 \cdot \omega^i & N < \sigma'(i) \le 2N \\ k_2 \cdot \omega^i & 2N < \sigma'(i) \le 3N \end{cases}$$

3. If $b = 0$,

$$\begin{aligned} \boldsymbol{q}_L(X) &= \textstyle\sum_{i=1}^{n} (\boldsymbol{q}_L)_i \cdot \mathcal{L}_i(X) \\ \boldsymbol{q}_R(X) &= \textstyle\sum_{i=1}^{n} (\boldsymbol{q}_R)_i \cdot \mathcal{L}_i(X) \\ \boldsymbol{q}_O(X) &= \textstyle\sum_{i=1}^{n} (\boldsymbol{q}_O)_i \cdot \mathcal{L}_i(X) \\ \boldsymbol{q}_M(X) &= \textstyle\sum_{i=1}^{n} (\boldsymbol{q}_M)_i \cdot \mathcal{L}_i(X) \\ \boldsymbol{q}_C(X) &= \textstyle\sum_{i=1}^{n} (\boldsymbol{q}_C)_i \cdot \mathcal{L}_i(X) \\ \boldsymbol{q}_C(X) &= \textstyle\sum_{i=1}^{n} (\boldsymbol{q}_C)_i \cdot \mathcal{L}_i(X) \\ S_{\sigma 1}(X) &= \textstyle\sum_{i=1}^{n} \sigma^*(i) \cdot \mathcal{L}_i(X) \\ S_{\sigma 2}(X) &= \textstyle\sum_{i=1}^{n} \sigma^*(n+i) \cdot \mathcal{L}_i(X) \\ S_{\sigma 3}(X) &= \textstyle\sum_{i=1}^{n} \sigma^*(2n+i) \cdot \mathcal{L}_i(X) \end{aligned}$$

   If $b = 1$

$$\begin{aligned} \boldsymbol{q}_L(X) &= \textstyle\sum_{i=1}^{n} (\boldsymbol{q}_L)_i \cdot \mathcal{L}_{n+i}(X) \\ \boldsymbol{q}_R(X) &= \textstyle\sum_{i=1}^{n} (\boldsymbol{q}_R)_i \cdot \mathcal{L}_{n+i}(X) \\ \boldsymbol{q}_O(X) &= \textstyle\sum_{i=1}^{n} (\boldsymbol{q}_O)_i \cdot \mathcal{L}_{n+i}(X) \\ \boldsymbol{q}_M(X) &= \textstyle\sum_{i=1}^{n} (\boldsymbol{q}_M)_i \cdot \mathcal{L}_{n+i}(X) \\ \boldsymbol{q}_C(X) &= \textstyle\sum_{i=1}^{n} (\boldsymbol{q}_C)_i \cdot \mathcal{L}_{n+i}(X) \\ S_{\sigma 1}(X) &= \textstyle\sum_{i=1}^{n} (\sigma^*(i) + n) \cdot \mathcal{L}_{n+i}(X) \\ S_{\sigma 2}(X) &= \textstyle\sum_{i=1}^{n} (\sigma^*(n+i) + n) \cdot \mathcal{L}_{n+i}(X) \\ S_{\sigma 3}(X) &= \textstyle\sum_{i=1}^{n} (\sigma^*(2n+i) + n) \cdot \mathcal{L}_{n+i}(X) \end{aligned}$$

4. Output proving key:

$$\mathsf{ipk} := (\boldsymbol{q}_L(X), \boldsymbol{q}_R(X), \boldsymbol{q}_O(X), \boldsymbol{q}_M(X), \boldsymbol{q}_C(X), S_{\sigma 1}(X), S_{\sigma 2}(X), S_{\sigma 3}(X))$$

   and verification key:

$$\mathsf{ivk} := ([\boldsymbol{q}_L]_1, [\boldsymbol{q}_R]_1, [\boldsymbol{q}_O]_1, [\boldsymbol{q}_M]_1, [\boldsymbol{q}_C]_1, [S_{\sigma 1}]_1, [S_{\sigma 2}]_1, [S_{\sigma 3}]_1)$$

$\underline{\mathsf{MergePK}(\mathsf{ipk}_b, \mathsf{ipk}_d) \to \mathsf{ipk}_\oplus}$

- *inputs:* a birth predicate proving key ipk_b and a death predicate proving key ipk_d.
- *outputs:* a merged proving key $\mathsf{ipk}_\oplus$.

1. Parse two input proving keys:

$$\begin{aligned} \mathsf{ipk}_b &:= \left(\boldsymbol{q}_L^b(X), \boldsymbol{q}_R^b(X), \boldsymbol{q}_O^b(X), \boldsymbol{q}_M^b(X), \boldsymbol{q}_C^b(X), S_{\sigma 1}^b(X), S_{\sigma 2}^b(X), S_{\sigma 3}^b(X)\right) \\ \mathsf{ipk}_d &:= \left(\boldsymbol{q}_L^d(X), \boldsymbol{q}_R^d(X), \boldsymbol{q}_O^d(X), \boldsymbol{q}_M^d(X), \boldsymbol{q}_C^d(X), S_{\sigma 1}^d(X), S_{\sigma 2}^d(X), S_{\sigma 3}^d(X)\right) \end{aligned}$$

2. Homomorphically add each element in the two verification keys:

$$\boldsymbol{q}_L^\oplus(X) = \boldsymbol{q}_L^b(X) + \boldsymbol{q}_L^d(X), \ldots, S_{\sigma 3}^\oplus(X) = S_{\sigma 3}^b(X) + S_{\sigma 3}^d(X)$$

3. Output the merged verification key:

$$\mathsf{ipk}_\oplus := \begin{pmatrix} \boldsymbol{q}_L^\oplus(X), \boldsymbol{q}_R^\oplus(X), \boldsymbol{q}_O^\oplus(X), \boldsymbol{q}_M^\oplus(X), \boldsymbol{q}_C^\oplus(X), \\ S_{\sigma 1}^\oplus(X), S_{\sigma 2}^\oplus(X), S_{\sigma 3}^\oplus(X) \end{pmatrix}$$

$\mathsf{MergeVK}(\mathsf{ivk}_b, \mathsf{ivk}_d) \to \mathsf{ivk}_\oplus$

- *inputs:* a birth predicate verification key ivk_b and a death predicate verification key ivk_d.
- *outputs:* a merged verification key $\mathsf{ivk}_\oplus$.

1. Parse two input verification keys:

$$\mathsf{ivk}_b := \big([\boldsymbol{q}_L^b]_1, [\boldsymbol{q}_R^b]_1, [\boldsymbol{q}_O^b]_1, [\boldsymbol{q}_M^b]_1, [\boldsymbol{q}_C^b]_1, [S_{\sigma 1}^b]_1, [S_{\sigma 2}^b]_1, [S_{\sigma 3}^b]_1\big)$$
$$\mathsf{ivk}_d := \big([\boldsymbol{q}_L^d]_1, [\boldsymbol{q}_R^d]_1, [\boldsymbol{q}_O^d]_1, [\boldsymbol{q}_M^d]_1, [\boldsymbol{q}_C^d]_1, [S_{\sigma 1}^d]_1, [S_{\sigma 2}^d]_1, [S_{\sigma 3}^d]_1\big)$$

2. Homomorphically add each element in the two proving keys:

$$[\boldsymbol{q}_L^\oplus]_1 = [\boldsymbol{q}_L^b]_1 + [\boldsymbol{q}_L^d]_1, \ldots, [S_{\sigma 3}^\oplus]_1 = [S_{\sigma 3}^b]_1 + [S_{\sigma 3}^d]_1$$

3. Output the merged proving key:

$$\mathsf{ivk}_\oplus := \big([\boldsymbol{q}_L^\oplus]_1, [\boldsymbol{q}_R^\oplus]_1, [\boldsymbol{q}_O^\oplus]_1, [\boldsymbol{q}_M^\oplus]_1, [\boldsymbol{q}_C^\oplus]_1, [S_{\sigma 1}^\oplus]_1, [S_{\sigma 2}^\oplus]_1, [S_{\sigma 3}^\oplus]_1\big)$$

$\mathsf{MergeWit}(\mathbb{w}_b, \mathbb{w}_d) \to \mathbb{w}_\oplus$

- *inputs:* birth predicate witness $\mathbb{w}_b$ and death predicate witness $\mathbb{w}_d$
- *outputs:* merged witness $\mathbb{w}_\oplus$

1. Parse birth and death witness $\in \mathbb{F}^{3n}$ (witness from the indexed relation is transformed as in Appx. C.1 first):

$$\mathbb{w}_b = (w_{b,1}, \ \ldots \ , w_{b,3n}) \,, \quad \mathbb{w}_d = (w_{d,1}, \ \ldots \ , w_{d,3n})$$

2. Outputs merged witness $\in \mathbb{F}^{3N}$ where $N = 2n$:

$$\mathbb{w}_\oplus = \begin{pmatrix} w_{b,1}, \ \ldots \ , w_{b,n}, w_{d,1}, \ \ldots \ , w_{d,n}, \\ w_{b,n+1}, \ \ldots \ , w_{b,2n}, w_{d,n+1}, \ \ldots \ , w_{d,2n}, \\ w_{b,2n+1}, \ \ldots \ , w_{b,3n}, w_{d,2n+1}, \ \ldots \ , w_{d,3n} \end{pmatrix}$$

where the merged public instance is $\mathbb{x}_\oplus = (w_{b,1}, \ \ldots \ , w_{b,\ell}, w_{d,1}, \ \ldots \ , w_{d,\ell})$

$\mathsf{BatchProve}(\mathsf{srs}, [\mathbb{x}_{\oplus,i}]_{i=1}^{n_{\mathsf{pf}}}, [\mathbb{w}_{\oplus,i}]_{i=1}^{n_{\mathsf{pf}}}, [\mathsf{ipk}_{\oplus,i}]_{i=1}^{n_{\mathsf{pf}}}) \to \pi_\circledast$

- *inputs:*
  - public parameter srs
  - list of merged instances, merged witnesses, and merged proving keys $[\mathbb{x}_{\oplus,i}]_{i=1}^{n_{\mathsf{pf}}}, [\mathbb{w}_{\oplus,i}]_{i=1}^{n_{\mathsf{pf}}}, [\mathsf{ipk}_{\oplus,i}]_{i=1}^{n_{\mathsf{pf}}}$ where n_{pf} denotes the number of instances to be proven.
- *outputs:* a batched proof $\pi_\circledast$

Round 1:

- For each $i \in [n_{\mathsf{pf}}]$, compute wire polynomials $a_i(X), b_i(X), c_i(X)$ as in Round 1 of [GWC19][27], and outputs $([a_i]_1, [b_i]_1, [c_i]_1)_{i \in [n_{\mathsf{pf}}]}$.

Round 2:

- Compute permutation challenge $\beta = H(\mathsf{transcript}, 0), \gamma = H(\mathsf{transcript}, 1)$.
- For each $i \in [n_{\mathsf{pf}}]$, compute permutation polynomials $z_i(X)$ as in Round 2 of [GWC19], and outputs $([z_i]_1)_{i \in [n_{\mathsf{pf}}]}$.

Round 3:

[27] Referring to the protocol presented in Section 8.3, similarly for all consequent steps

- Compute quotient challenge $\alpha = H(\mathsf{transcript})$.
- For each $i \in [n_{\mathsf{pf}}]$,
  - Parse merged instance $\mathbb{x}_{\oplus,i} = (x_i)_{i\in[2\ell]}$, compute public input polynomial: $\mathsf{PI}_i(X) = \sum_{i=1}^{\ell} (x_i \cdot \mathcal{L}_i(X) + x_{i+\ell} \cdot \mathcal{L}_{n+i}(X))$
  - Compute the following intermediate polynomials:

$$\begin{aligned} F_{gate,i}(X) &= a_i(X)\boldsymbol{q}_L^i(X) + b_i(X)\boldsymbol{q}_R^i(X) + a_i(X)b_i(X)\boldsymbol{q}_M^i(X) + \mathsf{PI}_i(X) - c_i(X)\boldsymbol{q}_O^i(X) \\ F_{\sigma 1,i}(X) &= (a_i(X) + \beta X + \gamma)(b_i(X) + \beta k_1 X + \gamma)(c_i(X) + \beta k_2 X + \gamma) \cdot z_i(X) \\ &\quad - (a_i(X) + \beta S_{\sigma 1}^i(X) + \gamma)(b_i(X) + \beta S_{\sigma 2}^i(X) + \gamma)(c_i(X) + \beta S_{\sigma 3}^i(X) + \gamma) \cdot z_i(\omega X) \\ F_{\sigma 2,i}(X) &= (z_i(X) - 1) \cdot \mathcal{L}_1(X) \end{aligned}$$

- Compute the *batched quotient polynomial* (note $|H| = N = 2n$):

$$t(X) = Z_H^{-1}(X) \left(\sum_{i=1}^{n_{\mathsf{pf}}} \alpha^{3i-3} \left(F_{gate,i}(X) + \alpha F_{\sigma 1,i}(X) + \alpha^2 F_{\sigma 2,i}(X) \right) \right)$$

- Split $t(X)$ into $t_1(X), t_2(X), t_3(X)$ as in Round 3 of [GWC19], and outputs $([t_1]_1, [t_2]_1, [t_3]_1)$.

Round 4:

- Compute evaluation challenge $\mathfrak{z} = H(\mathsf{transcript})$.
- For each $i \in [n_{\mathsf{pf}}]$, compute and output opening evaluations:

$$\bar{a}_i = a_i(\mathfrak{z}), \bar{b}_i = b_i(\mathfrak{z}), \bar{c}_i = c_i(\mathfrak{z}), \bar{s}_{\sigma_1,i} = S_{\sigma 1}^i(\mathfrak{z}), \bar{s}_{\sigma_2,i} = S_{\sigma 2}^i(\mathfrak{z}), \bar{z}_{\omega,i} = z_i(\mathfrak{z}\omega)$$

Round 5:

- Compute opening challenge $v = H(\mathsf{transcript})$.
- For each $i \in [n_{\mathsf{pf}}]$, compute the following intermediate polynomials:
  - Numerator polynomial of the quotient polynomial:

$$\begin{aligned} r_i(X) &= \bar{a}_i\boldsymbol{q}_L^i(X) + \bar{b}_i\boldsymbol{q}_R^i(X) + \bar{a}_i\bar{b}_i\boldsymbol{q}_M^i(X) + \boldsymbol{q}_C^i(X) + \mathsf{PI}(\mathfrak{z}) - \bar{c}_i\boldsymbol{q}_O^i(X) \\ &\quad + \alpha \cdot [(\bar{a}_i + \beta\mathfrak{z} + \gamma)(\bar{b}_i + \beta k_1\mathfrak{z} + \gamma)(\bar{c}_i + \beta k_2\mathfrak{z} + \gamma) \cdot z_i(X) \\ &\quad - (\bar{a}_i + \beta\bar{s}_{\sigma_1,i} + \gamma)(\bar{b}_i + \beta\bar{s}_{\sigma_2,i} + \gamma)(\bar{a}_i + \beta\bar{s}_{\sigma_3,i} + \gamma) \cdot \bar{z}_{\omega,i}] \\ &\quad + \alpha^2 \cdot [(z_i(X) - 1)\mathcal{L}_1(\mathfrak{z})] \end{aligned}$$

  - Polynomials evaluated at evaluation point $\mathfrak{z}$:

$$g_{\mathfrak{z},i}(X) = \begin{pmatrix} v \cdot (a_i(X) - \bar{a}_i) \\ +v^2 \cdot (b_i(X) - \bar{b}_i) \\ +v^3 \cdot (c_i(X) - \bar{c}_i) \\ +v^4 \cdot (S_{\sigma 1}^i(X) - \bar{s}_{\sigma_1,i}) \\ +v^5 \cdot (S_{\sigma 2}^i(X) - \bar{s}_{\sigma_2,i}) \end{pmatrix}$$

  - Polynomial(s) evaluated at evaluation point $\mathfrak{z}\omega$:

$$g_{\mathfrak{z}\omega,i}(X) = z_i(X) - \bar{z}_{\omega,i}$$

- Compute the *batched linearization polynomial*:

$$r(X) = \sum_{i=1}^{n_{\mathsf{pf}}} \alpha^{3i-3} \cdot r_i(X) - Z_H(\mathfrak{z}) \cdot \left(t_1(X) + \mathfrak{z}^N t_2(X) + \mathfrak{z}^{2N} t_3(X) \right)$$

- Compute the *batched opening proof polynomials*:

$$W_{\mathfrak{z}}(X) = \frac{r(X) + \sum_{i=1}^{n_{\mathsf{pf}}} v^{6i-6} \cdot g_{\mathfrak{z},i}}{X - \mathfrak{z}}, \quad W_{\mathfrak{z}\omega}(X) = \frac{\sum_{i=1}^{n_{\mathsf{pf}}} v^{i-1} \cdot g_{\mathfrak{z}\omega,i}}{X - \mathfrak{z}\omega}$$

- Outputs their commitments: $[W_{\mathfrak{z}}]_1, [W_{\mathfrak{z}\omega}]_1$.

Final Round:

- Compute multi-point evaluation challenge $u = H(\mathsf{transcript})$.
- For each $i \in [n_{\mathsf{pf}}]$, define: $\pi_i = \Big([a_i]_1, [b_i]_1, [c_i]_1, [z_i]_1, \bar{a}_i, \bar{b}_i, \bar{c}_i, \bar{s}_{\sigma 1,i}, \bar{s}_{\sigma 2,i}, \bar{z}_{\omega,i}\Big)$, outputs batched proof:

$$\pi_{\circledast} := \Big([t_1]_1, [t_1]_2, [t_3]_1, [W_{\mathfrak{z}}]_1, [W_{\mathfrak{z}\omega}]_1, [\pi_i]_{i\in[n_{\mathsf{pf}}]}\Big)\ .$$

Remark 7. Since we only use Plonk' for the inner predicate proofs, we don't need it to be zero-knowledge, as long as the SNARK proof for the outer circuit is zero-knowledge, then we would satisfy the function privacy of ZEXE. Specifically, we don't need random blinding scalars for wire polynomials in Round 1 and permutation polynomials in Round 2 as in [GWC19].

$\underline{\mathsf{BatchPartialVfy}(\mathsf{srs}, [\mathbb{x}_{\oplus,i}]_{i=1}^{n_{\mathsf{pf}}}, [\mathsf{ivk}_{\oplus,i}]_{i=1}^{n_{\mathsf{pf}}}, \pi_{\circledast}, s) \to \xi}$

- *inputs*:
  - public parameter srs
  - list of merged instances, merged verification keys $[\mathbb{x}_{\oplus,i}]_{i=1}^{n_{\mathsf{pf}}}, [\mathsf{ivk}_{\oplus,i}]_{i=1}^{n_{\mathsf{pf}}}$
  - the batched proof for all relations $\pi_{\circledast}$
  - masking randomness $s \in \mathbb{F}$
- *outputs*: an accumulator ξ containing two group elements to checked in a pairing equation in Decider in the accumulation scheme.

1. Validate all field elements and group elements in $\pi_{\circledast}$.
2. Compute challenges $\alpha, \beta, \gamma, \mathfrak{z}, v, u \in \mathbb{F}$ as in the $\mathsf{BatchProve}$ from the common inputs, public inputs, and elements of $\pi_{\circledast}$.
3. Compute:

$$[A]_1 = [W_{\mathfrak{z}}]_1 + u[W_{\mathfrak{z}\omega}]_1$$
$$[B]_1 = \mathfrak{z}[W_{\mathfrak{z}}]_1 + u\mathfrak{z}\omega[W_{\mathfrak{z}\omega}]_1 + \underbrace{\left[r(X) + \sum_{i=1}^{n_{\mathsf{pf}}} v^{6i-6} g_{\mathfrak{z},i}(X)\right]_1 + u\left[\sum_{i=1}^{n_{\mathsf{pf}}} v^{i-1} g_{\mathfrak{z}\omega,i}(X)\right]_1}_{\text{computable from all inputs to } \mathsf{BatchPartialVfy}.}$$

4. Mask both elements for hiding property:

$$[\tilde{A}]_1 = [A]_1 + s[x]_1, \quad [\tilde{B}]_1 = [B]_1 + s[1]_1$$

   where $[x]_1, [1]_1$ are in srs.
5. Outputs $\xi := \Big([\tilde{A}]_1, [\tilde{B}]_1\Big)$.

$\underline{\mathsf{Decide}(\mathsf{srs}, \xi) \to b}$

- *inputs*: public parameter srs and the accumulator (partial verification state) ξ
- *outputs*: accept or reject bit $b \in \{0, 1\}$

1. Parse $\xi := \Big([\tilde{A}]_1, [\tilde{B}]_1\Big)$, get $[1]_2, [x]_2$ from srs.
2. Check $e([\tilde{A}]_1, [1]_2) \overset{?}{=} e([\tilde{B}]_1, [x]_2)$, if equal output $b = 1$; output $b = 0$ otherwise.

E Modular Arithmetic Gadgets: Security Proof

We proceed to provide detailed proofs for the security properties of our modular arithmetic gadgets. Given set of relevant public parameters (and prerequisite assumption about them), a circuit design (w.r.t. a relation) has *completeness* when for any inputs and their valid witnesses, the circuit should always be satisfied; *soundness* when for any inputs and invalid witnesses, the circuit should never be satisfied.

Proof (Proposition 1). Soundness is straightforward since any invalid witnesses $w_0, w_1, z_0, z_1 \in \mathbb{F}_q$, either they are out of bound $\in [2^m, q)$ thus failed the first range check in the gadget, or they are values $\in [0, 2^m)$ that breaks the equation 1 and by violating any one of equation 1b,1c,1d, the corresponding steps in the gadget will fail.

$$(x_0 + 2^m \cdot x_1) \cdot (y_0 + 2^m \cdot y_1) \tag{1}$$
$$= z_0 + 2^m \cdot z_1 + (w_0 + 2^m \cdot w_1) \cdot (p_0 + 2^m \cdot p_1) \tag{1a}$$

$$\Updownarrow$$

$$\begin{cases} z_0 + w_0 \cdot p_0 - x_0 \cdot y_0 - 2^m \cdot c_0' = 0 & \text{(1b)} \\ z_1 + w_0 \cdot p_1 + w_1 \cdot p_0 - x_0 \cdot y_1 - x_1 \cdot y_0 + c_0' - 2^m \cdot c_1' = 0 & \text{(1c)} \\ w_1 \cdot p_1 - x_1 \cdot y_1 + c_1' = 0 & \text{(1d)} \end{cases}$$

For *completeness*, we need to further argue that (i) calculations within circuit won't overflow or underflow $\mathbb{F}_q$ at all steps; and (ii) there *exists* at least one witness for any possible inputs.

To see why (i) is true (with honest provers who follow the protocol and prepare witnesses and intermediate values like carriers c_0, c_1 properly):

- In step 2, LHS is $z_0 + w_0 \cdot p_0 \in [0, 2^m + 2^{2m}]$; RHS is $x_0 \cdot y_0 + 2^m \cdot (c_0 - 2^m) \in [0, 2^{2m} + 2^m \cdot (2^{m+k} - 2^m)) \in [0, 2^{2m+k})$. Since $q > 2^{2m+k+1}$, both LHS and RHS won't overflow. Note that RHS ≥ 0 because $c_0' \geq -2^m$ and $c_0 = c_0' + 2^m$.
- In step 3, LHS is $z_1 + w_0 \cdot p_1 + w_1 \cdot p_0 + (c_0 - 2^m) \in [0, 2^m + 2^{2m} \cdot 2 + 2^{m+k} - 2^m) \in [0, 2^{2m+1} + 2^{m+k})$; RHS is $x_0 \cdot y_1 + x_1 \cdot y_0 + 2^m \cdot (c_1 - 2^{m+1}) \in [0, 2^{2m} \cdot 2 + 2^m \cdot 2^{m+2}) \in [0, 2^{2m+3})$. Since $q > 2^{2m+k+1} \ \wedge \ K \leq 3$, both LHS and RHS won't overflow.
- In step 4, LHS is $w_1 \cdot p_1 + (c_1 - 2^{m+1}) \in [0, 2^{2m} + 2^{m+2}) \in [0, 2^{2m+1})$; RHS is $x_1 \cdot y_1 \in [0, 2^{2m})$. Clearly both sides won't overflow.

To see why (ii) is true, we emphasize that inputs x, y are assumed to be the canonical representation of $\mathbb{F}_p$ (namely, $x, y \in [0, p)$). Since for $x \cdot y = z + p \cdot w$ (the actual relation is expressed limb-wise), $x \cdot y \in [0, p^2)$ and $p \cdot w \in [0, p \cdot 2^{2m})$, given that $2^{2m} > p$, we know ensure existence of at least one $w \in [0, 2^{2m})$ for any inputs x, y.

Proof (Proposition 2). For *soundness*, it is straightforward, invalid witnesses w, y are either out of proper range and failed the range check in the first step of circuit, or they are in range but violate the relation $y + p \cdot w = x_1 + \ldots + x_N$ which will failed the second step.

For *completeness*, we need to further argue that (i) calculations within circuit won't overflow or underflow $\mathbb{F}_q$ at all steps; and (ii) there *exists* at least one witness for any possible inputs.

To see why (i) is true: given $N < \frac{K-1}{c} + 1, c \cdot p \geq 2^{2m} \geq p, \frac{q}{p} > c + K$ in step 2,
LHS is $0 \leq y + p \cdot w < 2^{2m} + 2^{2m} \cdot K < 2^{2m+k+1}$;
RHS is $0 \leq x_1 + \ldots + x_N < N \cdot 2^{2m}) < (\frac{K}{c} + 1) \cdot 2^{2m} < (\frac{K}{c} + 1) \cdot c \cdot p < (K + c) \cdot p < q$.
We can see both sides won't overflow $\mathbb{F}_q$.

To see why (ii) is true: since the tight upper bound of LHS is bigger than a loose upper bound of RHS:

$$2^{2m} + K \cdot p \geq (K + c - 1) \cdot p$$
$$> (\frac{K-1}{c} + 1) \cdot cp > (\frac{K-1}{c} + 1) \cdot 2^{2m} > N \cdot 2^{2m}$$

there must exists some witnesses w, y for any inputs.

Predicate commitment in $\mathcal{R}_{\mathsf{utxo}}$

$$\mathbb{x}_{\mathsf{utxo}} = \begin{pmatrix}\text{pred. commitment} & \mathsf{cm}_{\Phi}\end{pmatrix} \quad \mathbb{w}_{\mathsf{utxo}} = \begin{pmatrix}\text{input death pred. hashes} & [\mathsf{pid}_{d,i}]_1^m \\ \text{output birth pred. hashes} & [\mathsf{pid}_{b,j}]_1^n \\ \text{pred. comm. randomness} & r_{\Phi}\end{pmatrix}$$

Circuit (over $\mathbb{F}_r$)

1. The predicate commitment is valid:

$$\mathsf{cm}_{\Phi} = \mathsf{COM.Commit}([\mathsf{pid}_{d,i}]_1^m \parallel [\mathsf{pid}_{b,j}]_1^n; r_{\Phi}) = \mathsf{Blake2s}([\mathsf{pid}_{d,i}]_1^m \parallel [\mathsf{pid}_{b,j}]_1^n \parallel r_{\Phi})$$

Predicate commitment in $\mathcal{R}_{\Phi}$

$$\mathbb{x}_{\Phi} = \begin{pmatrix}\text{pred. commitment} & \mathsf{cm}_{\Phi}\end{pmatrix} \quad \mathbb{w}_{\Phi} = \begin{pmatrix}\text{input death pred. ver. key} & [\mathsf{ivk}_{d,i}]_1^m \\ \text{output birth pred. ver. key} & [\mathsf{ivk}_{b,j}]_1^n \\ \text{pred. comm. randomness} & r_{\Phi}\end{pmatrix}$$

Circuit (over $\mathbb{F}_p$)

1. For each $i \in \{1, \ldots, m\}$: the death predicate hash is computed correctly: $\mathsf{pid}_{d,i} = \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, \mathsf{ivk}_{d,i})$.
2. For each $j \in \{1, \ldots, n\}$: the birth predicate hash is computed correctly: $\mathsf{pid}_{b,j} = \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, \mathsf{ivk}_{b,j})$.
3. The predicate commitment is valid:

$$\mathsf{cm}_{\Phi} = \mathsf{COM.Commit}([\mathsf{pid}_{d,i}]_1^m \parallel [\mathsf{pid}_{b,j}]_1^n; r_{\Phi}) = \mathsf{Blake2s}([\mathsf{pid}_{d,i}]_1^m \parallel [\mathsf{pid}_{b,j}]_1^n \parallel r_{\Phi})$$

Fig. 13: Relation snippet for predicate commitment in [BCG+20].

F Optimized Predicate Commitment

For efficiency reason, we split $\mathcal{R}_e$ into two relations: an extended UTXO relation $\mathcal{R}_{\mathsf{utxo}}$ that checks the the well-formedness of input and output records among other things, and a predicates satisfiability relation $\mathcal{R}_{\Phi}$ that checks inner proofs for death/birth predicates of the input/output records. The circuit for $\mathcal{R}_{\Phi}$ takes a list of inner proofs and their corresponding verification keys as secret witnesses and checks their validity. To ensure that death/birth predicates involved in $\mathcal{R}_{\mathsf{utxo}}$ and $\mathcal{R}_{\Phi}$ are consistent, [BCG+20] proposes to make the hiding commitment cm_{Φ} to the predicates in a transaction as a public input for both circuits so that the verifier can check their equality (see Fig. 13). Furthermore, the authors suggest committing to the collision-resistant hashes of the predicate verification keys instead, to reduce the cost of computing cm_{Φ} in the circuit.

Concretely, the original ZEXE instantiate CRH with Pedersen hash, COM with Blake2s hash where the message is appended with a randomizer for the hiding property. The primary circuit cost comes from constraining non-algebraic Blake2s hash on a message size of $m+n+1$ for an m-input-n-output transaction. We emphasize that directly switching Blake2s to a SNARK-friendly hash is not immediately more advantageous, since we need to constrain this hash function in two different fields (over $\mathbb{F}_r$ for $\mathcal{R}_{\mathsf{utxo}}$ and over $\mathbb{F}_p$ for $\mathcal{R}_{\Phi}$), and constraining algebraic hashes over non-native fields is probably more expensive as it requires many range checks and modular arithmetics. Worse, the number of non-native operations grows linearly with the message size since longer messages require more invocations of the hash function.

We present an optimized circuit in Fig. 14. The high level idea is to encode the list of predicate hashes/identifiers as the coefficients of a univariate polynomial, and commit to these predicates by evaluating this polynomial at a binding point.

Let $\mathsf{HCOM}_1 : \mathbb{F}_r^{m+n} \mapsto \mathbb{F}_r, \mathsf{HCOM}_2 : \mathbb{F}_p^{m+n} \mapsto \mathbb{F}_r$ be two hash-based commitment scheme with different message spaces but the same digest space. In practice, we can instantiate $\mathsf{HCOM}_1, \mathsf{HCOM}_2, \mathsf{COM}$ with SNARK-friendly, hash-based commitments, and use modular arithmetic gadgets introduced in § 3.5 during step 4 of the predicate commitment circuit in $\mathcal{R}_{\Phi}$.

Predicate commitment in $\mathcal{R}_{\mathsf{utxo}}$

$$\mathbb{x}_{\mathsf{utxo}} = \begin{pmatrix} \text{pred. comm.} & \mathsf{cm}_{\Phi} \in \mathbb{F}_r \\ \text{pred. bindings} & c_1, c_2 \in \mathbb{F}_r \end{pmatrix} \qquad \mathbb{w}_{\mathsf{utxo}} = \begin{pmatrix} \text{input death pred. hashes} & [\mathsf{pid}_{d,i}]_1^m \\ \text{output birth pred. hashes} & [\mathsf{pid}_{b,j}]_1^n \\ \text{pred. comm. randomness} & r_{\Phi} \in \mathbb{F}_r \\ \text{pred. binding factor} & r_{\Phi,1} \in \mathbb{F}_r \end{pmatrix}$$

Circuit (over $\mathbb{F}_r$)

1. Check predicate binding is correct: $c_1 = \mathsf{HCOM}_1.\mathsf{Commit}(\mathsf{pp}_{\mathsf{HCOM}_1}, [\mathsf{pid}_{d,i}]_1^m \parallel [\mathsf{pid}_{b,j}]_1^n; r_{\Phi,1})$.
2. The predicate commitment is valid: $\mathsf{cm}_{\Phi} = \mathsf{COM}.\mathsf{Commit}(\mathsf{pp}_{\mathsf{COM}}, p_{\mathsf{utxo}}(c); r_{\Phi})$ where $c = c_1 + c_2$ and $p_{\mathsf{utxo}}(X) = \sum_{i=1}^m \mathsf{pid}_{d,i} \cdot X^{i-1} + \sum_{j=1}^n \mathsf{pid}_{b,j} \cdot X^{m+j-1}$.

Predicate commitment in $\mathcal{R}_{\Phi}$

$$\mathbb{x}_{\Phi} = \begin{pmatrix} \text{pred. comm.} & \mathsf{cm}_{\Phi} \\ \text{pred. bindings} & c_1, c_2 \in \mathbb{F}_r \end{pmatrix} \qquad \mathbb{w}_{\Phi} = \begin{pmatrix} \text{input death pred. ver. key} & [\mathsf{ivk}_{d,i}]_1^m \\ \text{output birth pred. ver. key} & [\mathsf{ivk}_{b,j}]_1^n \\ \text{pred. comm. randomness} & r_{\Phi} \in \mathbb{F}_r \\ \text{pred. binding factor} & r_{\Phi,2} \in \mathbb{F}_p \end{pmatrix}$$

Circuit (over $\mathbb{F}_p$)

1. For each $i \in \{1, \ldots, m\}$: the death predicate hash is computed correctly: $\mathsf{pid}_{d,i} = \mathsf{CRH}.\mathsf{Eval}(\mathsf{pp}_{\mathsf{CRH}}, \mathsf{ivk}_{d,i})$.
2. For each $j \in \{1, \ldots, n\}$: the birth predicate hash is computed correctly: $\mathsf{pid}_{b,j} = \mathsf{CRH}.\mathsf{Eval}(\mathsf{pp}_{\mathsf{CRH}}, \mathsf{ivk}_{b,j})$.
3. Check predicate binding is correct: $c_2 = \mathsf{HCOM}_2.\mathsf{Commit}(\mathsf{pp}_{\mathsf{HCOM}_2}, [\mathsf{pid}_{d,i}]_1^m \parallel [\mathsf{pid}_{b,j}]_1^n; r_{\Phi,2})$.
4. The predicate commitment is valid: $\mathsf{cm}_{\Phi} = \mathsf{COM}.\mathsf{Commit}(\mathsf{pp}_{\mathsf{COM}}, p_{\Phi}(c); r_{\Phi})$ where $c = c_1 + c_2$ and $p_{\Phi}(X) = \sum_{i=1}^m \mathsf{pid}_{d,i} \cdot X^{i-1} + \sum_{j=1}^n \mathsf{pid}_{b,j} \cdot X^{m+j-1}$. (addition, polynomial evaluation, and commitment are all computed over non-native field $\mathbb{F}_r$)

Fig. 14: Relation snippet for optimized predicate commitment.

Remark 8. Compared to the naïve solution of directly switching Blake2s to a SNARK-friendly hash, the main efficiency of our design in Fig. 14 comes from the fact that our non-native operations does not grow with message size (number of predicates committed). Because $\mathsf{HCOM}_1, \mathsf{HCOM}_2$ computations, used to bind all predicate identifiers and whose costs increase linearly with the input message size, are over the native field of their respective circuits and involve no modular arithmetics; whereas the only step involving non-native operations (the $\mathsf{COM}.\mathsf{Commit}$ in $\mathcal{R}_{\Phi}$ circuit) enjoys a fixed-size input, thus fixed cost, regardless of the number of predicates to be committed.

Proposition 3. *Assuming* $\mathsf{HCOM}_1, \mathsf{HCOM}_2$ *are random oracles,* COM *is a hiding commitment scheme, then the circuits in Fig. 14 ensure that the list of predicates being used in two circuits are consistent with overwhelming probability w.r.t. the randomly sampled binding factors* $r_{\Phi,1} \in \mathbb{F}_r, r_{\Phi,2} \in \mathbb{F}_p$*, and the predicate commitment randomness* $r_\Phi \in \mathbb{F}_r$*, and that* $\mathsf{cm}_\Phi, c_1, c_2$ *reveals nothing about the predicates involved.*

Proof (informal). Since $\mathsf{cm}_\Phi, c_1, c_2$ are public inputs, we know $\mathsf{cm}_\Phi, c = c_1 + c_2$ are guaranteed to be equal across two circuits. If $\mathcal{R}_{\mathsf{utxo}}$ and $\mathcal{R}_\Phi$ commits to two different lists of predicates, then the two polynomials would be different: $p_{\mathsf{utxo}}(X) \neq p_\Phi$. Given $\mathsf{cm}_\Phi = \mathsf{COM.Commit}(\mathsf{pp}_{\mathsf{COM}}, p_{\mathsf{utxo}}(c); r_\Phi) = \mathsf{COM.Commit}(\mathsf{pp}_{\mathsf{COM}}, p_\Phi(c); r_\Phi)$, there are two cases:

- $p_{\mathsf{utxo}}(c) \neq p_\Phi(c)$: this means we open cm_Φ to two different messages which breaks the computational binding property of COM.
- $p_{\mathsf{utxo}}(c) = p_\Phi(c)$: two different polynomials $\in \mathbb{F}_r^{<m+n-1}$ only evaluate to the same value at the same random point c (generated from random oracles) with negligible probability of $\frac{m+n-1}{|\mathbb{F}_r|}$ based on the Schwartz-Zippel lemma.

Therefore, we can conclude that the predicates are consistent across these two relations with overwhelming probability.

Furthermore, since COM is hiding, $\mathsf{HCOM}_1, \mathsf{HCOM}_2$ are random oracles, with randomly sampled randomnesses $r_{\Phi,1}, r_{\Phi,2}, r_\Phi$, we know that $\mathsf{cm}_\Phi, c_1, c_2$ reveals nothing about their committed messages.