VERIZEXE: Decentralized Private Computation with Universal Setup

Alex Luoyuan Xiong[1], Binyi Chen[2], Zhenfei Zhang[3], Benedikt Bünz[4], Ben Fisch[5], Fernando Krell[6], and Philippe Camacho[7]

[1,2,3,4,5,6,7]Espresso Systems
[1]National University of Singapore
[4]Stanford University
[5]Yale University

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 3.4x 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 $\sim 21k$ constraints.

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 [38], to a Turing complete virtual machine, as in Ethereum [44], 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 [6,14,18,19,35,39].The Zerocash protocol [6] is a privacy-preserving payment system that achieves user anonymity and amount confidentiality in transactions. Hawk [35] proposes a smart-contract framework that preserves program data privacy. Zether [14] 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 [12] 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 offline and update the ledger by sending a transaction with a publicly verifiable cryptographic proof attached, attesting to the correctness of the computation.

The core building block in a DPC construction is a *Succinct Non-interactive ARgument of Knowledge* (SNARK) proving system [9]. A SNARK system for a binary relation $\mathcal{R}$ provides a prover algorithm $\mathcal{P}(x,w)$ that on any valid *public inputs* and *private witness* pair $(x,w) \in \mathcal{R}$ outputs a valid and succinct proof π, and a verifier algorithm $\mathcal{V}(x,\pi)$ that always accepts valid proofs and rejects invalid proofs with overwhelming probability. A zkSNARK proof additionally guarantees the *zero-knowledge* property, thus leaking no information about the witness w. We generally encode $\mathcal{R}$ using "circuit" or various *constraint systems*, which outputs 1 on input (x,w) if and only if $(x,w) \in \mathcal{R}$. 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 [12] system uses a non-universal scheme [29, 31], thus requiring

Implementation	Universal Setup	Transaction Generation	Memory	Verification	Proof Size
Original ZEXE [12]	✗	14.3 s	6.56 GB	15 ms	0.482 KB
SnarkVM testnet-2	✓	151.4 s	22.81 GB	15 ms	0.482 KB
VERIZEXE (this work)	✓	16.9 s	6.61 GB	18 ms	4.138 KB

Table 1: Comparison of three DPC implementations for 2-input-2-output transaction. **Both verification time and proof size are constant and independent of transaction dimension** (number of input and output records), whereas transaction generation time and memory usage grow with larger transaction dimensions. Details in § 3.

a trusted setup for every application. As this is extraordinarily inconvenient in practice, the authors also suggest an alternative instantiation from universal SNARKs [20], which requires a one-time setup to support all future applications up to 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 proof for a circuit that encodes the SNARK verification logic. Specifically, its transaction generation speed is an order of magnitude slower than that in the original ZEXE. Hence we ask the following question:

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

1.1 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), while keeping constant on-chain verification cost independent of offline computation.
- Constraint designs for efficient variable-base MSM and modular arithmetics, leading to a PLONK verifier gadget taking only $\sim 21k$ PLONK constraints[2] which are of independent interests.[3]
- Implementation (open-sourced[4], written in Rust) and evaluation of VERIZEXE showing its practicality and most notably its 9x improvement on transaction generation time and 3.4x improvement on memory usage over the prior state-of-the-art.

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

[2] The PLONK constraint system is very extensible from just add and mul gate to customized gates (TURBOPLONK), to support the lookup argument (ULTRAPLONK). These different flavors only affect the concrete encodings of the same constraint logic. Technically we encode a TURBOPLONK verifier in a ULTRAPLONK constraint system using 21k constraints.

[3] Another promising application of our PLONK verifier gadget is *private zkRollup* protocols (such as Aztec [25]) which aggregates already private transactions into a single rollup transaction with proof attesting to the correctness of all private transactions, achieving higher computation compression for the blockchain validator (who now only needs to verify the final rollup proof) while maintaining privacy.

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

1.2 DPC Background and Use Cases

In a DPC scheme, the distributed ledger keeps track of *records*, the basic unit of data, each consisting of a data payload (state), a *birth* Φ_b and *death* Φ_d predicate governing the rules of its creation and consumption (program logic). A transaction, by consuming existing records and creating new records, represents the resulting state transition from some offline program execution. Execution correctness is captured in relation $\mathcal{R}_e$ which enforces ⓐ the existence and rightful ownership of the consumed records; ⓑ valid openings of new record commitments and consumed records *nullifiers*; and ⓒ satisfiability of death predicates of all consumed records and birth predicates of newly created records. For efficiency reason, $\mathcal{R}_e$ is split into $\mathcal{R}_{\mathsf{utxo}}$ for condition ⓐ and ⓑ, and $\mathcal{R}_\Phi$ for condition ⓒ.

The life cycle of transactions starts with ① a user (or a group collectively) executing programs offline by assembling existing records, creating new records with updated payload, and generating two zkSNARK proofs (one for $\mathcal{R}_{\mathsf{utxo}}$ and one for $\mathcal{R}_\Phi$) to testify the transaction validity. For $\mathcal{R}_\Phi$, users first generate several *inner proofs* each attesting to the satisfiability of one of the relevant predicates, then further generate a single *outer proofs* attesting the correctness of all inner proofs. Then ② users submit their transactions, containing unlinkable nullifiers of consumed records and hiding commitments of new records together with two zkSNARK proofs which reveal nothing about the program data or the predicates/programs involved, to ledger maintainers. Upon receiving transactions, ③ ledger maintainers verify SNARK proofs and update global states by inserting new record commitments and old record nullifiers into the ledger.

A motivating use case is a permissionless multi-asset trading platform that minimizes front-running, censorship, and traffic analysis. Trades on public blockchains like Ethereum have no privacy, leading to front-running arbitrage by the miners [21], and deanonymization as a service.[5] For blockchains with only data privacy, even though the exact amount and trading accounts are hidden, the programs the transaction is interacting with are still visible, which leaks the asset types involved in a trade. This metadata alone can result in preferential censorship by miners who want to block transactions of certain asset types, and traffic analysis with pattern association of side information off-chain which compromises privacy.

[5] Companies like Chain Analysis and TRM Labs can track and deanonymize real identities behind most Bitcoin and Ethereum addresses and offer it as a service.

Luckily, DPC-based blockchains would guarantee both data and function privacy of trading transactions, leaking no exploitable information to the miners or the public, thus greatly eliminating the previous issues.

We refer to the original paper [12] for detailed definitions, security properties of the DPC model, and more example applications.

1.3 Our Techniques

ZEXE [12] 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. 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 *Polynomial Commitment Scheme* (PCS) (such as KZG [33] and its variants [20], see definitions in Appx. A.2) require some pairing operations in their verification logic, which is very expensive and requires a large number of constraints in a circuit. [6]
- **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.
- **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 many times. However, commonly used hash functions are not SNARK friendly and result in high circuit costs.

We present an overview of our techniques that drastically reduce the outer circuit complexity whose proof generation dominates transaction generation cost.

Lightweight Verifier Circuit from Accumulation Scheme. Inspired by Halo [13], 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 the circuit, we output the $2\mathbb{G}_1$ points as public inputs and attach them 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 the ledger maintainer who receives the transaction validity proof and the two masked points.

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 savings 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 for SNARKs such as PLONK [27].

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 constitute the proving keys. The key observation is that with additive homomorphic polynomial commitment schemes (such as [33]), the commitment to the addition of two polynomials is simply the addition of their polynomial commitments.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 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 the 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 restricts 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 the proof batching technique to achieve a lower amortized cost. We leverage the fact that most universal SNARKs are cryptographically compiled from a *Polynomial Interactive Oracle Proof* (PIOP) using a PCS and

[6] Note that this is not a unique problem for universal SNARKs, as many non-universal counterparts [29, 31] also need pairing checks.

many choices of PCS support batch opening which reduces opening proof size and amortizes verification cost. Thus, we present a generic compiler in § 2.3 to transform a PIOP-based SNARK into a batched prover and verifier for a list of NP relations.

Variable-base MSM via Online Lookup Table. Instead of naïvely enforcing 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 [40] 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 a lookup table, any c-bit scalar multiplication on this specific base point becomes a table query rather than an elliptic curve group operation. Given that these bases are unfixed, the lookup table cannot be pre-processed – table values are only known during the online proving phase. Such online lookup tables are already possible with [26] although their presentation is limited to preprocessed query tables whose values are known ahead of time. We provide detailed gadget descriptions and circuit sizes in § 2.4.

Polynomial Evaluation over Non-native Field. To facilitate polynomial evaluation over a non-native field, we device efficient modular multiplication and modular addition gadgets by leveraging range check via lookup argument [26]. Compared to other modular arithmetic gadget designs, ours take advantage of (a) clever 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 § 2.5.

SNARK-friendly Symmetric Primitives. To reduce the number of non-algebraic operations in the circuit, we instantiate symmetric primitives used such as commitment schemes, Pseudorandom Function (PRF), and Collision-resistant Hash (CRH) with SNARK-friendly candidates which are designed to work natively with finite fields involving mostly algebraic operations. We specify our concrete implementations in § 3.1, most of which are based on Rescue hash functions [2]. More importantly, we carefully design customized gates in our TurboPlonk (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 the circuit. We further designed an optimized predicate commitment gadget in § 2.6 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.

1.4 Related Works

We refer readers to Section 1.2 in [12] for a comprehensive literature review on privacy-preserving computation on ledgers. Even though there are alternative private smart contract designs proposed afterwards [4, 34, 41–43] with different tradeoffs and limitations, Zexe remains the only concrete construction of DPC to date. DPC schemes like Zexe and VeriZexe have both data privacy and function privacy while maintaining high expressiveness.

Universal SNARKs. Our construction makes heavy use of Universal SNARKs [30], which strikes a good balance between efficiency and acceptable trust assumption. These systems support a universal and constantly *updatable* SRS [30] where anyone can contribute to the SRS in a verifiable way. As long as one of the contributors is honest, then no trapdoor exists. SNARKs with fully transparent setups [5, 15, 17] usually have worse performance or a much larger proof size.

We choose variants of Plonk [27] for our implementations primarily due to their performance, customizable gates, and importantly its support for lookup argument [11, 26] that some of our optimization techniques depend on. We provide a more detailed literature review of Universal SNARKs in the extended version of this paper.

2 VeriZexe: Practical Zexe with Universal SNARKs

To tackle the challenges of efficiently instantiating the DPC scheme with universal SNARKs described in Sec. 1.3, we propose numerous optimization techniques many of which can be applied to protocols beyond DPC. With all optimizations applied, we expect to bring VeriZexe to the realm of practicality. Detailed benchmark is reported in Sec. 3.

2.1 Lightweight Verifier Circuit from Accumulation Scheme

We apply a technique called *Accumulation Scheme* (AS), originally introduced in [13] and later generalized in [16], to move the expensive pairing check out of the SNARK verifier circuit. While the technique is not new, we try to cast part of the transaction generation procedure into an *incrementally verifiable computation* (Appx. A) and show explicitly how accumulation schemes can improve the performance of Zexe.

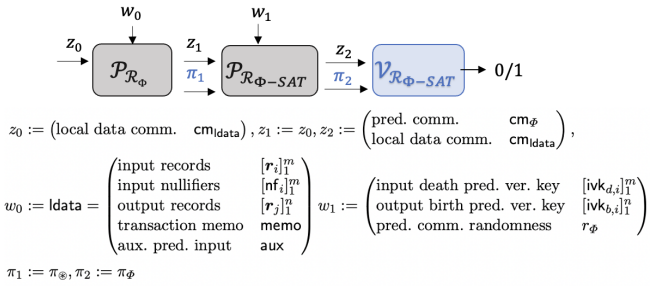

Figure 1: Casting $\mathcal{R}_\Phi$ proving into a two-step IVC (with different step function at each step).

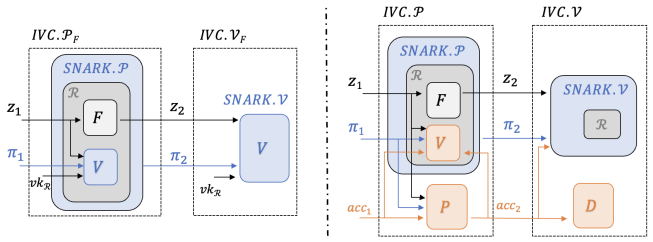

Figure 2: IVC from SNARK compositions (left) v.s. IVC from accumulation schemes (right). Blue boxes are SNARK prover $\mathcal{P}$ and verifier $\mathcal{V}$ for the relation $\mathcal{R}$, and orange boxes are accumulation prover P, verifier V (more lightweight than a SNARK verifier) and decider D. 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.

The core observations are that (i) proving satisfiability of user-defined predicates can be modeled as a two-step IVC, and (ii) original ZEXE instantiated this IVC using SNARK composition. To obtain a more lightweight IVC prover, we construct this IVC using a SNARK with an accumulation scheme. The key observation is that the verification of many universal SNARKs under the PIOP+PCS paradigm is efficient except for the final polynomial opening check. And with the help of an AC for the PCS, these opening checks can be separated from the SNARK verifier logic and delayed to another *decider* algorithm executed much later.

Modeling DPC executions as IVCs. As introduced in § 1.2, one of the proofs generated during transaction building is for the NP relation $\mathcal{R}_\Phi$ (for condition ©). As shown in Fig. 1, the 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 witnesses and running the 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 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 § 2.3), the IVC proof from the first step will be a single batched proof denoted as $\pi_\circledast$ instead of a list of predicate proofs.

ZEXE: IVC from SNARK composition. Next, we explain how the original ZEXE instantiates this IVC using SNARK composition (see the 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, the pairing checks in KZG for any PIOP+KZG universal SNARK are very expensive in the circuit and will slow 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 circuit (see the right half of Fig. 2).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 (transaction verification on-chain 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).

We emphasize that the accumulation must be zero-knowledge – the accumulator acc_2 and the 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 randomly masked and the ran-

domizer is an additional secret witness for the accumulation verifier. Note that authors of [16] already showed how to make accumulation schemes for inner-product-argument-based and pairing-based PCSs zero knowledge in their Appendix A and Section 8.

2.2 Instance Merging

Recall that a transaction builder 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 SNARK (Appx. A) and the homomorphism of polynomial commitment schemes (Appx. A.2), thus halving the number of proofs the outer circuit needs to verify.

Technique. In a SNARK based on polynomial IOP (such as *Algebraic Holographic Proof* (AHP) in MARLIN, and *idealized low-degree protocol* in PLONK), the preprocessing of circuit 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 arithmetization, for a birth predicate circuit $\mathcal{C}_1$ of size n, we pad the circuit to size of $2n$, with $\mathcal{C}_1$ being right padded (last n gates are dummy), and compute the proving key and verification key as usual; for a death predicate circuit $\mathcal{C}_2$ of size n, we perform similar operations but left pad the circuit (first n gates are dummy). Subsequently, whenever we want to merge $\mathcal{C}_1$ and $\mathcal{C}_2$, 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 the addition of two polynomials of the same degree, and the *merged* verification key (i.e. the commitments) can be similarly derived thanks to the additive homomorphism of PCS (such as KZG10 and its variants).

Syntax. We proceed to propose a slightly modified syntax for SNARKs that support instance merging. A *k-Mergeable SNARK* scheme

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

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 the merging of a death and a birth predicates into one.

- $\mathsf{srs} \leftarrow \mathsf{SNARK}_{\oplus}.\mathcal{G}(\lambda, N)$: same as $\mathsf{SNARK}.\mathcal{G}$ except $N = 2n$ where n is the size bound for each instance.
- $(\mathsf{ipk}_b, \mathsf{ivk}_b) \leftarrow \mathsf{SNARK}_{\oplus}.\mathcal{I}^{\mathsf{srs}}(\Phi_b, b)$: 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 $\mathcal{R}_{\oplus} := \{(\mathbb{X}_0 || \mathbb{X}_1, \mathbb{W}_0 || \mathbb{W}_1) : \phi_0(\mathbb{X}_0, \mathbb{W}_0) = 1 \wedge \phi_1(\mathbb{X}_1, \mathbb{W}_1) = 1\}$.
- $\mathsf{ipk} \leftarrow \mathsf{SNARK}_{\oplus}.\mathcal{M}_{\mathsf{ipk}}(\mathsf{ipk}_0, \mathsf{ipk}_1)$: Given any two complementarily slotted proving keys $\mathsf{ipk}_0, \mathsf{ipk}_1$, it outputs a merged proving key ipk.
- $\mathsf{ivk} \leftarrow \mathsf{SNARK}_{\oplus}.\mathcal{M}_{\mathsf{ivk}}(\mathsf{ivk}_0, \mathsf{ivk}_1)$: Given any two complementarily slotted verifying keys $\mathsf{ipk}_0, \mathsf{ipk}_1$, it outputs a merged verifying key ivk.
- $\mathbb{W} \leftarrow \mathsf{SNARK}_{\oplus}.\mathcal{M}_{\mathbb{W}}(\mathbb{W}_0, \mathbb{W}_1)$: 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 $\mathcal{R}_{\oplus}$.
- $\pi \leftarrow \mathsf{SNARK}_{\oplus}.\mathcal{P}(\mathsf{ipk}, \mathbb{X}, \mathbb{W})$: same as $\mathsf{SNARK}.\mathcal{P}$ except $N = 2n$.
- $b \leftarrow \mathsf{SNARK}_{\oplus}.\mathcal{V}(\mathsf{ivk}, \mathbb{X}, \pi)$: same as $\mathsf{SNARK}.\mathcal{V}$ except $N = 2n$.

We present a concrete construction of such a technique for PLONK in the extended version of this paper.

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 SNARK 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 C_n constraints; while a verifier gadget for a circuit of size $2n$ takes $C_{2n} = C_n + \delta$ constraints. Our instance merging techniques effectively reduce the outer circuit complexity from roughly $2m \cdot C_n$ to $m \cdot C_{2n}$, which is a significant saving as long as $\delta \ll C_n$. 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 C_n is orders of magnitude larger. Meanwhile, inevitably there is an 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.

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

- $\pi_{\circledast} \leftarrow \mathsf{SNARK}.\mathcal{P}_{\circledast}([\mathsf{ipk}_i]_{i=1}^{\ell}, [\mathbb{X}_i]_{i=1}^{\ell}, [\mathbb{W}_i]_{i=1}^{\ell})$: Given a list of ℓ proving keys, instances and witnesses, it proves them in batch and outputs a proof $\pi_{\circledast}$.
- $b \leftarrow \mathsf{SNARK}.\mathcal{V}_{\circledast}([\mathsf{ivk}_i]_{i=1}^{\ell}, [\mathbb{X}_i]_{i=1}^{\ell}, \pi_{\circledast})$: 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 the extended version of this paper.

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 preprocessing argument with universal SRS for the same relation (see Theorem 1 in [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 proof size and amortizes verification costs. We observe that many existing PCSs have a *linear combination scheme*, and thus support batch openings of multiple polynomials at multiple points (proven in Theorem 3 of [10] 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 the 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 returns 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 $\mathsf{PCS.Eval}.\mathcal{V}$.

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).

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

We generalize the lookup table argument in [26] by enabling a variant we call *online lookup table* to constrain MSM in the 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 computations where the bases

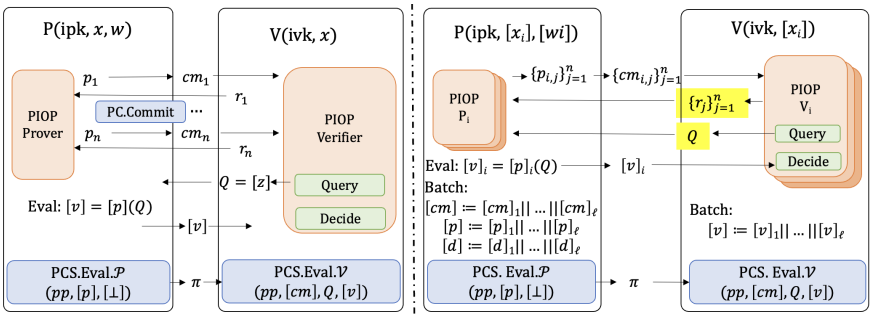

Figure 3: Generic compiler for batching PIOP-based SNARKs.

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 [40][7] 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,[8] 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 a lookup table, any c-bit scalar multiplication on this specific base point becomes a table query rather than an elliptic curve group operation. Given that these bases are unfixed, the lookup table cannot be pre-processed – table values are only known during the online proving phase which gives rise to our following technique.

Pre-processed v.s. Online Lookup Table. PLOOKUP [26] presents a polynomial IOP (PIOP) protocol for checking values of a *query table* $f := (f_1, \ldots, f_n) \in \mathbb{F}^n$ are contained in the values of a *lookup table* $t := (t_1, \ldots, t_d) \in \mathbb{F}^d$. 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 [26] 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 the online phase of the protocol run. Intuitively, the PIOP for online lookup tables still preserves 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.

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

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:

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

As a point of reference, with the TURBOPLONK circuit used to generate benchmark number in § 3, 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 $230,000$[9] – our optimized circuit is more than 6.5 factors smaller.

2.5 Polynomial Evaluation over Non-native Field

Inner proofs for predicate satisfiability and outer proofs for inner proof correctness are generated by circuits over different

[7] We only use a special case of a simplified Pippenger algorithm which is sometimes referred to as "the bucket method". For a detailed literature review and comparisons among different variants of Pippenger's predecessors, please see [7].

[8] 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" $\{S_1, \ldots, S_{2^c-1} \in \mathbb{G}\}$ 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 from maintaining $2^c - 1$ bucket sums and selectively updating the correct bucket sum for each base and its scalar – which is trivial outside the circuit, but expensive to enforce inside the circuit.

[9] Roughly, a naïve variable-base MSM can be done by decomposing the scalars to binary representation, then performing 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 $229,632$.

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 $Q = \sum_{j=0}^{m-1} \mathsf{wsum}_j \cdot 2^{cj}$.

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

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 $x_0,\ldots,x_{\ell-1} \in [0,K)$.

Figure 5: Range proof gadget.

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 a lookup table introduced in [26].

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 the 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 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:

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

for some c'_0, c'_1 carriers bounded by $-2^m \leq c'_0 < 2^{m+1}$ and $-2^{m+1} \leq c'_1 < 2^{m+2}$.[10] We present the modular multiplication gadget in Fig. 6 with the following notes:

[10] 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 $\frac{-2^m-2^{2m+1}}{2^m} = -1 - 2^{m+1} < -2^{m+1} < c'_1 < \frac{2^m+2^{2m+1}+2^{m+1}-0}{2^m} < 2^{m+2}$.

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 $2^{2m} = 2^{2\ell k} \geq p$ 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 $w_1 \cdot p_1 + (c_1 - 2^{m+1}) = x_1 \cdot y_1$

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

- To optimize gadget circuit size, we assume that the limbs of input x, y are already in the 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 the 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 the existence of witness $w \in \mathbb{F}_p$.[11]

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

See the proofs in the extended version of this paper.

Using the UltraPlonk constraint system specified in Def. 3, the circuit size of the 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.

[11] 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 $x \cdot y = z + p \cdot w$.

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 $y + p \cdot w = x_1 + \ldots + x_N$

Figure 7: Modular addition gadget.

Remark 1 *There are alternative non-native modular multiplication gadgets such as [24] that support more general field choices (e.g. it supports p, q where $q < p$). However, their designs render less efficient circuits. In comparison, we utilize the feature of two-chain curves for depth-2 proof recursion (instead of cycling curves for deeper proof recursions), which enables us to assume finer-grained requirements on p, q and 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 the foregoing modular multiplication gadget, we present the modular addition gadget in Fig. 7.

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

See the proofs in the extended version of this paper.

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.

2.6 SNARK-friendly Symmetric Primitives

Recall that the circuit for relation $\mathcal{R}_e$, which governs the rules of valid record creation and consumption, 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 involve many non-algebraic operations (*e.g.* bit-wise XOR, rotate in SHA256) which take lots of gates to constrain in an algebraic circuit. There are two main ways to constrain these primitives inside the circuit more efficiently:

1. precompute a lookup table containing legitimate (input, output) tuples[12] and the prover argues the witnesses (input and output of intermediate, non-algebraic steps) belong to the table [26].

2. use SNARK-friendly primitives specifically designed to work natively with finite field elements by using mostly algebraic computations (notably new hash functions: Rescue and Vision [2], Poseidon [28], MiMC [1]).

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 a lack of hardware acceleration. The former approach may allow better candidates with better security bounds or relies on weaker cryptographic assumptions.

Fiat-Shamir Transcript. Many SNARKs are made non-interactive by applying Fiat-Shamir transformation [23] 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 assumes these hash functions as random oracles. In practice, these random oracles are instantiated using Blake2s or the keccak permutation in SHA3, all of which incur 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 [22] designed a highly optimized circuit for SHA256 using lookup table with an overall cost of 2099 TURBOPLONK constraints; whereas CAP protocol [36] designed a CRH gadget by using Rescue permutation in a sponge construction with only 148 TURBOPLONK constraints. Granted that the arithemtizations in those two TURBOPLONK designs are slightly different, such numbers can only be used for informal comparison. We decide to use a 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.

[12] 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 $(x, y, x \oplus y)$.

Predicate Commitments. To ensure that death/birth predicates involved in $\mathcal{R}_{\mathsf{utxo}}$ and $\mathcal{R}_{\Phi}$ are consistent, [12] 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. Concretely, the original ZEXE instantiates 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 the extended version of this paper, we propose an efficient solution whose non-native operations do not grow regardless of the number of predicates committed.

3 Implementation and Evaluation

3.1 System Implementation

We implemented the DPC scheme and applied all optimizations (§ 2) except the predicate commitment technique.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, uses the following stack: we utilized `arkworks` library [3] as the underlying algebra backend for finite fields, elliptic curves, 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.

We break down our concrete instantiations of cryptographic building blocks used to generate benchmarks in § 3.2 in Appx. C.

3.2 Experimental Evaluation

Metrics and evaluation methodology. As an instantiation of the DPC scheme, our measurements focus on the resources required (including time, memory usage, and storage) during the execution of the three main algorithms of a DPC scheme (namely system setup, transaction generation, and verification). 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 transaction generation – which directly affects the

	System Setup		Transaction Generation				Verification	
			$\mathcal{R}_\Phi$ (outer circuit)					
Tx. Dim.	**Time** (s)	**SRS size** (MB)	**Constraints**	**Prover** (s)	**Time** (s)	**Memory** (GB)	**Verifier** (ms)	**Proof Size** (KB)
snarkVM testnet-2			R1CS					
2×2	176.8	5,254.2	4,235,068	138.5	151.4	22.8	15	0.482
3×3	246.0	7,056.6	6,330,496	202.7	223.0	26.8	21	0.482
4×4	370.1	10,454.9	8,447,588	293.2	321.1	40.6	21	0.482
VERIZEXE			ULTRAPLONK					
2×2	11.8	33.1	87,176	13.1	16.9	6.6	18	4.138
3×3	18.4	66.2	126,076	24.7	29.2	8.8	18	4.138
4×4	19.1	66.2	141,492	24.8	32.4	9.3	18	4.138

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). The "Prover" column refers to the prover time for the outer circuit whereas the "Time" column refers to the overall transaction generation time. snarkVM uses Groth16 for both $\mathcal{R}_{\mathsf{utxo}}$ and $\mathcal{R}_\Phi$, Marlin for birth/death predicates; whereas VERIZEXE uses TURBOPLONK for both $\mathcal{R}_{\mathsf{utxo}}$ and birth/death predicates, ULTRAPLONK for $\mathcal{R}_\Phi$. Notice that the SRS size for snarkVM contains the universal SRS of Marlin and preprocessed Groth16 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 the number of constraints reported for snarkVM are referring to R1CS constraints whereas the number for VERIZEXE are ULTRAPLONK constraints. All death and birth predicates require 2^{15} constraints in their respective constraint systems.

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 have mild or no effect on, such as 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 metrics. To the best of our knowledge, the most efficient and actively maintained implementation is *snarkVM* by the Aleo team many of whom are the co-authors of [12]. While there are a few versions of DPC instantiations inside snarkVM, we focus on its testnet-1 (the same implementation in Section 9 of [12]) and testnet-2 versions (see Table 1). [13]

Here we outline 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 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 uses GM17 [31] for birth/death predicates each of which requires a trusted setup. SnarkVM testnet-2 uses universal SNARK Marlin [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 $10.6\sim11.8$x improvement on outer proof generation, and a $9\sim10$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 is relatively low which results in faster proof generations. We also observe a $3\sim4.3$x improvement in 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 increasing the cost across the board from universal SRS generation to proving key indexing to proving[14].

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 difference in SRS size – snarkVM has a much larger SRS since it requires storing 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 joins the system.

Microbenchmarks. Since most of our techniques are attempts to reduce the outer circuit complexity, we now provide a microbenchmark on concrete circuit costs for major

[13] 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, 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 are outside the scope of this paper.

[14] We could further reduce the number of non-native operations when we implement the optimized predicate commitment in § 2.6.

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 (input: $\mathbb{F}^\ell$, output: $\mathbb{F}^k$)	native over BLS	$(\lceil \frac{\ell}{3} \rceil + k - 1) \cdot n_r + 4$
	native over BW	$(\lceil \frac{\ell}{3} \rceil + k - 1) \cdot n_p + 4$
Commitment (input: $\mathbb{F}^\ell$)	native over BLS	$\lceil \frac{\ell+1}{3} \rceil \cdot n_r + 4$
	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 TURBOPLONK constraints (see Def. 2), unless annotated with $*$ which refers to 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 $\mathbb{F}_p$.

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 § 2.5) for polynomial evaluation over non-native field and compact variable-based MSM gadget (see § 2.4). To illustrates the improvement attributed to our Pippenger-based vMSM gadget relying on the online lookup table technique, we provide a benchmark against a naïve implementation in Fig. 8.

Practicality. With significant improvements in memory usage, DPC transaction generations are possible on consumer-grade laptops or even on phones for the first time. As illustrated in Table 4, there is a general trade-off between prover time and peak memory usage – more cores enable higher parallelism which leads to faster proof generation at the cost of higher memory usage partially due to the overhead from multi-threading management.

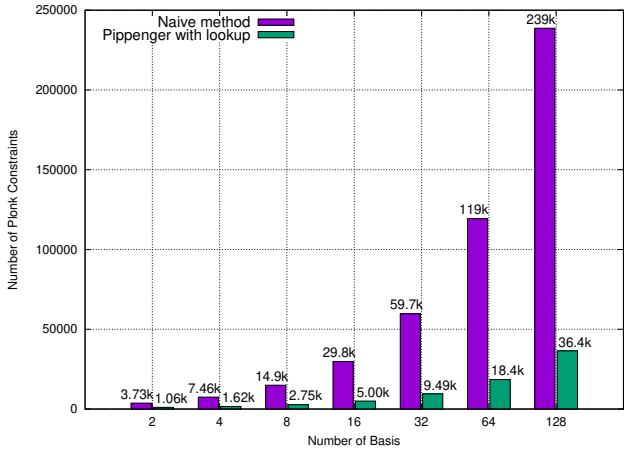

Figure 8: Circuit complexity for variable-based MSM.

	Inner Proofs		Outer Proof	
	Prover (s)	**Memory** (GB)	**Prover** (s)	**Memory** (GB)
Phone	46.3	1.1	270.0	2.6
Laptop	5.8	1.8	32.2	3.4
Server	3.2	5.0	12.7	6.6

Table 4: Proof generation time and memory usages for 2x2-transactions across different hardware environments. The first row simulates a phone environment with 4 CPU, 8GB RAM at 2.3 GHz. The second row simulates a customer-grade laptop environment with 16 CPU, 32GB RAM at 2.5 GHz. The third row simulates a powerful server environment with 64 CPU, 128GB RAM at 2.95 GHz.

Another observation is that inner proofs generations are easily manageable even for lower-resource hardware whereas the outer-proof generation is much more demanding. In quest of a balance between privacy and speed, resource-limited devices could produce inner proofs on-device, preserving the data privacy (all record states), and outsource the outer proof to a more powerful server, leaking only the predicates used in this transaction and nothing else. Note that the final transaction is still completely private to the world, we only sacrifice function privacy to the server. Alternatively, one could also use *Delegable DPC* (Sec.5 of [12]) which enables delegation of the entire transaction generation to untrusted workers who will learn about all transaction details but never produce valid transactions with invalid witnesses or without user's authorization. Our open-sourced VERIZEXE implementation supports Delegable DPC.

Threat Model and Security Proof. We emphasize that VERIZEXE is a concrete efficient construction of the DPC scheme, thus inherits all of its threat models, security properties, and DPC model level security proof. As long as our instantiations of cryptographic building blocks satisfy the necessary properties, then the ideal functionalities and security goals of DPC will be achieved.

The extra cryptographic assumptions, compared to [12], are Rescue permutation in Appx. C as a secure Pseudorandom Permutation and Rescue-based Hash as a random oracle.

Furthermore, our ULTRAPLONK are constraint system designs, not modifications to the underlying PLONK PIOP. The knowledge soundness error is proportional to the maximum degree of each gate times the number of gates. In our case, the maximum degree is 6 v.s. 2 in vanilla PLONK circuit; over the 256-bit field, the security difference is negligible (from Schwartz-Zippel lemma)

We will include more detailed security proofs in the extended version of this paper.

References

[1] 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.

[2] 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.

[3] arkworks contributors. `arkworks` zksnark ecosystem. https://arkworks.rs, 2022.

[4] 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.

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

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data from accumulation schemes. *IACR Cryptol. ePrint Arch.*, 2020:499, 2020.

[17] 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.

[18] 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.

[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.

[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.

[21] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *CoRR*, abs/1904.05234, 2019.

[22] The halo2 book. https://zcash.github.io/halo2/index.html. Accessed: 2022-04-26.

[23] 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.

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

[25] Ariel Gabizon, Zac Williamson, and Tom Walton-Pocock. Aztec yellow paper. https://hackmd.io/@aztec-network/ByzgNxBfd. Accessed: 2022-09-26.

[26] Ariel Gabizon and Zachary J Williamson. plookup: A simplified polynomial protocol for lookup tables. *IACR Cryptol. ePrint Arch.*, 2020:315, 2020.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, Dec 2008. Accessed: 2015-07-01.

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

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

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

[42] 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.

[43] 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.

[44] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

A Cryptographic Primitives: Definitions and Security Properties

Formal definitions and security properties of *Pre-processing SNARK with Universal SRS*, and *Incrementally Verifiable Computation* (IVC) will appear in the extended version of this paper.

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:

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

  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:

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

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

A.2 Polynomial Commitment Scheme

Introduced in [33], *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:

- $(\perp, b) \leftarrow \mathsf{PCS.Eval}(\mathcal{P}(\mathsf{pp}_{\mathsf{PCS}}, f, r), \mathcal{V}(\mathsf{pp}_{\mathsf{PCS}}, \mathbf{cm}, z, y))$ 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 $\mathsf{pp}_{\mathsf{PCS}}$ 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 $\mathcal{A}$:

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

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

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

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 [33]).

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 hold 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*.

B PLONK 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 the number of public inputs respectively.

Definition 1 (Plonk indexed relation) *The indexed relation $\mathcal{R}_{\mathsf{plonk}}$ is the set of all triples:*

$$\left(\mathbb{I} = (\mathbb{F}, n, m, \ell, a, \mathcal{Q}), \mathbb{X} = (w_j)_{j\in[\ell]}, \mathbb{W} = (w_j)_{j\in[\ell+1,m]}\right)$$

where the index vector $a \in [m]^{3n}$, selectors are $\mathcal{Q} := (q_L, q_R, q_O, q_M, q_C) \in (\mathbb{F}^n)^5$, such that $\forall i \in [n]$,

$$(q_L)_i \cdot w_{a_i} + (q_R)_i \cdot w_{a_{n+i}} + (q_M)_i \cdot w_{a_i} w_{a_{n+i}} + (q_C)_i + \mathsf{PI}_i = (q_O)_i \cdot w_{a_{2n+i}}$$

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 TURBOPLONK constraint system that allows for customized gates beyond just addition and multiplication gate. However, we note that TURBOPLONK proof system has a higher per-gate cost for proof generation and higher fan-in resulting in a 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 are 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:*

$$\left(\mathbb{I} = (\mathbb{F}, n, m, \ell, a, \mathcal{Q}), \mathbb{X} = (w_j)_{j\in[\ell]}, \mathbb{W} = (w_j)_{j\in[\ell+1,m]}\right)$$

where the index vector $a \in [m]^{5n}$, selectors are $\mathcal{Q} := (q_1, q_2, q_3, q_4, q_{M_{1,2}}, q_{M_{3,4}}, q_O, q_C, q_{H_1}, q_{H_2}, q_{H_3}, q_{H_4}, q_{ecc}) \in \mathbb{F}^{n\times 13}$, such that $\forall i \in [n]$,

$$\begin{aligned}(q_O)_i \cdot w_{a_{4n+i}} = {} & (q_1)_i \cdot w_{a_i} + (q_2)_i \cdot w_{a_{n+i}} \\ & + (q_3)_i \cdot w_{a_{2n+i}} + (q_4)_i \cdot w_{a_{3n+i}} \\ & + (q_{M_{1,2}})_i \cdot w_{a_i} w_{a_{n+i}} + (q_{M_{3,4}})_i \cdot w_{a_{2n+i}} w_{a_{3n+i}} \\ & + (q_{H_1})_i \cdot w_{a_i}^5 + (q_{H_2})_i \cdot w_{a_{n+i}}^5 \\ & + (q_{H_3})_i \cdot w_{a_{2n+i}}^5 + (q_{H_4})_i \cdot w_{a_{3n+i}}^5 \\ & + (q_{ecc})_i \cdot w_{a_i} w_{a_{n+i}} w_{a_{2n+i}} w_{a_{3n+i}} w_{a_{4n+i}} \\ & + (q_C)_i + \mathsf{PI}_i\end{aligned}$$

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 [26] with the previous TURBOPLONK 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 require 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 $a_{\mathsf{rg}} \in [m]^n$, such that for each $i \in [n]$, the witness value $w_{(a_{\mathsf{rg}})_i}$ is in the range table $\mathcal{T}_{\mathsf{rg}}$.
- To support multiple online lookup tables[15], each containing key-value tuples where the "keys" are scalars and the "values" are affine point variables (*i.e.* two variables for the x and y coordinates)[16], 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 $q_K \in \mathbb{F}^n$ to indicate whether a gate is performing online table entry insertion and query table insertion.
  3. Two domain separator selectors q_{lt}, q_{qt} for indicating the exact lookup sub-table and query sub-table an entry in the final merged table belongs to.

  More precisely, the i-th entry in our merged online lookup table is a key-value tuple $\mathcal{T}_i := (q_K)_i \cdot \left[(q_{lt})_i, (\mathcal{T}_{key})_i, w_{a_{3n+i}}, w_{a_{4n+i}}\right]$; the i-th entry in our merged online query table is a key-value tuple $\mathcal{Q}_i := (q_K)_i \cdot \left[(q_{qt})_i, w_{a_i}, w_{a_{n+i}}, w_{a_{2n+i}}\right]$. 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*

$$\mathbb{I} = (\mathbb{F}, n, m, \ell, a, a_{\mathsf{rg}}, \mathcal{Q}, \mathcal{T}_{\mathsf{rg}}, \mathcal{T}_{key})$$
$$\mathbb{X} = (w_j)_{j\in[\ell]}$$
$$\mathbb{W} = (w_j)_{j\in[\ell+1,m]}$$

where the TURBOPLONK *index vector* $a \in [m]^{5n}$*, the index vector for the range wire:* $a_{\mathsf{rg}} \in [m]^n$*, selectors are:* $\mathcal{Q} := (q_1, q_2, q_3, q_4, q_{M_{1,2}}, q_{M_{3,4}}, q_O, q_C, q_{H_1}, q_{H_2}, q_{H_3}, q_{H_4}, q_{ecc}, q_K, q_{lt}, q_{qt}) \in \mathbb{F}^{n\times 16}$*, such that:*

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

$$\mathcal{Q}_i := (q_K)_i \cdot \left[(q_{qt})_i, w_{a_i}, w_{a_{n+i}}, w_{a_{2n+i}}\right]$$

is in the lookup table

$$\mathcal{T} := \left\{\mathcal{T}_j = (q_K)_j \cdot \left[(q_{lt})_j, (\mathcal{T}_{key})_j, w_{a_{3n+j}}, w_{a_{4n+j}}\right]\right\}_{j\in[n]}$$

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

C System Implementation Details

Elliptic Curves. We use two pairing-friendly elliptic curves $E_{\mathsf{BLS}}, E_{\mathsf{BW}}$ in a similar fashion as [12] 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 [32] (outer circuits are over its scalar field which matches the base field of BLS12-377). Additionally, for some cryptographic primitives that require 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 pseudorandom permutation using *Rescue* algorithm [2].

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 $m, m' \in \mathbb{F}^4$.

[15] We merged multiple sub-tables into a single one by adding an 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.

[16] 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 variables.

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 [37]) but here is simplified to output a single field element. The PRF takes a secret key k of one field element, a message 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 into chunks of Rescue's state size, and absorbs 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[17].

CRH. We build our collision-resistant hash (CRH) using the Sponge construction [8] 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 the length is multiple of the rate. In short, we have a family of CRH that supports $H : \mathbb{F}^* \mapsto \mathbb{F}$ for an 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 the prevention of prefixing attacks and other formal security proofs, please refer to Section 4.1.8 in [36].

Commitments. We build a Rescue-based commitment scheme that takes in a message $m \in \mathbb{F}^n$ of some fixed length n and a randomly sampled blinding factor s, outputs a hiding commitment $c \leftarrow \mathsf{Commit}_n(s,m) = \mathsf{CRH}(s\|m\|0)$ where 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 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.

SNARK. We instantiate the SNARKs using KZG-based PLONK [27]. Concretely, our predicate circuit and circuit for the relation $\mathcal{R}_{\mathsf{utxo}}$ uses our TURBOPLONK 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 ULTRAPLONK 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. Note that our inner circuits don't need to be zero-knowledge, only the outer circuit requires zero-knowledge to reveal nothing about the predicate verification keys used in order to achieve the function privacy of a DPC scheme.

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