Succinct Zero-Knowledge Batch Proofs for Set Accumulators[*]

Matteo Campanelli
Protocol Labs
San Francisco, USA
matteo@protocol.ai

Dario Fiore
IMDEA Software Institute
Madrid, Spain
dario.fiore@imdea.org

Semin Han
Hanyang University
Seoul, Korea
seminhan@hanyang.ac.kr

Jihye Kim
Kookmin University
Seoul, Korea
jihyek@kookmin.ac.kr

Dimitris Kolonelos
IMDEA Software Institute, Spain
Universidad Politecnica de Madrid
Madrid, Spain
dimitris.kolonelos@imdea.org

Hyunok Oh
Hanyang University
Seoul, Korea
hoh@hanyang.ac.kr

ABSTRACT

Cryptographic accumulators are a common solution to proving information about a large set S. They allow one to compute a short digest of S and short certificates of some of its basic properties, notably membership of an element. Accumulators also allow one to track set updates: a new accumulator is obtained by inserting/deleting a given element. In this work we consider the problem of generating membership and update proofs for *batches* of elements so that we can succinctly prove additional properties of the elements (i.e., proofs are of constant size regardless of the batch size), and we can preserve privacy. Solving this problem would allow obtaining blockchain systems with improved privacy and scalability.

The state-of-the-art approach to achieve this goal is to combine accumulators (typically Merkle trees) with zkSNARKs. This solution is however expensive for provers and does not scale for large batches of elements. In particular, there is no scalable solution for proving batch membership proofs when we require zero-knowledge (a standard definition of privacy-preserving protocols).

In this work we propose new techniques to efficiently use zkSNARKs with RSA accumulators. We design and implement two main schemes: 1) HARiSA, which proves batch membership in zero-knowledge; 2) B-INS-ARiSA, which proves batch updates. For batch membership, the prover in HARiSA is orders of magnitude faster than existing approaches based on Merkle trees (depending on the hash function). For batch updates we get similar cost savings compared to approaches based on Merkle trees; we also improve over the recent solution of Ozdemir et al. [USENIX'20].

1 INTRODUCTION

Blockchains are decentralized and distributed systems in which a vast network of nodes maintain, distributed and replicated, a digital ledger. Core to blockchains is the maintenance of the global state of the system across its nodes. This state is usually large and is encoded in data structures such as a UTXO set (unspent transaction outputs, intuitively the unspent coins) in Bitcoin and Zcash [10, 67], the set of account-balances in Ethereum, or the set of identities in Decentralized Identity (DID) systems (e.g., Iden3, Sovrin, Hyperledger Indy) [62, 63, 66]. In these systems executing a transaction typically involves two steps, one "local" and one "global": *(i)* checking a given property with respect to the current state (e.g., the transaction is properly signed, some coins are spendable, some credentials exist), and *(ii)* modifying the global state (e.g., updating balances, adding a credential) and checking its correct update. The validity checks that are local to the transaction can for example involve checking a digital signature. Checking against the global state typically translate into *set-membership* ($x \in S$) or *set-update* ($S' \overset{?}{=} S \setminus \{x\} \cup \{x'\}$).

Blockchain systems grow through time and so do their global states (at the time of writing the UTXO set in Bitcoin is 4.6 GB). Verifying this state *at scale* is a challenging problem: every user, even one that only "passively" looks at the history of transactions, must re-execute them and store them to verify future ones.

A common approach to this problem uses authenticated data structures (ADS) [57], e.g., Merkle trees [47] as their most popular and deployed incarnation, or RSA accumulators [7, 11, 21, 45]. This idea [37, 54, 59] splits users into two groups. More "passive" users (aka verifiers) store only a *succinct digest* of the large set. A user proposing a transaction, on the other hand, has more information on the state (e.g., their account information) that it can use to *prove* either the membership of some elements, or the correctness of an update, with respect to the digest. This approach achieves scalable verification because ADS proofs are *succinct*, i.e., they are short and the time to verify is sublinear in the size of the set, e.g., it is logarithmic in Merkle trees, or constant in RSA accumulators.

While the efficiency benefits of this approach are clear, there are two additional challenges emerging in this space. They are the focus of our work.

1) *how to obtain privacy?* This is paramount whenever transactions data cannot be publicly exposed (e.g., to preserve anonymity or prevent front-running).

2) *how to improve throughput?* That is, the number of transactions we can process per unit of time.

The ADS-based approach above falls short on both issues. First, it generally requires that the global state is public. Second, it scales poorly when *proving many transactions*. Assume we want to batch prove m transactions at once, ADS either allow membership/update proofs for a single element only [47], or they have succinct batch proofs but the verifier must know and receive the elements [15]. This entails at least an $O(m)$ communication and verification time, affecting on-chain storage and work.

Both these problems can be solved via the use of *zkSNARKs* [13, 48], cryptographic proof systems that enable a prover to convince a verifier about the veracity of statements of the form "given a

---

[*] This is the full version of the paper that appears in the proceedings of ACM CCS 2022.

function F and a public input x, there is a secret w such that $F(x, w)$ is true". In particular, zkSNARK proofs are *zero-knowledge* and *succinct*. The former means that proofs do not reveal any information about the secret w and give solutions to the privacy challenge. For instance, in Zcash one proves the existence of a coin that is valid and part of the UTXO set, without revealing which is the coin so as to guarantee anonymity of a spend transaction. The other property, succinctness, means that proofs are short and efficient to verify, faster than the time it takes to execute F. This gives a solution to the throughput question above. The idea (known as zk-Rollup [8]) is that an aggregator can: collect a batch of m transactions; prove that they are valid; compute the updated global set and the corresponding digest; and finally broadcast the new digest with a succinct proof that its update is correct.

Even though zkSNARKs make verifiers' lives easier, the same cannot be said for provers. In these applications, the function to be proven includes the verification algorithm of an ADS which makes zkSNARK proving extremely expensive in terms of both computing time and RAM. To date the most deployed option is based on Merkle trees [47]. Proving their verification for a set of size n and a batch of m elements requires encoding about $\approx m \log n$ hash computations in the zkSNARK constraint system. Even by using hash functions that are optimized for zkSNARKs [1, 41], the proving time degrades very quickly (see section 6). In part, a reason of this cost is that Merkle trees do not allow batch openings.

RSA accumulators are a promising alternative as they enjoy constant-size batch proofs for membership and updates [15]. Yet, they have two potential limitations.

The first one is that generating a (batch) inclusion proof requires $O(n)$ RSA group operations, which is concretely more expensive than the analogous cost for Merkle trees, that is $O(n)$ hash computations. This issue, however, is heavily mitigated in several applications and does not represent a showstopper. For instance, in stateless blockchains [31, 60] a user can be provided the inclusion proof for her elements of the set (e.g., her account, identities) and does not need to recompute it from scratch, only to keep it updated, a task which in RSA accumulators can be performed at a constant-time cost per update.

The second limitation (and the focus of our work) has to do with applying SNARKs on them for succinct batch opening. The naive approach that encodes their verification as a constraint system for zkSNARKs is concretely prohibitive (verification consists of $O(m)$ RSA group operations becoming $\approx 1.8 \cdot m$ millions constraints).

Two recent papers [12, 50] consider this problem of efficiently combining RSA accumulators with zkSNARKs avoiding expensive encodings. Benarroch et al. [12] propose SNARKs for set-membership (proving that a committed element is in a large set, of which the verifier knows an RSA accumulator) but only support membership of a *single* element. Ozdemir et al. [50] propose a verifiable computation for batch *updates* of sets succinctly represented by RSA accumulators. Since in RSA accumulators, membership and updates are expressed through the same algebraic property, their techniques can be extended to support zero-knowledge *membership*. But this still entails a large fixed cost of 5 millions constraints to encode RSA group operations, in addition to $\approx 2 \cdot m$ thousands constraints, for a batch of size m. Also, for the problem of proving batch updates they improve over Merkle trees only beyond a fairly large threshold: 2^{20}-elements sets and batches of $\approx 1,300$ values.

1.1 Our work

We advance this research line by proposing new techniques to efficiently use zkSNARKs with RSA accumulators. We propose new, more scalable protocols for succinct zero-knowledge proofs of batch membership and updates. In what follows we give more details on our contributions.

Succinct proofs of batch membership. Our first result is a commit-and-prove [27] zkSNARK for batch membership, that is: given an RSA accumulator acc to a set $S = \{x_1, \dots, x_n\}$ and a succinct Pedersen commitment c_u to a vector of values $(u_1, \dots, u_m)$, it holds $u_i \in S$ for every $i = 1, \dots, m$. Thanks to the commit-and-prove feature, our scheme can be efficiently and modularly composed with other commit-and-prove[1] zkSNARKs [25] in order to prove further properties of the committed elements, e.g., $\forall i : u_i \in S \wedge P(u_1, \dots, u_m) = \mathsf{true}$ (P could be for example a numerical range check; see also our DID application in section 6.2.2). We dub our construction HARiSA[2].

Our technical contributions include: a new randomization method for RSA accumulator witnesses (needed to obtain zero-knowledge) and a new way to prove the accumulator verification in zero-knowledge in a SNARK *without* encoding RSA group operations in the constraint system. The latter is based on a novel combination of (non-succinct) sigma protocols, succinct proof of knowledge of exponent [15], and zkSNARKs for integer arithmetic.

Succinct proofs of batch insertion. Our second result concerns succinct proofs for batch insertion, that is: given two RSA accumulators acc, acc′ to sets S and S' respectively and a succinct Pedersen commitment c_u to $(u_1, \dots, u_m)$, it holds that $S' = S \cup \{u_1, \dots, u_m\}$.[3] We build our scheme for set insertions—dubbed B-INS-ARiSA[4]—by "scaling down" our techniques for set-membership, removing zero-knowledge and simplifying, finally obtaining a solution that is simpler and faster than our batch-membership scheme. Furthermore, following [50], we show how to use this scheme to obtain one for proving MultiSwaps, which in a nutshell means checking if S' is obtained by applying a sequence of "swaps" $\{(x_1, x'_1), \dots, (x_m, x'_m)\}$ (i.e., add x'_i, remove x_i) to S—essentially what we informally referred as set update. As shown in [50], proving MultiSwaps for accumulated sets has applications to verifiable outsourcing of state updates, applicable to Rollups [8] and efficient persistent RAM [17].

Implementation and evaluation. We implement our protocols[5] and evaluate them experimentally comparing with the state of the art. For zero-knowledge batch membership, we compare our solution with Merkle trees on two benchmarks: one that considers generic membership operations, and one that implements a DID application. For batch updates, we compare our MultiSwap solution with that of Ozdemir et al. [50] and with Merkle trees. We do the

---

[1] Roughly, the verification algorithm of the zkSNARK takes as input short commitments to a long (potentially private) input. This property is useful as the elements for which we prove set membership need to stay private, but still "referred to", e.g. for proving additional properties on them.

[2] HARiSA stands for "elements-Hiding Argument for RSA accumulators".

[3] More precisely, our schemes work with multisets.

[4] For Batch INSertion. It is pronounced as in the word *beans*.

[5] Part of implementation available at https://github.com/matteocam/libsnark-lego

latter comparison via a cost model analysis based on number of SNARK constraints. We also validate the case of set membership on a realistic application scenario (Decentralized Identity in section 6.2.2).

For batch membership, our experiments show that HARiSA saves at least *an order of magnitude in proving time* (depending on which hash function we use for Merkle trees in the comparison). As an example, proving batch-membership of 16 elements with SHA256 (resp. Poseidon) Merkle trees of depth 16 requires about 30 (resp. 1.5) minutes, while it requires less than 3 seconds with HARiSA. Our solution also enjoys >5× smaller public parameters than solutions based on Merkle trees, which also translates into less RAM consumption for the prover. A downside of HARiSA are slower verification and larger proofs; yet they remain competitive: verification takes ≈ 60ms (vs. 30ms for Merkle trees) and proofs are close to one kilobyte for any batch/set size.

For MultiSwaps, B-INS-ARiSA obtains similar improvements over Merkle-tree based solutions, i.e., more scalable prover and slightly worse verification and proof size. Also, B-INS-ARiSA surpasses Merkle-based swaps earlier than [50] (140 operations for 2^{20}-large sets).

2 TECHNICAL OVERVIEW

We now present a high-level overview of our main technical contributions.

Our core protocol is a succinct zero-knowledge proof of set membership for a batch of elements. Given a (public) set $S = \{x_1, \ldots, x_n\}$ and a commitment to $u_1, \ldots, u_m$, we aim at proving that $u_1, \ldots, u_m \in S$. We require for privacy that the u_i's remain hidden (and thus we provide them only as a commitment in the public input). We also require for efficiency that proof size and verification time should not depend either on the batch size m or the set size n.

We start from applying RSA accumulators [7] to compress the set into a succinct digest. Given random group element g in a group of unknown order (e.g. an RSA or class group [18]), one can produce a compressed representation of the set[6] as $\mathsf{acc} = g^{x_1 \cdot x_2 \cdots x_n}$. RSA accumulators enjoy succinct batch-membership proofs: to prove that $u_1, \ldots, u_m \in S$ it suffices to provide a single group element (a *witness*) $W = g^{\prod_{i \in [n]} x_i / \prod_{i \in [m]} u_j}$, which the verifier can check as $W^{u_1 \cdots u_m} = \mathsf{acc}$.

Though succinct, the batch-membership proofs of RSA accumulators do not hide the u_i elements, as the verifier should know them in order to perform the exponentiation. To address this problem, we could use a non-interactive zero-knowledge proof of exponentiation, which can be obtained using a Σ-protocol [34] (a three-message zero-knowledge scheme) made non-interactive through the Fiat-Shamir transform [38]. In it the prover computes: $R \leftarrow W^r$ for a sufficiently large random r; a random oracle challenge $h \leftarrow H(\mathsf{acc}||g||W||R)$, an integer $k \leftarrow r + h \cdot \prod_{i \in [m]} u_i$. The verifier accepts this zk-proof (R, h, k) if $h = H(\mathsf{acc}||g||W||R)$ and $R \cdot \mathsf{acc}^h = W^k$.

This protocol however does not yet achieve our goal, which is to generate a zero-knowledge batch-membership proof for committed u_i's. Towards this goal, we need to solve the following technical challenges. (A) The verifier needs to know the witness W, which can itself leak information about the elements it proves membership of. For example for $m = 1$ one can efficiently find the element u_1 by brute-force testing all elements of the set S, $W^{x_i} \overset{?}{=} \mathsf{acc}$ (recall that the set is public). The x_{i^*} for which the test passes will be u_1. (B) The proof (R, h, k) above simply shows existence of an exponent u such that $W^u = \mathsf{acc}$, in particular it does not link this statement to committed $(u_1, \ldots, u_m)$ such that $u = u_1 \cdots u_m$. (C) The proof is not succinct since the integer k is $O(m)$-bits long. (D) Most notably, the Σ-protocol described above is not even sound, unless the challenge is binary, $h \in \{0, 1\}$ [6, 58].

Our key contribution is an efficient technique to efficiently prove the verification of this Σ-protocol using a SNARK. Notably, we do not need encode any RSA group operations in the SNARK constraint system[7]. To obtain this result we combine three main ideas:

(1) We introduce a novel randomization method for an RSA accumulator witness, $W \mapsto \hat{W}$, so that $\hat{W}$ provably doesn't leak any information about the u_i's.
Our hiding-witness transformation works as follows: let $p_1, \ldots, p_{2\lambda}$ be the first 2λ prime numbers. We always (artificially) add these primes to the accumulator, i.e., the accumulator of a set S is an RSA accumulator $\hat{\mathsf{acc}}$ of $\hat{S} \leftarrow S \cup \{p_1, \ldots, p_{2\lambda}\}$: $\hat{\mathsf{acc}} \leftarrow \mathsf{acc}^{p_1 \cdots p_{2\lambda}}$ (we assume that S does not contain any of the p_i's). Then, to produce the hiding witness $\hat{W}$ we raise to the exponent each prime p_i with probability 1/2. A bit more formally, we sample $b_i \leftarrow\$ \{0, 1\}$ and set $\hat{W} \leftarrow W^{\prod_{i \in [2\lambda]} p_i^{1-b_i}}$, b_i's should remain hidden. We formally prove that under a cryptographic assumption (DDH-II, a variant of DDH [26]) $\hat{W}$ is computationally indistinguishable from random and thus $\hat{W}$, alone, hides u_i's (see section 4.1). Notice that $\hat{W}$ can be verified through the equality $\hat{W}^{\prod_{i \in [m]} u_i \cdot \prod_{i \in [2\lambda]} p_i^{b_i}} = \hat{\mathsf{acc}}$. Therefore we can use the NIZK for exponentiation described above, but for base $\hat{W}$ and exponent $e := \prod_{i \in [m]} u_i \cdot \prod_{i \in [2\lambda]} p_i^{b_i}$.
This technique solves the challenge (A) as it turns an RSA accumulator verification into a ZK verification. Yet challenges (B) and (C) remain: k is not short and the protocol only proves the existence of e such that $\hat{W}^e = \hat{\mathsf{acc}}$—which says nothing about membership of *legitimate* elements from S. For example e can contain only elements of $\{p_1, \ldots, p_{2\lambda}\}$ and no element from S.

(2) To solve (B) we "link" the Σ-protocol to $c_{\vec{u}}$, a commitment to all u_i's, by using a zkSNARK that proves the correct computation of k from the committed legitimate u_i's. Namely it proves that, for $c_{\vec{u}}$, a commitment to $\vec{u}$, and $c_{r,s}$, a commitment to integers $s = \prod_{i \in [2\lambda]} p_i^{b_i}$ and r, the equality $k = r + h \cdot s \cdot \prod_{i \in [m]} u_i$ holds over the integers and $u_i > p_{2\lambda}$ for each $i \in [m]$. Recall that $p_{2\lambda}$ is the largest of all p_i's, so $u_i > p_{2\lambda}$ translates to $u_i \neq p_j$ for all $j \in [2\lambda]$. This means that the exponent of e contains elements u_i's committed a-priori and that they are *legitimate* (not one of the artificially added p_i's).

(3) Although the above careful interplay between RSA Accumulators, Σ-protocols, zkSNARKs and our hiding technique for RSA accumulators witnesses gives a secure zero-knowledge proof of set membership, it is not yet succinct, as the verifier needs to receive the $O(m)$-long integer k. To solve this technical challenge, we apply a

---

[6] The elements of the set should be primes (or hashed to ones) for the RSA accumulator to securely apply.

[7] A "constraint system" is an encoding of the property proved by the SNARK. Its size, the number of constraints, is a key efficiency metric when evaluating proof schemes.

succinct proof of knowledge of exponent PoKE [15]. Instead of sending k, the prover sends $B = \hat{W}^k$ accompanied with a succinct proof that there is an integer k such that $B = \hat{W}^k$. Adding the PoKE proof however breaks the link between the Σ-protocol and the zkSNARK as the latter is supposed to generate a proof for a public k. To solve this last challenge, we "open the box" of PoKE verification and observe that the verifier receives the short integer $\hat{k} = k \mod \ell$, where ℓ is a random prime challenge of 2λ bits. Therefore, the last idea of our protocol is to let the zkSNARK prove the same statement as above but for $\hat{k}$, namely that $\hat{k} = r + h \cdot s \cdot \prod_{i\in[m]} u_i \mod \ell$.

A special mention needs to be made to (D), the soundness of the Σ-protocol. Standard impossibility results [6, 58] show that the Σ-protocols over groups of unknown order (as the groups of RSA accumulators) can have at most 1/2 soundness-error, meaning that they need many repetitions (e.g. $\lambda = 128$) to leverage them to fully sound (with negligible soundness-error). This usually makes the protocols prohibitively expensive.

The general intuition of the impossibility is that (using usual rewinding techniques) the extractor gets (R, h, k) and (R, h', k') such that $\mathsf{acc}^{h-h'} = W^{k-k'}$. However, we cannot imply to $acc = W^{(k-k')/(h-h')}$ because $(h-h')^{-1}$-in the exponent cannot be efficiently computed in groups of unknown order. So we are bound to set $h \in \{0, 1\}$ (so that $h - h' = 1$). In our solution, the zkSNARK proof described in (2) makes the extraction of the Sigma-protocol possible. This is possible because this proof guarantees that, in the two executions, $k = r + suh$ and $k' = r + suh'$, for committed r, s, u. This way, we get that $\mathsf{acc}^{h-h'} = W^{su(h-h')}$, from which we can conclude the desired result $\mathsf{acc} = W^{su}$.

Our technique of using a zkSNARK for the correct computation of the last message of a Σ-protocol over groups of unknown order, is generic for *any* such protocol and gives a way to efficiently bypass the impossibility results [6, 58] without inexpensive repetitions. We expect this to be of independent interest.

3 BACKGROUND

We give informal definitions for the main cryptographic primitives used in our constructions.

3.1 Commitments

Commitment schemes allow one to commit to a value, or a collection of values (e.g., a vector), in a way that is *binding*—a commitment cannot be opened to two different values—and *hiding*—a commitment leaks nothing about the value it opens to. In our work we also consider commitment schemes that are *succinct*, meaning informally that the commitment size is fixed and shorter than the committed value. Here is a brief description of the syntax we use in our work: $\mathsf{Setup}(1^\lambda) \rightarrow \mathsf{ck}$ returns a commitment key ck; $\mathsf{Comm}(\mathsf{ck}, x; o) \rightarrow c$ produces a commitment c on input a value x and randomness o (which is also the opening).

3.2 (Commit-and-Prove) SNARKs

Definition 3.1 (SNARK). A SNARK for a relation family $\mathcal{R} = \{\mathcal{R}_\lambda\}_{\lambda\in\mathbb{N}}$ is a tuple of algorithms $\Pi = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ with the following syntax:

- $\Pi.\mathsf{Setup}(1^\lambda, R) \rightarrow \mathsf{crs}$ outputs a relation-specific common reference string crs.
- $\Pi.\mathsf{Prove}(\mathsf{crs}, \mathbb{x}, \mathbb{w}) \rightarrow \pi$ on input crs, a statement $\mathbb{x}$ and a witness $\mathbb{w}$ such that $R(\mathbb{x}, \mathbb{w})$, it returns a proof π.
- $\Pi.\mathsf{Verify}(\mathsf{crs}, \mathbb{x}, \pi) \rightarrow b \in \{0, 1\}$ on input crs, a statement $\mathbb{x}$ and a proof π, it accepts or rejects the proof.

We require a SNARK to be complete, knowledge-sound and succinct. Completeness means that for any $\lambda \in \mathbb{N}, R \in \mathcal{R}_\lambda$ and $(\mathbb{x}, \mathbb{w}) \in R$ it holds with overwhelming probability that $\mathsf{Verify}(\mathsf{crs}, \mathbb{x}, \pi) = 1$ where $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda, R)$ and proof $\pi \leftarrow \mathsf{Prove}(\mathsf{crs}, R, \mathbb{x}, \mathbb{w})$. Knowledge soundness informally states we can efficiently "extract" a valid witness from a proof that passes verification. Succinctness means that proofs are of size $\mathsf{poly}(\lambda)$ (or sometimes $\mathsf{poly}(\lambda, \log|\mathbb{w}|)$) and can be verified in time $\mathsf{poly}(\lambda)\mathsf{poly}(|\mathbb{x}| + \log|\mathbb{w}|)$. A SNARK may also satisfy zero-knowledge, that is the proof leaks nothing about the witness (this is modeled through a simulator that can output a valid proof for an input in the language without knowing the witness). In this case we call it a zkSNARK. Whenever the relation family is obviously defined, we talk about a "SNARK for a relation R".

3.2.1 Commit-and-Prove SNARKs (CP-SNARKs). We use the framework for black-box modular composition of commit-and-prove SNARKs (or CP-SNARKs) in [25] and [12]. Informally a CP-SNARK is a SNARK that can efficiently prove properties of inputs committed through some commitment scheme C. In more detail, a CP-SNARK for a relation $R_{\text{inner}}(\mathbb{x}; u, \omega)$ is a SNARK that for a given commitment c can prove knowledge of $\mathbb{w} := (u, \omega, o)$ such that $c = \mathsf{Comm}(u; o)$ and $R_{\text{inner}}(\mathbb{x}; u, \omega)$ holds. We can think of ω as a non-committed part of the witness. In a CP-SNARK, besides the proof, the verifier's inputs are $\mathbb{x}$ and c.

Remark 1 (Syntactic Sugar for SNARKs/CP-SNARKs)**.** *For convenience we will use the following notational shortcuts. We make explicit what the private input of the prover is by adding semicolon in a relation and in a prover's algorithm (e.g., $R(\mathbb{x}; \mathbb{w})$). We explicitly mark relations as "commit-and-prove" by a tilde. We leave the assumed commitment scheme implicit when it's obvious from the context. Occasionally, we will also explicitly mark the commitment inputs by squared box around them (e.g. $\boxed{c_u}$) and we will assume implicitly that the relation includes checking the opening of these commitments (and we will not make explicit the openings). We assume that in the commitment $\boxed{c_u}$ the subscript u defines the variable u the commitment opens to. Analogously the opening for c_u is automatically defined as o_u. Example: $\tilde{R}_{\mathsf{ck}}(\boxed{c_u}, h; r) = 1 \Leftrightarrow h = \mathsf{SHA256}(u||r)$ is a shortcut for $\tilde{R}_{\mathsf{ck}}(c_u, h; r, u, o_u) = 1 \Leftrightarrow h = \mathsf{SHA256}(u||r) \wedge c_u = \mathsf{Comm}(\mathsf{ck}, u; o_u)$.*

3.2.2 Modular SNARKs through CP-SNARKs. We use the following folklore composition of (zero-knowledge) CP-SNARKs (cf. [25, Theorem 3.1]). Fixed a commitment scheme and given two CP-SNARKs $\mathsf{cp}\Pi_1, \mathsf{cp}\Pi_2$ respectively for two "inner" relations $\tilde{R}_1$ and $\tilde{R}_2$, we can build a (CP) SNARK for their conjunction (for a shared witness u) $\tilde{R}^*(\boxed{c_u}, \mathbb{x}_1, \mathbb{x}_2; \omega_1, \omega_2) = R_1(\boxed{c_u}, \mathbb{x}_1; \omega_1) \wedge R_2(\boxed{c_u}, \mathbb{x}_2; \omega_2)$ like this: the prover commits to u as $c_u \leftarrow \mathsf{Comm}(u, o)$; generates proofs π_1 and π_2 from the respective schemes; outputs combined proof $\pi^* := (c_u, \pi_1, \pi_2)$. The verifier checks each proof over respective inputs $(\mathbb{x}_1, c_u)$ and $(\mathbb{x}_2, c_u)$, with shared commitment c_u.

3.3 Accumulators to Multisets

A multiset is an unordered collection of values in which the same value may appear more than once. We denote by $S_1 \uplus S_2$ the union of multisets S_1 and S_2, i.e., the multiset S_3 where the multiplicity of any $x \in S_3$ is the sum of its multiplicity in S_1 and S_2. For two multisets $S_2 \subset S_1$, $S_1 \boxminus S_2$ denotes the multiset difference of S_1 and S_2, i.e., the multiset S_3 where the multiplicity of any $x \in S_3$ is the multiplicity of x in S_1 minus the multiplicity of x in S_2.

Cryptographic accumulators [11] are succinct commitments to sets that also allow one to generate succinct proofs of membership (and sometimes also non-membership). In our work we use accumulators that enjoy three additional properties. First, they support multisets. Second, they are *dynamic*, meaning that from an accumulator to S_1 one can publicly compute an accumulator to $S_1 \uplus S_2$ without knowing S_1. Third, one can create succinct membership proofs for batches of elements, i.e., to prove that $X \subset S$. In more detail, such an accumulator is a tuple of algorithms $\mathsf{Acc} = (\mathsf{Setup}, \mathsf{Accum}, \mathsf{PrvMem}, \mathsf{VfyMem}, \mathsf{Ins})$ such that:

- $\mathsf{Setup}(1^\lambda) \rightarrow \mathsf{pp}$ generates public parameters;
- $\mathsf{Accum}(\mathsf{pp}, S) \rightarrow \mathsf{acc}$ outputs accumulator acc for a multiset S;
- $\mathsf{PrvMem}(\mathsf{pp}, S, X) \rightarrow W_X$ outputs a membership proof ($X \subset S$);
- $\mathsf{VfyMem}(\mathsf{pp}, \mathsf{acc}, X, W_X) \rightarrow 0/1$ accepts or rejects a membership proof W_X;
- $\mathsf{Ins}(\mathsf{pp}, \mathsf{acc}, S') \rightarrow \mathsf{acc}'$ computes accumulator to $S \uplus S'$.

A multiset accumulator is secure if any PPT adversary has negligible probability of creating a valid membership proof for a multiset $X \not\subset S$, namely to output a tuple (S, X, W) such that there is an $x \in X$ such that $x \notin S$ and $\mathsf{VfyMem}(\mathsf{pp}, \mathsf{Accum}(\mathsf{pp}, S), X, W_X) = 1$.

We note that the popular RSA accumulator [7, 15, 21, 45] enjoys all the properties mentioned above.

3.4 Relations for batch set-membership and set-insertion

Our focus in this work is on building efficient CP-SNARKs for the following two relations parametrized by an accumulator scheme Acc and parameters $\mathsf{pp}_{\mathsf{Acc}}$:

$$\tilde{R}^{\mathrm{mem}}_{\mathrm{ck}}(\boxed{c_U}, \mathsf{acc}; W) = 1 \Leftrightarrow \mathsf{Acc.VfyMem}(\mathsf{pp}, \mathsf{acc}, U, W) = 1$$

$$\tilde{R}^{\mathrm{ins}}_{\mathrm{ck}}(\boxed{c_U}, \mathsf{acc}, \mathsf{acc}') = 1 \Leftrightarrow \mathsf{Acc.Ins}(\mathsf{pp}_{\mathsf{Acc}}, \mathsf{acc}, U) = \mathsf{acc}'$$

In a nutshell, a CP-SNARK for $\tilde{R}^{\mathrm{mem}}_{\mathrm{ck}}$ can prove that c_U is a commitment to a vector of values such that each of them is in the multiset accumulated in acc. A CP-SNARK for $\tilde{R}^{\mathrm{ins}}_{\mathrm{ck}}$ can instead prove that acc′ is a correct update of the accumulator acc obtained by inserting the elements committed in c_U. For the relation $\tilde{R}^{\mathrm{ins}}_{\mathrm{ck}}$ we are not interested in obtaining proofs that are zero-knowledge (i.e., so as to hide U), as the Ins algorithm is deterministic and thus simply having public accumulators acc, acc′ may leak information on the added elements.

The specific notion of knowledge soundness we assume for CP-SNARKs for these relations is the one where the malicious prover is allowed to select an arbitrary set S to be accumulated but the accumulator acc is computed honestly from S. Given an accumulator scheme Acc, we informally talk about this specific notion as "security under the Trusted Accumulator-Model for Acc". We do not provide formal details since this model corresponds to the notion of partial-extractable soundness in Section 5.2 in [12][8]; we refer the reader to this work for further details.

This trusted accumulator model fits several applications where the accumulator is maintained by the network.

On the other hand, we stress that in the R^{ins} relation, the trusted accumulator assumption is assumed only for acc but *not* for acc′. The interesting implication of this is that one can view a CP-SNARK for R^{ins} as a means to move from a trusted accumulator acc to a trustworthy one acc′. Thinking of acc_0 as the accumulator to the empty set that everyone knows and can efficiently compute, R^{ins} allows certifying the generation of an accumulator to any multiset.

The next sections show some interesting byproducts of having modular commit-and-prove SNARKs for relations $\tilde{R}^{\mathrm{mem}}_{\mathrm{ck}}$ and $\tilde{R}^{\mathrm{ins}}_{\mathrm{ck}}$.

3.4.1 *Composing (commit-and-prove) set-membership relations.*

The advantage of having CP-SNARKs for the set-membership relation (rather than just SNARKs) is that one can use the composition of section 3.2.2 to obtain *efficient* zkSNARKs for proving properties of elements in an accumulated set, e.g., to show that $\exists U = \{u_1, \ldots, u_n\}$ such that a property P holds for U (say, every u_i is properly signed) and $U \subset S$, where S is accumulated in some acc. In particular, such a zkSNARK can be obtained via the simple and efficient composition of a CP-SNARK for $\tilde{R}^{\mathrm{mem}}_{\mathrm{ck}}$ (like the ones we construct in our work) and any other CP-SNARK for P.

3.4.2 *From set-insertion to MultiSwap.*

Ozdemir et al. [50] introduce an operation over (RSA) accumulators called $\mathsf{MultiSwap}$. Consider two multisets S and S' and a sequence of pairs $(x_1, y_1), \ldots, (x_n, y_n)$, where each pair represents *in order* a "swap", namely removal of x_i and insertion of y_i. Verifying a $\mathsf{MultiSwap}$ means checking that $S' = S_n$ where $S_0 = S$ and $S_i = S_{i-1} \boxminus x_i \uplus y_i$. [50] shows that this check can be reduced to

$$\exists S_{mid} : S_{mid} = S \uplus \{y_i\}_i \land S_{mid} = S' \uplus \{x_i\}_i$$

So, when using accumulators, $\mathsf{MultiSwap}$ can be represented via the following relation:

$$R^{\mathrm{mswap}}(\mathsf{acc}, \mathsf{acc}'; X, Y) = 1 \Leftrightarrow$$
$$\exists \mathsf{acc}_{\mathrm{mid}} : R^{\mathrm{ins}}(\mathsf{acc}, \mathsf{acc}_{\mathrm{mid}}; Y) \land R^{\mathrm{ins}}(\mathsf{acc}', \mathsf{acc}_{\mathrm{mid}}; X)$$

Thus, a CP-SNARK for R^{mswap} can be obtained via the (self)composition of a CP-SNARK for R^{ins}.

3.4.3 *Chaining MultiSwap.*

Consider a scenario where an accumulator evolves in time, namely at time i a user returns a new accumulator acc_i along with a proof π_i that $(\mathsf{acc}_{i-1}, \mathsf{acc}_i) \in R^{\mathrm{mswap}}$ (and possibly additional proof that the elements added/removed satisfy a certain property, e.g., in Rollup they are valid transactions). It is easy to see that the concatenation $(\pi_1, \mathsf{acc}_1, \ldots, \mathsf{acc}_{n-1}, \pi_n)$ constitutes a proof for $(\mathsf{acc}_0, \mathsf{acc}_n) \in R^{\mathrm{mswap}}$.

3.5 Building blocks

3.5.1 *Pedersen Commitments of Integer values.*

The CP-SNARKs we construct are defined for commitments generated using the classical extension of Pedersen commitments to vectors. In particular, we sometimes use a variant of this scheme for committing to integers (instead of field elements); we describe it in fig. 1b. We assume a

---

[8] We notice that their model uses a slightly different language and formalizes accumulators as (binding-only) commitments for commit-and-prove NIZKs.

prime p and an algorithm $\mathcal{G}_p$ that generates appropriate parameters for groups of order p. Since we commit to an integer x whose size is potentially larger than p we split the integer into several "chunks", of size $\mathsf{ChkSz} \leq p$ specified in the parameters, and then we apply the standard vector-Pedersen on this split representation. We let the setup algorithm take as input a bound B denoting the max integer that we can commit to. The construction is perfectly hiding, and computationally binding under the discrete logarithm assumption.

3.5.2 RSA Accumulators. Another crucial component of our CP-SNARKs are RSA accumulators to multisets [7, 15], that we recall in fig. 1a. In particular, we assume their instantiation over any group of unknown order (including, e.g., classical RSA groups or class groups [18]) whose parameters are generated by an algorithm $\mathcal{G}_?$ and over which the Strong RSA [7] and the Adaptive Root [69] assumptions hold. We recall that for these Accumulators the set elements should be primes (or hashed-to-primes if not).

4 HARISA: ZERO-KNOWLEDGE CP-SNARK FOR BATCH SET-MEMBERSHIP

In this section we show the construction of a CP-SNARK for the relation $\tilde{R}^{\mathsf{mem}}_{\mathsf{ck}}$ defined in *section* 3.4.1, where: the accumulator is the classical RSA accumulator from Figure 1a where the accumulated elements are prime numbers larger than the 2λ-th prime (1619 for $\lambda = 128$), and the commitment scheme for the commit-and-prove functionality is the Pedersen scheme of Fig. 1b. In appendix C.1 we discuss how this construction can be easily extended to accumulate arbitrary elements via an efficient hash-to-prime function.

4.1 RSA Accumulators with hiding witnesses

We describe a method to turn a witness W of an RSA accumulator into another witness that computationally hides all the elements u_i it proves membership of. As discussed in Section 2 this constitutes the first building block towards achieving a zero-knowledge membership proof for committed elements.

Let $\mathbb{P}_n = \{2, 3, 5, 7, \ldots, p_n\}$ be the set of the first n prime numbers. Our method relies on two main ideas.

First, prover and verifier modify the accumulator acc so as to contain the first 2λ primes by computing $\hat{\mathsf{acc}} \leftarrow \mathsf{acc}^{\left(\prod_{p_i \in \mathbb{P}_{2\lambda}} p_i\right)}$. Note, $\hat{\mathsf{acc}} = g_?^{\left(\prod_{x_i \in S} x_i\right)\cdot\left(\prod_{p_i \in \mathbb{P}_{2\lambda}} p_i\right)} = \mathsf{Accum}(\mathsf{pp}, S \cup \mathbb{P}_{2\lambda})$.

Second, we build a randomized witness for $X \subset S$ as the witness for $(X \cup P) \subset (S \cup \mathbb{P}_{2\lambda})$ where P is a randomly chosen subset of $\mathbb{P}_{2\lambda}$. In more detail, given W, the prover computes $\hat{W}$ as follows:

- choose at random 2λ bits $b_1, \ldots, b_{2\lambda} \leftarrow\$ \{0, 1\}$ and let $s := \prod_{p_i \in \mathbb{P}_{2\lambda}} p_i^{b_i}$ and $\bar{s} := \prod_{p_i \in \mathbb{P}_{2\lambda}} p_i^{1-b_i}$.
- $\hat{W} \leftarrow W^{\bar{s}} = g_?^{\left(\prod_{x_i \in S \setminus X} x_i\right)\cdot\left(\prod_{p_i \in \mathbb{P}_{2\lambda}} p_i^{1-b_i}\right)}$.

Essentially, we have s as the product of the randomly chosen primes, $\bar{s}$ as the product of the primes not chosen, and we denote with $p^* := \prod_{p_i \in \mathbb{P}_{2\lambda}} p_i$ the product of all the first 2λ primes. Finally, by $\mathcal{D}_{2\lambda}$ we denote the distribution of $\bar{s}$, according to the sampling method described above. Note that $s\bar{s} = p^*$. Also, the new witness $\hat{W}$ could be verified by checking $\hat{W}^{s \prod_{x_i \in X} x_i} = \hat{\mathsf{acc}}$.

Our first technical contribution is proving that this randomization is sufficient. More precisely, we use a computational assumption over groups of unknown order, called DDH-II, and we show that under DDH-II $\hat{W}$ is computationally indistinguishable from a random $R \leftarrow\$ \mathbb{G}_?$. We stress that this hiding property holds only for the value $\hat{W}$ *alone*, i.e., when the random subset of $\mathbb{P}_{2\lambda}$ is not revealed. As we show later, this is sufficient for our purpose as we can hide the integer s in the same way as we hide the elements we prove membership of.

In the following section we state and explain the DDH-II assumption. In brief, this is a variant of the classical DDH assumption where the random exponents follow specific, not uniform, distributions. Next, we prove that under DDH-II $\hat{W}$ is computationally indistinguishable from random.

4.1.1 The DDH-II assumption. First, we state the DDH-II assumption, which is parametrized by a generator $\mathcal{G}_?(1^\lambda)$ of a group (of unknown order in our case) and by a well-spread distribution $\mathcal{WS}_{2\lambda}$ (in our case $\mathcal{D}_{2\lambda}$). A distribution $\mathcal{WS}_{2\lambda}$ with domain $\mathcal{X}_{2\lambda}$ is called *well-spread* if $\Pr[X = x | X \leftarrow\$ \mathcal{WS}_{2\lambda}] \leq 2^{-2\lambda}$ for each $x \in$

```
Setup(1^λ):
    $(\mathbb{G}_?, g_?) \leftarrow \mathcal{G}_?(1^\lambda)$
    return $\mathsf{pp} := (\mathbb{G}_?, g_?)$

Accum(pp, S):
    $\mathsf{prd} \leftarrow \mathsf{Prod}(S)$
    return $\mathsf{acc} := g_?^{\mathsf{prd}}$

Ins(pp, acc, S'):
    $\mathsf{prd}' \leftarrow \mathsf{Prod}(S')$
    return $\mathsf{acc}' := \mathsf{acc}^{\mathsf{prd}'}$

PrvMem(pp, S, X):
    $\mathsf{prd} \leftarrow \mathsf{Prod}(S)$, $\mathsf{prd}_X \leftarrow \mathsf{Prod}(X)$
    return $W := g_?^{\mathsf{prd}/\mathsf{prd}_X}$

VfyMem(pp, acc, X, W):
    $\mathsf{prd}_X \leftarrow \mathsf{Prod}(X)$
    Accept iff $W^{\mathsf{prd}_X} = \mathsf{acc}$
```

(a) RSA Accumulator for multisets of prime numbers. Above $\mathsf{Prod}(S)$ denotes the integer product of the elements in S.

```
Setup(1^λ, B ∈ ℕ, ChkSz ∈ ℕ, n ∈ ℕ):
    $(\mathbb{G}_p, f) \leftarrow \mathcal{G}_p(1^\lambda)$; If $\mathsf{ChkSz} > p$ then output $\perp$
    Let $N := n \cdot \left\lceil \frac{\mathsf{B}}{\mathsf{ChkSz}} \right\rceil$
    Sample $g_1, \ldots, g_N, h \leftarrow\$ \mathbb{G}_p$
    return $\mathsf{ck} := (\mathbb{G}, \mathsf{B}, \mathsf{ChkSz}, n, g_1, \ldots, g_N, h)$

Comm(ck, $\vec{x} \in \mathbb{Z}^n$; $r \in \mathbb{Z}_p$):
    If $\exists i : x_i > \mathsf{B}$ then output $\perp$
    Let $\left(x_1^{(i)}, \ldots, x_m^{(i)}\right)$ be the representation of $x_i$ in base ChkSz for $i \in [n]$
    $\vec{y} := \left(x_1^{(1)}, \ldots, x_m^{(1)}, \ldots, x_1^{(n)}, \ldots, x_m^{(n)}\right)$;   return $h^r \prod_{i=1}^{N} g_i^{y_i}$
```

(b) Pedersen Commitments for vectors of integers. B **is an upper bound over the integers we can commit to.** ChkSz **is the size of the chunks in which we divide each integer.** n **is the number of integers we can commit at the same time.** $m = \left\lceil \frac{\mathsf{B}}{\mathsf{ChkSz}} \right\rceil$ **is the number of chunks needed for each integer.**

Figure 1: Accumulator and commitment schemes we will use throughout this work

$\mathcal{X}_{2\lambda}$ (Intuition: the elements sampled from this distribution are "sufficiently random").

Assumption 1 (DDH-II). *Let* $\mathbb{G}_? \leftarrow \mathcal{G}_?(1^\lambda)$ *and* $g_? \leftarrow\$ \mathbb{G}_?$. *Let* $\mathcal{WS}_{2\lambda}$ *be a well-spread distribution with domain* $\mathcal{X}_{2\lambda} \subseteq [1, \mathsf{minord}(\mathbb{G}_?)]$. *Then for any PPT* $\mathcal{A}$*:*

$$\left|\Pr[\mathcal{A}(g_?^x, g_?^y, g_?^{xy}) = 0] - \Pr[\mathcal{A}(g_?^x, g_?^y, g_?^t) = 0]\right| = \mathsf{negl}(\lambda)$$

where $x \leftarrow\$ \mathcal{WS}_{2\lambda}$ *and* $y, t \leftarrow\$ [1, \mathsf{maxord}(\mathbb{G}_?)2^\lambda]$.[9]

Our distribution of interest $\mathcal{D}_{2\lambda}$ can be shown well-spread: there are $2^{2\lambda}$ outcomes and are all distinct, $\bar{s} = \prod_{p_i \in \mathbb{P}_{2\lambda}} p_i^{1-bi}$ are distinct since they are different products of the same primes (no p_i can be used twice). It follows that $\Pr[\bar{s} \leftarrow\$ \mathcal{D}_\lambda] = 1/2^{2\lambda}$ for every $\bar{s}$.

Remark 2. *The constraint that the domain should be in* $[1, \mathsf{minord}(\mathbb{G}_?)]$ *is for the following reason: If a sampled* x *is larger than* $\mathsf{ord}(g_?)$ *then in the exponent of* $g_?^x$ *a reduction modulo* $\mathsf{ord}(g_?)$ *will implicitly happen leading to a* $g_?^x = g_?^{x'}$ *for some* $x' \neq x$*. This can turn* $g_?^x$ *more frequently sampled, which can potentially help the adversary distinguish between* $(g_?^x)^y$ *and* $g_?^t$*.*

Different variants of DDH-II have been proven secure in the generic group model [46, 56] for prime order groups [9, 35]. We can prove it secure for groups of unknown order similarly with minor technical modifications related to GGM proofs in such groups [36].

Remark 3. *The need of an at least* $2^{2\lambda}$*-large domain* $\mathcal{X}_{2\lambda}$ *(and at most* $2^{-2\lambda}$ *probability) for* λ *security parameter comes from well-known subexponential attacks on DLOG [52, 53].*

4.1.2 *Security Proof of our hiding witnesses.*

Theorem 4.1. *For any parameters* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$*, set* S *(where* $S \cap \mathbb{P}_{2\lambda} = \emptyset$*),* $R \leftarrow\$ \mathbb{G}_?$ *and* $\hat{W}$ *computed as described above it holds:*

$$\left|\Pr[\mathcal{A}(\mathsf{pp}, S, \hat{W}) = 0] - \Pr[\mathcal{A}(\mathsf{pp}, S, R) = 0]\right| = \mathsf{negl}(\lambda)$$

for any PPT $\mathcal{A}$*, under the DDH-II assumption for* $\mathbb{G}_?$ *and* $\mathcal{D}_{2\lambda}$*.*

Proof. Call $\mathcal{A}$ an adversary achieving a non-negligible advantage ϵ above, i.e. $\epsilon := \left|\Pr[\mathcal{A}(\mathsf{pp}, S, \hat{W}) = 0] - \Pr[\mathcal{A}(\mathsf{pp}, S, R) = 0]\right|$ We construct an adversary $\mathcal{B}$ against DDH-II that, using adversary $\mathcal{A}$, gains the same advantage. $\mathcal{B}$ receives $(\mathbb{G}_?, g_?, g_?^{\bar{s}}, g_?^r, g_?^{b\bar{s}r+(1-b)t})$, where $\bar{s} \leftarrow\$ \mathcal{D}_{2\lambda}$ and $r, t \leftarrow\$ [1, \mathsf{maxord}(\mathbb{G}_?)2^\lambda]$. Then it chooses arbitrarily an element u and sets $S = \{u\}$, $\mathsf{pp} \leftarrow (\mathbb{G}_?, g_?^r)$ and $V = g_?^{b\bar{s}r+(1-b)t}$. $\mathcal{B}$ sends (pp, S, V) to the adversary $\mathcal{A}$, who outputs a bit b^*. Finally, $\mathcal{B}$ outputs b^*.

First, notice that $g_?^r$ is statistically close to a random group element of $\mathbb{G}_?$, meaning that $\mathcal{A}$ cannot distinguish pp from parameters generated by Acc.Setup(1^λ). Furthermore if $b = 0$ then V is again a (statistically indistinguishable element from a) uniformly random group element of $\mathbb{G}_?$ therefore $\Pr[\mathcal{B} = 0 | b = 0] = \Pr[\mathcal{A}(\mathsf{pp}, S, R) = 0]$. On the other hand, if $b = 1$ then $V = g_?^{r \cdot \bar{s}} = \hat{W}_u$ is a witness of u so $\Pr[\mathcal{B} = 0 | b = 1] = \Pr[\mathcal{A}(\mathsf{pp}, S, \hat{W}) = 0]$. Therefore we conclude that the probability of $\mathcal{B}$ to win the DDH-II is ϵ. □

4.2 Building Blocks

4.2.1 *Succinct proofs of knowledge of exponent (*PoKE*).* We recall the succinct proofs of knowledge of a DLOG for hidden order groups, introduced by Boneh et al. [15]. More formally, PoKE is a protocol for the relation

$$R^{\mathsf{PoKE}}(A, B; x) = 1 \Leftrightarrow A^x = B$$

parametrized by a group of unknown order $\mathbb{G}_?$ and a random group element $g_? \in \mathbb{G}_?$. The statement consists of group elements $A, B \in \mathbb{G}_?$ while the witness is an arbitrarily large $x \in \mathbb{Z}$.

Figure 2 gives a description of the protocol. For simplicity we directly expose its non-interactive version (after Fiat-Shamir). Although the interactive version of the protocol is secure with λ-sized challenges its non-interactive version is only secure with 2λ-sized challenges, due to a subexponential attack [14].

```
$\underline{\mathsf{Setup}(1^\lambda)}$ :
    $(\mathbb{G}_?, g_?) \leftarrow \mathcal{G}_?(1^\lambda)$
    return $\mathsf{crs} := (\mathbb{G}_?, g_?)$

$\underline{\mathsf{Prove}\ (\mathsf{crs}, A, B; x)}$ :
    $\ell \leftarrow H_{\mathrm{prime}}(\mathsf{crs}, A, B)$
    $Q \leftarrow A^{\lfloor \frac{x}{\ell} \rfloor}, res \leftarrow x \mod \ell$
    return $\pi = (Q, res)$

$\underline{\mathsf{Verify}\ (\mathsf{crs}, A, B, \pi)}$ :
    Parse $\pi$ as $(Q, res)$
    $\ell \leftarrow H_{\mathrm{prime}}(\mathsf{crs}, A, B)$
    Reject if $A, B, Q \notin \mathbb{G}_?$ or $res \notin [0, \ell - 1]$
    Reject if $Q^\ell A^{res} \neq B$
```

Figure 2: The succinct argument of knowledge PoKE **[15].** $H_{\mathbf{prime}}$ **denotes a cryptographic hash function that outputs a prime of size** 2λ**, modeled as a random oracle.**

Remark 4. *We note that the proof of fig. 2 is not originally secure for arbitrary bases A, but rather for random ones. For arbitrary bases extra care should be taken, that give a proof of additional* 2 *group elements. We wil show that the protocol still suffices for our needs, since we combine it with a SNARK for the relation* $res = x \mod \ell$*. In a nutshell, a* PoKE *for random bases with a SNARK for* $res = x \mod \ell$ *give a succinct proof of knowledge of exponent for arbitrary bases.*

This protocol is succinct: proof size and verifier's work are independent of the size of x, $O(\lambda)$ and $O(\|\ell\|) = O(\lambda)$ respectively.

4.2.2 *CP-SNARK for integer arithmetic relations.* We assume an efficient CP-SNARK $\mathsf{cp}\Pi^{\mathsf{modarithm}}$ for the following relation:

$$\tilde{R}^{\mathsf{modarithm}}_{\mathsf{ck}}(\boxed{c_{\vec{u}}}, \boxed{c_{s,r}}, h, \ell, \hat{k}) = 1 \Leftrightarrow$$
$$\hat{k} = s \cdot h \cdot \prod_{i \in [m]} u_i + r \mod \ell$$

Above, $\vec{u} = (u_1, \ldots, u_m) \in \mathbb{Z}^m$ is a vector of integers with a corresponding multi-integer commitment $c_{\vec{u}}$; $r, s \in \mathbb{Z}$ are integers

---

[9] Since the order of the group is unknown, we cannot efficiently produce uniformly random elements with $y, t \leftarrow\$ [1, \mathsf{ord}(g_?)]$. However, $y, t \leftarrow\$ [1, \mathsf{maxord}(\mathbb{G})2^\lambda]$ still produces statistically close to uniform elements.

committed with a corresponding multi-integer commitment $c_{s,r}$ and $\ell, h \in \mathbb{Z}$, $\hat{k} \in [0, \ell - 1]$ are (small) integers known as public inputs by both prover and verifier.

The above relation is equivalent to the integer relation:

$$\tilde{R}^{\text{arithm}}_{\text{ck}}(\boxed{c_{\vec{u}}}, \boxed{c_{s,r}}, h, \ell, \hat{k}; q) = 1 \iff q\ell + \hat{k} = s \cdot h \prod_i u_i + r$$

In fact this is how a modulo operation is encoded in a SNARK circuit. q here is a witness given to the SNARK.[10]

4.2.3 CP-SNARK for inequalities. We need a CP-SNARK $\mathsf{cp}\Pi^{\text{bound}}$ for the relation (where B is a public integer):

$$\tilde{R}^{\text{bound}}_{\text{ck}}(\boxed{c_{\vec{u}}}, B) = 1 \iff \bigwedge_{i \in [n]} u_i > B$$

4.3 Our Construction for Batched Set Membership (HARiSA)

Here we describe our CP-SNARK for the relation $\tilde{R}^{\text{mem}}_{\text{ck}}$ for RSA accumulators and Pedersen commitments to vectors of integers. Let us recall the setting in more detail.

Prover and verifier hold an accumulator acc to a set S and a commitment $c_{\vec{u}}$. The set's domain are prime numbers greater than $p_{2\lambda}$, the 2λ-th prime. The protocol works in the "trusted accumulator model" (section 3.4), which means the set is assumed to be public but the verifier does not take it as an input, it only uses acc, for efficiency reasons.[11]

The prover knows a batch of set elements $\vec{u} = (u_1, \ldots, u_m)$ that are an opening of the commitment $c_{\vec{u}}$, and its goal is to convince the verifier that all the u_i's are in S. To this end, we assume that the prover has an accumulator witness $W_{\vec{u}}$ as an input, either precomputed or given by a witness-providing entity. In this sense, the prover's goal translates into convincing the verifier that it has $W_{\vec{u}}$ such that $W_{\vec{u}}^{\prod_i u_i} = \text{acc}$ (see also remark 5 where we further refine this setting).

We give a full description of the CP-SNARK in Figure 3. We refer to the technical overview (sec. 2) for a high-level explanation. Below we provide additional comments.

To begin with, both prover and verifier transform the accumulator acc into $\mathrm{a\hat{c}c}$, the one corresponding to the same set with the additional small prime numbers from $\mathbb{P}_{2\lambda}$.[12] Next, the prover transforms W into a hiding witness as $\hat{W} = W^{\bar{s}}$ via our masking method of section 4.1, and then computes a (Fiat-Shamir-transformed) zero-knowledge Σ-protocol for the accumulator's verification $\hat{W}^{su^*} = \mathrm{a\hat{c}c}$. However, since the last message k of the protocol is not succinct, it computes a PoKE for the relation $(\mathrm{a\hat{c}c}^h R) = (\hat{W}_{\vec{u}})^k$ (exponent k), which is the verification equation of the Σ-protocol. The PoKE verification requires a check $Q^\ell \hat{W}_{\vec{u}}^{\hat{k}}$ where $\hat{k}$ is supposed to be $k \bmod \ell$. The last step of the proof is to show that $\hat{k}$ is not just "some exponent" but it is exactly $r{+}hsu^* \mod \ell$ with u^* being the product of all the u_i's committed in $c_{\vec{u}}$. To do so, the prover generates a proof with the $\mathsf{cp}\Pi^{\text{arithm}}$ CP-SNARK over the commitments $c_{\vec{u}}$, $c_{s,r}$ (r is the masking randomness of the Σ-protocol sampled in the first move). Also, for soundness we require that s and r are committed *before* receiving the random oracle challenge h. Finally, the prover generates a proof with $\mathsf{cp}\Pi^{\text{bound}}$ over the commitment $c_{\vec{u}}$ to ensure that the elements are in the right domain.[13]

We present our construction in fig. 3. This construction is obtained by applying Fiat-Shamir in the random oracle model (ROM) and additional optimizations to its interactive counterpart which we describe in the appendix (fig. 9).

Theorem 4.2. *Let H, H_{prime} be modeled as random oracles and $\mathsf{cp}\Pi^{modarithm}$, $\mathsf{cp}\Pi^{bound}$ be secure CP-SNARKs. The construction in fig. 3 for the relation $\tilde{R}^{mem}_{\text{ck}}$ is a secure CP-SNARK: succinct, knowledge-sound under the adaptive root assumption, and zero-knowledge under the DDH-II assumption.*

Proof. For succinctness, one can inspect that the proof size is proportional to that of $\mathsf{cp}\Pi^{\text{arithm}}$ and $\mathsf{cp}\Pi^{\text{bound}}$ plus some small constant overhead. Similarly for the verifier's cost. So succinctness is inherited from succinctness of $\mathsf{cp}\Pi^{\text{arithm}}$ and $\mathsf{cp}\Pi^{\text{bound}}$.

The proof for its interactive version (fig. 9) is in the appendix, theorem B.1. Then knowledge-soundness and zero-knowledge come directly from the (tight) security of the Fiat-Shamir transformation for constant-round protocols [4], in the random oracle model. ☐

Extensions. In Appendix C we show how to extend our CP-SNARK to support arbitrary—not necessarily prime—set elements, using an efficient hash-to-prime proof based on a single hash execution. We also discuss how to extend our protocol to prove (in zero-knowledge) batch non-membership.

5 B-INS-ARiSA: CP-SNARK FOR SET-INSERTION

We show a CP-SNARK for the relation $\tilde{R}^{\text{ins}}_{\text{ck}}$ (see sec. 3.4) and consequently for the MultiSwap relation R^{mswap}, using RSA accumulators. We call this construction B-INS-ARiSA.

For set-insertion we need to prove that $\mathsf{Acc.Ins}(\mathsf{pp}, \mathsf{acc}, U) = \mathsf{acc}'$, where acc and acc′ are public but the set of elements added U is not publicly provided[14], but instead a succinct commitment of it c_U. The accumulator acc is assumed to be trusted, in the sense that it is computed correctly from a set of valid elements, however for acc′ we do not make this assumption. In fact this is essentially the purpose of the protocol, to prove correctness of acc′.

5.1 Our construction for $\tilde{R}^{\text{ins}}_{\text{ck}}$ (B-INS-ARiSA)

We begin with a high-level overview of the scheme. Proving correctness of set-insertion in RSA accumulators roughly consists of proving the following:

1. $\text{acc}^{\prod_{u_i \in U} u_i} = \text{acc}'$.
2. $u_i \in \mathcal{D}$ for each $u_i \in U$.

---

[10] For the sake of our general protocol, it is not necessary that q remains hidden. It is only important that the proof is succinct w.r.t. its size. However, $\vec{u}$, s and r should remain hidden.

[11] This is a common consideration in scalable systems. The accumulator to the set is either computed once by the verifier or validated by an incentivized majority of parties that is supposed to maintain it.

[12] This operation can also be precomputed, we make it explicit only to show that they can both work with a classical RSA accumulator as an input.

[13] For the sake of generality we present π_2, π_3 as distinct proofs. In practice they can be proved by the same CP-SNARK and save on proof-size.

[14] As mentioned before, not giving U to the verifier is for the sake of succinctness. Hiding U is not in our scope.

```
$\mathsf{Setup}\left(1^\lambda, \mathsf{ck}, \mathsf{pp}\right)$ :
    $\mathsf{crs}_2 \leftarrow \mathsf{cp}\Pi^{\mathsf{modarithm}}.\mathsf{Setup}(1^\lambda, \mathsf{ck}, \tilde{R}^{\mathsf{modarithm}}_{\mathsf{ck}})$
    $\mathsf{crs}_3 \leftarrow \mathsf{cp}\Pi^{\mathsf{bound}}.\mathsf{Setup}(1^\lambda, \mathsf{ck}, \tilde{R}^{\mathsf{bound}}_{\mathsf{ck}})$
    return $\mathsf{crs} := (\mathsf{ck}, \mathsf{pp}, \mathsf{crs}_2, \mathsf{crs}_3)$
$\mathsf{Prove}\,(\mathsf{crs}, \mathsf{acc}, c_{\vec{u}}; W_{\vec{u}}, \vec{u}, o_{\vec{u}})$ :
    $\hat{\mathsf{acc}} \leftarrow \mathsf{acc}^{\prod_{p_i \in \mathbb{P}_{2\lambda}} p_i}$
    Let $u^* = \prod_i u_i,\ p^* = \prod_{p_i \in \mathbb{P}_{2\lambda}} p_i$
    Sample $b_1, \ldots, b_{2\lambda} \leftarrow\$ \{0,1\}$
    Let $s := \prod_{p_i \in \mathbb{P}_{2\lambda}} p_i^{b_i},\ \bar{s} := \prod_{p_i \in \mathbb{P}_{2\lambda}} p_i^{1-b_i}$
    $\hat{W}_{\vec{u}} \leftarrow W_{\vec{u}}^{\bar{s}}$
    Sample $r \leftarrow\$ \{0,1\}^{\|p^*\|+\|u^*\|+2\lambda}$
    $c_{s,r} \leftarrow \mathsf{Comm}_{\mathsf{ck}}(s, r; o_{s,r})$
    $R \leftarrow \hat{W}_{\vec{u}}^{r}$
    $h \leftarrow H(\mathsf{crs}||\mathsf{acc}||c_{\vec{u}}||c_{s,r}||\hat{W}_{\vec{u}}||R)$
    $k \leftarrow r + (u^* s)h$
    $\pi_1 \leftarrow \Pi^{\mathsf{PoKE}}.\mathsf{Prv}\big((\mathbb{G}_?, g_?), \hat{W}_{\vec{u}}, \hat{\mathsf{acc}}^h R; k\big)$
    Parse $\pi_1$ as $(Q, \hat{k})$
    $\ell \leftarrow H_{\mathrm{prime}}((\mathbb{G}_?, g_?), \hat{W}_{\vec{u}}, \hat{\mathsf{acc}}^h R)$
    $\pi_2 \leftarrow \mathsf{cp}\Pi^{\mathsf{modarithm}}.\mathsf{Prv}(\mathsf{crs}_2, c_{\vec{u}}, c_{s,r}, h, \ell, \hat{k}; \vec{u}, o_{\vec{u}}, r, s, o_{s,r})$
    $\pi_3 \leftarrow \mathsf{cp}\Pi^{\mathsf{bound}}.\mathsf{Prv}(\mathsf{crs}_3, c_{\vec{u}}, p_{2\lambda}; \vec{u}, o_{\vec{u}})$
    return $\pi = \big(\hat{W}_{\vec{u}}, R, c_{s,r}, \pi_1, \pi_2, \pi_3\big)$
$\mathsf{Verify}\,(\mathsf{crs}, \mathsf{acc}, c_{\vec{u}}, \pi)$ :
    $\hat{\mathsf{acc}} \leftarrow \mathsf{acc}^{\prod_{p_i \in \mathbb{P}_{2\lambda}} p_i}$
    Parse $\pi$ as $(\hat{W}_{\vec{u}}, R, c_{s,r}, \pi_1, \pi_2, \pi_3)$ and $\pi_1$ as $(Q, \hat{k})$
    $\ell \leftarrow H_{\mathrm{prime}}((\mathbb{G}_?, g_?), \hat{W}_{\vec{u}}, \hat{\mathsf{acc}}^h R)$
    $h \leftarrow H(\mathsf{crs}||\mathsf{acc}||c_{\vec{u}}||c_{s,r}||\hat{W}_{\vec{u}}||R)$
    Reject if $\Pi^{\mathsf{PoKE}}.\mathsf{Vfy}(\mathbb{G}_?, g_?), \hat{W}_{\vec{u}}, \hat{\mathsf{acc}}^h R, \pi_1) \neq 1$
    Reject if $\mathsf{cp}\Pi^{\mathsf{modarithm}}.\mathsf{Vfy}(\mathsf{crs}_2, c_{\vec{u}}, c_{s,r}, h, \ell, \hat{k}, \pi_2) \neq 1$
    Reject if $\mathsf{cp}\Pi^{\mathsf{bound}}.\mathsf{Vfy}(\mathsf{crs}_3, c_{\vec{u}}, p_{2\lambda}, \pi_3) \neq 1$
```

Figure 3: HARiSA: our scheme for proving set membership of a committed element. We let H denote a cryptographic hash function modeled as a random oracle.

Clearly the first point ensures that the insertion of the elements has been done correctly. However we still need to prove that the elements of U are in the correct domain $\mathcal{D}$. A usual domain for secure RSA Accumulators is the prime numbers, $\mathcal{D} = \mathbb{P}$. We will discuss later alternative domains.

5.1.1 *On the choice of set-membership protocol.* Notice that the first point is in fact a set-membership verification for the set of acc′ and acc is the corresponding witness of the membership. Therefore, we could in principle apply our batch set-membership protocol of sec. 4 and already obtain a construction. However, that construction would carry an overhead, due to zero-knowledge, unnecessary for the purposes of this section (for set-insertion we do not aim for zero-knowledge as discussed above). Therefore we use a simple PoKE proof for the exponentiation $\mathsf{acc}^{\prod_{u_i \in U} u_i} = \mathsf{acc}'$.

5.1.2 *On the choice of the domain.* For the second point, $u_i \in \mathcal{D}$, we need a domain that preserves the security of RSA accumulators but at the same time can be proven efficiently with a succinct protocol. Some examples of secure domains include: (1) prime numbers or prime numbers of a specific size, (2) outputs of a collision resistant hash-to-prime function, (3) outputs of a division-intractable hash function.

However, for the first two options there is no known efficient argument of knowledge; the only existing (succinct) solution is proving them with a general-purpose SNARK.[15] In particular, it is the primality check that is difficult to handle, and encoding it inside a SNARK circuit gets prohibitive as it usually requires many iterations.

Ozdemir et al. [50] observed that a variant of the division-intractable hash of Coron and Naccache [32] is (comparably) lightweight. Division intractability of a hash function H_{DI} with range in $\mathbb{Z}$ briefly means that it is hard for an adversary to find an element x and a set $\{x_i\}$ such that $x \notin \{x_i\}$ but $H_{\mathrm{DI}}(x) \mid \prod_i H_{\mathrm{DI}}(x_i)$. The function of [50] consists of a single hash computation and an addition (of 2048-bit integers). This function, denoted H_{DI}, works as follows: given a large public offset Δ of 2048-bits, the output of H_{DI} is

$$H_{\mathrm{DI}}(x) = \Delta + H(x)$$

where H is any collision-resistant hash function with image $[0, 2^{2\lambda}]$. Ozdemir et. al. showed that, under a plausible number-theoretic conjecture, H_{DI} is collision-intractable in the random oracle model (H is modeled as a random oracle). That is any output has at least a unique large prime factor, with overwhelming probability. This is a generalization of [32], where $H_{\mathrm{DI}}(x) = H(x)$, for a hash function with large outputs instead.

H can be any standard hash function as SHA256, or even a SNARK-friendly hash as Poseidon [41]. Proving a hash evaluation per element inserted inside a SNARK can be affordable in comparison to the rest of the solutions mentioned above that require primality checks. For this reason, we use division-intractable hashes as to produce accumulator elements. This technique, together with an implementation inside a SNARK, was introduced in [50].

The original elements of the set are arbitrary integers, $S \subset \mathbb{Z}$. Every element of the set x is mapped, through H_{DI}, to a division-intractable element $u = H_{\mathrm{DI}}(x)$ that are next accumulated to produce acc. Proving that $\{x_1, \ldots, x_m\}$ were inserted in S is equivalent to proving that the accumulator was updated with the corresponding $\{u_1, \ldots, u_m\} = \{H_{\mathrm{DI}}(x_1), \ldots, H_{\mathrm{DI}}(x_m)\}$. We refer the reader to [32] for a security analysis.

For our protocol we assume a CP-SNARK for the above DI-hash function evaluation:

$$\tilde{R}^{H_{\mathrm{DI}}}_{\mathsf{ck}}(\boxed{c_{\vec{u}}}; \vec{x}) = 1 \iff u_i = \Delta + H(x_i)$$

parametrized by a division-intractable hash, (H, Δ).

---

[15] Specialized solutions based on Σ-protocols exist [22] but are both inefficient and for a single prime, thus not succinct.

5.1.3 CP-SNARK for integer arithmetic relations. Again we assume an efficient CP-SNARK $\mathsf{cp}\Pi^{\text{modarithm}}$ for the relation: $\tilde{R}^{\text{modarithm}}_{\text{ck}}(\boxed{c_{\vec{u}}}, \ell, \hat{k}) = 1 \iff \hat{k} = \prod_{i\in[m]} u_i \mod \ell$ which is a simplification of the relation defined in section 4.

5.1.4 Summary of the construction. Putting things together, for our construction we prove that: a batch $U = \{u_1, \dots, u_m\}$ of committed elements is an output of H_{DI}, with $\mathsf{cp}\Pi^{H_{\text{DI}}}$; these elements are inserted in the accumulator, with a PoKE for $\text{acc}^{\prod_i u_i} = \text{acc}'$.

However, there should be a way to "link" the elements U in the two proofs. Essentially to show that the proofs are about the same batch of elements. In order to avoid encoding the RSA exponentiation acc^{u^*} inside the SNARK, which would be virtually infeasible,[16] we use an intermediate CP-SNARK that proves the following: the product u^* of the committed elements modulo the ℓ of the PoKE equals the $\hat{k}$ part of the PoKE proof, $\hat{k} = u^* \mod \ell$. As we show in the next section, this guarantees that the u^* of the PoKE is the same as the u^* (implicitly) committed, in $c_{\vec{u}}$.

A full description of our scheme is in Figure 4.

```
Setup $(1^\lambda, \mathsf{ck}, \mathsf{pp})$ :
    $\mathsf{crs}_2 \leftarrow \mathsf{cp}\Pi^{\text{modarithm}}.\mathsf{Setup}(1^\lambda, \mathsf{ck}, \tilde{R}^{\text{modarithm}}_{\text{ck}})$
    $\mathsf{crs}_3 \leftarrow \mathsf{cp}\Pi^{H_{\text{DI}}}.\mathsf{Setup}(1^\lambda, \mathsf{ck}, \tilde{R}^{H_{\text{DI}}}_{\text{ck}})$
    return $\mathsf{crs} := (\mathsf{ck}, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{crs}_2, \mathsf{crs}_3)$
Prove $(\mathsf{crs}, \text{acc}, \text{acc}', c_{\vec{u}}; \vec{u}, o_{\vec{u}}, \vec{x})$ :
    Let $u^* = \prod_i u_i$
    $\pi_1 \leftarrow \Pi^{\text{PoKE}}.\mathsf{Prv}((\mathbb{G}_?, g_?), \text{acc}, \text{acc}'; u^*)$
    Parse $\pi_1$ as $(Q, \hat{k})$
    $\ell \leftarrow H_{\text{prime}}((\mathbb{G}_?, g_?), \text{acc}, \text{acc}')$
    $\pi_2 \leftarrow \mathsf{cp}\Pi^{\text{modarithm}}.\mathsf{Prv}(\mathsf{crs}_2, c_{\vec{u}}, \ell, \hat{k}; \vec{u}, o_{\vec{u}})$
    $\pi_3 \leftarrow \mathsf{cp}\Pi^{H_{\text{DI}}}.\mathsf{Prv}(\mathsf{crs}_3, c_{\vec{u}}; \vec{u}, o_{\vec{u}}, \vec{x})$
    return $\pi = (\pi_1, \pi_2, \pi_3)$
Verify $(\mathsf{crs}, \text{acc}, \text{acc}', c_{\vec{u}}, \pi)$ :
    Parse $\pi$ as $(\pi_1, \pi_2, \pi_3)$ and $\pi_1$ as $(Q, \hat{k})$
    Reject if $\Pi^{\text{PoKE}}.\mathsf{Vfy}((\mathbb{G}_?, g_?), \text{acc}, \text{acc}', \pi_1) \neq 1$
    Reject if $\mathsf{cp}\Pi^{\text{modarithm}}.\mathsf{Vfy}(\mathsf{crs}_2, c_{\vec{u}}, \ell, res, \pi_2) \neq 1$
    Reject if $\mathsf{cp}\Pi^{H_{\text{DI}}}.\mathsf{Vfy}(\mathsf{crs}_3, c_{\vec{u}}, \pi_3) \neq 1$
```

Figure 4: b-ins-ARiSA: our scheme for proving correct set insertion of a committed batch of elements.

5.2 Multiswaps

As argued in [50] (we recall this in section 3.4.2) batch-insertion gives a succinct MultiSwap protocol: the relation R^{mswap} roughly consists of two set insertions.

$$R^{\text{mswap}}(\text{acc}, \text{acc}'; X, Y) = 1 \iff$$

$$\exists \text{acc}_{\text{mid}} : R^{\text{ins}}(\text{acc}, \text{acc}_{\text{mid}}; Y) \wedge R^{\text{ins}}(\text{acc}', \text{acc}_{\text{mid}}; X)$$

Given a set S, its corresponding (trusted) accumulator acc and a sequence of "swap" pairs $(x_1, y_1), \dots, (x_m, y_m)$ the prover computes acc_{mid}, acc' and two corresponding batch insertion proofs for $\text{acc} \xrightarrow{\text{Ins}} \text{acc}_{\text{mid}}$, $\text{acc}' \xrightarrow{\text{Ins}} \text{acc}_{\text{mid}}$. In short the prover publishes acc' and the proof for the multiswap is $\text{acc} \xrightarrow{\text{MultiSwap}} \text{acc}'$ is:

$$\pi \leftarrow (\pi_1^{\text{ins}}, \pi_2^{\text{ins}}, \text{acc}_{\text{mid}})$$

This proof can convince a verifier that the multiswap was done correctly (and that acc' is trusted).

5.2.1 Generating acc_{mid} and acc'. Computationally speaking the bottleneck in the above is the generation of acc'. Nevertheless, the intermediate value acc_{mid} is the result of the batch insertion of all y_i's, hence it can be efficiently computed in time $O(m)$ (m the size of the batch) by m sequential Acc.Ins. On the other hand, the value acc' is the result of "batch deletion" of all x_i's, an operation that cannot be done efficiently (in $O(m)$-time) and the only manner is to compute acc' from scratch, i.e. accumulate all remaining values: $\text{acc}' \leftarrow \mathsf{Acc.Accum}(\mathsf{pp}, S')$, where $S' = S \uplus \{y_i\}_i \boxminus \{x_i\}$. This requires time proportional to the size of the set, $O(n + m)$.

To this end, one can use a precomputation technique to speed-up the online computational cost. As shown by Boneh et al. [15], if one has precomputed a witness W_{x_1} then already $\text{acc}' = W_{x_1}$ is an accumulator for $S \boxminus \{x_1\}$. If one has precomputed witnesses W_{x_1} and W_{x_2} one can compute $\text{acc}' = W_{x_1,x_2}$, in $O(1)$-time by using Shamir's trick [55], which is essentially an accumulator for $S \boxminus \{x_1, x_2\}$. Generalizing this, if all witnesses are precomputed $W_{x_1}, \dots, W_{x_n}$ then one can compute acc' for any $S \boxminus \{x_{i_1}, \dots x_{i_m}\}$, in $O(m)$ time. This would require the prover to store additional $O(n)$ group elements.

To avoid storing linear-number of elements one can use another preprocessing method, introduced by Campanelli et al. [23], that offers storage-online time tradeoffs. The storage cost is $O(n/B)$ and the online time (worst-case) $O(mB)$, for any chosen parameter B. Essentially, the more elements one stores the less resources it uses online and vice-versa.

5.3 Comparison with [50]

Technically speaking our approach carries similarities with the one of Ozdemir et al. There are two distinguishing differences. The first is in the succinct protocol for the exponentiation $\text{acc}^{u^*} = \text{acc}'$. Ozdemir et al. make use of a Wesolowski proof (PoE protocol), while we propose the use of the Boneh et al. proof (PoKE protocol). The second is that we do not encode the verification of this proof inside the circuit of the SNARK.

The PoE protocol is a succinct proof of correct RSA exponentiation, introduced in [69]. It is defined for verifiers that know the exponent, i.e. the proof's input is $(\text{acc}, \text{acc}', u^*)$. For the non-interactive version, in order for the Fiat-Shamir transform to be sound the challenge should be generated as $\ell \leftarrow H_{\text{prime}}(\text{acc}, \text{acc}', u^*)$, meaning that it should take the large exponent as input. Since the verifier shall not receive the set U, it cannot generate the challenge ℓ itself.

---

[16] An RSA exponentiation of this size would require nearly 2 millions constraints per element of the batch.

Subsequently, the prover should, in addition to the rest of the computations, prove inside the SNARK that $\ell = H_{\text{prime}}(\text{acc}, \text{acc}', u^*)$. This computation gives a significant overhead to the prover's workload (see also further details below).

We replace the PoE protocol with a PoKE protocol. The PoKE proof, introduced in [15], is a proof *of knowledge* of exponent. That is, here the exponent u^* is a witness instead of an input. Meaning that the Fiat-Shamir challenge is now generated as $\ell \leftarrow H_{\text{prime}}(\text{acc}, \text{acc}')$. All inputs are public and known to the verifier. This translates to a save on the expensive SNARK computation $H_{\text{prime}}(\text{acc}, \text{acc}', u^*)$. This saves m hash computations for the SNARK and a hash-to-prime computation (applied on the output of the m hash-chain). The former has a cost of ≈ 300–$45,000$ constraints per input (depending on the choice of the hash), while the latter has a fixed cost of $217,703$ constraints [50]. So overall it has a significant impact that depends on batch size. For example, for SHA256 and a moderate-sized batch size $m = 1000$, our approach saves more than 45 million constraints.

Notably this replacement does not affect the security assumptions: although the PoKE itself is secure in the generic group model [46, 56], a careful security analysis shows that when combined with $\text{cp}\Pi^{\text{modarithm}}$ it can be proven secure in the standard model under the adaptive root assumption [69] (see proof of theorem B.1), which is the same assumption as in [50].

Instead of encoding the PoKE verification $Q^{\ell}\text{acc}^{\hat{k}} = \text{acc}'$ inside the SNARK, we let the verifier perform it itself. According to [50] (figure 3) the RSA operations needed for this verification—two RSA $|\ell|$-bit exponentiations and one RSA multiplication—overall cost about 5 million constraints. Our approach has the downside of having to additionally include the PoKE proof, $(Q, \hat{k})$, in the overall proof of set-insertion, which has an overhead of 1 RSA group elements and a 256-bit prime in the proof size. Therefore including this trick can be viewed as a tradeoff: 288 bytes in the proof vs 5 million constraints less for the prover (and vice versa).

6 EVALUATION

6.1 Instantiations and Implementation

We consider the variant of our construction in fig. 3 that supports arbitrary set elements, by hashing them to prime numbers (see appendix C) and proving this hash-to-prime by extending accordingly the relation proven by $\text{cp}\Pi^{\text{bound}}$. We use the Poseidon hash function [41] to instantiate this hash-to-prime. We instantiate the CP-SNARKs building blocks of the construction, $\text{cp}\Pi^{\text{modarithm}}$ and $\text{cp}\Pi^{\text{bound}}$, with LegoGroth16 from [25], an efficient commit-and-prove version of Groth16 [42]. Like Groth16, it requires an elliptic curve endowed with a bilinear map. We use the curve BLS12-381 [16] for our instantiations. The proof size of LegoGroth16 is constant (five group elements), amounting to 288 bytes in BLS12-381. For the accumulator scheme we use a 2048-bit RSA group. To be compatible with the assumptions of DDH-II in such a group, we must take at most $2\lambda = 232$ primes to hide the RSA witness in our construction. This does not affect the security provided by a 2048-bit RSA group.

Implementation. We provide an implementation of the instantiation described above comprising LegoGroth16, $\text{cp}\Pi^{\text{arithm}}$ and PoKE. Part of our code is an extension of the C++ SNARK library libsnark [65] with LegoGroth16. We use the Java library JSnark [64] to produce the circuit representation for the arithmetic relation in $\text{cp}\Pi^{\text{arithm}}$. We use a chunk size $\mathsf{ChkSz} = 32$ for commitments to integer (fig. 1).

Our code consists of ≈ 2000 lines of C++ code and 100 lines of Java code. We plan to release it under an open-source license. We ran our benchmarks single-threaded on Amazon EC2 using r5.8xlarge instances (248GB of memory). We ran DID-related benchmarks on an ordinary laptop (CPU i7-10510U with 16GB of RAM).

6.2 Benchmarks for Batch Membership

We evaluate our approach comparing it to Merkle Tree for benchmarks. Specifically we compare it to the following (the asterisk is a placeholder for the depth of the tree):

- MT-Pos-*: Merkle trees based on the Poseidon hash [41].
- MT-SHA-*: Merkle trees based on the SHA-256 hash.

These hash functions have different tradeoffs: while Poseidon has a much smaller encoding for SNARKs, it is hundreds of times slower when executed natively. For the case of SHA we (very conservatively) estimate timings for larger batches. Each of the Merkle-tree instantiations above is benchmarked by proving their (batch) opening using LegoGroth16 as a CP-SNARK. We compare these solutions on two benchmarks: a generic computation that consists only of batch membership statements, and a DID application in which one proves membership of a batch of elements as well as additional properties of these elements.

Remark 5 (On the witness for the batch set)**.** *Our evaluations measure the performance of proof generation, assuming that the proving user holds all the accumulator witnesses corresponding to the single elements it has interest to prove membership of. We do not include the cost of computing the accumulator witness from scratch since this task is application-dependent. For instance, in some applications (e.g., UTXO-sets and whitelists) the proving user may receive this witness and then have to keep it updated. In our construction the prover algorithm takes as input a single witness for the batch of elements. This batch-witness can be computed by aggregating the single ones held by the user; this aggregation is significantly cheaper than producing the witness for the batch subset from scratch. In our benchmarks* we do include *this aggregation cost, which does not impact our overall proving time significantly (it amounts to approximately 1% or less). See appendix A for more details.*

6.2.1 *General purpose Batch Membership of n Elements.*

We describe our evaluations in fig. 5. Notice that the performance of Merkle-tree solutions vary with the size of the accumulated set (ours does not). We benchmark both the minimal set size 2^{16} and the more realistic set size 2^{32}.

Our scheme shows an order of magnitude savings in proving time. Our verification time is slower but still highly practical: approximately 60ms vs 30ms for CP-SNARKs on Merkle trees for common set sizes. Our proof size is also competitive although 4x larger at 1.17 KB.[17]

CRS size and RAM consumption. Our constructions also show a better size of public parameters (not in figure). For batch sizes respectively 1, 16 and 64, we estimate the CRS size of our scheme to be lower than 1, 2.5, 8.5 MB respectively. In contrast, the smallest

---

[17] We use this fact: we can optimize the two LegoGroth16 proofs in fig. 3 as just one.

CRS for the Merkle-tree solutions (MT-Pos-16 for batch size 1) is already of approximately 5 MB, 5x larger than ours. We incur even better relative or absolute savings for more expensive hash functions—MT-SHA-16 has a CRS of more than 250MB for batch size 1—or larger batch sizes in larger sets—MT-Pos-32 has a CRS of more than 650MB for batch size 64. Notably, these savings on CRS size immediately translate into higher scalability due to less RAM consumption. For example, for a batch of size 64, MT-Pos-32 needs 5GB of RAM, MT-SHA-32 more than 64GB, our solution 420MB.

6.2.2 Decentralized Identity (DID). We experimentally validate our membership scheme in a realistic scenario: a Decentralized ID (DID) application on the blockchain. In this setting, the accumulator can be thought of as a portfolio of identity-related attributes/credentials of a user (e.g., bank account balance, value of monthly paycheck, identity information such as age, etc.). We are interested in privacy-preserving settings where we actually accumulate *hiding* commitments to the attributes and prove that a subset of them satisfy certain properties in zero-knowledge. We can assume the accumulator is maintained honestly, e.g, by a consensus or a smart contract that checks the signature of an authority issuing a new attribute before updating the accumulator.

Whenever a party aims to make a claim about some of her attributes, she sends a batch membership proof proving that 1) a commitment to the batch opens to a subset of the accumulator and 2) the elements in the batch refer to attributes satisfying a given property. We implement and evaluate our constructions in a concrete DID scenario where the attributes are used for *computing car insurance premiums* in a privacy-preserving way. See further details in appendix D.

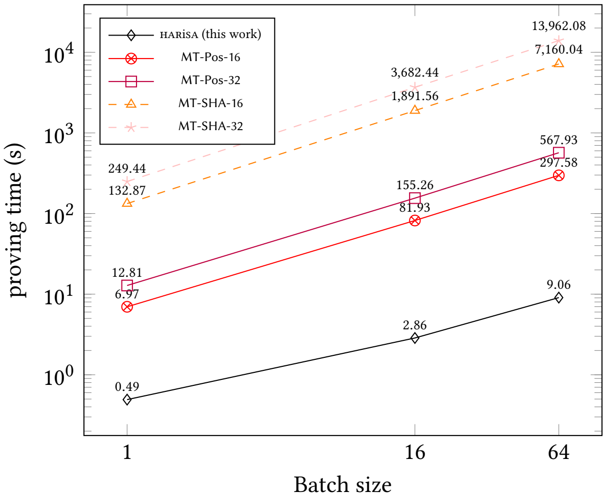
We compare a solution based on HARiSA, against a solution based on Merkle trees. We implement two instantiations of HARiSA for this protocol, one using SHA-256 for the hash-to-prime commitments to the attributes (HARiSA-DID_{sha}), and one using Poseidon (HARiSA-DID_{pos}); similarly we consider instantiations of the Merkle tree solution with both SHA-256 (MT-DID_{sha}) and Poseidon (MT-DID_{pos}).

In Figure 6 we report the proving time of the solutions for increasing batches of attributes. The accumulator contains a sets of 2^{16} committed attributes.

We find that HARiSA-DID_{pos}achieves the fastest proving time. For realistic batch sizes (16 and 64 attributes) HARiSA-DID_{pos}obtains a speedup of 12.29–12.57x compared to MT-DID_{pos}. When using SHA-256, the improvements for HARiSA-DID_{sha}vs. MT-DID_{sha}[18] are 8.57–9.26x. For larger sets we expect larger improvements, analogously to Figure 5. The tradeoffs in verification time and proof size are the same as those in fig. 5.

6.3 MultiSwap Benchmark

We evaluate our MultiSwap solution built on top of B-INS-ARiSA (sec. 5.2) and compare it with that of [50] (OWWB) and with a Merkle-tree based approach (Merkle-Swap). In all solutions we instantiate the hash functions with Poseidon. Our benchmark considers a computation consisting only of swap operations; we vary the number of swaps in 1–10, 000 and fix the set size to 2^{20}.



Scheme	V time (ms)	Proof size (KB)
MT-∗	31	0.29
HARiSA	63	1.17

Figure 5: Comparison of batched set-membership in zero-knowledge: our scheme (HARiSA) vs LegoGroth16 **on Merkle tree circuits. Plot is in log-scale. Verification time and proof size are $O(1)$ and independent of set/batch size.**

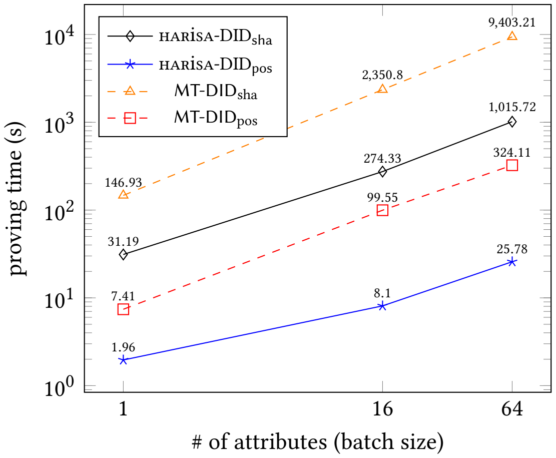


Figure 6: Comparison for our DID application: HARiSA-based solutions vs Merkle tree solutions. Plot is in log-scale.

Proving costs. For this evaluation, we use a cost-model analysis using, as metric, the number of constraints (see fig. 7). For Merkle-Swap the number of constraints is the only metric that conditions proving time. For our (resp. [50]) MultiSwap, the proving cost is made of the zkSNARK prover cost (which again depends on the number of constraints reported in fig. 7) plus the cost of RSA group operations to compute the accumulator acc′ after deleting elements and the

---

[18] Timings for MT-DID_{sha} for batch sizes larger than 1 are an extrapolated conservative lower bound because very RAM/CPU-intensive.

	Number of Constraints	
System	Per-operation cost	Per-proof cost
Merkle-Swap	$2(c_{H_e} + \log\lvert S\rvert \cdot c_H)$	—
MultiSwap from [50]	$2(c_{H_e} + c_{H_{in}} + c_{\text{split}} + c_{+\ell}(f) + c_{\times\ell})$	$4c_{e_{\mathbb{G}_?}}(\lvert\ell\rvert) + 2c_{\times\mathbb{G}_?} + c_{H_p} + c_{\text{mod}_\ell}(b_{H_{\text{DI}}})$
B-INS-ARISA-MultiSwap	$2(c_{H_e} + c_{\text{split}} + c_{+\ell}(f) + c_{\times\ell})$	$c_{\text{mod}_\ell}(b_{H_{\text{DI}}})$

λ	security parameter (128)	f	field elements size $\log_2 \lvert\mathbb{F}\rvert$ (255)
$\lvert\ell\rvert$	prime challenge bits (256 in ours, 352 in [50])	c_{split}	strict bit split in $\mathbb{F}$ (388)
$b_{H_{\text{DI}}}$	output size of division-intractable hash H_{DI} (2048)	$c_{+\ell}(b)$	addition mod ℓ of two inputs of b-bits $(16 + b)$
c_H	hash $\mathbb{F}^2 \rightarrow \mathbb{F}$ (varies)	$c_{\times\ell}$	multiplication mod ℓ (479)
c_{H_e}	set items hashing to $\mathbb{F}$, used in H_{DI} (varies)	$c_{mod_\ell}(b)$	reduction mod ℓ of b-bit input $(16 + b)$
$c_{H_{in}}$	per-operation cost of full-input hash in [50] (varies)	$c_{\times\mathbb{G}_?}$	multiplication in $\mathbb{G}_?$ (7563)
c_{H_p}	prime generation (217703)	$c_{e_{\mathbb{G}_?}}(b)$	exponentiation in $\mathbb{G}_?$ with b-bit exponent $(7044\, b)$

Figure 7: Constraints count model for Merkle swaps, the MultiSwap of [50], and our MultiSwap of section 5.2. The cost model and its parameters are from [50]. The hash functions cost "varies" depending on the hash instantiation.

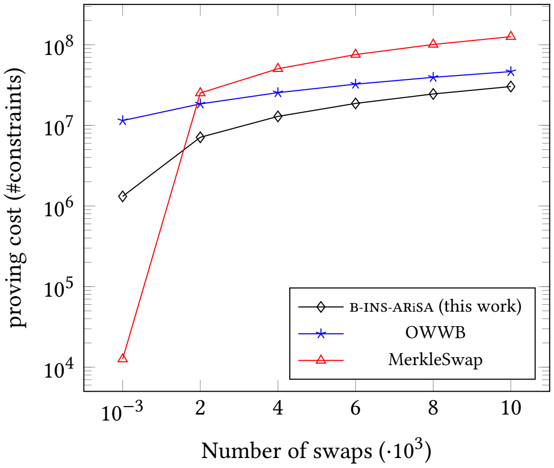


Figure 8: Proving cost vs. # swaps in MultiSwap. Set size $|S|$ is 2^{20} and y-axis is in log-scale.

PoKE (resp. PoE) proof. To estimate the latter costs, we extract an equivalent measure in number of constraints based on [50, Sec. 4.4].

We report our results in fig. 8. Our MultiSwap solution has proving cost larger than Merkle-Swap for small batches, but it breaks even at ≈ 140 swaps. Also, it strictly improves over OWWB [50] MultiSwap, which has a larger fixed cost, and a break-even point w.r.t. Merkle-Swap at ≈ 1400 swaps.

Verification time and proof size. For this evaluation we consider an instantiation of all solutions with LegoGroth16 as a CP-SNARK. Similarly to the batch-membership case, our solution has slower verification and larger proofs, which are still practical. Our MultiSwap proof is 1.4 KB whereas proofs for Merkle-Swap and [50] are 288 bytes. Our verification time is ≈ 120ms and is about 4 times slower than that of Merkle-Swap and OWWB [50].

7 RELATED WORK

Succinct proofs for RSA accumulators. The works that are closest to ours are those of Benarroch et al. [12] and Ozdemir et al. [50], both concerning the efficient use of RSA accumulators with zkSNARKs. Comparing to [12], we achieve constant-size proofs of membership for batches of elements whereas [12] can only prove membership of a single (committed) element. In particular, the technique of [12] does not seem extendable to support batching with constant size proofs: they mainly rely on a new sigma-protocol for proving that two commitments, one in a prime order group and one in a hidden-order group, open to the same integer value (the element in the accumulated set), but the size of its proof is linear in the integer's size. This means that extending this protocol to batch RSA witnesses would lead to linear-size proofs. Comparing to [50], our protocol for batch insertions is similar but has the following key differences. We employ a PoKE protocol instead of a proof-of-exponentiation (PoE); this allows us to generate the PoKE random oracle challenge based on the short and public verifier's input as opposed to the long and unknown exponent as in [50]. Thanks to this we can avoid encoding in the SNARK an expensive and long hashing along with a prime certification of its outputs. The second major difference is that Ozdemir et al. technique is not ZK-friendly and could not be used to do batch membership; to the best of our knowledge, hiding the RSA accumulator witness would require encoding an RSA group operation in the constraints.

Another work related to batch proofs is that of Boneh et al. [15] who construct such proofs for RSA accumulators with an efficient verification procedure. In their constructions, however, the verifier knows the elements for which it is verifying (non)membership. In contrast, our goal is to build proofs that can be verified by having only a succinct commitment to the batch of elements, over which one can also verify additional properties.

Verifiable computation with state. Verifiable computation and zkSNARKs have a vast literature; a complete coverage goes beyond the scope of this paper, e.g., see this recent survey [68] and references therein. More relevant to our work are some works that address the problem of verifiable computation (or zkSNARKs) with respect to succinct digests. Pantry [17] use Merkle trees to model RAM computations. Fiore et al. [39] propose hash&prove as well as accumulate&prove protocols that avoid expensive hash encodings in the circuit, but their solutions require the SNARK prover to do work *linear* in the hashed/accumulated set, which limits their scalability to large sets. The same limitation applies to the efficient commit-and-prove SNARKs [25, 33] as well as to the vSQL scheme of Zhang et al. [70] and TRUESET by Kosba et al. [43]. Also, all these schemes [25, 39, 70] require public parameters linear in the largest set. ADSNARK [5] can generate proofs on authenticated data; this setting is similar to accumulated sets except that inserting data in the set requires a secret authentication key; proofs in [5] are succinct *only* when verifiers know the secret authentication key.

Accumulators/vector commitments in stateless blockchains. In addition to the already mentioned accumulators from hidden-order groups and Merkle trees,[19] other popular schemes rely on bilinear pairings [20, 49]. Merkle trees actually generalize to vector commitments [28], of which we also know realizations from hidden-order groups and bilinear pairings. Recent works [15, 30, 40, 44, 61] have extended these two primitives with additional functionalities, including batch proofs, and shown applications to stateless blockchains. In the latter approach, transactions need to be sent and known for verification. Also, their use within zkSNARKs presents the same efficiency challenges—their verification (including elliptic curve operations and pairings) is considerably expensive when compiled into constraints [33, 64]—in addition to the fact that they need public parameters linear in the largest set to be accumulated. Finally, Chen et. al. studied the combination of Merkle-trees with recursive SNARKs [29], which allows a single proof for multiple state transitions (multiswaps) recursively. However, the performance of recursive SNARKs is currently much worse than the traditional ones, occuring to heavy limitations in practice, the main bottleneck being prover's computational time [29, Sec. 7.5].

ACKNOWLEDGMENTS

This work has received funding in part from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program under project PICOCRYPT (grant agreement No. 101001283), by the Spanish Government under projects SCUM (ref. RTI2018-102043-B-I00) and SECURITAS (ref. RED2018-102321-T), by the Madrid Regional Government under project BLOQUES (ref. S2018/TCS-4339), and by a grant from Nomadic Labs and the Tezos foundation. This work was also partly supported by the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korean government (MSIT) (No.2021-0-00518, Blockchain privacy preserving techniques based on data encryption, 50%, No.2021-0-00532, Blockchain scalability solutions supporting high performance/capacity transactions, 30%, No.2021-0-00727, A Study on Cryptographic Primitives for SNARK, 10%, No.2021-0-00528, Development of Hardware-centric Trusted Computing Base and Standard Protocol for Distributed Secure Data Box, 5% and No.2021-0-00590, Decentralized High Performance Consensus for Large-Scale Blockchains, 5%) and by a grant from Klaytn foundation. Other support was provided by the Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM) while Matteo Campanelli was affiliated with Aarhus University.

M. Campanelli, D. Fiore, S. Han, and D. Kolonelos are co-first authors. J. Kim and H. Oh are co-corresponding authors.

REFERENCES

[1] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. 2016. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In *ASIACRYPT 2016, Part I (LNCS)*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.), Vol. 10031. Springer, Heidelberg, 191–219. https://doi.org/10.1007/978-3-662-53887-6_7

[2] Jason Metz Ashley Kilroy. 2022. "9 Factors That Affect Your Car Insurance Rates". https://www.forbes.com/advisor/car-insurance/factors-in-rates/. (2022).

[3] Thomas Attema, Ronald Cramer, and Lisa Kohl. 2021. A Compressed Σ-Protocol Theory for Lattices. In *CRYPTO 2021, Part II (LNCS)*, Tal Malkin and Chris Peikert (Eds.), Vol. 12826. Springer, Heidelberg, Virtual Event, 549–579. https://doi.org/10.1007/978-3-030-84245-1_19

[4] Thomas Attema, Serge Fehr, and Michael Klooß. 2021. Fiat-shamir transformation of multi-round interactive proofs. *Cryptology ePrint Archive* (2021).

[5] Michael Backes, Manuel Barbosa, Dario Fiore, and Raphael M. Reischuk. 2015. ADSNARK: Nearly Practical and Privacy-Preserving Proofs on Authenticated Data. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 271–286. https://doi.org/10.1109/SP.2015.24

[6] Endre Bangerter, Jan Camenisch, and Stephan Krenn. 2010. Efficiency Limitations for S-Protocols for Group Homomorphisms. In *TCC 2010 (LNCS)*, Daniele Micciancio (Ed.), Vol. 5978. Springer, Heidelberg, 553–571. https://doi.org/10.1007/978-3-642-11799-2_33

[7] Niko Bari and Birgit Pfitzmann. 1997. Collision-Free Accumulators and Fail-Stop Signature Schemes Without Trees. In *EUROCRYPT'97 (LNCS)*, Walter Fumy (Ed.), Vol. 1233. Springer, Heidelberg, 480–494. https://doi.org/10.1007/3-540-69053-0_33

[8] barry WhiteHat. 2018. roll_up: Scale ethereum with SNARKs. https://github.com/barryWhiteHat/roll_up. (2018).

[9] James Bartusek, Fermi Ma, and Mark Zhandry. 2019. The Distinction Between Fixed and Random Generators in Group-Based Assumptions. In *CRYPTO 2019, Part II (LNCS)*, Alexandra Boldyreva and Daniele Micciancio (Eds.), Vol. 11693. Springer, Heidelberg, 801–830. https://doi.org/10.1007/978-3-030-26951-7_27

[10] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 459–474. https://doi.org/10.1109/SP.2014.36

[11] Josh Cohen Benaloh and Michael de Mare. 1994. One-Way Accumulators: A Decentralized Alternative to Digital Sinatures (Extended Abstract). In *EUROCRYPT'93 (LNCS)*, Tor Helleseth (Ed.), Vol. 765. Springer, Heidelberg, 274–285. https://doi.org/10.1007/3-540-48285-7_24

[12] Daniel Benarroch, Matteo Campanelli, Dario Fiore, Kobi Gurkan, and Dimitris Kolonelos. 2021. Zero-Knowledge Proofs for Set Membership: Efficient, Succinct, Modular. In *International Conference on Financial Cryptography and Data Security*. Springer, 393–414.

[13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2012. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS 2012*, Shafi Goldwasser (Ed.). ACM, 326–349. https://doi.org/10.1145/2090236.2090263

[14] Dan Boneh, Benedikt Bünz, and Ben Fisch. 2018. A Survey of Two Verifiable Delay Functions. *IACR Cryptol. ePrint Arch.* 2018 (2018), 712.

[15] Dan Boneh, Benedikt Bünz, and Ben Fisch. 2019. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. In *CRYPTO 2019, Part I (LNCS)*, Alexandra Boldyreva and Daniele Micciancio (Eds.), Vol. 11692. Springer, Heidelberg, 561–586. https://doi.org/10.1007/978-3-030-26948-7_20

[16] Sean Bowe. 2017. BLS12-381: New zk-SNARK elliptic curve construction. *Zcash Company blog, URL: https://z. cash/blog/new-snark-curve* (2017).

[17] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. 2013. Verifying computations with state. In *Proc. of the ACM SOSP*.

[18] Johannes Buchmann and Safuat Hamdy. 2011. A survey on IQ cryptography. In *Public-Key Cryptography and Computational Number Theory*. De Gruyter, 1–16.

[19] Ahto Buldas, Peeter Laud, and Helger Lipmaa. 2000. Accountable Certificate Management Using Undeniable Attestations. In *ACM CCS 2000*, Dimitris Gritzalis, Sushil Jajodia, and Pierangela Samarati (Eds.). ACM Press, 9–17. https://doi.org/10.1145/352600.352604

[20] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. 2009. An Accumulator Based on Bilinear Maps and Efficient Revocation for Anonymous Credentials. In *PKC 2009 (LNCS)*, Stanislaw Jarecki and Gene Tsudik (Eds.), Vol. 5443. Springer, Heidelberg, 481–500. https://doi.org/10.1007/978-3-642-00468-1_27

[21] Jan Camenisch and Anna Lysyanskaya. 2002. Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials. In *CRYPTO 2002 (LNCS)*, Moti Yung (Ed.), Vol. 2442. Springer, Heidelberg, 61–76. https://doi.org/10.1007/3-540-45708-9_5

[22] Jan Camenisch and Markus Michels. 1999. Proving in Zero-Knowledge that a Number Is the Product of Two Safe Primes. In *EUROCRYPT'99 (LNCS)*, Jacques Stern (Ed.), Vol. 1592. Springer, Heidelberg, 107–122. https://doi.org/10.1007/3-540-48910-X_8

---

[19] We can include under this category existing lattice-based schemes [51].

[23] Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. 2020. Incrementally Aggregatable Vector Commitments and Applications to Verifiable Decentralized Storage. In *ASIACRYPT 2020, Part II (LNCS)*, Shiho Moriai and Huaxiong Wang (Eds.), Vol. 12492. Springer, Heidelberg, 3–35. https://doi.org/10.1007/978-3-030-64834-3_1

[24] Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. 2020. Incrementally Aggregatable Vector Commitments and Applications to Verifiable Decentralized Storage. Cryptology ePrint Archive, Report 2020/149. (2020). https://eprint.iacr.org/2020/149.

[25] Matteo Campanelli, Dario Fiore, and Anaïs Querol. 2019. LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 2075–2092. https://doi.org/10.1145/3319535.3339820

[26] Ran Canetti. 1997. Towards Realizing Random Oracles: Hash Functions That Hide All Partial Information. In *CRYPTO'97 (LNCS)*, Burton S. Kaliski Jr. (Ed.), Vol. 1294. Springer, Heidelberg, 455–469. https://doi.org/10.1007/BFb0052255

[27] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. 2002. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*. ACM Press, 494–503. https://doi.org/10.1145/509907.509980

[28] Dario Catalano and Dario Fiore. 2013. Vector Commitments and Their Applications. In *PKC 2013 (LNCS)*, Kaoru Kurosawa and Goichiro Hanaoka (Eds.), Vol. 7778. Springer, Heidelberg, 55–72. https://doi.org/10.1007/978-3-642-36362-7_5

[29] Weikeng Chen, Alessandro Chiesa, Emma Dauterman, and Nicholas P Ward. 2020. Reducing participation costs via incremental verification for ledger systems. *Cryptology ePrint Archive* (2020).

[30] Alexander Chepurnoy, Charalampos Papamanthou, Shravan Srinivasan, and Yupeng Zhang. 2018. Edrax: A Cryptocurrency with Stateless Transaction Validation. Cryptology ePrint Archive, Report 2018/968. (2018). https://ia.cr/2018/968.

[31] Alexander Chepurnoy, Charalampos Papamanthou, and Yupeng Zhang. 2018. Edrax: A Cryptocurrency with Stateless Transaction Validation. Cryptology ePrint Archive, Report 2018/968. (2018). https://eprint.iacr.org/2018/968.

[32] Jean-Sébastien Coron and David Naccache. 2000. Security Analysis of the Gennaro-Halevi-Rabin Signature Scheme. In *EUROCRYPT 2000 (LNCS)*, Bart Preneel (Ed.), Vol. 1807. Springer, Heidelberg, 91–101. https://doi.org/10.1007/3-540-45539-6_7

[33] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. 2015. Geppetto: Versatile Verifiable Computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 253–270. https://doi.org/10.1109/SP.2015.23

[34] Ronald Cramer. 1996. Modular design of secure yet practical cryptographic protocols. *Ph. D. Thesis, CWI and University of Amsterdam* (1996).

[35] Ivan Damgard, Carmit Hazay, and Angela Zottarel. 2014. Short Paper On the Generic Hardness of DDH-II. (2014).

[36] Ivan Damgård and Maciej Koprowski. 2002. Generic Lower Bounds for Root Extraction and Signature Schemes in General Groups. In *EUROCRYPT 2002 (LNCS)*, Lars R. Knudsen (Ed.), Vol. 2332. Springer, Heidelberg, 256–271. https://doi.org/10.1007/3-540-46035-7_17

[37] Justin Drake. 2017. Accumulators, scalability of UTXO blockchains, and data availability. https://ethresear.ch/t/accumulators-scalability-of-utxo-blockchains-and-data-availability/176. (2017).

[38] Amos Fiat and Adi Shamir. 1987. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO'86 (LNCS)*, Andrew M. Odlyzko (Ed.), Vol. 263. Springer, Heidelberg, 186–194. https://doi.org/10.1007/3-540-47721-7_12

[39] Dario Fiore, Cédric Fournet, Esha Ghosh, Markulf Kohlweiss, Olga Ohrimenko, and Bryan Parno. 2016. Hash First, Argue Later: Adaptive Verifiable Computations on Outsourced Data. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, 1304–1316. https://doi.org/10.1145/2976749.2978368

[40] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. 2020. Pointproofs: Aggregating Proofs for Multiple Vector Commitments. In *ACM CCS 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, 2007–2023. https://doi.org/10.1145/3372297.3417244

[41] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. Poseidon: A new hash function for zero-knowledge proof systems. In *30th USENIX Security Symposium (USENIX Security 21)*.

[42] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *EUROCRYPT 2016, Part II (LNCS)*, Marc Fischlin and Jean-Sébastien Coron (Eds.), Vol. 9666. Springer, Heidelberg, 305–326. https://doi.org/10.1007/978-3-662-49896-5_11

[43] Ahmed E. Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, Mahmoud F. Sayed, Elaine Shi, and Nikos Triandopoulos. 2014. TRUESET: Faster Verifiable Set Computations. In *USENIX Security 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 765–780.

[44] Russell W. F. Lai and Giulio Malavolta. 2019. Subvector Commitments with Application to Succinct Arguments. In *CRYPTO 2019, Part I (LNCS)*, Alexandra Boldyreva and Daniele Micciancio (Eds.), Vol. 11692. Springer, Heidelberg, 530–560. https://doi.org/10.1007/978-3-030-26948-7_19

[45] Jiangtao Li, Ninghui Li, and Rui Xue. 2007. Universal Accumulators with Efficient Nonmembership Proofs. In *ACNS 07 (LNCS)*, Jonathan Katz and Moti Yung (Eds.), Vol. 4521. Springer, Heidelberg, 253–269. https://doi.org/10.1007/978-3-540-72738-5_17

[46] Ueli M. Maurer. 2005. Abstract Models of Computation in Cryptography (Invited Paper). In *10th IMA International Conference on Cryptography and Coding (LNCS)*, Nigel P. Smart (Ed.), Vol. 3796. Springer, Heidelberg, 1–12.

[47] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *CRYPTO'87 (LNCS)*, Carl Pomerance (Ed.), Vol. 293. Springer, Heidelberg, 369–378. https://doi.org/10.1007/3-540-48184-2_32

[48] Silvio Micali. 1994. CS Proofs (Extended Abstracts). In *35th FOCS*. IEEE Computer Society Press, 436–453. https://doi.org/10.1109/SFCS.1994.365746

[49] Lan Nguyen. 2005. Accumulators from Bilinear Pairings and Applications. In *CT-RSA 2005 (LNCS)*, Alfred Menezes (Ed.), Vol. 3376. Springer, Heidelberg, 275–292. https://doi.org/10.1007/978-3-540-30574-3_19

[50] Alex Ozdemir, Riad S. Wahby, Barry Whitehat, and Dan Boneh. 2020. Scaling Verifiable Computation Using Efficient Set Accumulators. In *USENIX Security 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2075–2092.

[51] Charalampos Papamanthou, Elaine Shi, Roberto Tamassia, and Ke Yi. 2013. Streaming Authenticated Data Structures. In *EUROCRYPT 2013 (LNCS)*, Thomas Johansson and Phong Q. Nguyen (Eds.), Vol. 7881. Springer, Heidelberg, 353–370. https://doi.org/10.1007/978-3-642-38348-9_22

[52] Stephen Pohlig and Martin Hellman. 1978. An improved algorithm for computing logarithms over GF (p) and its cryptographic significance (corresp.). *IEEE Transactions on information Theory* 24, 1 (1978), 106–110.

[53] John M Pollard. 1978. Monte Carlo methods for index computation mod p. *Mathematics of computation* 32, 143 (1978), 918–924.

[54] Tomas Sander and Amnon Ta-Shma. 1999. Auditable, Anonymous Electronic Cash. In *CRYPTO'99 (LNCS)*, Michael J. Wiener (Ed.), Vol. 1666. Springer, Heidelberg, 555–572. https://doi.org/10.1007/3-540-48405-1_35

[55] Adi Shamir. 1983. On the generation of cryptographically strong pseudorandom sequences. *ACM Transactions on Computer Systems (TOCS)* 1, 1 (1983), 38–44.

[56] Victor Shoup. 1997. Lower Bounds for Discrete Logarithms and Related Problems. In *EUROCRYPT'97 (LNCS)*, Walter Fumy (Ed.), Vol. 1233. Springer, Heidelberg, 256–266. https://doi.org/10.1007/3-540-69053-0_18

[57] Roberto Tamassia. 2003. Authenticated Data Structures. In *ESA*.

[58] Björn Terelius and Douglas Wikström. 2012. Efficiency Limitations of S-Protocols for Group Homomorphisms Revisited. In *SCN 12 (LNCS)*, Ivan Visconti and Roberto De Prisco (Eds.), Vol. 7485. Springer, Heidelberg, 461–476. https://doi.org/10.1007/978-3-642-32928-9_26

[59] Peter Todd. 2016. Making UTXO Set Growth Irrelevant With Low-Latency Delayed TXO Commitments. https://petertodd.org/2016/delayed-txo-commitments. (2016).

[60] Peter Todd. 2016. Making UTXO Set Growth Irrelevant With Low-Latency Delayed TXO Commitments. https://petertodd.org/2016/delayed-txo-commitments. (2016).

[61] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. 2020. Aggregatable Subvector Commitments for Stateless Cryptocurrencies. In *SCN 20 (LNCS)*, Clemente Galdi and Vladimir Kolesnikov (Eds.), Vol. 12238. Springer, Heidelberg, 45–64. https://doi.org/10.1007/978-3-030-57990-6_3

[62] V.A. 2022. Hyperledger Indy. https://www.hyperledger.org/use/hyperledger-indy. (2022).

[63] V.A. 2022. Iden3. https://iden3.io. (2022).

[64] V.A. 2022. jsnark. https://github.com/akosba/jsnark. (2022).

[65] V.A. 2022. libsnark. https://github.com/scipr-lab/libsnark. (2022).

[66] V.A. 2022. Sovrin. https://sovrin.org. (2022).

[67] V.A. 2022. Zcash. https://z.cash. (2022).

[68] Michael Walfish and Andrew J. Blumberg. 2015. Verifying Computations without Reexecuting Them. *Commun. ACM* 58, 2 (jan 2015), 74–84. https://doi.org/10.1145/2641562

[69] Benjamin Wesolowski. 2019. Efficient Verifiable Delay Functions. In *EUROCRYPT 2019, Part III (LNCS)*, Yuval Ishai and Vincent Rijmen (Eds.), Vol. 11478. Springer, Heidelberg, 379–407. https://doi.org/10.1007/978-3-030-17659-4_13

[70] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases. In *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 863–880. https://doi.org/10.1109/SP.2017.43

A MORE ON THE GENERATION AND MAINTENANCE OF ACCUMULATOR WITNESSES

The setting. When evaluating our proof system for batch membership we assume that the prover already has access to precomputed witnesses of its singletons of elements (fig. 5); we do account for witness generation in our MultiSwap benchmark (section 6.3).

There are different scenarios where it is plausible that users hold precomputed witnesses for their set elements of interest. These scenarios include for example UTXO-like settings and whitelists (where the elements represent respectively an unspent transaction and an identity).

Aggregating witnesses for singletons. Consider a party holding a "set of interest" $\hat{U}$ (the subset of accumulated elements in which it has a stake to prove set membership). As mentioned above we assume that each party holds an accumulator witness for each of the elements in $\hat{U}$. When requested to batch prove membership for $u_1, \ldots, u_m \in \hat{U}$, the party can obtain a single witness for the whole batch (like the one assumed as input in fig. 3) without recomputing it from scratch. In RSA accumulators, we can in fact apply a process of *aggregation* among the witnesses. Aggregation uses Shamir's trick [55][20] and proceeds in a tree-like fashion. For a batch of size m it consists of roughly m GCD computations, and a similar number of products and RSA exponentiations with integer inputs of varying size. As we observe in remark 5, aggregation does not significantly impact overall proving time.

Witness generation and maintenance. Here we discuss how proving parties can obtain and maintain witnesses for elements in their set of interest[21].

A straightforward way for a user to obtain a witness to their elements of interest is to precompute it from scratch. For a single witness, this involves performing roughly N exponentiations with exponents of 256 bits in an RSA group (where N is the whole set size). There are efficient ways to reuse work and distribute it in parallel for subsets of elements. The naive approach to generate a witness requires less than a minute on an ordinary laptop for a set of size 2^{16}, but it can be costly for larger sets. In order to mitigate this, there exist more sophisticated highly-parallelizable approaches to generate witnesses. For example, those described in Section 4.4 in [50]. As an alternative this can be delegated to a service provider as described for updates in [15] (notice that the witnesses from this service providers does not need to be trusted and can be efficiently verified through the standard accumulator verification algorithm).

Does a party need to recompute their witness from scratch if the accumulator (and its underlying set) changes over time? Fortunately not. A party observing updates to the accumulators can update their witnesses cheaply. For example, appending an element x to the set requires updating the witness by simply exponentiating the old witness to x. Other types of updates (e.g., removal of an element) can be handled through Shamir's trick[22]

If a party cannot observe all updates or if the update process is too demanding, this can be delegate to a (non trusted) service provider as described in [15].

A note on storage: if storing all witnesses for singletons of interests is too demanding, this can be mitigated through some of the disaggregation & aggregation techniques described in [23] and storing only witnesses for "chunks" of elements of interest[23]. A similar technique can also be useful to reduce the complexity of handling updates.

B DEFERRED SECURITY PROOFS

B.1 Security of the construction of section 4

In fig. 9 we describe an interactive version of our construction.

Theorem B.1. *Let* $\mathsf{cp}\Pi^{modarithm}$, $\mathsf{cp}\Pi^{bound}$ *be secure CP-SNARKs then the construction in fig. 9 for the relation* $\tilde{R}^{mem}_{\mathsf{ck}}$ *is a secure CP-NIZK: succinct, knowledge-sound under the adaptive root assumption and zero-knowledge under the DDH-II assumption.*

Proof. *Succinctness*: Comes from inspection and from the assumption that $\mathsf{cp}\Pi^{\mathrm{modarithm}}$ and $\mathsf{cp}\Pi^{\mathrm{bound}}$ are succinct.

$(2, M)$*-Special Soundness*: assume that we have a tree of $(2, M)$ successful transcripts, for $M = \mathrm{poly}(\lambda) > \left\lceil \frac{\|p^*\|+\|u^*\|+\lambda}{2\lambda} \right\rceil$, i.e.

$$\left\{\left(\hat{W}_{\vec{u}}, c_{s,r}, R\right), h, \ell^{(j)}, \left(Q^{(i)}, res^{(j)}\right), \pi_2^{(j)}, \pi_3^{(j)}\right\}_{j=1}^{M}$$

and

$$\left\{\left(\hat{W}_{\vec{u}}, c_{s,r}, R\right), \tilde{h}, \tilde{\ell}'^{(j)}, \left(\tilde{Q}^{(i)}, \tilde{res}^{(j)}\right), \tilde{\pi}_2^{(j)}, \tilde{\pi}_3^{(j)}\right\}_{j=1}^{M}$$

We construct an extractor Ext that works as follows.

Ext uses the extractor of $\mathsf{cp}\Pi^{\mathrm{modarithm}}$ to extract $\vec{u}^{(j)}, s^{(j)}, r^{(j)}$, openings of $c_{\vec{u}}$ and $c_{s,r}$ respectively, such that $res^{(j)} = s^{(j)} h \prod_i u_i^{(j)} + r^{(j)} \mod \ell^{(j)}$. From the binding of the commitments we get that $\vec{u}^{(j)} = \vec{u}^{(j')}, s^{(j)} = s^{(j')}, r^{(j)} = r^{(j')}$ for each transcript $j \neq j'$ and $j, j' \in [M]$, since they refer to the same commitments. So we denote the extracted values as $\vec{u}, s, r$ and get:

$$sh \prod_i u_i + r = res^{(j)} \mod \ell^{(j)}, \text{ for each } j \in [M]$$

Using the Chinese Remainder Theorem we get a k such that

$$k = sh \prod_i u_i + r \mod \left(\prod_{j=1}^{M} \ell^{(j)} \right)$$

M can be set sufficiently large (but still polynomial-sized) so that $\prod_{j=1}^{M} \ell^{(j)} > sh \prod_i u_i + r$ and thus $k = sh \prod_i u_i + r$ over the integers. Furthermore, $k = res^{(j)} \mod \ell^{(j)}$ for each $j \in [M]$.

As shown in [15] the fact that for any accepting proof, (ℓ, Q, res), it holds that $Q^{\ell} \hat{W}_{\vec{u}}^{res} = \mathsf{acc}^h R$ and $k = res \mod \ell$ (the latter in our case is ensured by the SNARK) then under the adaptive root assumption we get:

$$\hat{W}_{\vec{u}}^{k} = \hat{\mathsf{acc}}^h R$$

---

[20] See page 12 in [24].

[21] Here we provide an overview of techniques that can be useful to make these issues practical. Which approach is best is something highly sensitive to idiosyncratic aspects of the domain and a full analysis is out of the scope of this paper.

[22] These operations can be concretely inexpensive for meaningful sizes of the subset of interest. For example, we measure the time required to update 64 witnesses after an element is removed from the set to be around $0.3s$ on an ordinary laptop. Performing a similar update in the event of an element being added to the set is even faster.

[23] The concrete costs for aggregation and disaggregation correspond to the cost of updating witnesses for respectively deletions and additions of elements since they use the same techniques. See also numbers reported in footnote 22.

(we refer to [15] appendix C.2 for the formal reduction).

Then the extractor does the same for the second set of transcripts to get $\tilde{k}, \tilde{\vec{u}}, \tilde{s}, \tilde{r}$ such that $\hat{W}_{\tilde{\vec{u}}}^{\tilde{k}} = \hat{\mathrm{acc}}^{\tilde{h}} R$ and $\tilde{k} = \tilde{s}\tilde{h} \prod_i \tilde{u}_i + \tilde{r}$ over the integers. Now since $\tilde{\vec{u}}, \tilde{s}, \tilde{r}$ refer to the same commitment as $\vec{u}, s, r$ (recall that the commitment were sent a priori) from the binding of the pedersen commitment we get that $\tilde{\vec{u}} = \vec{u}, \tilde{s} = s, \tilde{r} = r$, which gives us that $\tilde{k} = s\tilde{h} \prod_i u_i + r$.

From the above we have: $\hat{W}_{\vec{u}}^{k} = \hat{\mathrm{acc}}^{h} R$ and $\hat{W}_{\tilde{\vec{u}}}^{\tilde{k}} = \hat{\mathrm{acc}}^{\tilde{h}} R$. Combining the two we get that

$$\hat{W}_{\vec{u}}^{k-\tilde{k}} = \hat{\mathrm{acc}}^{h-\tilde{h}} \Leftrightarrow$$

$\underline{\mathsf{Setup}\left(1^\lambda, \mathsf{ck}, \mathsf{pp}_{\mathsf{Acc}}\right):}$

$\mathsf{crs}_2 \leftarrow \mathsf{cp}\Pi^{\mathrm{arithm}}.\mathsf{Setup}(1^\lambda, \mathsf{ck}, \tilde{R}_{\mathsf{ck}}^{\mathrm{arithm}})$

$\mathsf{crs}_3 \leftarrow \mathsf{cp}\Pi^{\mathrm{bound}}.\mathsf{Setup}(1^\lambda, \mathsf{ck}, R^{\mathrm{bound}})$

return $\mathsf{crs} := (\mathsf{ck}, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{crs}_2, \mathsf{crs}_3)$

$\underline{P\,(\mathsf{crs}, \hat{\mathrm{acc}}, c_{\vec{u}}; W_{\vec{u}}, \vec{u}, o_{\vec{u}}, S)}$ $\underline{V\,(\mathsf{crs}, \hat{\mathrm{acc}}, c_{\vec{u}})}$

Sample $b_1, \ldots, b_{2\lambda} \leftarrow\$ \{0, 1\}$

let $s := \prod_{p_i \in \mathbb{P}_{2\lambda}} p_i^{b_i}, \ \bar{s} := \prod_{p_i \in \mathbb{P}_{2\lambda}} p_i^{1-b_i}$

$\hat{W}_{\vec{u}} \leftarrow W_{\vec{u}}^{\bar{s}}$

Sample $r \leftarrow\$ \{0, 1\}^{\|p^*\|+\|u^*\|+\lambda}$

$c_{s,r} \leftarrow \mathsf{Comm}_{\mathsf{ck}}(s, r; o_{s,r})$

$R \leftarrow \hat{W}_{\vec{u}}^{r}$

$$\xrightarrow{\pi_3, \hat{W}_{\vec{u}}, c_{s,r}, R}$$

$h \leftarrow\$ \{0, 1\}^\lambda$

$$\xleftarrow{h}$$

$k \leftarrow r + (s \prod_i u_i) h$

$\ell \leftarrow\$ \mathbb{P}_{2^{2\lambda}}$

$$\xleftarrow{\ell}$$

$Q \leftarrow A^{\left\lfloor \frac{k}{\ell} \right\rfloor}, \mathit{res} \leftarrow k \mod \ell$

$\pi_2 \leftarrow \mathsf{cp}\Pi^{\mathrm{modarithm}}.\mathsf{Prv}(\mathsf{crs}_2, \boxed{c_{\vec{u}}}, \boxed{c_{s,r}}, h, \ell, \mathit{res})$

$\pi_3 \leftarrow \mathsf{cp}\Pi^{\mathrm{bound}}.\mathsf{Prv}(\mathsf{crs}_3, c_{\vec{u}}, p_{2\lambda}; \vec{u}, o_{\vec{u}})$

$$\xrightarrow{(Q, \mathit{res}), \pi_2}$$

Accept if all are true:

- $Q \in \mathbb{G}_?$ and $\mathit{res} \in [0, \ell - 1]$
- $Q^\ell \hat{W}_{\vec{u}}^{\mathit{res}} = a\hat{c}c^h R$ and
- $\mathsf{cp}\Pi^{\mathrm{modarithm}}.\mathsf{Vfy}(\mathsf{crs}_2, c_{\vec{u}}, c_{s,r}, h, \ell, \mathit{res}, \pi_2) = 1$
- $\mathsf{cp}\Pi^{\mathrm{bound}}.\mathsf{Vfy}(\mathsf{crs}_{\mathsf{bound}}, c_{\vec{u}}, p_{2\lambda}, \pi_3) = 1$

Figure 9: Interactive version of our protocol for batch membership.

$$\hat{W}_{\vec{u}}^{sh\prod_i u_i + r - s\tilde{h}\prod_i u_i - r} = \mathrm{a\hat{c}c}^{h-\tilde{h}} \iff$$
$$\hat{W}_{\vec{u}}^{(s\prod_i u_i)(h-\tilde{h})} = \mathrm{a\hat{c}c}^{h-\tilde{h}}$$

From the low order assumption (which is implied by the adaptive root assumption) we get $\hat{W}_{\vec{u}}^{s\prod_i u_i} = \mathrm{a\hat{c}c}$.

Finally, the extractor runs once the extractor of $\mathsf{cp}\Pi^{\mathrm{bound}}$ to get that $u_i > 2\lambda$.

To conclude the proof, $(2, M)$-special soundness implies knowledge-soundness [3].

Zero-Knowledge: It comes directly from the standard rewinding-simulation Σ-method and the use of the simulators of $\mathsf{cp}\Pi^{\mathrm{modarithm}}$ and $\mathsf{cp}\Pi^{\mathrm{bound}}$. □

B.2 Security of the construction of section 5

We give a formal statement of the security of the scheme and then give an overview of the security proof. The proof can be seen as a simplification of the proof of theorem B.1.

Theorem B.2. *Let H_{prime} a hash-to-prime function modeled as a random oracle, H_{DI} be a division-intractable hash function and $\mathsf{cp}\Pi^{modarithm}$, $\mathsf{cp}\Pi^{H_{\mathrm{DI}}}$ be secure CP-SNARKs. The construction in fig. 4 for the relation $\tilde{R}_{\mathrm{ck}}^{ins}$ is a succinct and knowledge-sound, under the adaptive root assumption, CP-SNARK.*

Succinctness: is inherited from the succinctness of $\mathsf{cp}\Pi^{\mathrm{modarithm}}$, $\mathsf{cp}\Pi^{H_{\mathrm{DI}}}$ and PoKE.

Knowledge-Soundness intuition: the extractor proceeds similarly to a PoKE extractor, it rewinds the prover until it gets $M = \mathrm{poly}(\lambda)$ proofs $\{Q^{(i)}, res^{(i)}, \pi_2^{(i)}\}_{i\in[M]}$ and $\ell^{(i)}$ such that each proof verifies, $Q^{(i)\ell^{(i)}}\mathrm{acc}^{res^{(i)}} = \mathrm{acc}'$. For each proof it runs the corresponding extractor $\mathsf{cp}\Pi^{\mathrm{modarithm}}$ and gets a (common) $\vec{u}$ such that $u^* = res^{(i)} \bmod \ell^{(i)}$ for each $i \in [M]$. As argued in the proof of theorem B.1 for a sufficiently large $M > \|u^*\|/\|\ell\|$ we get that $\mathrm{acc}^{u^*} = \mathrm{acc}'$. This by using the CRT and a reduction to the adaptive root assumption from [15].

To conclude the extraction we additionally need a single π_3 and run the extractor of $\mathsf{cp}\Pi^{H_{\mathrm{DI}}}$ to get that $u_i = H_{\mathrm{DI}}(x_i)$.

For the non-interactive version, security come directly from the (tight) security of the Fiat-Shamir transformation for constant-round protocols [4], in the random oracle model.

C EXTENDING OUR CP-SNARK FOR BATCH MEMBERSHIP

C.1 Dealing with sets of arbitrary elements

The scheme described in the section 4 works for sets whose elements are suitably large prime numbers. Working with primes can be a limitation in practical applications. Here we describe how to get rid of this limitation and can support sets of arbitrary elements, such as binary strings. The idea is common in previous work and is to use a suitable collision-resistant hash function that maps arbitrary strings to prime numbers. What is a bit more complicated in our setting is that in order to prove membership of an arbitrary element, we need to prove the mapping to a prime.

Thanks to the commit-and-prove modularity of our protocol we can do this extension easily. This is the same idea used in [12]. Say that the prover holds a commitment $\hat{c}$ to a vector of binary strings $(\hat{u}_1, \ldots, \hat{u}_m)$. To prove the mapping the prover creates a commitment c to the primes $(u_1, \ldots, u_m)$ such that $u_i = H_{\mathrm{prime}}(\hat{u}_i)$, runs our CP-SNARK with c and adds a proof $\pi_{H_{\mathrm{prime}}}$ showing that $c, \hat{c}$ commit to elements such that $\forall i : u_i = H_{\mathrm{prime}}(\hat{u}_i)$. The latter proof can be generated via a CP-SNARK for this hashing relation. In particular, although a computation of H_{prime} involves several computations of a collision resistant hash function until reaching a prime, for the sake of proving one can use nondeterminism and prove a single hash evaluation (see [12] for details).

C.2 Succinct batch proofs of non-membership

We observe that by using a CP-SNARK for batch membership it is also possible to prove batch non-membership, if one accumulates sets using an interval-based encoding. The idea is that the elements of the set $S = \{x_i\}_i$ are ordered, $x_1 < x_2 < \cdots < x_n$, and the accumulator actually contains hashes of consecutive pairs $u_i = H(x_{i-1}, x_i)$. This way, proving that $x \notin S$ translates into proving that there is an element $u_j = H(x_{j-1}, x_j)$ in the accumulator such that $x_{j-1} < x < x_j$. The idea of interval-based non-membership proofs was introduced by Buldas et al. in the context of Merkle trees [19].

D DECENTRALIZED IDENTITY (DID) IMPLEMENTATION

We now give further details regarding our application to Decentralized Identity. While we focus and implement the specific scenario of cars insurance, we remark that the same approach can be applicable in other DID scenarios, such as financial instruments subscription and loans.

D.1 Scenario overview

We assume a car insurance scenario to show an example of how our approach can be used in a DID scenario. In many cases, a person who wants to take out car insurance has to submit her sensitive information such as health record, income, or diploma since the insurance company computes a premium through those information. We propose a privacy-preserving solution based on zero-knowledge and that does not require sending any data in the clear. In the remainder, we will denote by "*attribute*" the bits of information that can be useful for computing the premium and by "*holder*" the customer of the insurance (the one interested in preserving their information).

At the high level we store the attributes in a public accumulators. For privacy, we do not let the attributes in plain be the elements of the accumulators. Instead, the accumulated set consists of hiding commitments to the attributes (this commitment can be realized as a randomized hashing-to-prime). The idea is that the holder can compute her premium herself without revealing her attributes. The accumulator can be thought of as a portfolio of the (private) attributes of the holder. It gets updated with new credentials by authorized issuers and maintained publicly (through a consensus or a smart contract). When adding a new credential, the issuer broadcasts a signed transaction consisting of a hiding commitment to a valid attribute of the holder. The holder is given access to the commitment together with its opening.

Whenever necessary, the holder can compute the premium with the attributes using a public formula and show in zero-knowledge that the premium is computed correctly (we provide an example in appendix D.2. To do that, the holder does roughly the following: she generates a fresh Pedersen commitment $\mathsf{cm}_{\text{batch}}$ to the attributes of interest, then it produces a batch membership proof showing that 1) the freshly committed $\mathsf{cm}_{\text{batch}}$ actually contains members of the accumulator, 2) the computed premium is correctly computed with the attributes. At last, the verifier validates the batch membership proof with the accumulator in the network. In the description above we ignored for simplicity the fact that the elements in the accumulators are actually *commitments* themselves. The Pedersen commitment $\mathsf{cm}_{\text{batch}}$ should then actually be a commitment to commitments. The proof system should then show, besides membership, that the opening of these commitments (the elements in the set) satisfy the required premium relation. We stress that our implementation and evaluation do account for this.

D.2 Formula for computing insurance premium

We assume that 16 attributes are required to compute a premium of a single holder, and thus the holder has 16 commitments to these credentials, one for each attribute. In our benchmark we consider both 16 and 64 as the batch size. The former corresponds to the case of a single holder making a proof for her own premium; the latter corresponds to the case of a user generating a proof for 4 insurance premiums, a use case which plausibly applies to a family's or company's group subscription. The 16 attributes (a_i) and their weight (w_i) to compute the premium are as follows. Our attributes are variations from real world settings [2].

Attributes:

- a_0 : Driving history(date of license acquisition)
- a_1 : Married or single
- a_2 : Having a child or not
- a_3 : Completing a safe driving education
- a_4 : Driver's age
- a_5 : Driver's diploma(engineer)
- a_6 : Residential area
- a_7 : Income
- a_8 : Credit score
- a_9 : Driving habit(average driving hour per day)
- a_{10} : 1-year recent accident record
- a_{11} : $1 \sim 5$ year recent accident record
- a_{12} : penalty point record
- a_{13} : Specialized job
- a_{14} : property
- a_{15} : Health record(number of family history like acute myocardial infraction).

Weights (they are such that $[w_i]_{i=0}^{15} = 0$):

- w_0 : if $a_0 < 3$, then $w_0 = 600$, else if $3 \leq a_0 < 10$, then $w_0 = 300$, else if $a_0 \geq 10$, then $w_0 = -200$.
- w_1 : if $a_1 = 1$,(holder is married), then $w_1 = -120$.
- w_2 : if $a_2 = 1$, (holder has child), then $w_2 = -120$.
- w_3 : if $a_3 = 1$, (holder completed training), then $w_3 = -200$
- w_4 : if $20 \leq a_4 < 30$ or $50 \leq a_4 < 60$, then $w_4 = 0.02$, else if $30 \leq a_4 < 40$, then $w_4 = 0$, else if $a_4 \geq 60$, then $w_4 = 0.05$.
- w_5 : if $a_5 = 1$, (holder has engineer diploma) then $w_5 = -150$.
- w_6 : if $a_6 = 0$, (holder resides where the traffic accident rate is low), then $w_6 = -200$, else if $a_6 = 1$ (holder resides where the traffic accident rate is high), then $w_6 = 200$.
- w_7 : if $a_7 < 35000$, then $w_7 = 350$, else if $35000 \leq a_7 < 65000$, then $w_7 = 200$, else if $65000 \leq a_7 < 100000$, then $w_7 = 100$.
- w_8 : if $a_8 = 1$ (credit score is medium), then $w_8 = 100$, else if $a_8 = 2$ (credit score is low), then $w_8 = 150$.
- w_9 : if $a_9 < 2$, then $w_9 = -200$, $a_9 > 4$, then $w_9 = 150$.
- w_{10} : if $a_{10} = 0$, $w_{10} = -150$, else $w_{10} = a_{10} * 0.01$.
- w_{11} : $w_{11} = a_{11} * 100$.
- w_{12} : $w_{12} = a_{12} * 10$.
- w_{13} : if $a_{13} = 1$, then $w_{13} = -200$.
- w_{14} : if $a_{14} < 50000$, then $w_{14} = 500$, if $50000 \leq a_{14} < 100000$, then $w_{14} = 300$, if $100000 \leq a_{14} < 300000$, then $w_{14} = 200$, and if $300000 \leq a_{14} < 500000$, then $w_{14} = 100$.
- w_{15} : $w_{15} = a_{15} * 500$.

Formula to compute premium (Basic fee=1800 USD):
If $w_{10} = -150$, then:

$$premium = (1800 + \Sigma_{i \in [0,16]-\{4,10\}} w_i) * (1 + w_4 + w_{10})$$

otherwise:

$$premium = (1800 + \Sigma_{i \in [0,16]-\{4\}} w_i) * (1 + w_4).$$

Example Assume that the holder has the following attributes:

$$[a_0 = 20], [a_1 = 1], [a_2 = 1], [a_3 = 1], [a_4 = 49], [a_5 = 1], [a_6 = 0],$$
$$[a_7 = 70000], [a_8 = 0], [a_9 = 3], [a_{10} = 1], [a_{11} = 1], [a_{12} = 0],$$
$$[a_{13} = 0], [a_{14} = 300000], [a_{15} = 0].$$

Then, the premium must be 1121.1(USD) and the prover shows that the formulas above lead to this premium.