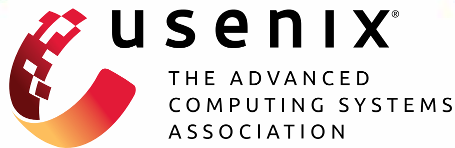

Notus: Dynamic Proofs of Liabilities from Zero-knowledge RSA Accumulators

Jiajun Xin, Arman Haghighi, Xiangan Tian, and Dimitrios Papadopoulos,
The Hong Kong University of Science and Technology

https://www.usenix.org/conference/usenixsecurity24/presentation/xin

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.

Notus: Dynamic Proofs of Liabilities from Zero-knowledge RSA Accumulators

Jiajun Xin   Arman Haghighi   Xiangan Tian   Dimitrios Papadopoulos

The Hong Kong University of Science and Technology

Abstract

Proofs of Liabilities (PoL) allow an untrusted prover to commit to its liabilities towards a set of users and then prove independent users' amounts or the total sum of liabilities, upon queries by users or third-party auditors. This application setting is highly dynamic. User liabilities may increase/decrease arbitrarily and the prover needs to update proofs in *epoch* increments (e.g., once a day for a crypto-asset exchange platform). However, prior works mostly focus on the static case and trivial extensions to the dynamic setting open the system to windows of opportunity for the prover to under-report its liabilities and rectify its books in time for the next check, *unless all users check their liabilities at all epochs*. In this work, we develop Notus, the first *dynamic PoL* system for general liability updates that avoids this issue. Moreover, it achieves $O(1)$ query proof size, verification time, and auditor overhead-per-epoch. The core building blocks underlying Notus are a novel zero-knowledge (and SNARK-friendly) RSA accumulator and a corresponding zero-knowledge MultiSwap protocol, which may be of independent interest. We then propose optimizations to reduce the prover's update overhead and make Notus scale to large numbers of users (10^6 in our experiments). Our results are very encouraging, e.g., it takes less than 2ms to verify a user's liability and the proof size is 256 Bytes. On the prover side, deploying Notus on a cloud-based testbed with eight 32-core machines and exploiting parallelism, it takes $\sim$3 minutes to perform the complete epoch update, after which all proofs have already been computed.

1 Introduction

The emergence of the decentralized finance (DeFi) ecosystem and cryptocurrency assets has led to increased demand for auditing and regulation of involved markets [22,24,27,41,63]. For instance, cryptocurrency holders may store their assets in exchange platforms lacking "traditional" financial regulation and policies [1,5,28,50], which leaves them open to incidents such as the recent collapse of the very popular exchange of

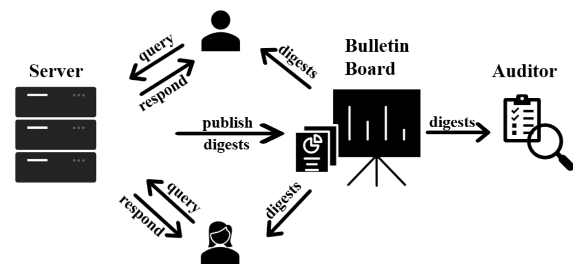

Figure 1: Model of a dynamic proof-of-liabilities system.

FTX due to malpractice [1]. This has led researchers and industry to explore the design of mechanisms that can provide strong security guarantees, regarding the availability and liquidity of funds in such systems. One particular such mechanism is *proofs of liabilities (PoL)*, a cryptographic primitive that allows a party to prove the amounts of funds it "owes" to its customers to a third-party auditor—e.g., as part of proving its financial solvency during an audit process.

In more detail, a PoL system (see Figure 1) consists of one *server* that stores liability (or transaction) records associated with its *users* and publishes a succinct *digest* of this dataset (e.g., a short, binding, cryptographic representation) on a public bulletin board, e.g., a blockchain platform. Users can query the server to get their latest record and check it is up-to-date and accurate with respect to the latest digest. The model also includes a third-party *auditor* who can periodically request the server to prove its total sum of liabilities with respect to the public digest or ask for the liabilities of individual users. From a security point of view, a dishonest server is clearly motivated to *under-report* these values [41], but not the other way around—artificially increasing liabilities offers no benefit as it makes it appear to "owe more" (see also our threat model discussion in Section 3.1). We also stress that a malicious server operates in a very favorable setting as it can readily tamper user records or even introduce artificial transactions and the only entities that can "catch" such behavior are the affected users themselves (we are not assuming any public-key or certificate infrastructure for authenticating records).

PoL has been the focus of a growing line of works and

various systems have been proposed in the academic literature [22–24, 27, 30, 36, 39, 41, 56] and adopted in practice, e.g., by Binance [6] and OKX [52]. Focusing on academic proposals, the state-of-the-art constructions include the DAPOL+ system [41], and the TAP system [56]. DAPOL+ proposed a scheme based on a Merkle tree of Pedersen commitments [54], with user liabilities stored at the leaves and the root published as the digest. The homomorphic property of Pedersen combined with efficient range proofs [13] makes it easy to prove individual liabilities (as well as sums of liabilities) in a secure and privacy-preserving manner (i.e., no "sibling" node information is revealed when proving membership). TAP takes a different approach, relying on *append-only dictionaries* [59] based on Merkle sum trees [40] for proving sums of liabilities; this is similar to DAPOL+'s Merkle tree with the additional requirement of sorting all tree leaves in ascending order.

Focusing on PoLs that have been deployed in practice, two highly-visible approaches come from exchange platforms Binance [6] and OKX [52]. Besides *third-party auditability*, they aim for a comprehensive set of features to accommodate their business model, including the ability to prove global sums across *multiple assets* while considering varying exchange rates. Binance [6] commits users' liabilities in a Merkle tree where each leaf is a hash digest of one user's liability information. It then uses succinct non-interactive arguments (SNARKs) to prove the sum of all users' liabilities committed in the Merkle tree across multiple assets. Because of the large number of users (45 million) they have, it is infeasible to generate a single SNARK proof that checks all users in the Merkle tree. Instead, Binance separates users into multiple batches, with each batch containing several hundreds of users, and constructs proofs for the entire set of users incrementally. Starting from an empty tree, it builds the entire Merkle tree with all users' liability information gradually. OKX [52] also commits users' liability information with Merkle trees. However, it employs scalable transparent arguments of knowledge (STARK) [3] as the proving system to achieve a faster prover and a transparent setup. Unfortunately, this comes at the cost of a much larger proof size. For ∼25 million users, the proof that needs to be checked (either by a trusted auditor, or by *each user* independently) is of size 1.2GB!

Challenges of Designing PoL in the Dynamic Setting. The applications PoL target clearly operate in a *dynamic* setting that can be expressed via a series of *epochs* during which users can perform transactions (deposits, withdrawals, and transfers) that will alter their liabilities. Correspondingly, the server should be able to provide proofs that are "fresh" with respect to the latest epoch (and the latest corresponding digest). For instance, a cryptocurrency exchange could be required by regulatory authorities to publish a new digest corresponding to its state regularly, e.g., Binance and OKX currently publish new digests and prove liabilities *every month*. This epoch duration may vary greatly for different applications from a few minutes (e.g., the time it takes to publish a new blockchain block) to once a year (e.g., for financial tax auditing). Despite this, most prior PoL works focus their security analysis and their system design on the *static* case. While this yields conceptually "simpler" schemes, one can argue it does not provide the comprehensive auditing capabilities that would be necessary to guarantee accountability. For instance, the monthly digests provided by Binance and OKX provide monthly independent *snapshots* to the auditor that are not "linkable", e.g., the auditing authority has no information about what transactions took place from one month to the next, in order to reach the new liabilities. Ideally, in a dynamic PoL, the auditing process should "link" the prior epoch's digest to the new in a way that guarantees certain properties. As we discuss next, such an extension of PoLs to the dynamic case introduces subtle challenges.

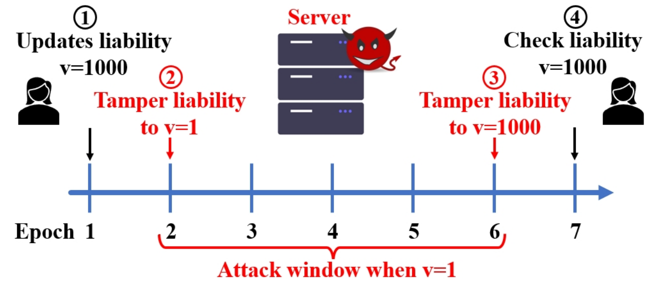

Figure 2: This figure illustrates a window-of-opportunity attack between the times a user checks her record. The user updates her liability in Step 1 and checks again in epoch 7. The malicious server reduces the user's liabilities to 1 in Step 2, and restores to 1000 in epoch 6, leading to a temporary decrease in its liability sum, without being caught.

Window-of-Opportunity for Attacks. In PoL systems, users check and verify their corresponding records to avoid foul play by the server. In the dynamic setting, this means that to guarantee security *all users must query for their records at all epochs*. If this does not happen, this gives a "window of opportunity" for the server to perform an attack by first lowering a user's liabilities, producing a decreased sum to the auditor that passes verification, and then increasing it again—all this within the time interval between two checks by the affected user (see Figure 2). To the best of our knowledge, existing PoL works suffer from this "catastrophic" attack. Indeed, [41] discussed the effect of users not verifying their records. Therefore, trivially extending static PoL schemes from the literature [6, 23, 24, 41, 52] would impose a very stringent condition for users to remain largely online and would incur massive communication cost to the system.

One exception to the above limitation is the recent TAP system [56] which is also the only work that is explicitly focused and designed for the dynamic case. However, by design, TAP can only support positive *monotonically increasing liabilities*. In practice, this is a limiting condition for several applications. E.g., in the Binance PoL [6], currency exchanges (say,

	Dynamic	Negative Updates	Update Pattern Privacy	Trusted Setup	Prover Overhead	Verifier Overhead for m epochs	Auditor Overhead	Proof of Sum size	Cryptographic Techniques
DAPOL+ [41]	No	N/A	N/A	No	$O(n\log n)$	N/A	N/A	$O(1)$	PedersenTree+RangeProof
TAP [56]	Yes	No	No	No	$O(n'\log n')$	$O(m\log n')$	$O(n')$	$O(m)$	PedersenTree+RangeProof
OKX [52]	No	Yes	N/A	No	$O(n\log^2 n)$	$O(m\log n)$	$O(\log^2 n)$	$O(1)$	STARK+Mekle Tree
Binance [6]	No	Yes	N/A	Yes	$O(n\log n)$	$O(m\log n)$	$O(1)$	$O(1)$	SNARK+Mekle Tree
Notus (Ours)	Yes	Yes	Yes	Yes	$O(n\log n)$	$O(1)$	$O(1)$	$O(1)$	SNARK+zkRSA Acc.

Table 1: Comparison of Notus with prior state-of-the-art PoL schemes. DAPOL+ is designed for the static case. TAP can only support monotonically increasing liabilities. Notation: n is the number of users in the system, n' the number of users whose records changed in one epoch, m is the number of consecutive epochs for which a user did not check its record.

from ETH to BTC) are encoded as increases *and decreases* in the respective currency accounts of the user, following the specified exchange rate. As another example of an application that requires both liability increases and decreases, consider decentralized loans [63]. When a user gets a loan, she needs to show her liabilities are less than her lending capacity. When the loan is issued, her liabilities increase, but after the loan is repaid, her liabilities should again decrease. Finally, with TAP, if a user delays checking by m epochs, eventually, she needs to perform an amount of work that is *linear* to m to "catch up", effectively checking her status for all intermediate epochs. Ideally, we would like to make this process efficient, e.g., it should only take a single check to catch up.

This Work. In this paper, we propose Notus, the first *dynamic proof-of-liabilities (**DPoL**)* system that supports general liability updates and achieves provable security and privacy guarantees in the dynamic setting. In particular, Notus overcomes the need for all users to check their records at all epochs. Instead, users can verify their records at arbitrary times while: (i) verification overhead and proof size is $O(1)$, and (ii) they have the guarantee that any (prior or ongoing) tampering of their record by the server will be identified. Compared to TAP [56], our system supports arbitrary (positive and negative) liability updates, m times faster verification after m unchecked epochs, and much lower overhead for the auditor's check in every epoch—$O(1)$ for Notus vs. linear in the number of transactions that took place in TAP. A side-effect of the last property is that making the auditor so lightweight makes it possible to instantiate it in a "trustless" manner, e.g., via a smart contract [15,48,57]. Finally, compared to commercially deployed systems [6,52], our system supports dynamic updates, shorter proof size, and concretely faster prover. Table 1 asymptotically compares Notus with prior works.

At a technical level, Notus combines a novel hidden-order-group *zero-knowledge accumulator* [34] for individual user queries, with an "accompanying" new *zero-knowledge MultiSwap* [53] protocol based on SNARKs for efficiently proving and verifying the execution of updates between consecutive epochs. To allow users to check their records at arbitrary times, within the accumulator we encode and connect the liability information of each user via a hash chain. Notably, updating a *private* dataset while enforcing certain checks via SNARKs is challenging from an efficiency perspective. To propose a system with good practical performance, we designed our accumulator to be "SNARK-friendly" to keep the prover overhead low. At the same time, to keep the "online" overhead for the server minimal, we adopt a massive pre-computation strategy for the accumulator proofs. To scale this approach to numerous users (as we expect to be the case for PoL applications) we introduce a series of important optimizations.

Our overall contributions can be summarized as follows:

1. We propose security and privacy definitions for **DPoL** and the first scheme that satisfies them for general liabilities called Notus. It achieves $O(1)$ proof size, verification cost, and auditor overhead (Section 3).
2. We propose the first zero-knowledge RSA accumulator, which supports precomputing all zero-knowledge membership proofs in $O(n\log n)$ time. We also propose the first zero-knowledge MultiSwap (ZK-MultiSwap) protocol based on our accumulator. Its overhead is similar to that of prior (non-zero-knowledge) MultiSwaps [16]. (Section 4). Finally, we propose optimization techniques for our RSA accumulator and Notus. (Section 5).
3. We experimentally evaluate the performance of our constructions. Notus achieves the fastest update speed among current PoL schemes (e.g., 3 mins to update 1 million users) and achieves extremely fast verification cost (2ms for user liabilities) and proof size 256Bytes (Section 6).
4. Our zero-knowledge RSA accumulator also works as a zero-knowledge lookup system. Compared to the state-of-the-art Caulk [67] system for zero-knowledge lookups, our accumulator achieves up to *three orders of magnitude faster* online prover time with similar offline time.

2 Preliminaries

Notation. Let λ denote a security parameter. A function $\text{negl}(\lambda)\colon \mathbb{N} \to \mathbb{R}^+$ is negligible if for every positive polynomial $poly(\lambda)$ there exists a $\lambda_0 \in \mathbb{N}$, such that for all $\lambda > \lambda_0$: $\varepsilon(\lambda) < 1/poly(\lambda)$. Let $[i,j] = \{i, i+1, \ldots, j-1, j\}$. We denote by $x \xleftarrow{\$} F$ sampling uniformly at random from domain F. $\mathcal{H}_\lambda$ is a public function family, where every $H \in \mathcal{H}_\lambda$ is modeled as a random oracle that maps an arbitrary string to a random number with λ bits. Formally, $H : \{0,1\}^* \to \{0,1\}^\lambda$.

Proof of Exponentiation. Given a hidden order group $\mathbb{G}$, one integer x and two group elements g, y, a Proof of Exponentiation (PoE) protocol proves exponentiation relation $g^x = y \in \mathbb{G}$. We follow the approach of Wesolowski [62],

generalized in [9]. It allows the verifier to check the exponentiation in constant calculations, where the verifier first sends a prime challenge l, the prover calculates q, r s.t. $x = ql + r, r \in [0, l)$, and sends $\pi = g^q \in \mathbb{G}$ to the verifier. The verifier computes $r = x \mod l$ and checks $\pi^l g^r \stackrel{?}{=} y$. Boneh *et al*. extended the above idea and provided a non-interactive zero-knowledge proof of knowledge of exponentiation (NI-ZKPoKE) in [9]. We refer to their protocol as (1) a prove function $\pi \leftarrow$ **NI-ZKPoKE.Prove** $(G, g; x)$ where π is the proof, x is the witness and $G = g^x$, and (2) a verification function $1/0 \leftarrow$ **NI-ZKPoKE.Verify**(G, g, π).

RSA Accumulators. An RSA accumulator [4, 9, 45] is a cryptographic commitment to a set $S = \{e_1, \ldots, e_n\}$. It has pros/cons compared to Merkle tree-based accumulators and bilinear accumulators as discussed in [59, 60]. Its main advantage is constant-sized membership proof for any set size.

Setup. An RSA accumulator can be set up by picking two random large prime p', q', calculating $N = p'q'$, and finding one large group $\mathbb{G} \subseteq \mathbb{Z}_N^*$ and its generator g such that strong RSA assumption [2] and adaptive root assumption [62] holds in $\mathbb{G}$. We additionally require that p', q' are two primes, i.e., $p' = 2p + 1$ and $q' = 2q + 1$, and that the group order is odd (pq). The most commonly used such group is the quadratic residue group of N $(QR(N))$. This setup process can either be performed by a trusted party, or via a multi-party protocol [10, 14] to generate the group $\mathbb{G}$. Alternatively, one can also rely on class groups [12] which do not need a trusted setup but are slower in performance and less studied.

Membership proofs. Let $S = \{e_1, e_2, \ldots, e_n\}$ denote a set of elements that are either prime numbers or output of Division Intractable (DI) hashes [33]. $acc(S)$ denotes the digest of committing set S using accumulator $acc(S) = g^{\prod_{e_i \in S} e_i}$. To prove one element $e' \in S$, the prover calculates $\pi = g^{\prod_{e_i \in S \setminus \{e'\}} e_i}$. Verifier checks if $\pi^{e'} \stackrel{?}{=} acc(S)$. All such membership proofs can be generated in batch (e.g., in pre-computation) in time $n \log n$ using divide-and-conquer (e.g., see [60]).

MultiSwap. MultiSwap is a cryptographic primitive for proving the update state of an accumulator C after removing a set X and adding a set Y is C'. It was introduced by Ozdemir et al. [53] and improved by Campanelli et al. [16]. Both schemes rely on SNARKs for this check, which we introduce next.

SNARKs. A **S**uccinct **N**on-interactive **AR**gument of **K**nowledge (SNARK) for a relation $\mathcal{R}$ comprises of three algorithms as follows:

- Π.**Setup**$(1^\lambda, \mathcal{R}) \rightarrow$ crs. crs is a common reference string.
- Π.**Prove**$(\text{crs}, x; w) \rightarrow \pi$. x is a statement and w is the witness such that $\mathcal{R}(x, w)$ holds. It outputs π as proof.
- Π.**Verify**$(\text{crs}, x, \pi) \rightarrow \{0, 1\}$. It inputs the crs, statement x and proof π. If the proof checks correctly, it outputs 1; otherwise, it outputs 0.

A SNARK is *complete, knowledge-sound, and succinct*. Completeness means that if $\mathcal{R}(x, w)$ holds, an honest prover fails to generate the proof π with probability less than negl(λ). Knowledge soundness means that given a valid proof, there exists an extractor to extract the witness for that statement. Finally, a SNARK is succinct if the proof and verification time are poly-logarithmic in the witness size. This property allows the verifier to check any statement in $\mathcal{R}$ faster than checking the statement and witness (if given) directly. If a SNARK is also zero-knowledge, it leaks no information of the witness and is called zkSNARK [8]. For our implementation, we use a Groth16-type SNARK [38], which requires a trusted setup.

In this paper, we use a Commit-and-Prove SNARK (CP-SNARK) [17] for the security and composition of different SNARK proofs. Informally, the prover can commit the inputs and witnesses of a SNARK through some commitment scheme, e.g., extended Pedersen commitments which are perfect hiding and computationally binding. We denote $\boxed{c_{w1}}$ the committed witness $w1$ for a CP-SNARK proof $\pi \leftarrow \Pi$.**Prove**$(\text{crs}, \boxed{c_{w1}}, x; w2)$ where $w2$ is the non-committed part of the witness. As discussed in [17], CP-SNARK allows the modular composition of SNARKs through the committed witness. For example, given two relations $\mathcal{R}_1$ and $\mathcal{R}_2$, the prover can prove $\mathcal{R}_1(\boxed{c_{w1}}; w_2)$ and $\mathcal{R}_2(\boxed{c_{w1}}; w_3)$ hold with two CP-SNARKs for the same committed witness $\boxed{c_{w1}}$.

Definitions of range proofs, universal accumulators, and division intractable hashes can be found in Appendix A. Definitions of cryptographic assumptions and the generic group model can be found in the extended version of our paper [65].

3 Definition of Dynamic Proofs of Liabilities

Here we propose our new definition for dynamic PoL. Looking ahead, in our dynamic PoL, the server demonstrates the "correct update" between successive epochs and their respective digests. This is in contrast to relying on a static PoL and simply publishing a series of unrelated digests, computed from scratch each time. Note that, during consecutive epochs, some users' liabilities may remain unchanged. Any modifications should be accurately captured in the updated digest while also updating the total sum of liabilities. Next, we first provide an overview of the **DPoL** system and explain its data structures and threat model. Then we present function definitions along with formal security and privacy definitions for the **DPoL**. Finally, we discuss the limitations of current works and why they do not satisfy such security requirements.

3.1 System overview and data structures

A **DPoL** system is a versioned ledger system, where time is split into epochs, and each epoch bounds a unique ledger digest. It involves three entities: users, a server, and an auditor (Figure 1). We also assume a public bulletin board where digests are published (e.g., a public blockchain). We denote $\mathcal{U} = \{u_1, \ldots, u_t\}$ as the set of t users.

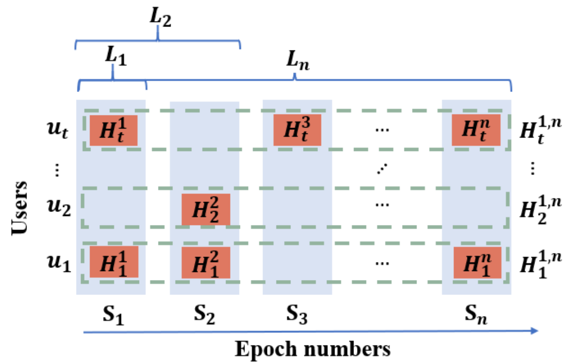

Figure 3: The ledger can be examined both vertically and horizontally. Each row represents the transaction history of an individual user, whereas each column represents the updated users within a single epoch. As the epoch number increases, the ledger naturally expands column-wise.

Users update their liabilities through transactions, which are subsequently stored on the server. The server, denoted by $\mathcal{S}$, maintains the versioned ledger in an append-only way for all transaction histories. It provides a digest of the whole ledger, the sum of all liabilities, and the lookup proof for each user's transaction history. The server provides an updated digest and corresponding proofs in each new epoch to show the ledger is append-only. It publishes the digest and the append-only proof on the bulletin board. Users retrieve the latest digest from the public bulletin board and can check the correctness of their records (full transaction histories) in the ledger versus this digest. The auditor checks the correctness of the updated digest in every epoch and the correctness of the sum of liabilities and user histories when delegated.

Data structures. The ledger is a set of transactions (txs) where each tx is a tuple of user ID, this user's final liability after the update, and the timestamp of the updated epoch. We use $tx.id$, $tx.lia$ and $tx.upd$ to denote these values respectively. Without loss of generality, we assume these have a fixed bit length, and user IDs are unique. Within a single epoch, users may perform multiple operations on their liabilities, but only one transaction is needed to summarize the liability updates. A user's liability can increase or decrease across different epochs but must always remain non-negative.

The ledger operates on an append-only basis, maintaining a complete history of every transaction. Let L_i represent the state of the ledger at epoch i. Since each transaction includes a timestamp indicating when it was added, it is easy to deduce all previous versions of the ledger, denoted as L_j, for all $j < i$, given the current ledger state L_i. For a user denoted by u, $H_u^{i,j}$ represents the history of transactions (i.e., the complete set of all transactions) occurring between epochs i and j. It is evident that $H_u^{1,i} \subseteq H_u^{1,j}$, for all instances where $i \leq j$. We use H_u^j to denote the transactions of user u in epoch j and $H_u^j.lia$ to represent her final liability in epoch j. In cases where a user does not update during epoch j, her final liability value $H_u^j.lia$ will remain the same as her previous liability, $H_u^{j-1}.lia$.

We denote S_i as the set of transactions in epoch i. The notation $u \in S_i$ refers to a user who performed an update in epoch i. Clearly, $S_i = L_i \setminus L_{i-1} = \bigcup_{\forall u \in S_i} H_u^i$ and $L_j = \bigcup_{i=1}^{j} S_i = \bigcup_{\forall u \in \mathcal{U}} H_u^{1,j}$. We illustrate one example of such data structure in Figure 3. The sum of liabilities for ledger L_i is the sum of all users' final liabilities in epoch i. For example, let's consider epoch 3 in Figure 3. In this epoch, one update happened for user u_t. The sum of liabilities at epoch 3 equals to $H_1^2.lia + H_2^2.lia + H_t^3.lia$.

Threat Model. In the **DPoL** system, a malicious server tries to opportunistically *decrease* the sum of all liabilities even for brief periods [41]. We stress that artificially increasing the amount of liabilities is not a valid attack vector as it would make the adversary appear to have more debt (in practice when used to prove the financial solvency of a party, a PoL will typically be combined with a, conceptually simpler, proof of reserves [6, 41, 52] that complements it by stopping the adversary from over-reporting).

To achieve its goal, a malicious server may employ various tactics, such as deleting transactions, modifying transactions' values or their order, intentionally delaying transaction updates, and introducing counterfeit transactions (e.g., with negative liabilities), including the possibility to *fabricate and manipulate* any number of "virtual users" to further its objectives. Honest users update their own transactions as they occur and need to check their state "*only occasionally*" (as opposed to mandatorily checking at every epoch) through membership queries. Our auditor is assumed honest. For every epoch, it receives the updated public digest and proof from the server that allows it to check if the server has correctly updated the system, according to the following properties: (1) the update is append-only; (2) the update only appends transactions with a timestamp of the current epoch number; (3) the update does not cause any user's liability to become negative; (4) the sum of all liabilities is updated consistently with the transactions. Finally, regarding the privacy of the ledger, we consider both users and the auditor to be curious, i.e., they may try to infer additional information about the data, besides what they know (from their specific transactions and the sum of liabilities), e.g., transactions of other users, update patterns of users, etc.

Users colluding with the server & fake accounts. We stress that our threat model already considers the possibility of users colluding with the server, to help under-report liabilities, trivially since the latter can always "spawn" fake users anyway. Looking ahead to our solution, as long as we ensure that the liabilities of all user accounts in the system remain "positive", creating fake accounts or colluding with users can only *increase* the server's liability, and does not benefit the adversary.

Window-of-opportunity attacks & fake transactions. Since we do not assume a public-key or certificate infrastructure for authenticating records, neither do we require all users to check their history at every epoch, "window-of-opportunity"

(Figure 2) attacks are always possible. Recall that this entails the server issuing fake transactions to artificially lower liabilities, pass the audit, and rectify the liabilities in time before affected users perform their checks. Our threat model addresses this by ensuring the ledger is append-only. If the server artificially lowers liabilities by fake transactions, evidence will remain "in perpetuity" (captured by subsequent digests). Thus, affected users can *eventually* catch the cheating server and have evidence of this malpractice.

3.2 DPoL API

In **DPoL**, the prover implements the following algorithms:

- $(ek, vk) \leftarrow$ **Setup**(1^{λ}). Randomized algorithm that returns evaluation key ek used by the prover and a verification key vk used by users. Here, λ is the security parameter.
- $d_i \leftarrow$ **Digest**(ek, L_i, aux_i). Randomized algorithm that generates digest d_i. aux_i is the auxiliary information (e.g., randomness) needed to generate the digest up to epoch i.
- $\pi_{\text{mem}} \leftarrow$ **LookupProof**(ek, L_i, u, aux_i). This algorithm provides a lookup proof for user u's full history in epoch i with a membership proof π_{mem}.
- $\pi_{\text{sum}} \leftarrow$ **ProveSum**(ek, L_i, aux_i). This algorithm proves the sum of all liabilities with a sum proof π_{sum}.
- $\pi_i \leftarrow$ **ProveConsistent**(ek, L_i, aux_i). Randomized algorithm that proves the ledger committed in d_i is append-only and consistent, resulting from the application of changes in epoch i to the ledger committed in d_{i-1}.

The auditor implements the following algorithms:

- $\{1/0\} \leftarrow$ **VerConsistent**$(vk, i, d_{i-1}, d_i, \pi_i)$. This algorithm ensures the transactions are append-only and consistent after the changes in epoch i.

The auditor and users implement the following algorithms:

- $\{1/0\} \leftarrow$ **VerLookup**$(vk, d_i, H_u^{1,i}, \pi_{\text{mem}})$. This algorithm verifies proofs returned by **LookupProof**$(\cdot)$ against the digest d_i for user u. It verifies the full history of user u is in the set via the membership proof π_{mem}.
- $\{1/0\} \leftarrow$ **VerSum**$(vk, d_i, \text{sum}, \pi_{\text{sum}})$. This algorithm verifies proofs returned by **ProveSum**$(\cdot)$ against the digest d_i. It verifies sum is the total final liabilities committed in d_i.

3.3 Correctness and Security of DPoL

We require that a Dynamic Proof of Liabilities (DPoL) system satisfies the following properties: Completeness, Undeniability, Update Soundness, Sum Soundness, and Privacy. We provide their intuitive explanations here and defer their formal definitions to Appendix B.

Completeness ensures that when all participating entities act honestly and adhere to the established protocol, verification will always succeed. Note that this definition handles all possible orders of appending elements.

Undeniability ensures the non-equivocation of the committed ledger and the sum: for any digest d_i, it bounds a unique sum and a unique ledger. It also implies *collision-resistance*: i.e., different ledgers and sums cannot have the same digest.

Update Soundness ensures that: 1) the update is append-only, the ledger L_i is a superset of ledger L_{i-1}; 2) all transactions updated in this epoch ($tx \in S_i$) have the update epoch number the same as the current epoch number i; 3) all updated users' final liabilities are still positive; 4) the sum of all liabilities is also updated accordingly and consistently. One special case of this definition is inserting new users. For a new user u in S_i, H_u^{i-1} does not exists, we set $H_u^{i-1}.lia = 0$ as a special case.

Sum Soundness ensures that the sum of all liabilities of the ledger is not smaller than the total sum of liabilities of those honest users who have checked their own entries.

Privacy ensures the proofs do not reveal any additional information about the **DPoL** system except for what is explicitly provided by the server. We formulate this by a real/ideal simulation paradigm for an adversary that corrupts users and the auditor, chooses the ledger, and makes lookup/sum queries. We require that the view of this adversary throughout multiple epochs can be simulated without access to the ledger.

Having defined these security properties, we now examine why existing schemes fail to meet them, setting aside any functional constraints they may have and the fact that some of them are only set on the static setting. A common issue impacting Soundness in static PoL systems is the necessity for users to examine all intermediate states across every epoch to ascertain their full history. DAPOL+ [41] does not satisfy **DPoL**'s Update Soundness definition: for any epoch, the sum of total liabilities is only "correct" when all users verify their membership proofs. In other words, DAPOL+'s soundness hinges on all transactions being verified by some "entities" one by one, either by the union of all honest users cooperatively, or by a "powerful" auditor with access to all users' information. TAP [56] does not satisfy **DPoL**'s Privacy definition. In each epoch, the TAP system's server generates a tree consisting of sorted leaves. The number of these leaves discloses the number of updated users. When a user's transaction is present in the leaves, she can identify her transaction value's position within the sorted set, thereby obtaining additional information about the distribution of other users' transaction values. This presents a privacy issue, as multiple colluding users could merge their insights to deduce statistical information about other participants in the system. Binance [6] does not satisfy **DPoL**'s Update Soundness definition. In their system, the auditor's role is limited to ensuring that the sum is accurately calculated based on all liabilities committed to the Merkle tree without guaranteeing the ledger is strictly append-only. It also does not satisfy **DPoL**'s Privacy definition. Binance employs fixed user IDs to denote users' positions within the Merkle tree and relies on the Merkle path for membership proof, inadvertently exposing information about each user's neighbors.

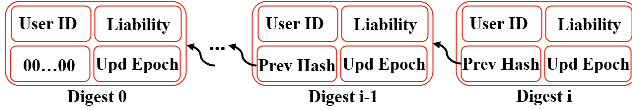

Figure 4: HashChain used in Notus system. Each hash combines the user ID, user's liability, update epoch, and the previous hash as its input. The first hash in the chain uses all "0"s as a special mark.

Likewise, OKX [52] does not satisfy **DPoL**'s Update Soundness definition similar to Binance as it has no append-only guarantees.

4 Notus Design

In this section, we present our instantiation of a **DPoL** system, named Notus. Our objectives, in addition to the security and privacy requirements of the **DPoL** system, include ensuring constant-sized digests, membership proofs, and proof-of-consistency. Moreover, we aim to improve server efficiency by allowing membership proofs to be pre-computed in-batch.

To achieve these goals, Notus encodes each user's history using a HashChain, which results in a succinct representation that makes it easier to verify the append-only property. Notus stores the HashChain digests of all users in a zero-knowledge RSA accumulator and employs zero-knowledge MultiSwaps to confirm two consecutive epochs are updated correctly. The remainder of this section is organized as follows. We first introduce three building blocks of Notus, namely the HashChain, zero-knowledge RSA accumulators, and zero-knowledge MultiSwaps. Next, we provide an in-depth explanation of the Notus system, using the three building blocks as black boxes.

4.1 HashChain

HashChain is a fundamental data structure in the design of secure and tamper-evident systems. It operates by incorporating a series of hash values and linking them in a sequential manner to form a chain. Its key property is immutability, as any attempt to alter the data or its sequence would result in a different digest. This feature allows for the demonstration and verification of the append-only nature of the HashChain. Notus utilizes the HashChain data structure to encode users' transaction histories to achieve the following goals:

1. Ensure the server can unambiguously demonstrate that updates to users' histories are strictly append-only, thus maintaining the integrity of the transaction records.
2. Enable users and the auditor to inspect the most recent HashChain outcome, which in turn facilitates the validation of all intermediate states in the transaction history.
3. Grant users and the auditor the capability to initiate subsequent examinations from a checkpoint once they have verified the transaction histories up until that specific point.

Notus commits each user's transaction history into a HashChain digest k_u^i where u denotes the user and i denotes the epoch number. Notus builds the HashChain with hash functions $\textbf{Hash}(\cdot)$ that are collision resistant and pre-image hard. More specifically, $k_u^i = \text{HashChain}(H_u^{1,i}) = \textbf{Hash}(tx.id||tx.lia||tx.upd||\text{previous Hash})$ where tx denotes the latest transaction H_u^i, $||$ denotes the string concatenation and previous Hash denotes k_u^{i-1}. For the initial Hash where the previous Hash does not exist, we use all "0" as a special mark. We illustrate one example in Figure 4.

Because the hash function is collision resistant and $tx.id$, $tx.lia$, $tx.upd$ have fixed bit length, we summarize HashChain's properties as follow:

- Transitive. For $H_u^{1,i+1} = H_u^{1,i} \bigcup tx(u, lia, i+1)$, $k_u^{i+1} = \textbf{Hash}(u||lia||i+1||k_u^i) = \text{HashChain}(H_u^{1,i+1})$.
- Collision resistant. For any two different transactions histories $H_u^{1,i}$, $\hat{H}_u^{1,i}$, $\Pr\left[\text{HashChain}(H_u^{1,i}) = \text{HashChain}(\hat{H}_u^{1,i})\right]$ is negligible in λ.

We denote the digest of the server for epoch i as d_i. For one user u, a valid membership proof should convince her that k_u^i is summarized in d_i. Based on the transitive and collision resistant property of HashChain, once user u checks her membership proof in epoch i, to check her membership proof in epoch $j > i$, she only needs to hold her HashChain digest k_u^i and the transactions history since epoch i, which is $H_u^{i+1,j}$. In other words, once a user checks her membership proof in epoch i, epoch i becomes a checkpoint for her.

4.2 Zero-knowledge RSA accumulator

Zero-knowledge accumulators enable a prover to efficiently commit to a set of values while preserving binding (undeniability) and hiding (zero-knowledge) properties. Adversaries cannot obtain any information from the digest and (non)membership proofs, apart from the information explicitly provided by the prover. The functional definitions of zero-knowledge accumulators resemble those of Universal accumulators in Appendix A. The primary distinction between the two lies in the use of "blinding" random values while generating digests in zero-knowledge RSA accumulators, which is essential for privacy, as demonstrated in [34].

Zero-knowledge RSA accumulator construction. To achieve undeniability and zero-knowledge properties in RSA accumulators, a straightforward strategy is to employ two distinct generators, labeled g and h. Specifically, the generator g is dedicated to committing the product of elements, represented by x, while h is designated for committing random numbers, denoted by t, e.g., $g^x h^t \mod N$. This method is similar to the classical Fujisaki-Okamoto integer commitments [31]. However, the two-generator structure necessitates additional overheads for membership-proof precomputation (and, looking ahead, for the incorporation of the accumulator checks into a SNARK). To avoid this, we choose to employ

$\underline{\textbf{RSA.KeyGen}(1^\lambda)}$	$\underline{\textbf{RSA.Acc}(ek, X, t)}$	$\underline{\textbf{RSA.Witness}(X, x, C, ek)}$	$\underline{\textbf{RSA.Verify}(C, x, w_x, b, vk)}$
return: (ek, vk)	**return:** C	**return:** (b, w_x)	**return:** $0/1$

(a) Zero-knowledge RSA accumulator functionalities. **Acc**$(\cdot)$ function accumulates the set X and at least one random number t, and outputs the digest C. **Witness**$(\cdot)$ function generates a (non)membership proof, while its output b denotes it is a membership (when $b = 1$) or a non-membership (when $b = 0$) proof.

$\underline{\Pi.\textbf{Setup}(1^\lambda, \mathcal{R}^{tent})}$	$\underline{\Pi.\textbf{Prove}(\text{crs}, C, C', C_{mid}, \boxed{c_{\vec{u}0}}, \boxed{c_{\vec{u}1}}; \tau_0, \tau_1, X, Y)}$	$\underline{\Pi.\textbf{Verify}(\text{crs}, C, C', C_{mid}, \boxed{c_{\vec{u}0}}, \boxed{c_{\vec{u}1}}, \pi)}$
return: crs	**return:** π	**return:** $0/1$

(b) ZK-MultiSwap functionalities. Proof of knowledge of set X, Y s.t. removing X and inserting Y for C gets C', and a tentative relation $\mathcal{R}^{tent}$ holds for (X, Y). crs is the common reference string, C, C', C_{mid} are zero-knowledge RSA accumulators. Hash values of set X and τ_0 are committed in $\boxed{c_{\vec{u}0}}$, and hash values of set Y and τ_1 are committed in $\boxed{c_{\vec{u}1}}$. Hash of τ_0 and τ_1 are used as randomizers.

Figure 5: Zero-knowledge RSA accumulators and ZK-MultiSwaps functions we use to build Notus. We use RSA and Π as prefixes to denote them, respectively.

a single generator to commit the product x of all elements and at least one randomizer t to maintain privacy, e.g., g^{xt} mod N. This RSA accumulator configuration allows for easy precomputation of all membership proofs and can be input into SNARKs with a simple PoKE proof (as we will show later). To further optimize our RSA accumulator for SNARK compatibility, we construct it directly with DI hashes. For ease of presentation, we defer the details of our construction in Appendix C and prove its security and privacy in the extended version of our paper [65]. Here, we only provide the API of zero-knowledge RSA accumulators in Figure 5.

Zero-knowledge Subset. Given two zero-knowledge RSA accumulators, where one is a subset of the other, we can provide a zero-knowledge subset proof of constant size. This subset proof enables efficient updates to zero-knowledge RSA accumulators and serves as a building block for the ZK-MultiSwap protocol, which will be discussed later. Boneh et al. [9] introduced both PoKE and ZK-PoKE protocols for RSA accumulators. It is possible to integrate a ZK-PoKE protocol into our accumulator to prove zero-knowledge subsets. However, the ZK-PoKE necessitates additional elements and incurs increased proving/verifying overhead compared to a PoKE, particularly when combined with SNARKs. Instead, we next design an new zero-knowledge subset protocol for our accumulator based on PoKE protocol and SNARKs.

The high-level idea of our zero-knowledge subset is putting redundant randomizers into the zero-knowledge RSA accumulator beforehand, and utilizing the randomizer to "hide" the subset (exponent). In this way, the group elements and integers in the PoKE transcript become simulatable and therefore the whole protocol achieves zero-knowledge. Due to space limitations, we defer construction and proofs to Appendix D. We also show that our zero-knowledge subset protocol implies an efficient zero-knowledge set insertion protocol for the relation $\mathcal{R}^{Ins}(C, \boxed{c_{\vec{u}}}; \tau, X) = C'$ where the witness vector $\vec{u}$ are hash outputs of τ and X, i.e., $H_K(\cdot)$ is a secure DI hash that can be modeled as a random oracle, $H_K(\tau) = t$ and t is the randomizer for accumulator C'.

4.3 ZK-MultiSwap

ZK-MultiSwap proves that removing a subset X from a zero-knowledge RSA accumulator C and inserting a new set Y leads to the new zero-knowledge RSA accumulator C'. Tentative relations between (X, Y) can be checked by SNARKs and the transcript of the protocol needs to be zero-knowledge. Based on the same analysis in [53], given two zero-knowledge accumulators C, C', a ZK-MultiSwap relation $\mathcal{R}^{MSwap}$ is equivalent to proving an intermediate accumulator C_{mid}, and two set insertion proofs for inserting sets (X, Y) s.t. $\mathcal{R}^{Ins}(C_{mid}, \boxed{c_{\vec{u}0}}; \tau_0, X) = C \wedge \mathcal{R}^{Ins}(C_{mid}, \boxed{c_{\vec{u}1}}; \tau_1, Y) = C'$, while maintaining zero-knowledge. Based on our zero-knowledge RSA accumulator and zero-knowledge subset protocol, we construct a protocol for ZK-MultiSwap. The detailed construction and proofs are in the extended version of our paper [65], and we provide here its function interfaces in Figure 5. Compared with Harisa [16], our ZK-MultiSwap achieves zero-knowledge with the cost of verifying two $H_K(\cdot)$ inside a SNARK, instantiated with two SNARK-friendly sponge hashes like [37].

Dynamic ZK-MultiSwap. We further prove in the dynamic setting of ZK-MultiSwap, the above scheme is still complete, sound, and zero-knowledge. In the dynamic setting, the prover is required to demonstrate a series of chained MultiSwaps where the updated accumulator becomes the original accumulator in the subsequent epoch. Let's represent the series of accumulators as $C_0, C_1, C_2, \ldots, C_n$ and the sequence of acc_{mid} as $C_{0,1}, C_{1,2}, \ldots, C_{n-1,n}$. We employ a "zig-zag" approach to randomize the accumulators. This method ensures that at least one random number is assigned to each accumulator. We denote these random numbers as t_i. Specifically, C_0 is randomized using t_0 and t_1, C_1 is randomized using t_1 and t_2, and so on. Similarly, $C_{0,1}$ is randomized using t_1, $C_{1,2}$ is randomized using t_2, and so forth. In the dynamic setting, the

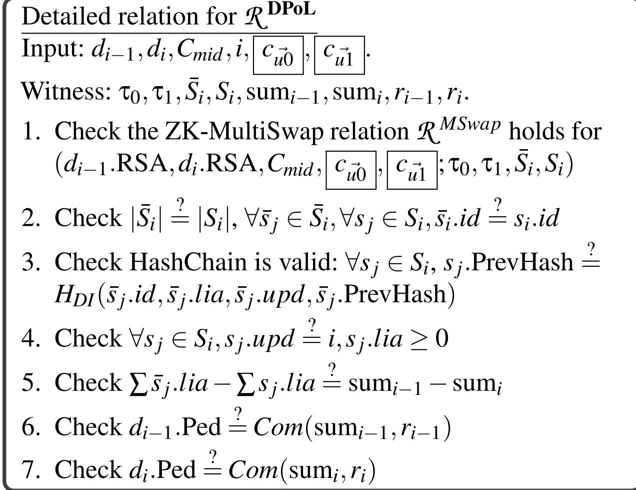

Figure 6: Detailed relation for **DPoL**. We denote transactions in $\bar{S}_i$ as $\bar{s}_1, \dots, \bar{s}_j$ and transactions in S_i as $s_1, \dots, s_j$.

relation $\mathcal{R}^{MSwap}$ holds for all consecutive pairs, i.e., (C_0, C_1), (C_1, C_2), ..., and (C_{n-1}, C_n). The proof is in the extended version of our paper [65].

4.4 Notus construction

Finally, we can provide details about the construction of Notus. We construct the HashChain using a DI hash H_{DI}, which is collision resistant, and division intractable. The data structure we input into the DI hash is the concatenation of $(tx.id||tx.lia||tx.upd||\text{previous Hash})$. For user u updated in epoch i, her HashChain digest $k_u^i = \text{HashChain}(H_u^{1,i}) = H_{DI}(tx.id||tx_t^i.lia||i||k_u^{i^-})$ where $k_u^{i^-}$ is the hash digest when previous updated in epoch i^-. Given a ledger L_i, we denote D_i as the collection of all users' latest HashChain digest. We use $D_i \leftarrow \mathbf{LedgerDigest}(L_i)$ as a helper function. Recall that we denote S_i the set of transactions accumulated in the accumulator for epoch i. We denote $\bar{S}_i$ as the set containing the previous states of transactions that are updated in epoch i. Transactions in $\bar{S}_i$ and S_i are pairwise, i.e. $\forall u \in S_i$, $u \in \bar{S}_i$ and $\bar{S}_i \in L_{i-1}$. During an update, updated users' old digests are removed, and corresponding new digests based on $\bar{S}_i$ and S_i are generated and inserted. $L_i \setminus S_i$ is the set of transactions not updated in epoch i. We use D_{mid} to denote the *un-updated set of users'* latest HashChain digest in epoch i.

We use the helper function $\mathrm{sum} \leftarrow \mathbf{GetSum}(L_i)$ to calculate the sum based on the ledger L_i for epoch i, denoted as sum_i. Additionally, we require a prime order group for Pedersen commitments, which offers hiding, binding, and compatibility with CP-SNARKs to commit the sum of all liabilities. As our focus lies on the hidden order group, we do not explicitly describe operations within the prime order group. Instead, we consider the prime order group parameters are included in the *crs* of the SNARKs, thus reducing ambiguity. We represent a Pedersen commitment of value s randomized by r (a uniform random number within the prime field) as $Com(s, r)$. The digest d_i for epoch i consists of two parts: d_i.RSA represents the digest of the RSA accumulator; d_i.Ped denotes the digest of the Pedersen commitment.

We construct the Notus system based on the ZK-MultiSwap with the following relation:

$$\mathcal{R}^{\mathbf{DPoL}}(d_{i-1}, d_i, C_{mid}, i, \boxed{c_{\vec{u}0}}, \boxed{c_{\vec{u}1}}; \tau_0, \tau_1, \bar{S}_i, S_i, \mathrm{sum}_{i-1}, \mathrm{sum}_i, r_{i-1}, r_i).$$

The relation $\mathcal{R}^{\mathbf{DPoL}}$ needs to check that *(i)* the number of deleted users is the same as the inserted users and all deleted users are re-inserted in the HashChain; *(ii)* the update only appends transactions with a timestamp of the current epoch number i; *(iii)* the updated users' liabilities are positive; *(iv)* the sum of liabilities is correctly updated; *(v)* the Pedersen commitments commit the sums. Figure 6 includes the details of relation $\mathcal{R}^{\mathbf{DPoL}}$. Our construction for Notus based on the above components is shown in Figure 7. We refer to the formal theorem statement and proof of its security in the extended version of our paper [65] and details for membership proof precomputation in Appendix E.

5 Performance Optimizations

In this section, we present optimizations aimed at improving the performance of both the RSA accumulator and Notus. During every epoch, the server must perform membership proof precomputation, which has been identified as a bottleneck in this system and other similar systems based on RSA groups [59, 61]. Our goal is to enhance its efficiency.

5.1 User Grouping Strategy

Dividing users into different groups is a common approach when dealing with a large number of users in PoL, e.g., OKX [52]. In our system, we partition users into smaller groups, or subsystems, and provide a single proof of total liabilities as in the original system. Users only need to query their membership proof within their own subsystem, while the auditor is responsible for verifying the correctness of all subsystems. This approach does not negatively impact users' performance, as each user only queries their membership within their respective subsystem (but the auditor must check all subsystems). For subsystems without updates, the server still randomizes each RSA accumulator and provides SNARK proofs. Thus, the server overhead is fully parallelizable; given m groups, it can potentially become m times faster.

Note that, information about users' subgroup participation is potentially revealed in this way. However, we considered this "benign" leakage since this partitioning may be based on a random allocation or even on public information (e.g., user names or user ids). Furthermore, each RSA accumulator remains zero-knowledge, preventing the parties from learning

Setup:(1^λ):

$(\text{RSA}.ek, \text{RSA}.vk) \leftarrow \textbf{RSA.KeyGen}(1^\lambda)$
$\text{crs} \leftarrow \Pi.\textbf{Setup}(1^\lambda, \mathcal{R}^{DPoL})$
return: $ek = (\text{RSA}.ek, \text{crs}), vk = (\text{RSA}.vk, \text{crs})$

Digest:(ek, L_i, aux_i):

$D_i \leftarrow \textbf{LedgerDigest}(L_i)$
$acc_i \leftarrow \textbf{RSA.Acc}(\text{RSA}.ek, D_i, t_i, t_{i+1})$
$\text{sum} \leftarrow \textbf{GetSum}(L_i)$
return: $(d_i.\text{RSA} = acc_i,\ d_i.\text{Ped} = Com(\text{sum}, r_i))$

LookupProof:(ek, L_i, u, aux_i):

$k_u^i = \text{HashChain}(H_u^{1,i})$
$(b, w_x) \leftarrow \textbf{RSA.Witness}(D_i, k_u^i, acc_i, \text{RSA}.ek)$
If $b = 0$, **return:** $\perp$
If $b = 1$, **return:** π_{mem}

ProveSum:(ek, L_i, aux_i):

$\text{sum} \leftarrow \textbf{GetSum}(L_i)$, $\pi_{\text{sum}} = r_i$
return: $(\text{sum}, \pi_{\text{sum}})$

ProveConsistency:(ek, L_i, aux_i):

$D_{mid} \leftarrow \textbf{LedgerDigest}(L_i \setminus S_i)$
$C_{mid} \leftarrow \textbf{RSA.Acc}(\text{RSA}.ek, D_{mid}, t_{i-1})$
$\pi_i \leftarrow \Pi.\textbf{Prove}(\text{crs}, d_{i-1}, d_i, C_{mid}, i, \boxed{c_{\vec{u}0}}, \boxed{c_{\vec{u}1}}; \tau_0, \tau_1, \bar{S}_i, S_i, \text{sum}_{i-1}, \text{sum}_i, r_{i-1}, r_i)$
return: π_i

VerConsistency:$(vk, i, d_{i-1}, d_i, \pi_i)$:

Check $\Pi.\textbf{Verify}(\text{crs}, d_{i-1}, d_i, C_{mid}, i, \boxed{c_{\vec{u}0}}, \boxed{c_{\vec{u}1}}, \pi_i)$
return: 0 if rejected and 1 otherwise

VerLookup:$(vk, d_i, H_u^{1,i}, \pi_{\text{mem}})$:

$k_u^i = \text{HashChain}(H_u^{1,i})$
Check $\pi_{\text{mem}}^{k_u^i} \stackrel{?}{=} d_i.\text{RSA}$
return: 0 if rejected and 1 otherwise

VerSum:$(vk, d_i, \text{sum}, \pi_{\text{sum}})$:

Check $d_i.\text{Ped} \stackrel{?}{=} Com(\text{sum}, \pi_{\text{sum}})$
return: 0 if rejected and 1 otherwise

Figure 7: Description of Notus. For each epoch, the server samples two random values, (τ_i, r_i), where $t_i = H_K(\tau_i)$, t_i is used for zero-knowledge RSA accumulators and r_i is used for Pedersen commitment. aux_i is the collection of all random numbers used till epoch i.

any useful information within a given group. In practice, we utilize Groth16 [38] with a public upper bound on the number of updates that can be checked. To accommodate groups with more active users or a surge of transactions, we raise the upper bound as a safeguard, effectively concealing the actual quantity of updates. It should be noted that raising the upper bound could lead to increased overhead for the prover. Nevertheless, the SNARK proving process accounts for only a small fraction of the total runtime and does not introduce a bottleneck in the overall system performance.

	# of bits=200,000		# of bits=2,000,000	
	Time(ms)	Rate	Time(ms)	Rate
2×NaiveExponent	911.9	1	9443.7	1
DoubleExponent	646.4	70.9%	6640.3	70.3%
4×NaiveExponent	1827.6	1	18830.5	1
FourfoldExponent	733.9	40.2%	7482.0	39.7%
precomFourfoldExponent	359.7	19.7%	3596.5	19.1%

Table 2: Multi-output exponentiation function completion time and performance gain rate (rate= $\frac{\text{optimized algorithm time}}{\text{naive algorithm time}}$).

5.2 Multi-output Exponentiations

When precomputing membership proofs using the divide-and-conquer trick, at every layer, we calculate the following: $g^x \mod N$ and $g^y \mod N$ where g is a new group element in every layer except for the first layer, and x, y are two large integers with similar bitlengths. The most advanced method to calculate the modular exponent is Montgomery modular multiplication [49] with a binary expression of exponents. Informally, calculating $g^x \mod N$ using left-to-right binary exponentiation keeps squaring g, and based on the bit of x to decide if the squared values should be multiplied into the final result. We observe that to calculate $g^x \mod N$ and $g^y \mod N$, we can combine them using the same squaring of g. Furthermore, we can extract the common "1" bits of x and y as z and have $x' = x - z$, $y' = y - z$ to save common multiplications during the process.

Based on the same idea, we can further combine the process of $g^{x_1} \mod N$, $g^{x_2} \mod N$, $g^{x_3} \mod N$, $g^{x_4} \mod N$ by extracting all combinations of their common bits. We first extract common bits for $\{x_1, x_2, x_3, x_4\}$ and update the original values; then we extract common bits of $\{x_1, x_2, x_3\}$, $\{x_1, x_2, x_4\}$, $\{x_1, x_3, x_4\}$ and $\{x_2, x_3, x_4\}$ respectively and update the original values; finally, we extract $\binom{4}{2}$ combinations for each pair of values and update. In this way, we get four updated values $\{x_1', x_2', x_3', x_4'\}$ that are sparse in terms of "1" bit, and their original common "1" bits that can be calculated at once. This is optimal for calculating the four exponents via two layers of divide-and-conquer for membership precomputing. We show in our experiments that this significantly improves efficiency with 2 and 4 multi-output exponentiations. Besides, this works with precomputation tables for even better speed-up.

In our implementation, we optimize multi-output exponentiations till 4 numbers. For larger values, combining more exponents becomes harder. For example, for the common bits of k integers, the sum of all the combinations is $O(2^k)$, which makes implementation much more complicated.

6 Experimental Evaluation

In this section, we evaluate the performance of Notus. We implemented our scheme and optimizations discussed in Section 5 and our code is available[1] online. Our implementation is in Golang and we used gnark [11] for Groth16-type SNARKs [38], optimized with DIZK [64]. The elliptic curve

[1] https://github.com/notus-project/rsa_accumulator

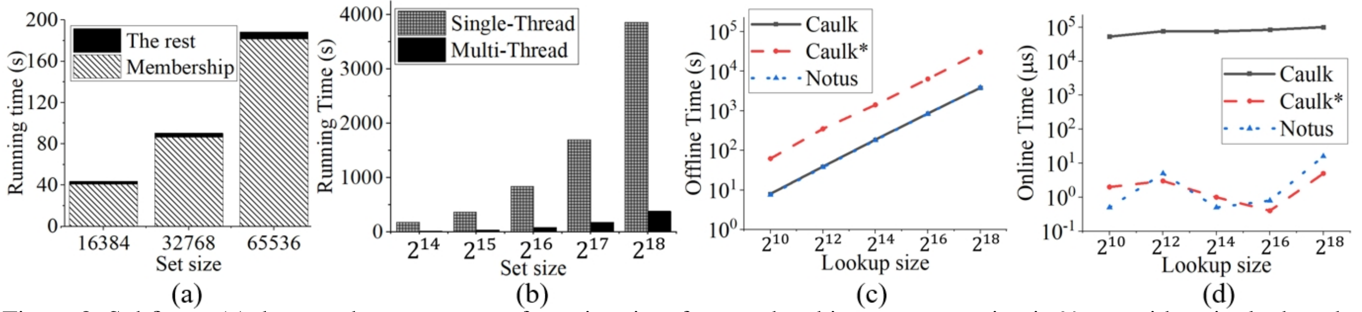

Figure 8: Subfigure (a) denotes the percentage of running time for membership precomputation in Notus with a single thread; Subfigure (b) indicates the membership proof precomputation time under different set sizes with both single-thread and multi-thread; Subfigure (c) and (d) compare our RSA accumulator with Caulk as a lookup argument for both the offline (precomputation) time and online time. Caulk* denotes generating all membership proofs in the offline phase.

we use is bn254. Our experiments were conducted on an Amazon EC2 M6a instance with 32 vCPUs and 128GB of memory. We use DI-Hash with 1024 bits (achieving 80 bits of security) based on Ozdemir et al. [53] as the representatives into RSA accumulators. We use Poseidon hash [37] to make our DI-Hash more SNARK-friendly. First, we evaluate our general optimization applied to multiple output exponentiations. Second, we evaluate our zero-knowledge RSA accumulator, especially the performance of precomputation of membership with different optimizations and parallelization. We also run Caulk as a comparison for efficient lookup proofs for single-element memberships. Finally, we evaluate Notus with different optimizations and compare it with TAP and Binance.

6.1 Effect of Multi-output Exponentiations

Based on Golang's official implementation for Montgomery modular multiplication, we implement the Multi-output exponentiation optimization. We generate a random base g and an odd modulus N with 2048 bits. For consistency, we use the same g and N in each run with randomly selected exponents ranging from 200K to 2M bits. We report the average results across 10 runs. Our findings are summarized in Table 2, which presents the function completion time and the percentage performance gain of our "optimized" execution over the "naive" one. We examine specific functions as follows: 1) 2×NaiveExponent vs. DoubleExponent: We calculate $g^{x_1} \bmod N$ and $g^{x_2} \bmod N$ "naively" by determining each exponent separately and refer to this as the 2×NaiveExponent function. We compare it against our optimized version, DoubleExponent. 2) 4×NaiveExponent vs. FourfoldExponent: For random values x_1, x_2, x_3, x_4, we "naively" calculate $g^{x_1} \bmod N$, $g^{x_2} \bmod N$, $g^{x_3} \bmod N$, and $g^{x_4} \bmod N$. We refer to this function as 4×NaiveExponent and compare it against our optimized FourfoldExponent function. Our optimizations result in DoubleExponent being approximately 30% faster and FourfoldExponent being approximately 60% faster than their original counterparts. Furthermore, we evaluate the performance of FourfoldExponent when combined with a precomputation table containing every single bit precomputed, referred to as the precomFourfoldExponent function. More elaborate precomputation table settings (e.g., including four combinations of every two precomputed bits) lead to faster calculations and larger table sizes. It is important to note that the precomFourfoldExponent optimization is only effective for fixed generators, allowing the exponents to be precomputed. When applying precomFourfoldExponent to membership precomputations, we use it only for the first layer of the divide-and-conquer algorithm, which is the most time-consuming layer. For the remaining layers, we employ FourfoldExponent.

6.2 RSA Accumulator Performance

Next, we evaluate the performance of our zero-knowledge RSA accumulator. A membership proof consists of one RSA group element, which is 256 bytes in size, and verifying a membership proof takes approximately 1 ms. A non-membership proof contains 10 RSA group elements and 18 integers of varying lengths, resulting in a total length of 8960 bytes. The proof generation time depends on the set size. Subsequently, we executed the Notus system with RSA precomputation for lookup proofs implemented using Fourfold Exponent optimizations. We recorded the time taken for each part of the system and the results are shown in Figure 8 (a). The precomputation of membership proofs accounts for the majority of the running time, while the remaining tasks (generating accumulators, ZKPoKE proofs, and SNARK proofs) contribute to no more than 5% of the total running time. Next, we focus more on membership precomputation.

Membership precomputations. In Figure 8 (b), we illustrate the membership precomputations under different set sizes with a single thread and with 32 threads. Note that, due to the divide-and-conquer nature of our membership precomputation, it is not perfectly parallelizable; however, it is still around 9-10× faster than the single-thread version. E.g., for set size 2^{18}, the single-thread precomputation takes 3857 seconds, whereas a multi-thread takes only 384 seconds.

Lookup arguments. We also assess the performance of Notus when used as a lookup argument [67]. We use the single-threaded version of our implementation to directly compare

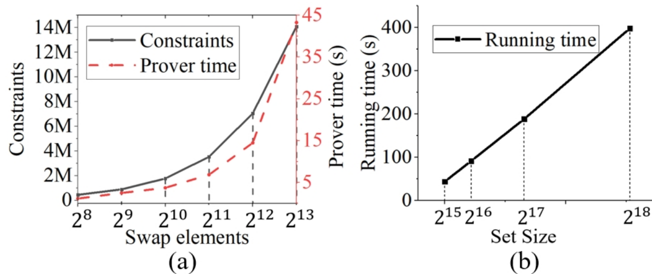

Figure 9: Subfigure (a) illustrates the number of constraints and the prover's time for the Notus SNARK circuit; Subfigure (b) illustrates the running time required to execute Notus under different set sizes with 32 cores and 128GB memory.

with the prior state-of-the-art Caulk [67]. We observe that the offline time for Notus is similar to Caulk. However, unlike Notus, Caulk requires an online phase to "complete" the lookup proof even after precomputation has taken place. In contrast, with Notus, after precomputation, one simply needs to retrieve proofs, which is almost instantaneous (e.g., $< 20\mu sec$). This results in much faster performance, as shown in Figure 8 (c) and (d). We also explore an alternative version of Caulk which generates all membership proofs in an offline phase (similar to Notus) with no additional online overhead, to guarantee full fairness in our comparison; we call this version Caulk*. As shown, the offline time of Caulk* (calculated as offline time of Caulk + (Lookup size × online time)) is nearly 3 orders of magnitude slower than Caulk and Notus.

6.3 Notus Performance

Finally, we evaluate the performance of Notus and compare it with other PoL approaches. We begin with a comparison of the time it takes for the server to complete an update between epochs. Proving correct updates between consecutive epochs can be a bottleneck for PoL systems when updates are frequent and users are highly active. We compare the proving time, proof size, and verification of Notus with TAP and Binance, as shown in Table 3. Among prior research in PoL, we compare performance with TAP because it is the only solution designed and implemented for the dynamic setting. We also compare with the PoL deployed by Binance as it uses the same gnark libary [11] implementation for SNARK as Notus, facilitating direct comparison of the two system designs. Moreover, as explained in our introduction, the approach of OKX produces a proof of size more than 1GB, making comparisons unrealistic. Recall that TAP is based on Merkle Sum trees, combined with Pedersen commitments and range proofs. For fairness, we run TAP ignoring its increasing proving overhead as epochs progress, and although its code does not support parallelism, we assume it achieves "ideal" perfect parallelization for comparison purposes. As shown, TAP has a fast prover but its proof size grows linearly with the number of updates, resulting in very large proofs.

Binance combines Merkle trees with SNARKs, considering a SNARK-friendly hash [37]. Its code is developed for production and currently considers a PoL for 350 different cryptocurrencies, with all datasets stored in a Merkle tree of depth 28. As we do not have access to the sample data required to run Binance, we first report their own benchmark, also tested on a system with 32 cores and 128GB memory [7], and normalize their results for 2^{10} updates. To conduct a fair comparison, we also ran SNARKed MerkleTree (MT). This is a "simplified" version of the technique behind Binance where we eliminated business-specific details, and built a Merkle tree of depth 28 incrementally within the SNARK by verifying the Merkle path for each newly inserted leaf.

	Prover time (sec)	Proof size (Byte)	Verify time (sec)
TAP [56]	3.8	610K	2.0
Binance* [6]	124.4	256	0.001
SNARKed MT	15.0	256	0.001
Notus	3.7	256	0.001

Table 3: Comparison of proving correct update between TAP, Binance, Merkle Insert, and Notus for 2^{10} updates. Results of Binance* come from its own benchmark with similar settings.

Finally, our results show that Notus achieves the fastest prover time, around 3× faster than SNARKed MT, and even marginally outperforms the range-proof-based system while maintaining succinct proof sizes and verification times.

Next, we turn our attention to the performance of Notus. Having already discussed the zero-knowledge accumulator part in section 6.2, we provide an evaluation for the SNARK part of Notus in terms of its circuit size (number of constraints) and prover time, for a variable number of updated transactions S_i (swap elements) in Figure 9 (a). Then, Figure 9 (b) provides the total prover overhead for Notus run on an AWS server with 32 cores and 128GB memory under different user sizes. Clearly, the running time increases almost linearly with the set size. On the verifier side, the auditor needs to spend 1ms and the proof size is 256 bytes for SNARK and 1024 bytes for MultiSwap, regardless of the set size. For users, membership proofs are one RSA element (256 bytes). Verifying the transaction history from the last checkpoint requires generating a hash chain and one membership proof in less than 2ms.

Super-efficient Auditing. Our auditor overhead per epoch is $O(1)$ and concretely so low that we can actually run it on a smart contract, which no prior PoL system could achieve. This is important as it makes auditing transparent and *eliminates the need to trust the auditor* blindly [15, 55, 57]. We wrote a Solidity smart contract for the auditor check, and the Gas cost per epoch is only 750K (270K for verifying a Groth16 proof and 480K for verifying two PoKEs).

Scaling to Larger User Sets. Increasing the user set size becomes challenging according to Figure 8 (d) as the prover time grows. Moreover, the precomputation table we maintain for exponentiations will quickly cause memory issues as the size increases. However, based on our user-grouping strategy from Section 5, we can naturally scale the Notus prover assuming a setting with multiple machines handling separate

subgroups. E.g., let's aim for 2^{20} users (assuming probability $1/2^6$ for users to update in an epoch), splitting them into just 8 groups, each one has 2^{17} users, $\sim 2^{11}$ of them update within the epoch. (For comparison, OKX currently splits users into $\sim 2K$ groups [52]). We can use eight cloud servers, each with 32 cores, to run this in parallel in approximately *3 minutes* according to Figure 8 (d). The extremely favorable side-effect of our design of Notus is that this approach *only increases the auditor's overhead and not the users'* as each user only verifies with respect to her own subgroup. Hence, if we split the 2^{20} users into 32 groups of 2^{15} users, and we utilize 32 servers to run in parallel, the prover time would become merely $\sim$43 seconds, while the auditor's overhead would still be in the order of milliseconds and the proof a few hundred KB!

These results show the practicality of Notus and indicate its potential as a good candidate for real-world applications. Binance currently operates a static PoL system with 45 million users, proving its solvency every month. Assuming one server with 32 cores and 64G memory, for 45 million users, it takes 68 *days* for Binance to generate the proof. Even with 100 servers, it would still take 16 *hours* [7]. If we extrapolate our results, Notus could build a **DPoL** system for 45 million users for the same configuration, proving its solvency every $43 \times 45 = 1935$ seconds! This is an over-simplified estimation but it clearly shows the potential qualitative difference.

7 Related Work

As discussed in Section 1, prior works on PoL mostly focus on the static case and cannot be readily extended to updates. The authors of [41] briefly mention the possibility of extending their construction to handle updates of liabilities. Identifying the privacy issues that arise from proofs across different epochs, they theoretically discuss relying on oblivious RAM [35] to hide update patterns. However, this introduces the problem of proving statements involving this ORAM [21], either via expensive zero-knowledge proofs, or via relying on trusted hardware, such as Intel SGX, which we consider too strong a security assumption for PoL applications (e.g., due to known attacks such as [43,47]). One could consider extending TAP [56] to handle general liability modifications, e.g., by maintaining one data structure for increasing liabilities and a "dual" one solely for decreasing modifications. Queries would then involve requesting proofs from both instances and computing the joint result. However, this hypothetical approach would require some sort of consistency checking between the two instances during updates, most likely via a SNARK, which would significantly blow up the overhead. Considering DAPOL+, Buterin suggested in a blogpost [15] replacing the Merkle tree with KZG polynomial commitments [42] combined with a "large" SNARK to prove all values are positive. Again, moving to the dynamic case, it is not clear how much this would blow up the overhead. In very recent independent work, Falzon et al. [32] designed a *static* PoL system that improved DAPOL+ by offering a shorter membership proof size at the cost of increased computation overhead for the prover. However, it still operates in *static* setting and compared to Notus, its proof size grows logarithmically with the number of users (concretely, it appears to be larger than ours, which is constant 256 bytes).

Finally, besides proving financial solvency, PoL has also been considered for other applications, e.g., disapproval voting, decentralized lending, taxing, reports of public data, and dynamic pricing based on public data [22,41,56].

Relation to Append-only Authenticated Dictionaries (AADs). A relevant research area to PoL is that of AADs [25, 40,51,56,59,61] that have found use cases in Certificate and Key Transparency [44]. Like PoLs, these schemes consider an untrusted server maintaining a dataset, however, in that setting, the server only resolves key-lookup queries (e.g., a user's public certificate). Due to this, prior AADs cannot be directly converted to a PoL. TAP [56] extends this line of work and it essentially provides an AAD with added functionalities beyond lookups, which it then uses to build a PoL. Another line of work uses this approach to build append-only relational databases [66]. While supporting more general queries, existing works for verifiable database queries either do not consider privacy [68] (which is crucial for PoLs) or require the auditor to essentially replicate the server's workload [66].

Zero-knowledge Accumulators and Vector Commitments. Zero-knowledge accumulators allow a prover to succinctly commit to a set of values with hiding and binding properties. Zero-knowledge (commonly referred to as *hiding*) vector commitments (VCs) further require commitments to be position-binding. There are limited zero-knowledge accumulators in the literature [18,34,69]. Ours is the first such scheme that is SNARK-friendly, supports efficient membership precomputations, and has constant-sized membership proofs. On the other hand, there are a number of choices for several zero-knowledge VCs [17, 19, 20, 42] and zero-knowledge sets [26, 69]. When designing Notus, we also considered a VC approach. However, there are limited candidates that are simultaneously zero-knowledge, SNARK-friendly, and support efficient lookups. The state-of-the-art candidate for such a VC approach is Caulk [67]. In our experimental evaluation for lookups, we show that our lookup can significantly outperform Caulk by more than 1000 times depending on the set size. To make Caulk support instant online responses like ours, we mimic Caulk* by precomputing all the membership proof offline. Caulk*'s offline line time becomes extremely heavy, almost 3 orders of magnitude slower than Caulk and our zero-knowledge RSA accumulator.

Zero-knowledge Membership Proofs. Zero-knowledge membership proofs allow a party to convince a verifier that it "knows" (e.g., as a commitment pre-image) an element or set of elements belonging to a committed set, without necessarily revealing the element(s). This primitive has been

built as an additional feature for RSA [9, 16] and bilinear accumulators [58], and Merkle trees combined with verifiable random functions [25]. This is different than the privacy property of our zero-knowledge accumulator whose goal is to also hide the committed set. For instance, zero-knowledge membership proofs are also meaningful when the set itself is public, e.g., when designing anonymous credentials and decentralized identities, and they can usually be adapted to hide the element itself (e.g., as the pre-image of a separate commitment). Boneh et al. [9] gave the state-of-the-art RSA accumulator with zero-knowledge membership proofs for a public set. Campanelli et al. [16] extends [9] by providing zero-knowledge subset proofs also for a publicly committed set. Our zero-knowledge RSA accumulator shares a similar basis with [16] but has orthogonal aims. We focus on protecting the privacy of the whole set while providing zero-knowledge membership/subset proofs. Besides, our subset proof is statistically zero-knowledge while subset proof in [16] is computationally zero-knowledge.

MultiSwap. First proposed in [53] for efficient batch "off-chain" verification of blockchain transaction, MultiSwap allows proving the correct updated status of a set after removals and insertions, usually by relying on a SNARK to check the validity of the removed and added subsets. More recently, Harisa [16] improved [53] in efficiency. We improve MultiSwap by making it zero-knowledge with minimal overhead.

8 Conclusion

We presented Notus, the first PoL system that supports general liability updates without requiring users to check their transactions at every epoch. Our system achieves $O(1)$ proof size, and verification and auditing overhead. Contrary to prior attempts, users in Notus can verify their status by checking just one proof, despite how many epochs they were absent. Notus utilizes as building blocks our novel zero-knowledge RSA accumulator and the first zero-knowledge MultiSwap protocol that may find other applications. Our experimental evaluation shows that Notus can scale to large numbers of users while maintaining good overall performance. One future direction in this line of work is to potentially reduce the prover's overhead to linear in the number of transactions in an epoch (as in [56] but for arbitrary updates).

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd for their constructive feedback. This work was partially supported by Hong Kong RGC under grant 16200721.

References

[1] "Bankruptcy of ftx," https://en.wikipedia.org/wiki/Bankruptcy_of_FTX/, accessed: 2023-04-19.

[2] N. Barić and B. Pfitzmann, "Collision-free accumulators and fail-stop signature schemes without trees," in *International conference on the theory and applications of cryptographic techniques*, 1997.

[3] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," *Cryptology ePrint Archive*, 2018.

[4] J. Benaloh and M. De Mare, "One-way accumulators: A decentralized alternative to digital signatures," in *EUROCRYPT*, 1993.

[5] "Binance exchange," https://www.binance.com/.

[6] "Binance proof-of-reserves," https://www.binance.com/en/proof-of-reserves/.

[7] "Binance proof-of-reserves code," https://github.com/binance/zkmerkle-proof-of-solvency/.

[8] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again," in *Innovations in Theoretical Computer Science*, 2012.

[9] D. Boneh, B. Bünz, and B. Fisch, "Batching techniques for accumulators with applications to iops and stateless blockchains," in *CRYPTO*, 2019.

[10] D. Boneh and M. Franklin, "Efficient generation of shared rsa keys," in *CRYPTO*, 1997.

[11] G. Botrel, T. Piellard, Y. E. Housni, I. Kubjas, and A. Tabaie, "Consensys/gnark: v0.8.0," Feb. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.5819104

[12] J. Buchmann and H. C. Williams, "A key-exchange system based on imaginary quadratic fields," *Journal of Cryptology*, 1988.

[13] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *IEEE symposium on security and privacy (SP)*, 2018.

[14] J. Burkhardt, I. Damgård, T. Frederiksen, S. Ghosh, and C. Orlandi, "Improved distributed rsa key generation using the miller-rabin test," in *Proceedings of the ACM CCS*, 2023.

[15] V. Buterin, "Having a safe cex: proof of solvency and beyond," https://vitalik.ca/general/2022/11/19/proof_of_solvency.html, accessed: 2023-05-10.

[16] M. Campanelli, D. Fiore, S. Han, J. Kim, D. Kolonelos, and H. Oh, "Succinct zero-knowledge batch proofs for set accumulators," in *Proceedings of the ACM CCS*, 2022.

[17] M. Campanelli, D. Fiore, and A. Querol, "Legosnark: Modular design and composition of succinct zero-knowledge proofs," in *Proceedings of the ACM CCS*, 2019.

[18] M. Campanelli, M. Hall-Andersen, and S. H. Kamp, "Curve trees: Practical and transparent {Zero-Knowledge} accumulators," in *USENIX Security Symposium*, 2023.

[19] D. Catalano and D. Fiore, "Vector commitments and their applications," in *PKC*, 2013.

[20] D. Catalano, D. Fiore, and M. Messina, "Zero-knowledge sets with short proofs," in *EUROCRYPT*, 2008.

[21] E. Cecchetti, F. Zhang, Y. Ji, A. E. Kosba, A. Juels, and E. Shi, "Solidus: Confidential distributed ledger transactions via PVORM," in *Proceedings of the ACM CCS*, 2017.

[22] K. Chalkias, P. Chatzigiannis, and Y. Ji, "Broken proofs of solvency in blockchain custodial wallets and exchanges," *IACR Cryptol. ePrint Arch.*, p. 43, 2022.

[23] K. Chalkias, K. Lewi, P. Mohassel, and V. Nikolaenko, "Practical privacy preserving proofs of solvency," *Amsterdam ZKProof Community Event*, 2019.

[24] ——, "Distributed auditing proofs of liabilities," *Cryptology ePrint Archive*, 2020.

[25] M. Chase, A. Deshpande, E. Ghosh, and H. Malvai, "Seemless: Secure end-to-end encrypted messaging with less trust," in *Proceedings of the ACM CCS*, 2019.

[26] M. Chase, A. Healy, A. Lysyanskaya, T. Malkin, and L. Reyzin, "Mercurial commitments with applications to zero-knowledge sets." in *EUROCRYPT*, 2005.

[27] P. Chatzigiannis, F. Baldimtsi, and K. Chalkias, "Sok: auditability and accountability in distributed payment systems," in *ACNS*, 2021.

[28] "Coinbase exchange," https://www.coinbase.com/.

[29] G. Couteau, T. Peters, and D. Pointcheval, "Removing the strong RSA assumption from arguments over the integers," in *EUROCRYPT*, 2017.

[30] G. G. Dagher, B. Bünz, J. Bonneau, J. Clark, and D. Boneh, "Provisions: Privacy-preserving proofs of solvency for bitcoin exchanges," in *Proceedings of the ACM CCS*, 2015.

[31] I. Damgård and E. Fujisaki, "A statistically-hiding integer commitment scheme based on groups with hidden order," in *ASIACRYPT*, 2002.

[32] F. Falzon, K. Elkhiyaoui, Y. Manevich, and A. D. Caro, "Short privacy-preserving proofs of liabilities," in *Proceedings of the ACM CCS*, 2023.

[33] R. Gennaro, S. Halevi, and T. Rabin, "Secure hash-and-sign signatures without the random oracle," in *EUROCRYPT*, 1999.

[34] E. Ghosh, O. Ohrimenko, D. Papadopoulos, R. Tamassia, and N. Triandopoulos, "Zero-knowledge accumulators and set algebra," in *ASIACRYPT 2016*, 2016.

[35] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996.

[36] S. Gorbunov, L. Reyzin, H. Wee, and Z. Zhang, "Pointproofs: Aggregating proofs for multiple vector commitments," in *Proceedings of the ACM CCS*, 2020.

[37] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A new hash function for zero-knowledge proof systems," in *USENIX Security Symposium*, 2021.

[38] J. Groth, "On the size of pairing-based non-interactive arguments," in *EUROCRYPT*, 2016.

[39] K. Hu, Z. Zhang, and K. Guo, "Breaking the binding: Attacks on the merkle approach to prove liabilities and its applications," *Computers & Security*, 2019.

[40] Y. Hu, K. Hooshmand, H. Kalidhindi, S. J. Yang, and R. A. Popa, "Merkle^2: A low-latency transparency log system," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

[41] Y. Ji and K. Chalkias, "Generalized proof of liabilities," in *Proceedings of the ACM CCS*, 2021.

[42] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *ASIACRYPT*, 2010.

[43] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, 2020.

[44] B. Laurie, A. Langley, and E. Kasper, "Rfc 6962: Certificate transparency," 2013.

[45] J. Li, N. Li, and R. Xue, "Universal accumulators with efficient nonmembership proofs," in *ACNS*, 2007.

[46] H. Lipmaa, "On diophantine complexity and statistical zero-knowledge arguments," in *ASIACRYPT*, 2003.

[47] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[48] A. Luthra, J. Cavanaugh, H. R. Olcese, R. M. Hirsch, and X. Fu, "Zeroaudit," in *Annual Computer Security Applications Conference*, 2020, pp. 798–812.

[49] C. McIvor, M. McLoone, and J. V. McCanny, "Modified montgomery modular multiplication and rsa exponentiation techniques," *IEE Proceedings-Computers and Digital Techniques*, vol. 151, no. 6, pp. 402–408, 2004.

[50] R. McMillan, "The inside story of mt. gox, bitcoin's $460 million disaster," https://www.wired.com/2014/03/bitcoin-exchange/, accessed: 2023-01-02.

[51] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "Coniks: Bringing key transparency to end users," in *USENIX Security Symposium*, 2015.

[52] "OKX proof-of-reserves," www.okx.com/proof-of-reserves/.

[53] A. Ozdemir, R. Wahby, B. Whitehat, and D. Boneh, "Scaling verifiable computation using efficient set accumulators," in *USENIX Security Symposium*, 2020.

[54] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *CRYPTO*, 1992.

[55] Y. Peng, M. Du, F. Li, R. Cheng, and D. Song, "Falcondb: Blockchain-based collaborative database," in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, 2020.

[56] D. Reijsbergen, A. Maw, Z. Yang, T. T. A. Dinh, and J. Zhou, "TAP: transparent and privacy-preserving data services," in *USENIX Security Symposium*, 2023.

[57] A. M. Rozario and M. A. Vasarhelyi, "Auditing with smart contracts." *International Journal of Digital Accounting Research*, vol. 18, 2018.

[58] S. Srinivasan, I. Karantaidou, F. Baldimtsi, and C. Papamanthou, "Batching, aggregation, and zero-knowledge proofs in bilinear accumulators," in *Proceedings of the 2022 ACM CCS*, 2022.

[59] A. Tomescu, V. Bhupatiraju, D. Papadopoulos, C. Papamanthou, N. Triandopoulos, and S. Devadas, "Transparency logs via append-only authenticated dictionaries," in *Proceedings of the ACM CCS*, 2019.

[60] I. A. Tomescu Nicolescu, "How to keep a secret and share a public key (using polynomial commitments)," Ph.D. dissertation, Massachusetts Institute of Technology, 2020.

[61] N. Tyagi, B. Fisch, A. Zitek, J. Bonneau, and S. Tessaro, "VeRSA: Verifiable registries with efficient client audits from rsa authenticated dictionaries," in *Proceedings of the ACM CCS*, 2022.

[62] B. Wesolowski, "Efficient verifiable delay functions," in *EUROCRYPT*, 2019.

[63] E. G. Weyl, P. Ohlhaver, and V. Buterin, "Decentralized society: Finding web3's soul," *Available at SSRN 4105763*, 2022.

[64] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica, "DIZK: A distributed zero knowledge proof system," in *USENIX Security Symposium*, 2018.

[65] J. Xin, A. Haghighi, X. Tian, and D. Papadopoulos, "Notus: Dynamic proofs of liabilities from zero-knowledge RSA accumulators," Cryptology ePrint Archive, Paper 2024/395, 2024. [Online]. Available: https://eprint.iacr.org/2024/395

[66] C. Yue, T. T. A. Dinh, Z. Xie, M. Zhang, G. Chen, B. C. Ooi, and X. Xiao, "Glassdb: An efficient verifiable ledger database system through transparency," *Proceedings of the VLDB Endowment*, 2023.

[67] A. Zapico, V. Buterin, D. Khovratovich, M. Maller, A. Nitulescu, and M. Simkin, "Caulk: Lookup arguments in sublinear time," in *Proceedings of the ACM CCS*, 2022.

[68] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases," in *IEEE Symposium on Security and Privacy, SP*, 2017.

[69] Y. Zhang, J. Katz, and C. Papamanthou, "An expressive (zero-knowledge) set accumulator," in *2017 IEEE European Symposium on Security and Privacy, EuroS&P*, 2017.

A Extended Preliminaries

A.1 Range proof

Range proofs are zero-knowledge proofs where the prover convinces the verifier that the committed values are in specific ranges. For hidden order group-based commitments, range proofs are usually [29, 46] composed by zero-knowledge argument of positivity (ZKAoP): in order to prove $x \in [a,b]$, the prover proves $x-a>0$ and $b-x>0$. In this work, we use the ZKAoP provided in [29], based on the RSA assumption. In their protocol, the statement is the integer commitment [31] in the format $g^x h^r$ where x is the committed value, and r is a randomness. We modify the statement to the format g^x. We can consider the statement as a special case where $r=0$. We denote the non-interactive version as NI-ZKAoP. We denote the protocol as (1) $\pi \leftarrow$ **NI-ZKAoP.Prove**$(G,g;x)$ where π is the proof, x is the witness such that $x>0$ and $G=g^x$; (2) $1/0 \leftarrow$ **NI-ZKAoP.Verify**(G,g,π). Besides, the authors in [29] also proved its knowledge extractability.

A.2 Universal Accumulators

We follow the universal accumulator definitions provided by [34].

Definition 1. *Given security parameter λ, a universal accumulator is a tuple of four PPT algorithms* **(KeyGen, Acc, Witness, Verify)***:*

- $(ek,vk) \leftarrow$ **KeyGen**(1^λ). *This algorithm takes as input the security parameter and outputs a public evaluation key ek that will be used to generate witnesses and a public verification key vk that will be used to verify witness.*
- $C \leftarrow$ **Acc**(ek,X). *This algorithm takes as input a set $X \subset \mathcal{X}_\lambda$ where $\mathcal{X}_\lambda$ is the input domain for the elements to be accumulated and the evaluation key ek and outputs one accumulation of the set $C \in \mathcal{G}_\lambda$ where $\mathcal{G}_\lambda$ is the output domain for* **Acc***.*
- $(b,w_x) \leftarrow$ **Witness**(X,x,C,ek). *This algorithm takes as input a set $X \subset \mathcal{X}_\lambda$, one element $x \in \mathcal{X}_\lambda$, the evaluation key ek and the accumulation of the set C. It outputs a boolean value b indicating whether the element is in the set and a witness w_x for the answer. $b=1$ indicates the element is in the set and this witness is a membership witness. Otherwise, $b=0$ and this witness is a non-membership witness.*
- $0/1 \leftarrow$ **Verify**(C,x,w_x,b,vk). *This algorithm takes as input the accumulation value $C \in \mathcal{G}_\lambda$, one element $x \in \mathcal{X}_\lambda$, the witness w_x, a bit b and the public verification key vk. It outputs 1 if it accepts the proof and 0 otherwise.*

A.3 Division Intractable Hash

Division Intractable (DI) hash is a special kind of hash function where it is hard to find any set of distinct preimages such that one hash result divides the product of the remaining hash results. It can be formally defined as follows.

Definition 2. *Division intractable. A function family $\mathcal{H}$ is called division intractable (DI) if given security parameter λ, finding $H \in \mathcal{H}$ and distinct inputs $X_1,\ldots,X_m,Y$, $m<poly(\lambda)$ such that*

$$\Pr[H(Y)|H(X_1)\ldots H(X_m)] < \mathrm{negl}(\lambda).$$

Clearly, all Hash-to-prime hash functions are DI. However, most Hash-to-prime functions are time-consuming. According to Lemma 6 in [33], a random oracle with λ^2-bit output is also DI. Informally, random numbers with large bit-length tend to have large prime factors. A large prime factor makes the random number hard to divide from other random numbers with different large prime factors.

In particular, Ozdemir *et al.* [53] conjectured that the density of integers in $[1,\alpha]$ with small factors also holds for a large interval around α, more specifically, $[\alpha,\alpha+\alpha^{1/8}]$. Based on this conjecture, we can construct a DI-hash more efficiently in computation. Let Δ be a large integer with λ^2-bit, and $H(x)$ be a collision resistant and preimage-hard hash function with $\alpha^{1/8}$ bits output. $H_\Delta(x)=\Delta+H(x)$, $H_\Delta(x)$ is also DI.

B Correctness and Security Definitions of DPoL

Definition 3 (**DPoL** Completeness)**.** *$\forall$ ordered sequences of appends S_i, for all users $u \in \mathcal{U}$ and $1 \le i \le m$ epochs,*

$$\Pr\left[\begin{array}{c}(ek,vk) \leftarrow \mathbf{Setup}(1^\lambda),\\ d_i \leftarrow \mathbf{Digest}(ek,L_i,aux_i)^+,\\ \forall u \in \mathcal{U}, \pi_{\text{mem}} \leftarrow \mathbf{LookupProof}(ek,L_i,u,aux_i),\\ \pi_{\text{sum}} \leftarrow \mathbf{ProveSum}(ek,L_i,aux_i),\\ \forall i \in [1,m], \pi_i \leftarrow \mathbf{ProveConsistent}(ek,L_i,aux_i):\\ \forall u \in \mathcal{U}, \mathbf{VerLookup}(vk,d_i,H_u^{1,i},\pi_{\text{mem}})=1,\\ \mathbf{VerSum}(vk,d_i,\text{sum},\pi_{\text{sum}})=1,\\ \forall i \in [1,m], \mathbf{VerConsistent}(vk,i,d_{i-1},d_i,\pi_i)=1\end{array}\right] \ge 1-\mathrm{negl}(\lambda)$$

Definition 4 (**DPoL** Undeniability)**.** *$\forall$ adversaries $\mathcal{A}$ running in time poly(λ), for any user $u \in \mathcal{U}$ and any epoch $i \in [1,m]$,*

$$\Pr\left[\begin{array}{c}(ek,vk) \leftarrow \mathbf{Setup}(1^\lambda),\\ (d_i,\text{sum},\text{sum}',\pi_{\text{sum}},\pi'_{\text{sum}},L_i,L'_i,\\ aux_i,aux'_i) \leftarrow \mathcal{A}(1^\lambda,ek,vk):(\text{sum} \neq \text{sum}'\\ \wedge \mathbf{VerSum}(vk,d_i,\text{sum},\pi_{\text{sum}})=1\\ \wedge \mathbf{VerSum}(vk,d_i,\text{sum}',\pi'_{\text{sum}})=1) \vee\\ (L_i \neq L'_i \wedge d_i \leftarrow \mathbf{Digest}(ek,L_i,aux_i) \wedge\\ d_i \leftarrow \mathbf{Digest}(ek,L'_i,aux'_i))\end{array}\right] \le \mathrm{negl}(\lambda)$$

Definition 5 (**DPoL** Update Soundness). *$\forall$ adversaries $\mathcal{A}$ running in time poly(λ),*

$$\Pr\left[\begin{array}{c}(ek,vk) \leftarrow \textbf{Setup}(1^\lambda),\\ (L_{i-1},L_i,d_{i-1},d_i,\pi_i,aux_i,aux_{i-1},\text{sum}_{i-1},\\ \text{sum}_i,\pi^{i-1}_{\text{sum}},\pi^i_{\text{sum}}) \leftarrow \mathcal{A}(1^\lambda,ek,vk):\\ d_i \leftarrow \textbf{Digest}(ek,L_i,aux_i),\\ d_{i-1} \leftarrow \textbf{Digest}(ek,L_{i-1},aux_{i-1}),\\ L_{i-1} \subset L_i,\ S_i = L_i \setminus L_{i-1},\\ \textbf{VerConsistency}(vk,i,d_{i-1},d_i,\pi_i) = 1,\\ \textbf{VerSum}(vk,d_{i-1},\text{sum}_{i-1},\pi^{i-1}_{\text{sum}}) = 1,\\ \textbf{VerSum}(vk,d_i,\text{sum}_i,\pi^i_{\text{sum}}) = 1,\\ (\exists tx \in S_i : tx.upd \neq i \vee tx.lia < 0)\vee\\ \text{sum}_i - \text{sum}_{i-1} \neq \sum_{u\in S_i}(H^i_u.lia - H^{i-1}_u.lia)\end{array}\right] \leq \text{negl}(\lambda)$$

Definition 6 (**DPoL** Sum Soundness). *$\forall$ adversaries $\mathcal{A}$ running in time poly(λ),*

$$\Pr\left[\begin{array}{c}(ek,vk) \leftarrow \textbf{Setup}(1^\lambda),\\ (d_i,\text{sum}_i,\pi^i_{\text{sum}},H^{1,i}_{u_0},\ldots,H^{1,i}_{u_j},\\ \pi_{\text{mem}_0},\ldots,\pi_{\text{mem}_j}) \leftarrow \mathcal{A}(1^\lambda,ek,vk):\\ \textbf{VerSum}(vk,d_i,\text{sum}_i,\pi^i_{\text{sum}}) = 1\wedge\\ \forall j, \textbf{VerLookup}(vk,d_i,H^{1,i}_{u_j},\pi_{\text{mem}_j}) = 1\wedge\\ \text{sum}_i < \sum_{\forall u} H^{1,i}_{u_j}.lia \wedge \forall H^{1,i}_{u_j}.lia > 0\end{array}\right] \leq \text{negl}(\lambda)$$

Additional discussions for these definitions can be found in the extended version of our paper [65].

Definition 7 (**DPoL** Privacy). *Let F_1 be a function checking user u's history for a given ledger L_i, $H^{1,i}_u \leftarrow F_1(L_i,u)$. Let F_2 be a function checking the sum of liabilities for a given ledger L_i, sum $\leftarrow F_2(L_i)$. Let $\textbf{Real}_{\textbf{Adv}}(1^\lambda)$, $\textbf{Ideal}_{\textbf{Adv,Sim}}(1^\lambda)$ be games between a challenger, an adversary $\mathcal{A}$ and a simulator $\textbf{Sim}$=$(\textbf{Sim}_1,\textbf{Sim}_2)$, defined as follows:*

$\textbf{Real}_{\textbf{Adv}}(1^\lambda)$:

- ***Setup.*** *The challenger runs $(ek,vk) \leftarrow \textbf{Setup}(1^\lambda)$ and forwards (ek,vk) to $\mathcal{A}$, $\mathcal{A}$ chooses S_1 with $|S_1| \leq poly(\lambda)$ and sends to the challenger. The challenger runs $d_1 \leftarrow \textbf{Digest}(ek,L_1 = S_1,aux_1)$ and sends d_1 to $\mathcal{A}$.*
- ***Query.*** *For $i = 1,\ldots,\zeta$ where $\zeta \leq poly(\lambda)$, the $\mathcal{A}$ outputs $op = (lookup,u)$, op=sum or $op = (update,S_{i+1})$.*

  – If $op = (lookup,u)$, the challenger runs $\pi_{\text{mem}} \leftarrow \textbf{LookupProof}(ek,L_i,u,aux_i)$ if user u has transaction history in the ledger in epoch i and returns π_{mem} to $\mathcal{A}$.

  – If op = sum, the challenger runs $\pi_{\text{sum}} \leftarrow \textbf{ProveSum}(ek,L_i,aux_i)$, returns $(\text{sum},\pi_{\text{sum}})$ to $\mathcal{A}$.

  – If $op = (update,S_{i+1})$, the challenger runs $L_{i+1} = L_i \bigcup S_{i+1}$, $d_{i+1} \leftarrow \textbf{Digest}(ek,L_{i+1},aux_{i+1})$, $\pi_{i+1} \leftarrow \textbf{ProveConsistency}(ek,L_{i+1},aux_{i+1})$ and returns (d_{i+1},π_{i+1}) to $\mathcal{A}$.
- ***Respond.*** *$\mathcal{A}$ outputs a bit $b = 0/1$.*

$\textbf{Ideal}_{\textbf{Adv,Sim}}(1^\lambda)$:

- ***Setup.*** *The simulator $\textbf{Sim}_1$ inputs (1^λ) and forwards vk to $\mathcal{A}$, $\mathcal{A}$ chooses S_1 with $|S_1| \leq poly(\lambda)$. The simulator (without seeing S_1) responds with d_1 to $\mathcal{A}$ and maintains state $\textbf{State}_S$.*
- ***Query.*** *For $i = 1,\ldots,\zeta$ where $\zeta \leq poly(\lambda)$, $\mathcal{A}$ outputs $op = (lookup,u)$, op=sum or $op = (update,S_{i+1})$.*

  – If $op = (lookup,u)$, the simulator runs $\pi_{\text{mem}} \leftarrow \textbf{Sim}_2(ek,\textbf{State}_S,F_1(L_i,u))$ if $F_1(L_i,u)$ outputs history $H^{1,i}_u$ and returns π_{mem} to $\mathcal{A}$.

  – If op = sum, the simulator runs $\pi_{\text{sum}} \leftarrow \textbf{Sim}_2(ek,\textbf{State}_S,F_2(L_i))$ and returns $(\text{sum},\pi_{\text{sum}})$ to $\mathcal{A}$.

  – If $op = (update,S_{i+1})$, the simulator runs $(d_{i+1},\pi_{i+1}) \leftarrow \textbf{Sim}_2(ek,\textbf{State}_S,d_i)$ and returns (d_{i+1},π_{i+1}) to $\mathcal{A}$.
- ***Respond.*** *$\mathcal{A}$ outputs a bit $b = 0/1$.*

*A **DPoL** scheme is zero-knowledge if there exists a probabilistic polynomial time (PPT) simulator $\textbf{Sim}$=$(\textbf{Sim}_1,\textbf{Sim}_2)$ such that for all **Adv**,*

$$|\Pr[\textbf{Real}_{\textbf{Adv}}(1^\lambda) = 1] - \Pr[\textbf{Ideal}_{\textbf{Adv,Sim}}(1^\lambda) = 1]| \leq \text{negl}(\lambda).$$

*If the above probabilities are equivalent, the **DPoL** scheme is perfect zero-knowledge. If the inequality only holds for PPT **Adv**, the **DPoL** scheme is computational zero-knowledge.*

C Zero-knowledge RSA accumulator

Let λ be the security parameter. We set $\mathcal{X}_\lambda = \mathbb{Z}_{2^{\lambda^2}}$. $\mathcal{X}_\lambda$ is the input domain for the elements to be accumulated. Denote by $X = \{x_1,\ldots,x_m\}$ the set of m elements to be accumulated. $\mathcal{H}_{DI}$ is a DI-hash family.

We use QR_N to denote the subgroup of $\mathbb{Z}^*_N$ of squares (quadratic residues modulo N). In order to let adaptive root assumption hold in $\mathbb{Z}^*_N$, we usually eliminate elements with trivial roots, $\{1,-1\}$. We use $\mathbb{G} = QR_N \setminus \{1,-1\}$ to denote excluding the use of $\{1,-1\}$ in challenges and proofs so that strong RSA assumption, adaptive root assumption hold in $\mathbb{G}$. Denote $K > \max\{\text{maxord}(\mathbb{G})2^{\lambda+2}, 2^{\lambda^2}\}$ where maxord$(\mathbb{G})$ is the upper bound for the group order and 2^{λ^2} is the lower bound to make random numbers DI as discussed in Appendix A.

We use $(a,b) \leftarrow \textbf{EEA}(x,y)$ to denote the extended Euclidean algorithm that calculates Bézout coefficients (a,b) s.t. $ax + by = \gcd(x,y) = z$. We present our construction in Figure 10. In the **KeyGen**, we generate two generators g,h. h is not explicitly used in the construction and is only used for the **NI-ZKAoP** for the non-membership proof. For the function **Acc**, it inputs at least one random number $t \xleftarrow{\$} [1,K]$. As shown in our security proofs, one random number uniform in $[1,K]$ is enough to achieve statistically zero knowledge, and random numbers in $[1,K]$ also satisfy the DI property. We generalize to multiple random numbers for dynamic cases, which are used in the dynamic MultiSwap.

KeyGen(1^λ):

$p, q \xleftarrow{\$} \text{Primes}(\lambda)$, $p' = 2p+1$, $q' = 2q+1$, s.t. p', q' are prime, $N = p'q'$, $\mathbb{G} = QR_N \setminus \{1, -1\}$

$g, h \xleftarrow{\$} \mathbb{G}$

return: $ek = vk = (N, g, h)$

Acc(ek, X, t):

$u = \prod_{i=1}^{m} H_{DI}(x_i)$, $C = g^{ut} \mod N$

return: C

Witness(X, x, C, ek):

-When $x \in X$, $\xi = \prod_{x_i \in X/x} H_{DI}(x_i)$, $w_x = g^{\xi t} \mod N$

return: $b = 1, w_x$

-When $x \notin X$, $\xi = H_{DI}(x)$, $a, b \leftarrow$ **EEA**(ut, ξ)

$\gamma \xleftarrow{\$} [1, K]$, $A = g^{(a+\gamma\xi)ut} \mod N$, $B = g^{b - \gamma ut} \mod N$

$D = (A \times B^{\xi})^{-1} \times g^{\xi} = g^{\xi - z} \mod N$

$\pi_1 \leftarrow$ **NI-ZKPoKE.Prove**$(A, C; a - \gamma\xi)$

$\pi_2 \leftarrow$ **NI-ZKAoP.Prove**$(D, g; \xi - z)$,

$\pi_3 \leftarrow$ **NI-ZKAoP.Prove**$(A \times B^{\xi}, g; z)$

return: $b = 0, w_x = (A, B, \pi_1, \pi_2, \pi_3)$

Verify(C, x, w_x, b, vk):

-When $b = 1$,

Reject if $(w_x)^{H_{DI}(x)} \neq C \mod N$

-When $b = 0$,

Parse w_x as $(A, B, \pi_1, \pi_2, \pi_3)$

$D = (A \times B^{\xi})^{-1} \times g^{\xi} \mod N$

Reject if $0 \leftarrow$ **NI-ZKPoKE.Verify**(A, C, π_1),

Reject if $0 \leftarrow$ **NI-ZKAoP.Verify**(D, g, π_2)

Reject if $0 \leftarrow$ **NI-ZKAoP.Verify**$(A \times B^{\xi}, g, \pi_3)$

return: 0 if rejected and 1 otherwise

Figure 10: Our zero-knowledge RSA accumulator. **Acc** inputs at least one random number $t \xleftarrow{\$} [1, K]$. If **Acc** inputs more than one random, then accumulate all of them. For example, if the inputs include t_1, t_2, then $C = g^{ut_1t_2} \mod N$.

Protocol zero-knowledge subset

Setup(1^λ): $N, \mathbb{G} = QR_N \setminus \{1, -1\}$, $g \xleftarrow{\$} \mathbb{G}$;
$\text{crs}_1 \leftarrow \Pi.$**Setup**$(1^\lambda, \mathcal{R}^{Hash})$;
$\text{crs}_2 \leftarrow \Pi.$**Setup**$(1^\lambda, \mathcal{R}^{mod})$.
Input: $C, C' \in \mathbb{G}$, $\boxed{c_{\vec{u}}}$.

Witness: (τ, S) s.t. $x = \prod_{s_i \in S} H_{DI}(s_i)$, $\tau \xleftarrow{\$} [K]$, $t = H_K(\tau)$, $C' = C^{xt}$.

Claim: Proof of knowledge of set S s.t. accumulating S into C and randomized with t gets C', and hash values of set S and τ are committed in $c_{\vec{u}}$.

1. Verifier sends $l \xleftarrow{\$} \text{Primes}(\lambda)$ to the prover.
2. Prover computes the quotient $q = \lfloor xt/l \rfloor \in \mathbb{Z}$ and residue $r = xt \mod l$ s.t. $xt = ql + r$. $Q = g^q \mod N$.
3. $\pi_1 \leftarrow \Pi.$**Prove**$(\text{crs}_1, \boxed{c_{\vec{u}}}; \tau, S)$.
4. $\pi_2 \leftarrow \Pi.$**Prove**$(\text{crs}_2, \boxed{c_{\vec{u}}}, l, r)$.
5. Prover sends $\pi = (\boxed{c_{\vec{u}}}, Q, r, \pi_1, \pi_2)$ to the verifier.
6. Verifier accepts if $r \in [l]$ and $Q^l C^r = C'$ holds in $\mathbb{G}$, $\Pi.$**Verify**$(\text{crs}_1, \boxed{c_{\vec{u}}}, \pi_1) = 1$ and $\Pi.$**Verify**$(\text{crs}_2, \boxed{c_{\vec{u}}}, l, r, \pi_2) = 1$.

Figure 11: Our protocol for zero-knowledge subset accumulator while the complementary set is committed.

D Zero-knowledge Subset

We construct our zero-knowledge subset protocol using two basic CP-SNARKs: $\mathcal{R}^{Hash}$ and $\mathcal{R}^{mod}$.

$\mathcal{R}^{Hash}(\boxed{c_{\vec{u}}}; \tau, S) = 1 \Leftrightarrow \forall s_i \in S, u_i = H_{DI}(s_i), u_{i+1} = H_K(\tau)$, which proves the committed values are perspective hash outputs of S and τ, i.e., H_{DI} for each s_i and H_K for τ.

$\mathcal{R}^{mod}(\boxed{c_{\vec{u}}}, l, r) = 1 \Leftrightarrow \prod_{\forall i} u_i = r \mod l$, which proves the product of committed values modulo l equals r.

We describe the construction of our zero-knowledge subset accumulator in Figure 11.

E Membership proof precomputation

Our zero-knowledge RSA accumulator, similar to general RSA accumulators, supports precomputing all membership proofs in $O(n \log n)$ time using the divide-and-conquer method [60] with minor modifications. Consider an accumulator $g^{x_1x_2x_3x_4t_1t_2} \mod N$ containing four elements (x_1, x_2, x_3, x_4) and two randomizers (t_1, t_2). The membership for x_1 is represented by the element $g^{x_2x_3x_4t_1t_2} \mod N$, which includes all elements and randomizers except x_1.

In the first round of the divide-and-conquer approach, our method diverges slightly from the general method by computing $g^{x_1x_2t_1t_2} \mod N$ and $g^{x_3x_4t_1t_2} \mod N$. For subsequent rounds, the membership precomputation remains the same, with each branch accumulating half of the remaining elements to generate membership proofs. For instance, in the second round, we compute $(g^{x_1x_2t_1t_2})^{x_3}, (g^{x_1x_2t_1t_2})^{x_4} \mod N$ to obtain the membership proofs for x_3 and x_4, respectively.