Generalized Proof of Liabilities

Yan Ji
Cornell Tech & IC3
yj348@cornell.edu

Konstantinos Chalkias
Novi / Facebook
kostascrypto@fb.com

ABSTRACT

Proof of liabilities (PoL) allows a prover to prove his/her liabilities to a group of verifiers. This is a cryptographic primitive once used only for proving financial solvency but is also applicable to domains outside finance, including transparent and private donations, new algorithms for disapproval voting and publicly verifiable official reports such as COVID-19 daily cases. These applications share a common nature in incentives: it's not in the prover's interest to increase his/her total liabilities. We generalize PoL for these applications by attempting for the first time to standardize the goals it should achieve from security, privacy and efficiency perspectives. We also propose DAPOL+, a concrete PoL scheme extending the state-of-the-art DAPOL protocol but providing provable security and privacy, with benchmark results demonstrating its practicality. In addition, we explore techniques to provide additional features that might be desired in different applications of PoL and measure the asymptotic probability of failure.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**.

KEYWORDS

distributed audit; blockchains; range proofs; sparse Merkle trees; solvency; tax reporting; disapproval voting; fundraising; COVID-19 reporting; credit score

ACM Reference Format:
Yan Ji and Konstantinos Chalkias. 2021. Generalized Proof of Liabilities. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3460120.3484802

1 INTRODUCTION

Companies that accept monetary deposits from consumers, such as banks and blockchain custodial wallets, are periodically being audited for accounting and oftentimes financial solvency purposes. A company being solvent means that it has enough assets to pay its customers. In other words, the amount of total assets owned by the company is no less than its total liabilities (customers' deposits).

Insolvency occurs not only when a company makes bad investments but also due to panic, i.e., lack of trust from its customers that it's running in a healthy and solvent state. Furthermore, the panic can spread to other companies to which the insolvent company owes money, and lead to a domino effect, i.e., an insolvency of one company can lead to a "cascade" of insolvencies [61].

Without proper regulation and protection, customers are not able to get deposits back in full when the bank becomes insolvent. Americans lost $140 billion due to bank failures in the Great Depression by the time the Federal Deposit Insurance Corporation (FDIC) [44] was created. Similar tragedies of bankruptcy and money loss occur in the cryptocurrency world as well, even with newly emerging technologies. A number of exchanges, a type of custodial cryptocurrency wallets taking users' deposits and trading on their behalf, have lost their deposits and declared bankruptcy. At the collapse of Mt. Gox, one of the oldest exchanges in Bitcoin's history, over $450M in customer assets were lost [52].

Traditionally, to secure customer's deposits, a third party auditor undertakes the role of verifying solvency by crosschecking transaction records in the company books. However, this is not sufficient to guarantee that the reported amount of total liabilities of a company is correct. First of all, the records can be manipulated and the auditor can hardly find out unless he confirms with the corresponding customer. Even if he does so, a misbehaving company can omit some accounts and report smaller liabilities. In addition, the auditor can learn sensitive information during the auditing, including the company's individual liabilities (each user's balance) and possibly transaction histories. More importantly, customers cannot tell whether an auditor is colluding with the company [42, 72]. Therefore, to preserve stability and public confidence in financial systems, we need a transparent and reliable audit of solvency.

Decentralized solutions [8, 13, 19, 20, 25, 73] that require customers to jointly participate on the auditing process have been recently proposed as an alternative or complementary method to conventional auditing. Decentralized auditing places less trust on auditors and is more promising because customers can make sure their own balances are not omitted, which cannot be achieved by centralized auditing solely. There is a rising demand in standardizing proof of solvency in the digital assets industry [20, 56].

Proof of liabilities (PoL) [73] is a cryptographic primitive to solve one half of solvency auditing, the other half being proof of reserves. The goal of PoL is to prove the size of funds a bank owes to its customers. Most of existing schemes follow the same principle: a prover aggregates all of the user balances using some accumulator and consumers can verify balance inclusion in the reported total amount. This process is probabilistic and the more the users that verify inclusions, the better the guarantee of a non-cheating prover. We provide a formal analysis of failure probability of PoLs.

While PoL seemed limited to custodial services, DAPOL [20] made the first attempt to apply it to a wide range of applications where an entity (prover) needs to transparently publish an absolute value that represents its total obligations against a set of users and

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484802

allow users to individually verify the inclusion of the prover's liabilities to them. For example, a charity can use PoL to prove the total amount of funds raised to donors for transparency. Non-financial use cases include anything related to obligations or negative reviews, including reporting COVID cases and complaining about harmful speech. We revise the applications mentioned in DAPOL and provide a complete list in appendix A. These applications, including PoL for solvency, share a common nature in participants: there is no incentive for the prover to exaggerate (over-report) the total liabilities; there is an incentive for individuals to make sure their values are included in the reported total liabilities. The concepts of "liability" and "incentives" are crucial in the formalization of PoL constructions because they relax the security requirements, i.e., there's no need to prevent the prover from arbitrarily adding dummy values for "fake" users without being detected.

Existing PoL works [13, 19, 20, 25, 73] target the same security goal, i.e., the prover cannot cheat without detection, but privacy against customers to different extents varies between the two extremes of leaking everything such as individual liabilities and number of customers and leaking no side information at all. DAPOL aims to provide the latter extreme for the solvency case but lacks formal PoL definitions and security proofs for the scheme. We formalize PoL and propose DAPOL+, a concrete design of PoL extending DAPOL. DAPOL+ provides provable security and privacy, and indicates that a PoL leaking no side information at all is possible.

Apart from the basic security and privacy, different applications might desire different additional features. E.g., in applications accepting updates in individual liabilities such as negative reviews and solvency, a prover may need to preserve privacy for two sequential PoLs before and after the updates. Additionally, the prover knowing which user verified inclusions for the previous PoLs may be able to predict which are less likely to perform verifications thus there is a lower risk to omit the liabilities to these users. Therefore, another desired property is to hide the identities of users that query for proofs. We study these additional features and explore solutions.

Our major contributions are as follows:

- We formalize PoL as a general cryptographic primitive for applications not limited to financial uses, and make the first attempt to standardize the goals for PoL from three aspects: security, privacy and efficiency;
- We propose DAPOL+, the first PoL scheme with provable security and strong privacy, and demonstrate its practicality by benchmarking a proof-of-concept (PoC) implementation;
- We categorized additional features desirable in different applications and propose solutions and accumulator variants;
- We analyze failure probability, i.e., the probability that a malicious prover evades detection when a subset of users verify proofs, aiming to provide insights on the effectiveness of distributed verification.

Paper Organization. We compare existing schemes from aspects of security, privacy and efficiency in section 2. We formalize security and privacy for PoL in section 3. In section 4, we introduce DAPOL+, discuss accumulator variants and explore additional features optional in various applications. In section 5, we analyze failure probability. We benchmark a PoC implementation of DAPOL+ to demonstrate practicality in section 6 and conclude in section 7.

2 RELATED WORK

There have been a few PoL schemes proposed in the literature [13, 19, 20, 25, 73]. They follow the same basic idea: the prover commits to the total liabilities and each individual user checks if the prover's liabilities to him/her is properly included. In this section, we review these protocols and compare the following aspects of them:

- *Security.* In a PoL protocol, the prover should not be able to cheat (claiming a smaller value than the fact) about the total liabilities without being detected by any verifier.
- *Privacy.* Different applications require different extents of privacy for PoL protocols. In the fundraising case, for example, donors might wish to keep the amounts of their donations between the charity organization and themselves, and conceal from a third party. We call this privacy of individual liabilities. In the proof of solvency case, in contrast, this might not be sufficient. The total liabilities of a bank/company and the number of its users could be sensitive information about its business, so a good PoL should preserve the privacy of them. Privacy guarantees that no adversary can learn any information they shouldn't throughout the execution of the protocol. We give a formal definition of privacy in section 3.5. For demonstration, we examine the privacy of total liabilities, the number of users and individual liabilities in this section.
- *Efficiency.* The efficiency of a PoL protocol include that of the proof generation time, verification time, individual proof size and the commitment size on the public bulletin board (PBB). The first three are straightforward as it is always desired to minimize the computation and bandwidth complexities for protocol participants. A PBB is a piece of universally accessible and append-only memory allowing everyone to have the same view of the contents [41]. A PBB is necessary in a PoL protocol to prevent the prover from committing different total liabilities to users and cheating without being detected. Although PBB has been a standard assumption in cryptography for tens of years, implementing a PBB is not cost free. Today, blockchains are widely considered to be a practical implementation of a PBB [33], but writing data on a blockchain could be expensive. E.g., on Ethereum, a blockchain with the second largest crypto market cap [24], writing 1KiB costed approximately $140 [26, 74] on May 5, 2021. This is impractical for an entity with 1M users to prove liabilities. The expensive cost stems from the strong guarantees we assume for a PBB, as implicated by the CAP theorem [36, 48] and the Blockchain Trilemma [67]. Therefore, a practical PoL scheme should minimize the data written on a PBB.

We summarize the comparisons in table 1. Note that ○ indicates the property is not guaranteed, ● indicates it is guaranteed, and ◑ indicates the design only provides a partial solution.

Maxwell-Todd. Maxwell and Todd [73] proposed a summation Merkle tree construction to prove total liabilities. Merkle trees are a data structure enabling a set owner to prove element membership in the set efficiently. In their design, each customer is mapped to a leaf node in the tree. Each node in the Merkle tree not only has a field of hash h, but also a value c. The c field in a leaf node indicates the prover's liabilities to the user, denoted by l, and that in an internal node is the sum of the values in its two child nodes lch

Table 1: *Comparison between existing schemes for PoLiab.*

	Security	Privacy			Efficiency			
Scheme	Known vulnerability	Total liabilities	Population	Individual liabilities	Proving time	Verification time	Proof size	Commitment size on PBB
Maxwell-Todd [73]	[43]	○	○	○	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Maxwell+ [43]	-	○	○	○	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Maxwell++ [19]	-	○	◑	◑	$O(n \cdot M)$	$O(\log(n \cdot M))$	$O(\log(n \cdot M))$	$O(1)$
Camacho [13]	[43]	●	○	●	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Provisions [25]	-	●	◑	●	$O(n+B)$	$O(n+B)$	$O(1)$	$O(n+B)$
DAPOL [20]	-	●	◑	●	$O(n \cdot H)$	$O(H)$	$O(H)$	$O(1)$
DAPOL+	-	●	●	●	$O(n \cdot H)$	$O(H)$	$O(H)$	$O(1)$

and rch, i.e., $c = c_{lch} + c_{rch}$. For the hash field, unlike hashing the concatenation of two child nodes in a standard Merkle tree, i.e., $h = \mathcal{H}(h_{lch}||h_{rch})$, Maxwell-Todd includes the value field at hashing, i.e., $h = \mathcal{H}(c||h_{lch}||h_{rch})$. For each leaf node mapped to a user, $h = \mathcal{H}(c||CI)$, where CI is the credential information of the user. We demonstrate in fig. 1 a summation Merkle tree for 4 users.

$\mathcal{P}$ generates the summation tree and commits the Merkle root (h_{root}, c_{root}) to a PBB. The value c_{root} is the amount of $\mathcal{P}$'s total liabilities. A user queries the prover for a Merkle proof to make sure the amount is included in the tree. For instance, in fig. 1, the prover sends the Merkle path $(h_2, c_2), (h_6, c_6)$ to User 1. Possessing (h_1, c_1), User 1 computes $c'_5 = c_1 + c_2$ and $c'_7 = c'_5 + c_6$, and checks if $c'_7 = c_{root}$. And similarly for the hash field.

Assuming there are n users, $\mathcal{P}$'s proof generation time is $O(n)$ and the complexities of the individual proof size and verification time are both $O(\log n)$. The commitment size on the PBB is constant $O(1)$. However, there is a security flaw in the design and the prover is able to claim smaller liabilities [43], i.e., a malicious prover can set the value field $c = \max(c_{lch}, c_{rch})$ while also being able to generate inclusion proofs that pass verifications successfully. We demonstrate the flaw by an example in appendix B.1. Moreover, this scheme doesn't provide any privacy of our concern here. The value of the total liabilities is public, and the population n and individual liabilities can be inferred via inclusion proofs. For example, each user observes the prover's liabilities to a user next to him/her in the Merkle tree via the proof. The right-most user can infer the population by the position of his/her leaf node in the tree, and everyone can speculate n by the tree height.

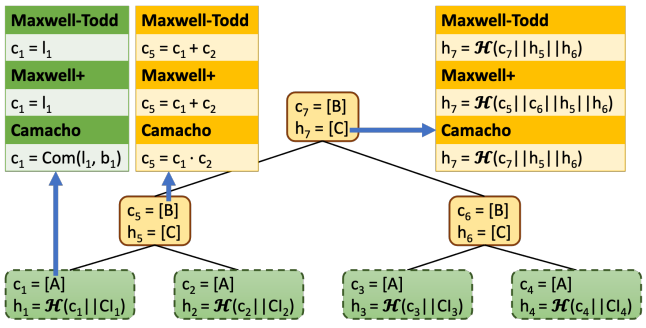


Figure 1: *Summation Merkle tree constructions in Maxwell-Todd, Maxwell+ and Camacho.*

Maxwell+. We refer by Maxwell+ the Maxwell-Todd's protocol adopting the fix in [43]. The fix is demonstrated in fig. 1, modifying the merge function of the hash field to include both values and hashes of child nodes, i.e., $h = \mathcal{H}(c_{lch}||c_{rch}||h_{lch}||h_{rch})$. Intuitively, this binds the value of each node in the parent hash to provide verifiability and prevent the prover from manipulating the values on the Merkle path. Maxwell+ doesn't introduce additional complexity on proving/verification time and proof/commitment size; it does not offer better privacy than Maxwell either.

Maxwell++. Maxwell++ [19] is a protocol extending Maxwell+ to provide privacy of population and individual liabilities. Maxwell++ splits individual liabilities into small units and shuffles them, then maps each unit to a leaf node in a summation Merkle tree. Denoted by M, the splitting factor is the average number of entries each user's value is split into. The data written on the PBB is still $O(1)$, but the proving time is $O(n \cdot M)$ and the proof size and verification time are increased to $O(\log(n \cdot M))$, because the number of leaf nodes is $n \cdot M$. This scheme provides privacy of the number of users and individual liabilities to some extent highly depending on the selection of M. The greater M is, the better the privacy Maxwell++ provides and the higher the complexity it ends up with. Full privacy is achieved only when each user's value is split into the smallest possible unit, but this would be inefficient. Therefore, we say that Maxwell++ offers partial privacy. Besides, Maxwell++ still cannot conceal the total liabilities.

Camacho. Camacho [13] proposed a PoL protocol guaranteeing privacy of individual liabilities and total liabilities by using homomorphic commitments, Pedersen commitments in particular, and zero-knowledge range proofs (ZKRPs). Cryptographic commitments guarantee hiding (an adversary cannot learn the value from the commitment) and binding (a prover cannot open the commitment to a different value). Denote by $\text{Com}(l, b)$ a commitment to $l \in \mathbb{Z}_q$, where b is the blinding factor randomly selected from $\mathbb{Z}_q$. By homomorphism, $\text{Com}(l_1, b_1) \cdot \text{Com}(l_2, b_2) = \text{Com}(l_1 + l_2, b_1 + b_2)$. Details about Pedersen commitments are in appendix C.3. As depicted in fig. 1, Camacho's scheme is based on Maxwell-Todd's construction, but replaces the value field in each node with its Pedersen commitment to hide the value. When required to reveal the total liabilities to an eligible auditor, the prover can open the commitment in the Merkle root with the blinding factor. The prover may also prove the range of the total liabilities with the commitment.

To prove inclusion of the prover's liabilities to a user, the prover sends a Merkle proof as always. However, that's not sufficient

because there might be an overflow when multiplying two commitments, i.e., $\mathrm{Com}(l_1+l_2, b_1+b_2)$ is opened to a smaller value $(l_1+l_2) \bmod q$ when $l_1+l_2 \geq q$. Therefore, the prover additionally generates a ZKRP for each commitment on the Merkle path to prevent overflows. ZKRPs guarantees that the committed value is within a certain range without leaking the value, details in appendix C.4.

Camacho guarantees privacy of individual liabilities and total liabilities, but not the number of users. It doesn't address the security flaw in Maxwell-Todd's summation Merkle tree either.

Provisions. Provisions [25] also uses homomorphic commitments and zero-knowledge proofs to preserve the privacy of liabilities but without using a Merkle tree. The prover simply publishes a commitment to his/her liabilities to each user with a range proof on the PBB. Each user verifies that his/her commitment and all range proofs are valid. A commitment to the total liabilities can be extracted by multiplying all commitments on the PBB. Provisions attempts to obscure the number of users by padding with dummy users. Denoting by B the number of dummy users padded, the proving and verification time, and the commitment size on the PBB are $O(n+B)$. Each verifier only receives a blinding factor of their commitment for verification so the proof size is constant. Since the complexity grows with B and B determines the level of privacy provided, we say that privacy of population is partially satisfied.

If there is a third party auditor verifying the range proofs for all commitments, each user only needs to verify that a valid commitment to his/her amount is on the PBB. In this case, the verification complexity for each client is $O(1)$ while $O(n+B)$ for the auditor.

The most severe practicality issue here is the massive data committed on a PBB. For example, the commitment size is over 8GiB when $n+B = 1M$ [25]. Even if the prover writes only the hash of all commitments on the PBB, each user will need to download all the commitments to verify consistency with the hash, which makes the bandwidth complexity intolerable. Provisions argued against the use of a Merkle tree by claiming that the proof size for each user will be several hundred KiB due to the expensive ZKRPs for multiple commitments. However, this is out-dated. Using Bulletproofs [11], which hadn't been proposed at the time of Provisions' publishing, the range proofs can be aggregated so the size can be much smaller, only a few KiB for each user as we show in section 6.2.

DAPOL. DAPOL [20] builds on Camacho adopting the fix in [43] and using sparse Merkle trees (SMT) to hide the number of users. Assuming the maximum potential population is N, the sparse Merkle tree is of height $H \geq \lceil \log N \rceil$. Each user is mapped to a random bottom-layer node in an empty SMT. The prover then builds this tree with minimal nodes such that each node either has two child nodes or none. We call the nodes with no child and not mapped to a user *padding nodes*, each with a commitment to 0. The number of nodes in an SMT is $O(n \cdot H)$, thus the proving time is $O(n \cdot H)$. Each inclusion proof consists of H tree nodes and range proofs, thus the proof size and verification time for each user are both $O(H)$.

DAPOL targets full privacy of the population but its padding node construction is flawed, probably due a typo. Padding nodes have a hash field $h = \mathcal{H}(\text{“}pad\text{”}||idx)$, where idx is the unique identifier of a node, e.g., a natural number, or $(height, position)$ such as $(3, 001)$ for Node 2 in fig. 1. The input to $\mathcal{H}(\cdot)$ is deterministic, so is the hash of a padding node, making padding nodes distinguishable from nodes of other types. Thus given an inclusion proof with padding nodes at heights $\{x_1, \cdots, x_m\}$ (root at height 0), the population can be bounded within range $[H-m+1, 2^H - \sum_{i=1}^{m} 2^{H-x_i}]$. An example of this privacy leak is presented in appendix B.2. Besides, DAPOL utilizes expensive verifiable random function (VRF) to construct SMT without clarifying necessity. Additionally, each user is mapped to a leaf node in a verifiable way in DAPOL, i.e., a user's ID determines which leaf node to map. Therefore, there is a chance that more than one user is mapped to the same leaf. E.g., if there are 1M users and the tree height is 32, it's almost certain that this will happen. To avoid collision, the tree height needs to be sufficiently large, leading to higher complexities of proving/verification time and proof size. Moreover, DAPOL lacks formal analysis of its security and privacy.

3 DEFINITIONS

There has not been a satisfactory formal definition of PoL so far. The authors of Provisions only presented the security definitions of proof-of-solvency which consists of PoL as a subtask, and informally the properties their PoL protocol satisfies. As mentioned, in some applications the prover may want to keep the number of users secret. In the definitions in Provisions, however, the population is assumed to be public by default. Because of this, their security definitions are scheme specific, not applicable for the examination of other protocols such as DAPOL which aims to provide population privacy. In contrast, DAPOL generalizes PoL for a wider range of applications and provides a better privacy definition allowing the population to be concealed. Nevertheless, it mixes the failure probability of distributed auditing in the security definition thus makes it tedious. In this section, we address these issues and formalize PoL as a generalized cryptographic primitive for various applications with minimal security requirements.

3.1 Entities

A generalized PoL protocol PoL involves two entities:

- *Users*: Denoted by $\mathcal{U} = \{u_1, \cdots, u_n\}$, this is a set of n users.
- *Prover*: Denoted by $\mathcal{P}$, the prover is the subject liable to the users for certain obligations. For example, in the proof-of-solvency case, $\mathcal{P}$ is liable to users in $\mathcal{U}$ for their deposits. Note that $\mathcal{P}$ has no incentive to increase the total liabilities.

3.2 Functions

A PoL consists of algorithms defined as below:

- Setup: $(PD, SD) \xleftarrow{\$} \mathsf{Setup}(1^\kappa, DB)$. Executed by $\mathcal{P}$, the probabilistic polynomial-time (p.p.t.) algorithm takes as input κ, the security parameter, and $DB = \{(id_u, l_u)\}_{u \in \mathcal{U}}$, the data set of user ID and individual liability pairs, and outputs PD, the public data committed on the PBB, and SD, $\mathcal{P}$'s private data which is kept secret. Note that for each user $u \in \mathcal{U}$, id_u and l_u denote u's ID and $\mathcal{P}$'s liabilities to u, respectively.
- ProveTot: $(L, \Pi) \leftarrow \mathsf{ProveTot}(DB, SD)$. Executed by $\mathcal{P}$, the polynomial-time algorithm takes as input the data set DB and $\mathcal{P}$'s private data SD, and outputs $\mathcal{P}$'s total liabilities L and its associated proof Π.

- VerifyTot: $\{0,1\} \leftarrow \mathsf{VerifyTot}(PD, L, \Pi)$. Given the total liabilities L and its associated proof Π, anyone can audit the validity of L according to the public data PD committed by $\mathcal{P}$ on the PBB. The polynomial-time algorithm returns 1 if the verification succeeds and 0 otherwise.
- Prove: $\pi \leftarrow \mathsf{Prove}(DB, SD, id)$. Executed by the prover, the polynomial-time algorithm takes as input the data set DB, $\mathcal{P}$'s private data SD and a user ID id, and outputs a proof π indicating the inclusion of $\mathcal{P}$'s liabilities to the user in the total liabilities.
- Verify: $\{0,1\} \leftarrow \mathsf{Verify}(PD, id, l, \pi)$. Executed by a user, the polynomial-time algorithm takes as input the public data PD committed by $\mathcal{P}$, the user's ID id, $\mathcal{P}$'s liabilities to the user l and the associated inclusion proof π. It returns 1 if the verification succeeds and 0 otherwise.

A PoL is a collection of the algorithms as defined above, i.e., $\mathsf{PoL} = (\mathsf{Setup}, \mathsf{ProveTot}, \mathsf{VerifyTot}, \mathsf{Prove}, \mathsf{Verify})$. Note that we define ProveTot and VerifyTot as above for simplicity and generality. In some scenarios, it may not be the exact value of $\mathcal{P}$'s total liabilities that is of concern, but the range the total liabilities falls in or its comparison with another value. Depending on the actual requirements of a particular application, instead of revealing the total liabilities for verification, the prover might compute different verifiable claims about it. For instance, if we are only interested in solvency but not the exact values of liabilities and assets, the prover may generate a zero-knowledge proof showing that the total liabilities are no more than the total assets. We will go into details of these claims in section 4.4.4.

3.3 Threat Model

A malicious prover potentially corrupting any number of users may attempt to reduce his/her total liabilities, e.g., via manipulating or discarding the liabilities to non-corrupted users. However, there is no motivation for the adversarial prover to increase the total liabilities. Note that this assumption on incentives is key to PoL, and it relaxes the security requirements so simpler solutions are possible. Without this, a valid PoL scheme might be more complicated because it needs to further prevent a prover from raising the value by inserting or duplicating positive entries.

Users can establish secure communication channels with $\mathcal{P}$ and authenticate their identities. The authentication process is out of scope for this paper. Meanwhile, users may be corrupted by an adversary to break privacy, i.e., the adversary attempts to learn more information than she should from the corrupted users, such as the number of users or $\mathcal{P}$'s liabilities to non-corrupted users.

The PBB provides a consistent view of the data on it to everyone. Anyone can read and write data on the PBB and the content cannot be tampered with. In PoL, the public data PD is written on the PBB. This is necessary to prevent the malicious prover from showing inconsistent commitments to different users.

3.4 Security Definitions

Definition 3.1. Valid data set. A dataset $DB = \{(id_u, l_u)\}_{u \in \mathcal{U}}$ is $(N, MaxL)$-valid, N and $MaxL$ being two positive integers, iff the following conditions are met:

- there are at most N users, i.e., $n = |DB| \leq N$;
- for any two distinct users $u, u' \in \mathcal{U}$, $id_u \neq id_{u'}$;
- for any user $u \in \mathcal{U}$, $0 \leq l_u < MaxL$.

Denote by $\mathsf{PoL}(N, MaxL)$ a PoL protocol targeted for all of the $(N, MaxL)$-valid data sets. A $\mathsf{PoL}(N, MaxL)$ is *secure* iff both completeness and soundness as defined below are satisfied:

Definition 3.2. Completeness. A $\mathsf{PoL}(N, MaxL)$ is complete if for any $(N, MaxL)$-valid data set DB,

$$\begin{aligned}\Pr[(PD, SD) \xleftarrow{\$} \mathsf{Setup}(1^\kappa, DB),\\ (L, \Pi) \leftarrow \mathsf{ProveTot}(DB, SD),\\ \forall u \in \mathcal{U}, \pi_u \leftarrow \mathsf{Prove}(DB, SD, id_u) :\\ \mathsf{VerifyTot}(PD, L, \Pi) = 1 \wedge\\ \forall u \in \mathcal{U} : \mathsf{Verify}(PD, id_u, l_u, \pi_u) = 1 \wedge\\ L \geq \textstyle\sum_{u \in \mathcal{U}} l_u] = 1\end{aligned}$$

Completeness guarantees that if all parties are honest and follow the protocol, the verifications should all succeed and the proved total liabilities should be no less than the sum of the prover's liabilities to individual users.

Definition 3.3. Soundness. A $\mathsf{PoL}(N, MaxL)$ is sound if for any $(N, MaxL)$-valid data set DB, for any p.p.t. adversarial prover $\mathcal{A}^*$ potentially corrupting any number of users, there exists a negligible function $\epsilon(\cdot)$ such that for any subset V of non-corrupted users,

$$\begin{aligned}\Pr[(PD, L, \Pi, \{\pi_u\}_{u \in V}) \xleftarrow{\$} \mathcal{A}^*(1^\kappa, DB) :\\ \mathsf{VerifyTot}(PD, L, \Pi) = 1 \wedge\\ \forall u \in V, \mathsf{Verify}(PD, id_u, l_u, \pi_u) = 1 \wedge\\ L < \textstyle\sum_{u \in V} l_u] \leq \epsilon(\kappa)\end{aligned}$$

Soundness guarantees that a computationally bounded adversarial prover is not able to cheat on the total liabilities. In particular, for any subset of non-corrupted users that successfully verify the inclusion of the prover's liabilities to them, if the committed total liabilities is associated with a valid proof, its value is no less than the sum of the prover's liabilities to these users.

Note that we define the probability over any subset of honest users instead of the entire set of them to capture the nature of distributed auditing. In other words, the amount of $\mathcal{P}$'s total liabilities is guaranteed with respect to the users that perform the verification. If we define it over the entire set of honest users instead, the total liabilities won't be bounded when some user is given an invalid proof (i.e., $\mathsf{Verify}(\cdot)$ returns 0). A protocol satisfying so-defined soundness is not useful in practice because this particular user might never perform verification. In this scenario, other users will not detect and report a misconduct of the prover even if the amount of total liabilities committed is an arbitrary value. In contrast, with our soundness definition, the committed total liabilities is at least bounded by the total liabilities to users who verify.

PoL falls under the more general category of transparency solutions where a prover trusted for privacy but not honesty maintains a dataset. The general notion of soundness in most of these works, e.g., SEEMless [21], is non-equivocation. Our soundness is an analog of non-equivocation, making the values of users that verify concretely counted in the total amount, thus lower-bounding the total to the sum of users that verify. We restrict ourselves to non-equivocation because this is the only known definition achievable

without generic SNARKs which would lead to a significant degradation in efficiency and mobile-friendliness.

3.5 Privacy Definitions

We define *user privacy* against $V \subseteq \mathcal{U}$, a subset of users corrupted by an adversary. The adversary has access to the ID and liability pairs of users in V. She can also send queries to the prover for inclusion proofs of $\mathcal{P}$'s liabilities to the corrupted users, so possesses $\pi_u \leftarrow \mathsf{Prove}(DB, SD, id_u)$ for all $u \in V$. We aim to guarantee that the view of the adversary throughout an execution of PoL can be simulated by a simulator given limited information. In particular, in an execution of PoL upon a valid data set DB, letting $(PD, SD) \xleftarrow{\$} \mathsf{Setup}(1^\kappa, DB)$ and $\pi_u \leftarrow \mathsf{Prove}(DB, SD, id_u)$ for all $u \in V$, the view of the adversary corrupting $V \subseteq \mathcal{U}$ is $\mathsf{view}_V^{\mathsf{user}} = (PD, DB[V], \{\pi_u\}_{u \in V})$. Note that $DB[V] = \{(id_u, l_u)\}_{u \in V}$, which is the data set of user ID and liability pairs of users in $V \subseteq \mathcal{U}$. User privacy against V corrupted by an adversary requires that $\mathsf{view}_V^{\mathsf{user}}$ can be simulated by a p.p.t. simulator that does not have access to DB but only to 1^κ, $DB[V]$ and the leakage function $\Phi^{\mathsf{user}}(DB, V)$. More formally:

Definition 3.4. User privacy. A $\mathsf{PoL}(N, MaxL)$ is Φ^{user}-*private* against $V \subseteq \mathcal{U}$, a subset of users corrupted by an adversary, if there exists a p.p.t. simulator $\mathcal{S}$ such that for any $(N, MaxL)$-valid data set DB, the following two distributions are computationally indistinguishable:

- $\{(PD, SD) \xleftarrow{\$} \mathsf{Setup}(1^\kappa, DB), \forall u \in V, \pi_u \leftarrow \mathsf{Prove}(DB, SD, id_u) : PD, DB[V], \{\pi_u\}_{u \in V}\}$
- $\{\mathcal{S}(1^\kappa, DB[V], \Phi^{\mathsf{user}}(DB, V))\}$

We also define *auditor privacy* against an adversary that has access to the output of ProveTot and corrupts a subset of users $V \subseteq \mathcal{U}$. Similarly, in an execution of PoL upon a valid data set DB, letting $(PD, SD) \xleftarrow{\$} \mathsf{Setup}(1^\kappa, DB)$, $(L, \Pi) \leftarrow \mathsf{ProveTot}(DB, SD)$ and $\pi_u \leftarrow \mathsf{Prove}(DB, SD, id_u)$ for all $u \in V$, the view of the adversary corrupting $V \subseteq \mathcal{U}$ is $\mathsf{view}_V^{\mathsf{auditor}} = (PD, L, \Pi, DB[V], \{\pi_u\}_{u \in V})$. Auditor privacy requires that $\mathsf{view}_V^{\mathsf{auditor}}$ can be simulated by a p.p.t. simulator that does not have access to DB but only to 1^κ, L, $DB[V]$ and the leakage function $\Phi^{\mathsf{auditor}}(DB, V)$.

Definition 3.5. Auditor privacy. A $\mathsf{PoL}(N, MaxL)$ is Φ^{auditor}-*private* against any malicious auditor corrupting any subset of users $V \subseteq \mathcal{U}$, if there exists a p.p.t. simulator $\mathcal{S}$ such that for any $(N, MaxL)$-valid data set DB, the following two distributions are computationally indistinguishable:

- $\{(PD, SD) \xleftarrow{\$} \mathsf{Setup}(1^\kappa, DB), (L, \Pi) \leftarrow \mathsf{ProveTot}(DB, SD), \forall u \in V, \pi_u \leftarrow \mathsf{Prove}(DB, SD, id_u) : PD, L, \Pi, DB[V], \{\pi_u\}_{u \in V}\}$
- $\{\mathcal{S}(1^\kappa, L, DB[V], \Phi^{\mathsf{auditor}}(DB, V))\}$

Note that the privacy definitions above cover the case where everything can be public, i.e., $DB \subseteq \Phi^{\mathsf{user/auditor}}(DB, V)$. For example, in charity applications, the donations of each donor might be public. There are also special voting systems, e.g., parliaments, where for transparency reasons, all votes should be revealed.

4 DESIGN SPECIFICATIONS

In this section, we present concrete PoL schemes. First, we propose DAPOL+, a PoL protocol extending DAPOL but fixing its privacy issue, getting rid of VRF and deterministic mapping as mentioned in section 2. We then formally prove that DAPOL+ protocol satisfies the security and privacy properties as defined earlier. Second, we discuss how to resolve disputes in DAPOL+. Third, we discuss different accumulator variants that can be used in DAPOL+ and their trade-offs. Fourth, we consider other additional features potentially desired by different applications and propose solutions.

4.1 DAPOL+

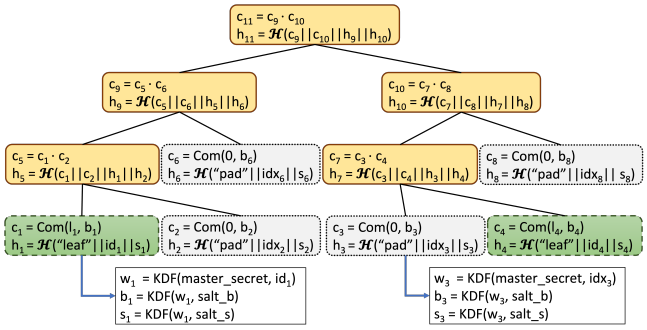


Figure 2: *Padded height-3 DAPOL+ tree.*

Recall that among existing schemes of PoL [13, 19, 20, 25, 73], DAPOL aims to provide the strongest security and privacy guarantees. However, there is a privacy issue in their construction of the SMT, i.e., the padding nodes are distinguishable from tree nodes of other types in their design, which could leak the number of users. Moreover, they mentioned utilizing VRFs in the construction of SMT nodes, although as an alternative, without analyzing the necessity of such expensive cryptographic primitives. In addition, each user is deterministically mapped to a leaf node in the SMT in DAPOL, which requires the height of the SMT to be sufficiently large to avoid collision.

We propose DAPOL+, by extending DAPOL with the basic idea of using SMT for privacy and efficiency benefits, and further making the following improvements:

- fixing the privacy leak in DAPOL;
- getting rid of VRF for efficiency, mobile and post-quantum friendliness;
- utilizing random mapping instead of deterministic shuffling, thus allowing smaller SMT heights and proof size (SMT variants discussed in section 4.3).

DAPOL+ provides provable security and privacy.

Briefly speaking, in DAPOL+, the prover first makes a commitment to the total liabilities on the PBB so that all users have the same view of the commitment and the prover cannot open to different values afterwards. Each user then may check if the amount of $\mathcal{P}$'s liabilities to him/her is included in the total liabilities to make sure the prover is not cheating and the amount of total liabilities is properly bounded. To generate a proof of inclusion, DAPOL+

leverages homomorphic Pedersen commitments to hide the exact values of the prover's liabilities, and an SMT as an accumulator to conceal the number of users and minimize usage of the PBB. In particular, the inclusion proof is a Merkle proof in a summation Merkle tree together with range proofs for all nodes on the Merkle path to guarantee no overflow in the sum of the committed values when multiplying the Pedersen commitments.

Denote by $\mathsf{Prot}_{\mathsf{DAPOL+}}(N, MaxL)$ the main protocol of DAPOL+ targeted for all $(N, MaxL)$-valid data sets. We now present the details of $\mathsf{Prot}_{\mathsf{DAPOL+}}(N, MaxL)$ in the following flow, with pseudocode in appendix D:

(1) Select public protocol parameters.
(2) Set up, i.e., generate the SMT and commitments.
(3) Prove the total liabilities.
(4) Verify the total liabilities.
(5) Prove individual liabilities.
(6) Verify individual liabilities.

Public protocol parameters. All of the protocol parameters of $\mathsf{Prot}_{\mathsf{DAPOL+}}(N, MaxL)$ are fixed in the protocol, and we don't need a trusted setup. First of all, N and $MaxL$ are public protocol parameters. And we need to publicly fix the height H of the SMT in $\mathsf{Prot}_{\mathsf{DAPOL+}}(N, MaxL)$. Note that we only require $H \geq \lceil \log N \rceil$. Although security and privacy of a PoL can be guaranteed as long as this condition is met, different selections of H reflect different tradeoffs. We discuss this in section 4.3.

In addition, we have a public group G of prime order $q \geq (H \cdot N + 1) \cdot MaxL$. Let g_1 and g_2 be fixed public generators of G, whose relative discrete logarithm is unknown to anyone, i.e., no entity has the knowledge of x such that $g_1^x = g_2$. Note that g_1 and g_2 can be selected deterministically. In our implementation in section 6.2, for instance, G is the Ristretto group for Curve25519, g_1 is the base point in G, and g_2 is a point converted from the hash of g_1. Note that although Curve25519 is an elliptic curve, we use the conventional multiplicative notation throughout this paper for uniformity, i.e., xg_1 is written as g_1^x.

Apart from N, $MaxL$, the SMT height H, the group G, its two generators g_1 and g_2, we also need to fix two strings as public identifiers $salt\_b$ and $salt\_s$ for deterministically constructing SMT nodes, which we'll explain in detail later.

Setup. The prover first picks a random secret $master\_secret$. Next, $\mathcal{P}$ randomly maps each user to a bottom-layer leaf node in an SMT of a fixed height H and then generates the whole SMT. An example of a summation SMT of height 3 containing 2 users is depicted in fig. 2. There are three types of nodes:

- Leaf nodes, denoted by green dashed blocks in fig. 2, are bottom-layer tree nodes mapped to users. Note that this is different from the conventional definition of leaf nodes that have no child nodes and can be in any layer of a tree. Each leaf node contains a Pedersen commitment to $\mathcal{P}$'s liabilities to the corresponding user, i.e., $c_u = Com(l_u, b_u) = g_1^{l_u} \cdot g_2^{b_u}$ where b_u is the blinding factor, and a hash of the user's ID id_u concatenated with a mask s_u, i.e., $h_u = \mathcal{H}(\text{"}leaf\text{"}||id_u||s_u)$. Note that the uniqueness of h_u is guaranteed by the uniqueness of users' IDs (in practice we can use users' credential information such as phone number or email address), which is also key to a valid data set. Therefore, a malicious prover cannot map two users with the same l_u to the same leaf node to claim smaller liabilities without being detected.
  Both b_u and s_u should be hard to guess for privacy concerns. If a non-corrupted user u happens to be mapped to the sibling node of a corrupted user, $\mathcal{A}$ might receive (c_u, h_u) as a part of the inclusion proof of the corrupted user. In this case, if b_u can be easily guessed, $\mathcal{A}$ can infer l_u from c_u by looking up the table of g_1^x for all $x \in [0, MaxL)$. Similarly, if s_u can be guessed, $\mathcal{A}$ can infer whether the bottom-layer node in a Merkle proof is a leaf node corresponding to a user or a padding node by brute forcing all possible IDs due to their low entropy. This gives the adversary additional information about the number of users.
  We want to extract b_u and s_u deterministically from $w_u = KDF(master\_secret, id_u)$, which is the seed deterministically extracted by a key derivation function (KDF) taking $\mathcal{P}$'s secret $master\_secret$ and id_u as inputs. This is to allow $\mathcal{P}$ to reproduce the contents in leaf nodes with minimized storage, i.e., DB together with $\mathcal{P}$'s $master\_secret$. The property of having a minimized data base can be beneficial in terms of the data transferred when an external auditor initiates a full investigation requiring $\mathcal{P}$ to send everything to the auditor. We also want to extract b_u and s_u independently, i.e., knowledge of either doesn't help learn the other, for selective information disclosure, which we explain in section 4.4.5. Therefore, we extract them via $b_u = KDF(w_u, salt\_b)$ and $s_u = KDF(w_u, salt\_s)$ respectively, where $salt\_b$ and $salt\_s$ are two public identifiers in the protocol.
- Padding nodes, denoted by gray dotted blocks, are nodes that have no child nodes in the tree apart from leaf nodes mapped to users. Padding nodes do not contribute to the total liabilities but are dummy nodes guaranteeing that each node in the tree has either two child nodes or none, allowing generation of Merkle proofs for leaf nodes. Each padding node contains a Pedersen commitment to 0, i.e., $c_i = Com(0, b_i) = g_2^{b_i}$, and a hash of the node index concatenated with a mask s_i, i.e., $h_i = \mathcal{H}(\text{"}pad\text{"}||idx_i||s_i)$. The extraction of b_i and s_i is the same as that for leaf nodes, i.e., $b_i = KDF(w_i, salt\_b)$ and $s_i = KDF(w_i, salt\_s)$, where w_i is extracted from $\mathcal{P}$'s $master\_secret$ and the node index idx_i by $w_i = KDF(master\_secret, idx_i)$. The prefix "$pad$" of the preimage of w_i can be used to prove padding nodes for random sampling, which we discuss in section 4.4.5. Note that the seed w_i for each padding node should be kept secret to $\mathcal{P}$, otherwise an adversary can distinguish between a padding node and a node of another type via the hash, which leaks information about the number of users.
- Internal nodes, denoted by yellow solid blocks, are the tree nodes that have two child nodes. Each internal node contains the multiplication of the commitments in its child nodes, i.e., $c_i = c_{lch_i} \cdot c_{rch_i}$, and a hash of commitments and hashes in its child nodes, i.e., $h_i = \mathcal{H}(c_{lch_i}||c_{rch_i}||h_{lch_i}||h_{rch_i})$, where lch_i and rch_i denote the left and right child nodes of i respectively. The Merkle root is an internal node.

The SMT can be generated layer by layer starting from the bottom (at height H). Initially the SMT only contains leaf nodes in its bottom layer. We insert a padding node whenever the sibling of a leaf node, i.e., having the same parent as the leaf node, doesn't exist in the SMT. Then we insert an internal node as a parent for each pair of existing sibling nodes in the bottom layer. Next we move to the layer above, inserting padding nodes as siblings to existing nodes in this layer and internal nodes as parents in the upper layer. We repeat the step above until we reach to the root (at height 0). The complexity of this procedure is linear in the number of nodes in the SMT, depending on the number of users and the tree height.

In the function $(PD, SD) \xleftarrow{\$} \mathsf{Setup}(1^\kappa, DB)$, PD is the commitment and hash pair of the Merkle root, i.e., $PD = (c_{root}, h_{root})$. In fig. 2, e.g., $PD = (c_{11}, h_{11})$. On the other hand, SD includes *master_secret* and the mapping from users to bottom-layer leaf nodes in the SMT. Apart from that, the SMT could also be part of SD but this is not necessary. Although storing the whole SMT allows faster generation of inclusion proofs later in response to users' queries, the prover can reproduce the same SMT deterministically from *master_secret*, DB and the mapping only. We discuss the possibility of having SD = *master_secret* via deterministic mapping in section 4.3.

Prove total liabilities. To prove the total liabilities, $\mathcal{P}$ simply reveals the blinding factor in the Merkle root. By the homomorphism of Pedersen commitments, for any $c_1 = Com(l_1, b_1)$ and $c_2 = Com(l_2, b_2)$, we know $c = c_1 \cdot c_2 = Com(l_1 + l_2, b_1 + b_2)$. Therefore, $c_{root} = Com(\sum_{u \in \mathcal{U}} l_u, \sum_{u \in \mathcal{U}} b_u + \sum_{\text{padding node } i} b_i)$.

In the function $(L, \Pi) \leftarrow \mathsf{ProveTot}(DB, SD)$, L is $\mathcal{P}$'s total liabilities to users, i.e., $L = \sum_{u \in \mathcal{U}} l_u$, and Π is the sum of blinding factors in all leaf and padding nodes, i.e., $\Pi = \sum_{u \in \mathcal{U}} b_u + \sum_{\text{padding node } i} b_i$.

Verify total liabilities. To verify the total liabilities, anyone receiving the proof can act as a verifier and check if c_{root} committed on the PBB is a Pedersen commitment to the total liabilities.

In the function $\{0, 1\} \leftarrow \mathsf{VerifyTot}(PD, L, \Pi)$, if $c_{root} = g_1^L \cdot g_2^\Pi$ (note that $PD = (c_{root}, h_{root})$), the function returns 1. Otherwise it returns 0.

Prove individual liabilities. To query for the inclusion proof of l, a user can establish a secure communication channel with $\mathcal{P}$ and prove his/her identity with respect to id. The implementation of the authentication is out of the scope of this paper so we don't go into details. $\mathcal{P}$ ignores the query when the authentication fails. Thus, a bounded adversary has no access to proofs of non-corrupted users but only to corrupted users.

Upon receiving an authenticated query, $\mathcal{P}$ locates the leaf node mapped to the user, and retrieves the Merkle path $\{(c_i, h_i)\}_{i \in [1,H]}$, where (c_i, h_i) is the commitment and hash pair in the sibling of the node at height i on the path from the user's leaf node to the root. The Merkle path proves the inclusion of the leaf node in the SMT, but when multiplying the commitments along the path, there might be an overflow, i.e.,the sum of two values exceeds the group order $q \geq (H \cdot N + 1) \cdot MaxL$, so c_{root} might commit to a smaller value. Therefore, we need to prove each c_i commits to a value within the range $[0, N \cdot MaxL)$. In particular, we adopt Bulletproofs [11] which enables aggregation of zero-knowledge range proofs for multiple values efficiently and succinctly. Additionally, non-interactive Bulletproofs via the Fiat-Shamir transform is proved to be secure [17, 35]. Overall, the inclusion proof of individual liabilities consists of the user's blinding factor b, his/her mask s, a Merkle path in the summation tree and aggregated range proofs for commitments on the path.

In the function $\pi \leftarrow \mathsf{Prove}(DB, SD, id)$, the prover generates $\pi = (b, s, \{(c_i, h_i)\}_{i \in [1,H]}, \pi_{range})$, where (c_i, h_i) is the commitment and hash pair in the node at height i on the Merkle path, and π_{range} is a zero-knowledge proof that each c_i on the Merkle path commits to a value within range $[0, N \cdot MaxL)$.

Verify individual liabilities. To verify $\mathcal{P}$'s individual liabilities to a user, he/she first verifies the Merkle path, i.e., computing the internal nodes on the path from the leaf node to the root and checking if the root matches with PD committed on the PBB. The user also verifies the range proofs to make sure each commitment on the Merkle path commits to a value within the proper range.

In the function $\{0, 1\} \leftarrow \mathsf{Verify}(PD, id, l, \pi)$, the verifier computes $(c'_H = \mathsf{Com}(l, b), h'_H = \mathcal{H}(\text{“}leaf\text{”}||id||s))$ which is the content in the leaf node. Then the verifier computes $c'_i = c'_{i+1} \cdot c_{i+1}$ with i iterating from $H - 1$ to 0, where c_i is contained in π. And similarly for h'_i in the internal nodes. If $(c'_0, h'_0) = PD$ and the range proofs in π are valid, the function returns 1. Otherwise it returns 0.

We claim that the following security and privacy properties hold for $\mathsf{Prot}_{\mathsf{DAPOL+}}$ under the discrete logarithm (DL) assumption in the random oracle model and the algebraic group model [30]. We provide detailed proofs in appendix E.

Theorem 4.1. $\mathsf{Prot}_{\mathsf{DAPOL+}}(N, MaxL)$ *is secure.*

Theorem 4.2. $\mathsf{Prot}_{\mathsf{DAPOL+}}(N, MaxL)$ *is* Φ^{user}*-private, where* $\Phi^{\text{user}} = \emptyset$.

Theorem 4.3. $\mathsf{Prot}_{\mathsf{DAPOL+}}(N, MaxL)$ *is* Φ^{auditor}*-private, where* $\Phi^{\text{auditor}} = \emptyset$.

Note that $\Phi^{\text{user/auditor}} = \emptyset$ indicates that DAPOL+ provides the strongest privacy not leaking any additional information.

4.2 Dispute Resolution

When $\mathcal{P}$ misbehaves, e.g., fails to respond to a user with a valid inclusion proof, the user should be able to raise a dispute with evidence. We divide this into two subtasks: 1. dispute resolution for PoL assuming agreement on individual liabilities; 2. dispute resolution for disagreement on individual liabilities. Note that the former task makes a prerequisite assumption that $\mathcal{P}$ agrees with each user on l_u in DB. To achieve this, users may obtain a proof of the value of l_u from $\mathcal{P}$. The form of the proof varies across applications, e.g., a receipt for each transaction in the solvency case, or a donation certificate in the charity fund raising case. The proof could also be $\mathcal{P}$'s digital signature on l_u.

For the former task, an invalid proof issued by $\mathcal{P}$ can be a concrete evidence in a dispute. The only exception is when the proof is not available. Data availability is a hard task [12] because the case of $\mathcal{P}$ not sending the proof is indistinguishable with the case when a user raises a false alarm. This is inevitable in any centralized system. A probabilistic workaround is to have a third party auditor

to query proofs on users' behalf when there is suspicion, which however, enables the auditor to lower bound the number of users.

The latter task depends on the specific application where PoL is used. It remains as an open problem for many applications [22]. We take the solvency case as an example and empirically analyze all possible scenarios of dispute resolution in appendix F.

4.3 Accumulator Variants

In the main protocol of DAPOL+, we utilize an SMT as an accumulator and randomly map users to a bottom-layer node in the tree. In this section, we explore four accumulator variants that can be plugged into DAPOL+. Without sacrificing the security and privacy of DAPOL+, each variant may provide additional features desired in some applications, as summarized in table 2.

Table 2: *Comparison between accumulator variants.*

Variant	Cost	Features	Applications
NDM-SMT	Low	Simple and efficient	Fundraising
DM-SMT	Medium	Hide proof access patterns combined with PIR	Negative reviews
ORAM-based	High	Allow updates and hide patterns	Solvency
Hierarchical	Low	Distribute work to hierarchical institutions	COVID reports

4.3.1 Non-deterministically mapping SMT (NDM-SMT). NDM-SMTs is a simplest type of SMT posing no constraints on the SMT other than that $H \geq \lceil \log N \rceil$. For a population of 10B, e.g., an NDM-SMT of height 33 is sufficient. Therefore, the proof size and the complexity of proof generation and verification which take H as a factor could be rather small. This efficiency benefit makes NDM-SMT suitable for applications where the prover entity varies from companies to individuals with different business sizes and computation resources, e.g., fundraising and syndicated loans.

To implement this variant, however, we cannot simply select a random bottom-layer node for each user due to non-negligible probability of collisions. A rudimentary method is to retry when collision occurs, of which the expected number of random number generation (RNG) operations is $2^H \cdot \sum_{i=2^H-n+1}^{2^H} \frac{1}{i}$ which approaches to $H \cdot 2^H$ when n is close to 2^H. Thus this method is practical only when $n << 2^H$, which does not apply to the use case of NDM-SMTs. We need a scheme to map users to leaf nodes more efficiently. It turns out Durstenfeld's shuffle algorithm optimized by a HashMap can achieve the goal. The HashMap originally is empty. We start from the first user and repeat the following steps for n times to generate a random mapping. For the i-th user, randomly select a number $k \in [i, 2^H]$. If k exists as a key in the HashMap, the i-th user is mapped to $\mathsf{HashMap}(k)$. Otherwise the user is mapped to k. At the end of each iteration, update the HashMap by $\mathsf{HashMap}(k) = i$. In this algorithm, only n RNG operations are performed, and both computation and memory complexities are $O(n \log n)$ in the worst case if the HashMap is optimized by some balanced search tree.

4.3.2 Deterministically mapping SMT (DM-SMT). In contrast to NDM-SMTs, a DM-SMT maps users to leaf nodes deterministically, e.g., the node index mapped to u is determined by $KDF(w_u, salt\_m)$, where $salt\_m$ is a public identifier. Note that since u and $\mathcal{P}$ need to agree on l_u for PoL, $\mathcal{P}$ can in the meanwhile send each user w_u in a similar application-specific registration process before $\mathcal{P}$ generates the SMT, which won't affect the security or privacy guarantees of DAPOL+. In this case, u keeps w_u secret, and $\mathcal{P}$ doesn't need to include b_u or s_u in the inclusion proof for u because they can be deterministically extracted. At the same time, users can figure out the positions of their leaf nodes on their own and $\mathcal{P}$ cannot change the mapping at his/her will. The determinism feature doesn't provide extra security, but minimizes the data base for $\mathcal{P}$ to reproduce the SMT, unlike NDM-SMTs which require $\mathcal{P}$ to store the mapping. In addition, when combined with private information retrieval, DM-SMT may allow users to query inclusion proofs without leaking their identities. This is a desired feature for periodical auditing of PoL such as negative reviews. $\mathcal{P}$ proves the amount of negative reviews he receives every month, e.g., and if it identifies which users seldom query for proofs, there is a good chance to evade detection when discarding the reviews from them. We discuss this in detail in section 4.4.3.

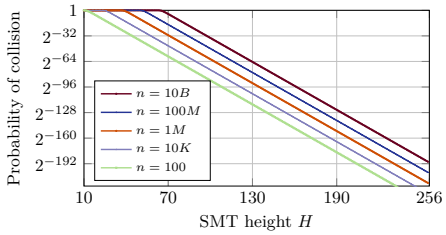


Figure 3: *Probability of collision*

To allow each user to have a deterministic position in the bottom layer of the SMT, the tree height must be large enough to avoid collisions, i.e., two users mapped to the same leaf node. The probability of collision with n users in a H-height tree equals to that of the birthday paradox and is plotted in fig. 3. Each two consecutive curves differ by 13 bits. Assuming there are at most 10B clients, for 128-bit collision resistance we need at most a 193-height SMT. If we tolerate collision probability of 2^{-64}, we need a 130-height SMT.

4.3.3 ORAM-based SMT. Some applications such as proving solvency allow $\mathcal{P}$'s liabilities to users to be updated. If we allow updates to an NDM-SMT directly, privacy leakage might occur because a user observes the update history of his sibling node. Taking the solvency case as an example, a user can link two transactions both changing his sibling node to the same user. Although the values are hidden, it is possible to link transactions and identities in practice. To guarantee complete privacy, $\mathcal{P}$ can run an independent execution of PoL periodically, i.e., generate a new SMT with a new pair of identifiers $salt\_s$ and $salt\_b$. Although this doesn't leak any information, the generation complexity might be high. We propose ORAM-based SMT which combines NDM-SMTs with Oblivious RAM (ORAM) to provide efficient privacy-preserving updates.

ORAM allows users to read and write data on a database without leaking access patterns [37, 38]. A tree-based ORAM [63] is a full binary tree and each block of data is uniformly randomly mapped to a leaf node but may reside in any node along the path from the root to the leaf. Each tree node is a bucket containing B blocks of data, and we call B the bucket size. To read/write a block of data,

the data owner remaps the block to a new random leaf node, reads and rewrites (re-encrypts and shuffles) the whole path the block previously resides on following certain rules. From the server's point of view, each access by a user is just rewriting a few random paths, thus two access sequences are indistinguishable.

In our setting, colluding users trying to infer information from their inclusion proofs can be viewed as the server in ORAM, and $\mathcal{P}$ performing updates acts as a user aiming to conceal access patterns. We adopt tree-based ORAM and modify it to support the functionalities of summation Merkle trees. Basically, we add a hash field h and a commitment field c to each ORAM tree node. The merging function is $h = \mathcal{H}(c_{lch}||c_{rch}||c_{\beta_1}||\cdots||c_{\beta_B}||h_{lch}||h_{rch}||h_{\beta_1}||\cdots||h_{\beta_B})$, $c = c_{lch} \cdot c_{rch} \cdot \prod_{i=1}^{B} c_{\beta_i}$, where β_i is the i-th block in the tree node. $\mathcal{P}$ updates the tree following ORAM rules and proves inclusion of liabilities by a Merkle proof of size $O(B \cdot H)$. In the worst case, even if all users are colluding and sharing all their proofs to reconstruct the tree, they won't be able to distinguish between two sequences of updates. The only privacy leakage in this scheme is the number of updates but $\mathcal{P}$ can always add dummy updates to make constant number of updates periodically. Note that it is $\mathcal{P}$ that accesses the database and performs ORAM operations, just like the institutional prover in Solidus [16]. The complexity overhead on users' side is constant compared with other SMT variants. Tuning the bucket size and the stash size (number of blocks in the root) to optimize PoL proof size remains future work.

Another direction is to make updates privacy preserving and publicly verifiable. Users only need to verify inclusions at the initiation stage. PVORM [16] combines Circuit ORAM [69] with zero-knowledge proofs to achieve this. We can modify it similarly to support summation for PoL. With update proofs, a single auditor/verifier can guarantee the validity of updates and liabilities. However, the proofs need to be committed on a PBB, which is more expensive compared with the distributed auditing manner.

4.3.4 Hierarchical SMTs. For some applications such as official liability reports like COVID-19 cases, the amount of liabilities is tracked by different institutions. We can construct hierarchical SMTs, i.e., an SMT of the i-th level is an accumulator of tree roots of $(i-1)$-th level SMTs. In the COVID-19 tracking case, for example, each hospital may generate a first-level SMT to prove the number of confirmed cases in the hospital. Then a second-level SMT can be generated for the total number in a city/state, and a federal institution may generate a third-level SMT for COVID-19 cases nation wide. The prover's work is therefore distributed and parallelized.

4.4 Additional Features

Although the fundamental security and privacy for PoL have been formally defined in section 3, there remain other interesting properties desirable in certain applications. Since these features are optional and application-specific, we informally discuss them in this section, aiming to inspire future work for particular use cases.

4.4.1 Privacy of distinct PoLs. Privacy of distinct PoLs means for two different $\mathsf{PoL}_1, \mathsf{PoL}_2$ involving provers $\mathcal{P}_1, \mathcal{P}_2$ and user sets $\mathcal{U}_1, \mathcal{U}_2$ respectively, where $\mathcal{P}_1$ and $\mathcal{P}_2$ may or may not be the same entity, and $\mathcal{U}_1$ may intersect with $\mathcal{U}_2$, for an adversary $\mathcal{A}$ corrupting a set of users $V \subseteq \mathcal{U}_1 \bigcup \mathcal{U}_2$, throughout the executions of PoL_1 and PoL_2, $\mathcal{A}$ should not learn anything more than she should, depending on the requirements of particular applications. This indicates independence between two PoLs. In the solvency case, e.g., this property prevents linking two accounts from two banks to the same user. The way DAPOL+ generates leaf nodes and padding nodes as indicated in fig. 2 prevents $\mathcal{A}$ from linking users across PoLs. If PoLs are executed by the same prover, because b_u and s_u are extracted from w_u and public unique PoL identifiers, u only needs to register once and get w_u. Then u can deterministically compute b_u and s_u for all future PoLs and inclusion proofs don't need to contain b_u and s_u. In addition, for rebuilding the SMTs, $\mathcal{P}$ also only needs to store $master\_secret$, DB and possibly the mapping for each PoL depending on the SMT variants being used.

4.4.2 Privacy of sequential PoLs. Privacy of sequential PoLs means for two sequential $\mathsf{PoL}_1, \mathsf{PoL}_2$ involving $\mathcal{P}$ and $\mathcal{U}$, where PoL_2 is subsequent to PoL_1 with updates of $\mathcal{P}$'s liabilities to some users in $\mathcal{U}$, for any adversary $\mathcal{A}$ corrupting users $V \subseteq \mathcal{U}$, throughout the executions of PoL_1 and PoL_2, $\mathcal{A}$ should not learn anything more than they should, depending on the requirements of particular applications. For example, in the solvency case, banks need to process transactions and prove solvency periodically while preserving privacy of trading patterns. This is exactly what ORAM-based SMTs in section 4.3.3 are designed for. However, an inclusion proof needs to contain b_u and s_u due to the re-encryption operations in ORAM, which disallows users from determining these factors on their own.

4.4.3 Privacy of verifier identity. A desirable property for PoL to enhance security is to hide from $\mathcal{P}$ the identities of users that query and verify inclusion proofs. Knowing which users seldom verify, $\mathcal{P}$ can simply discard the accounts to decrease the total liabilities without being detected. Provisions can achieve this property with the trick in DAPOL+ allowing users to deterministically extract blinding factors thus no need to obtain any personal proof from the prover. However, the commitment size on a PBB is large.

Our goal is similar to that of private information retrieval (PIR). PIRs allow users storing their data on a database to retrieve data without revealing which piece of data is accessed [23]. Viewing the inclusion proof for each user as a block of data with a unique index, users try to obtain inclusion proofs with block indexes, and the prover acts as the untrusted server in PIR aiming to learn which block of data is retrieved. This matches with the setting of PIR. Note that the size of the database here should be at least N instead of n because we want to hide n to users. So the interaction between the user and the prover when $n = 1$ should be indistinguishable with that when $n = N$. Additionally, unless a user receives the index of the leaf node he should be mapped to in the SMT at registration, to allow users to deterministically know the correct indexes of leaf nodes for their queries while avoiding collision, we need a DM-SMT of which the tree is tall and the size of the database is 2^H. Although most blocks are dummy blocks and can be produced at accessing so not costing much storage for the prover, the computation complexity for both the prover and users is usually high [50, 59, 62]. A potential direction to solving the practicality problem is to allow each user to be mapped to one of a deterministic set of leaf nodes, and this remains future work. A practical workaround, however, is to use PIR protocols utilizing a trusted hardware [68].

We can also place some trust on a third party auditor. $\mathcal{P}$ can outsource proofs to a semi-honest auditor that doesn't leak verifier identities to $\mathcal{P}$. Users obtain proofs from the auditor directly and individual verifiability is still guaranteed. Note that now the auditor learns nothing except the number of users and verification patterns. To preserve privacy of population against users, we can either have $\mathcal{P}$ encrypt proofs so that each can only be decrypted by its corresponding user, or require users to authenticate their identities to the auditor in a zero-knowledge manner.

4.4.4 Selective privacy preserving claims. Sometimes the prover may need to make zero-knowledge claims that are publicly verifiable, such as the range of the total liabilities without leaking the exact value. For example, in the solvency case, the prover commits to the total liabilities and assets, thus can generate a zero-knowledge proof of solvency, i.e., the amount of total liabilities is no more than the total assets. Multiple entities can also make claims using multi-party computation without leaking sensitive information to anyone. In the case of disapproval voting, for instance, multiple candidates can run the distributed Bulletproofs [11] MPC approach to jointly generate a range proof and determine the best/worst candidate without leaking the final tallies or tally differences.

4.4.5 Random Sampling. In some applications there might be third parties auditing liabilities and PoL can be complementary to that. For example, in the solvency case, auditors may actively sample some users and ask them if their balances are correctly included. In practice, an auditor doesn't have access to the set of users, which makes it hard to sample users for auditing. A rudimentary scheme for the auditor is to randomly sample a node index and ask the prover to return the inclusion proof of that node and the credential information of the corresponding user. With a response, the auditor contacts the user and asks whether the leaf contains correct balance. However, in an SMT, especially in an DM-SMT of large height, a randomly sampled leaf node does not always exist, and the auditor has an overwhelming probability of querying a non-existing node. Moreover, the prover can always pretend that the queried node doesn't exist, since the existence of a leaf node is not verifiable.

To solve this, we propose a concrete scheme of random sampling, the idea of which stems from DAPOL. The auditor randomly samples a leaf index and sends it to $\mathcal{P}$. $\mathcal{P}$ responds with the inclusion proof and user credential data if the queried node exists. To provide user credential data, $\mathcal{P}$ opens the hash of the leaf node and returns the preimage, i.e., id and s, so the auditor can verify the validity of the credential. In addition, we offer *selective disclosure* by generating s and b independently, i.e., disclosing id and s to the auditor doesn't leak b, so the user's balance remains private.

If the queried node doesn't exist, however, the prover returns Merkle paths of the leaf neighbor(s) closest to the queried index, together with a neighboring proof indicating that the returned node(s) are the closest real leaf nodes, which allows the non-existence of a sampled leaf index to be verifiable. The neighboring proof consists of a set of padding nodes at certain positions and the proofs indicating that they are padding nodes. For example, in fig. 2, if the auditor samples Node 2 (a padding node), the two closest leaf nodes are Node 1 and Node 4. The prover should respond with inclusion proofs of Node 1 and Node 4, user IDs (credential information) and the mask of the two nodes, and a neighboring proof indicating that Node 2, 3 and 6 are padding nodes thus no leaf node exists between Nodes 1 and 4. To prove padding nodes, the prover provides the preimage of the node hash so the verifier can check. The number of padding nodes in a neighboring proof is no more than $2H$, so the sizes of a random sampling proof is linear in the tree height.

There could be privacy leakage to the auditor other than the identities of sampled users, i.e., the population. This is because the auditor sees the positions of neighboring leaf nodes and thus can estimate a tighter upper bound on the population.

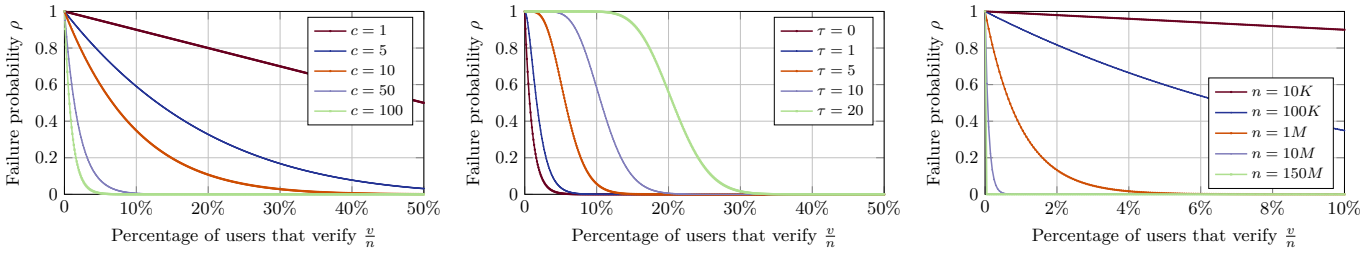
5 FAILURE PROBABILITY

A PoL protocol can only bound the total liabilities to the sum of $\mathcal{P}$'s individual liabilities to users that perform verifications. A prover manipulating or discarding the individual liabilities to users that never check cannot be detected. When a prover misbehaves while undetected, which is undesired, we say PoL fails, and we denote the probability for this to happen the *failure probability*. Failure probability is independent of PoL security and fundamental to distributed verification. It sheds light on how many verifiers are sufficient to jointly prevent the prover from cheating, thus further helps evaluate the effort needed to encourage verifiers in practice.

In this section, we analyze the failure probability of PoL, i.e., the probability that a malicious prover misbehaves and evades detection when a subset of users verify the inclusion proofs. Note that although Provisions also attempted to analyze this concept, it only reached to the result of a special case as our eq. (3). We analyze failure probability more generally for applications not limited to the solvency case. In addition, Provisions claimed that the failure probability is independent of the balances zeroed out in the solvency case, which we show is not true by eq. (6).

Denote by $\rho : \mathbb{N}^* \times \mathbb{N}^* \times \wp(\mathcal{U}) \to [0, 1]$ the failure probability function, where $\wp(\mathcal{U})$ is the power set of $\mathcal{U}$. The function $\rho(v, \tau, C)$ takes as input the number of users that verify v, the tolerance parameter τ and the cheating set $C \in \mathcal{U}$ of size c, i.e., the set of users to whom $\mathcal{P}$ cannot provide a valid proof. In short, C can be viewed as the set of users whose amounts are manipulated by the prover. The function $\rho(v, \tau, C)$ outputs the probability for the prover, when queried by v users, to successfully prove inclusions to at least $v - \tau$ users while manipulating liabilities to users in C. In short, the failure probability depicts how likely a malicious prover can avoid detection of more than τ invalid proofs when cheating on the liabilities to users in C. The tolerance factor $\tau \leq \min(v, c)$ might vary in different applications. In the strictest case, $\tau = 0$, indicating that a single invalid inclusion proof triggers an investigation. In other cases such as tax reporting, the IRS might tolerate $\tau > 0$ incidents before they start an investigation. Note that to cheat on liabilities to c users in DAPOL+, the prover doesn't necessarily need to manipulate the leaf nodes but internal nodes instead, which makes the number of tree nodes manipulated smaller than c. In this case, we still count it as c because we only care how many users are affected, i.e., who would receive an invalid inclusion proof.

Assume each user $u \in \mathcal{U}$ has a probability $p_u = F_u(l_u)$ to check her inclusion proof depending on the prover's liabilities to her. In real-world applications such as the solvency case, users with higher



(a) $n = 1M$, $\tau = 0$ and varying c **(b)** $n = 1M$, $c = 100$ and varying τ **(c)** $\tau = 0$, $c = 0.01\% \cdot n$ and varying n

Figure 4: *Failure probabilities with varying c, τ and n, respectively.*

balances are more likely to verify inclusion proofs. We have

$$\rho(v,\tau,C) = \Pr[|V \cap C| \le \tau \Big| |V| = v] = \frac{\sum\limits_{\substack{|V|=v \wedge \\ |V\cap C|\le\tau}} \prod^{u\in V} p_u}{\sum\limits_{|V|=v} \prod_{u\in V} p_u} \tag{1}$$

If all users follow the same uniform distribution $F_u(l_u) = p$ to verify inclusion proofs, which is the simplest case, we have

$$\rho(v,\tau,C) = \frac{\sum\limits_{\substack{|V|=v \wedge \\ |V\cap C|\le\tau}} \prod^{u\in V} p}{\sum\limits_{|V|=v} \prod_{u\in V} p} = \frac{\sum\limits^{0\le i\le\tau} \binom{c}{i}\cdot\binom{n-c}{v-i}}{\binom{n}{v}} \tag{2}$$

When $\tau = 0$, we can further simplify the formula:

$$\rho(v,0,C) = \frac{\sum\limits_{\substack{|V|=v \wedge \\ |V\cap C|\le\tau}} \prod^{u\in V} p}{\sum\limits_{|V|=v} \prod_{u\in V} p} = \frac{\binom{n-c}{v}}{\binom{n}{v}} \tag{3}$$

Consider a slightly more complicated setting in which users don't have the same distribution but can be categorized into multiple non-intersecting sets and users in each set follow the same uniform distribution to verify inclusion proofs. More formally, suppose all users can be categorized into m non-intersecting sets $V_1, \cdots, V_m$ such that $\forall i \ne j, V_i \cap V_j = \emptyset$ and $\cup_{1\le i\le m} V_i = \mathcal{U}$. For any $u \in V_i$, the user has uniform probability $p_u = F_u(l_u) = \hat{p}_i$ to check the inclusion proof. If the prover manipulates c_i accounts in $C_i \subseteq V_i$, then the probability for v_i users in V_i to encounter at most τ_i verification failures equals to $\rho(v_i, \tau_i, C_i)$ in the simplest same uniform distribution case as in eq. (2). Define $\rho'(\vec{v}, \vec{\tau}, \vec{C})$ as the probability of all subsets of users encounter no more verification failures than tolerable, i.o.w., the prover's misbehavior in all m sets evades detection, where $\vec{v} = (v_1, \cdots, v_m)$, $\vec{\tau} = (\tau_1, \cdots, \tau_m)$, $\vec{C} = (C_1, \cdots, C_m)$. We have

$$\rho'(\vec{v},\vec{\tau},\vec{C}) = \prod_{1\le i\le m} \rho(v_i,\tau_i,C_i) \le \min_{1\le i\le m}(\rho(v_i,\tau_i,C_i)) \tag{4}$$

By definition, the failure probability in the multi-set case is bounded by the subset of users with the lowest risk.

We plot the failure probability for users following the same uniform distribution to verify under various choice of n, c, τ in fig. 4. The smaller the failure probability of PoL is, the lower the risk users take. The risk is much lower as the population grows, thus allowing users to individually verify their contributions to the total liabilities works in a large scale. Given the fact that 150.6 million mobile users accessed the Amazon App in September 2019 [54], assuming $n = 150M$ and $\tau = 0$, only 0.05% users verifying inclusion proofs can guarantee an overwhelming chance of detecting an adversarial prover manipulating 0.01% accounts.

Without the condition that exactly v users verify, denote by $\varrho(\tau, C)$ the failure probability when the prover manipulates accounts in C without being detected by more than τ users. We have

$$\varrho(\tau,C) = \sum_{v=0}^{n} \rho(v,\tau,C)\cdot\Big(\sum_{|V|=v}\Big(\prod_{u\in V} p_u \cdot \prod_{u\in U-V}(1-p_u)\Big)\Big) \tag{5}$$

When $\tau = 0$, we have

$$\varrho(0,C) = \prod_{u\in C}(1-p_u) \tag{6}$$

Making the same assumption as in Provisions that users with more deposits are more likely to verify in the solvency case, eq. (6) implies a negative correlation between the failure probability and $\mathcal{P}$'s liabilities to users in C. In other words, the malicious prover is more likely to be caught when zeroing out the largest accounts than the smallest, which contradicts the claim in Provisions.

6 EVALUATION

We implemented [18] in Rust, a PoC DAPOL+ with NDM-SMT, the simplest accumulator, and benchmark the performance. The theoretical complexities of existing schemes are compared in table 1. Note that DAPOL+ is the first PoL protocol that satisfies security and privacy as defined. With this functionality advantage, our purpose of evaluation is not to compare DAPOL+ with other schemes but to provide concrete numbers to demonstrate its practicality.

DAPOL+ can be viewed as working in two phases: 1. generating the SMT; 2. responding to users' queries. We evaluate the performance of them separately. All experiments are run with a single thread on a recent model Apple M1 with memory size 16GiB.

6.1 SMT Generation (Setup)

We first evaluate the generation time of NDM-SMT used for DAPOL+. For Pedersen commitments, we adopt the Ristretto255 group on top of Curve25519 [51] as it is the curve used in the Bulletproofs library [65]. We use Blake3 [9], which is fast, secure and highly parallelizable, as the hash function.

We plot in fig. 5 the generation time vs. the number of nodes in the tree with different settings of n and $H \ge 32$ assuming $N = 4B$ which is approximately half of the world population [75] and should

suffice in most applications. The label to each point indicates (n, H). We can see that the complexity of tree generation is linear in the number of nodes in the SMT which varies by the distribution of leaf nodes. In the best case, when users take up a consecutive range of leaves and are positioned next to each other, the total number of nodes is roughly $2n + 2H$. In the worst case, when users take up the whole range of leaves and each consecutive pair has the same largest distance, there are approximately $2n \cdot (1 + H - \log n)$ nodes.

When $n = 1M$ and $H = 50$, it takes about half an hour to generate the whole tree. Considering the frequency of generating SMTs, which is per day for COVID cases, and per year for solvency and tax reporting, this is efficient enough. Meanwhile, the generation process is highly parallelizable by generating subtrees concurrently.

6.2 Inclusion proofs

Table 3: *Aggregated range proofs performance.*

#Ranges aggregated	1	32	64	128	256
Proving time (s)	0.008	0.277	0.506	1.009	2.005
Verification time (s)	0.002	0.048	0.096	0.191	0.381
Proof size (B)	672	992	1056	1120	1184

We integrated Bulletproofs [11] to the NDM-SMT so the basic DAPOL+ functionality is complete. Regarding range proofs, we prove all values are within range $[0, 2^{64} - 1]$, which is sufficient for most real-world applications. The Bulletproofs library [65] doesn't support aggregated proofs when the number of ranges is not a power of two. To address this, we can apply padding or splitting. By padding, we deterministically add $2^{\lceil \log H \rceil} - H$ dummy ranges to generate a proof aggregating $2^{\lceil \log H \rceil}$ ranges. By splitting, we split $H = \sum_{i=0}^{k} c_i \times 2^i$ ranges to at most $k = \log H$ sets of ranges, the size of each is of a power of two.

We show DAPOL+ is practical via an example of generating a 32-height SMT for 1M users. The two mechanisms for generating aggregated range proofs, i.e., padding and splitting, are equivalent in this case. We present the range proof size, generation and verification time in table 3. The time for retrieving the Merkle path and verifying Merkle path is negligible compared to that of range proofs, not up to 1ms. The time for generating one inclusion proof (aggregating 32 range proofs) is 0.277s and the verification time is 0.048s. In the proof, each node has a hash of 32 bytes and a commitment of 32 bytes, and the size of the aggregated range proof is 992 bytes, so the size of an inclusion proof is $992B + 32 * (32 + 32) = 3040B = 3KiB$.

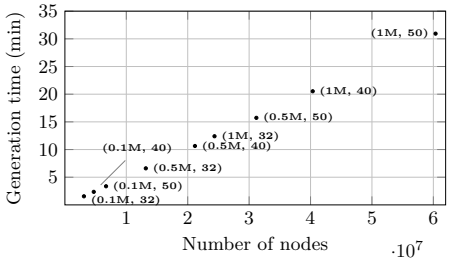


Figure 5: *Generation time vs. number of nodes.*

There is a tradeoff between proving time and proof size because the size of the aggregated range proof grows logarithmically with the number of ranges but the proof generation time grows linearly. The generation time for an inclusion proof in a 32-height SMT is 0.277s when using aggregated Bulletproofs. Given 1M clients, the total time for generating all inclusion proofs is $0.277s \times 10^6 \approx 77h$ without parallelization. Note that the nodes in upper layers of the SMT are involved in multiple aggregated range proofs for different clients. One optimization option is to sacrifice proof size for faster proof generation. We can generate a range proof for every node in the SMT without aggregation and provide all non-aggregated range proofs on the proof path to clients. Then there is no duplication of computing range proofs for the same tree node. In this way, the proof size is $23.6KB$ which is still tolerable for a user receiving proofs via a mobile device. The total inclusion proof generation time varies by the number of nodes thus from 4.5h to 57h without any parallelization. Clients can verify the range proofs in batch though. Another optimization option is to aggregate range proofs for nodes in upper layers of the SMT and generate non-aggregated range proofs for lower-layer nodes. The previous two mechanisms are like two extremes, one optimized in proof size while the other optimized in proof generation time. We can tune the number of tree nodes aggregated to achieve a satisfactory balance between proof size and computation complexity. Anyway, the prover can deploy multiple servers to generate range proofs and respond to users simultaneously and the response latency is absolutely tolerable.

7 CONCLUSION

Applications of PoL share a common nature: the prover has no incentive to exaggerate the total liabilities; individuals, in contrast, have an incentive to make sure their values are included in the reported total liabilities. We have formalized PoL as a general cryptographic primitive utilizing the incentive feature, and standardized PoL from three aspects: security, privacy and efficiency. We presented DAPOL+, a PoL protocol based on DAPOL but providing provable security and privacy, and demonstrated its practicality by evaluation results. We informally discussed other interesting properties optional in different real-world applications and their potential solutions. Formal treatment of the additional features for a particular application remains future work. Although PoL cryptographically bounds the reported total liabilities to some extent, it doesn't prevent a malicious prover from discarding the liabilities to users that never verify proofs while remaining undetected. We analyzed failure probability to understand the effectiveness of distributed verification. Note that failure probability is independent of particular PoL schemes but fundamental to the distributed nature of PoL and other interesting problems as well, such as voting where voters need to verify their own individual ballots are correctly tallied. It's important to carefully evaluate the effort needed to incentivize sufficient verifiers for jointly preventing the prover from cheating when applying PoL in practice.

8 ACKNOWLEDGEMENTS

We would like to thank Harjasleen Malvai, Irakliy Khaburzaniya, Dahlia Malkhi and anonymous reviewers for their constructive feedback on definitions and implementation considerations.

REFERENCES

[1] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the 2017 acm sigsac conference on computer and communications security*. 2087–2104.

[2] Niko Barić and Birgit Pfitzmann. 1997. Collision-free accumulators and fail-stop signature schemes without trees. In *Eurocrypt*.

[3] Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and J-F Raymond. 2002. Breaking the O (n/sup 1/(2k-1)/) barrier for information-theoretic Private Information Retrieval. In *IEEE FOCS*.

[4] Michael Ben-Or, Oded Goldreich, Silvio Micali, and Ronald L Rivest. 1990. A fair protocol for signing contracts. *IEEE Transactions on Information Theory* 36, 1 (1990), 40–46.

[5] Josh Benaloh and Michael De Mare. 1993. One-way accumulators: A decentralized alternative to digital signatures. In *Eurocrypt*.

[6] Josh Benaloh and Eric Lazarus. 2011. *The trash attack: An attack on verifiable voting systems and a simple mitigation.* Technical Report. MSR-TR-2011-115, Microsoft.

[7] Bitfury. 2013. On Blockchain Auditability. White Paper.

[8] Bitfury. 2016. On Blockchain Auditability. https://bitfury.com/content/downloads/bitfury_white_paper_on_blockchain_auditability.pdf.

[9] Blake3-team. 2021. Blake3. https://github.com/BLAKE3-team/BLAKE3/.

[10] Sean Bowe, Jack Grigg, and Daira Hopwood. 2019. *Recursive proof composition without a trusted setup.* Technical Report. Cryptology ePrint Archive, Report 2019/1021, 2019. https://eprint. iacr. org ….

[11] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short proofs for confidential transactions and more. In *S&P*.

[12] Vitalik Buterin. 2021. Why sharding is great: demystifying the technical properties. https://vitalik.ca/general/2021/04/07/sharding.html.

[13] Philippe Camacho. 2014. Secure Protocols for Provable Security. https://www.slideshare.net/philippecamacho/protocols-for-provable-solvency-38501620.

[14] Philippe Camacho, Alejandro Hevia, Marcos Kiwi, and Roberto Opazo. 2008. Strong accumulators from collision-resistant hashing. In *ISC*.

[15] Jan Camenisch and Anna Lysyanskaya. 2002. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Crypto*.

[16] Ethan Cecchetti, Fan Zhang, Yan Ji, Ahmed Kosba, Ari Juels, and Elaine Shi. 2017. Solidus: Confidential distributed ledger transactions via PVORM. In *ACM CCS*.

[17] Konstantinos Chalkias, François Garillot, Yashvanth Kondi, and Valeria Nikolaenko. [n. d.]. Non-interactive half-aggregation of EdDSA and variants of Schnorr signatures. ([n. d.]).

[18] Konstantinos Chalkias, Yan Ji, and Irakliy Khaburzaniya. 2021. Rust DAPOL+ library. https://github.com/novifinancial/solvency.

[19] Konstantinos Chalkias, Kevin Lewi, Payman Mohassel, and Valeria Nikolaenko. 2019. Practical Privacy Preserving Proofs of Solvency. Amsterdam ZKProof Community Event.

[20] Konstantinos Chalkias, Kevin Lewi, Payman Mohassel, and Valeria Nikolaenko. 2020. Distributed Auditing Proofs of Liabilities. 3rd ZKProof Workshop. *ZKProof* (2020).

[21] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. 2019. SEEMless: Secure End-to-End Encrypted Messaging with less Trust. In *ACM CCS*.

[22] Panagiotis Chatzigiannis, Foteini Baldimtsi, and Konstantinos Chalkias. [n. d.]. SoK: Auditability and Accountability in Distributed Payment Systems. ([n. d.]).

[23] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private information retrieval. In *IEEE FOCS*.

[24] CoinMarketCap. 2021. Today's Cryptocurrency Prices by Market Cap. https://coinmarketcap.com/.

[25] Gaby G Dagher, Benedikt Bünz, Joseph Bonneau, Jeremy Clark, and Dan Boneh. 2015. Provisions: Privacy-preserving proofs of solvency for bitcoin exchanges. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 720–731.

[26] Etherscan. 2021. Ethereum Gas Tracker. https://etherscan.io/gastracker.

[27] FATF. 2020. 12-Month Review of the Revised FATF Standards on Virtual Assets and Virtual Asset Service Providers.

[28] Centers for Disease Control and Prevention. 2020. Cases and Deaths in the U.S. https://www.cdc.gov/coronavirus/2019-ncov/cases-updates/us-cases-deaths.html.

[29] Reid Forgrave. 2018. The Man Who Cracked the Lottery. The New York Times.

[30] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. 2018. The algebraic group model and its applications. In *Annual International Cryptology Conference*. Springer, 33–62.

[31] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. 2019. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. *IACR Cryptol. ePrint Arch.* 2019 (2019), 953.

[32] Chaya Ganesh. 2017. *Zero-knowledge Proofs: Efficient Techniques for Combination Statements and their Applications.* Ph. D. Dissertation. New York University.

[33] Christina Garman, Matthew Green, and Ian Miers. 2014. Decentralized Anonymous Credentials.. In *NDSS*. Citeseer.

[34] Peter Gaži, Aggelos Kiayias, and Alexander Russell. 2018. Stake-bleeding attacks on proof-of-stake blockchains. In *Crypto Valley Conference on Blockchain Technology*.

[35] Ashrujit Ghoshal and Stefano Tessaro. 2020. *Tight State-Restoration Soundness in the Algebraic Group Model.* Technical Report. Cryptology ePrint Archive, Report 2020/1351.

[36] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News* 33, 2 (2002), 51–59.

[37] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *ACM STOC*.

[38] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[39] Seena Gressin. 2017. The Equifax Data Breach: What to Do. Federal Trade Commission.

[40] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 305–326.

[41] James Heather and David Lundin. 2008. The append-only web bulletin board. In *International Workshop on Formal Aspects in Security and Trust*. Springer, 242–256.

[42] Chabeli Herrera. 2016. Largest lawsuit against an auditor goes to court for $5.5 billion. https://www.miamiherald.com/news/business/banking/article92700782.html.

[43] Kexin Hu, Zhenfeng Zhang, and Kaiwen Guo. 2019. Breaking the binding: Attacks on the Merkle approach to prove liabilities and its applications. *Computers & Security* (2019).

[44] Julia Kagan. 2020. Bank Failure. https://www.investopedia.com/terms/b/bankfailure.asp.

[45] Karen Kitching. 2009. Audit value and charitable organizations. *Journal of Accounting and Public Policy* (2009).

[46] Hugo Krawczyk. 2010. Cryptographic extraction and key derivation: The HKDF scheme. In *Crypto*.

[47] Ralf Kusters, Tomasz Truderung, and Andreas Vogt. 2012. Clash attacks on the verifiability of e-voting systems. In *IEEE S&P*.

[48] Andrew Lewis-Pye and Tim Roughgarden. 2020. Resource pools and the cap theorem. *arXiv preprint arXiv:2006.10698* (2020).

[49] Helger Lipmaa. 2005. An oblivious transfer protocol with log-squared communication. In *ISC*.

[50] Helger Lipmaa. 2009. First CPIR protocol with data-dependent computation. In *ICISC*.

[51] Isis Agora Lovecruft and Henry de Valence. [n. d.]. curve25519-dalek. https://github.com/dalek-cryptography/curve25519-dalek.

[52] Robert McMillan. 2014. The Inside Story of Mt. Gox, Bitcoin's $460 Million Disaster. https://www.wired.com/2014/03/bitcoin-exchange/.

[53] Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *Crypto*.

[54] Maryam Mohsin. 2020. 10 AMAZON STATISTICS YOU NEED TO KNOW IN 2020 [INFOGRAPHIC]. https://www.oberlo.com/blog/amazon-statistics.

[55] Tyler Moore and Nicolas Christin. 2013. Beware the middleman: Empirical analysis of Bitcoin-exchange risk. In *FC*.

[56] Chamber of Digital Commerce. 2021. Proof of Reserves – Establishing Best Practices to Build Trust in the Digital Assets Industry. https://4actl02jlq5u2o7ouq1ymaad-wpengine.netdna-ssl.com/wp-content/uploads/2021/05/Proof-of-Reserves-.pdf.

[57] U.S. Bureau of Labor Statistics. 2015. How the Government Measures Unemployment. https://www.bls.gov/cps/cps_htgm.htm.

[58] U.S. Bureau of Labor Statistics. 2017. Counting injuries and illnesses in the workplace: an international review. https://www.bls.gov/opub/mlr/2017/article/counting-injuries-and-illnesses-in-the-workplace.htm.

[59] Femi Olumofin and Ian Goldberg. 2011. Revisiting the computational practicality of private information retrieval. In *International Conference on Financial Cryptography and Data Security*. Springer, 158–172.

[60] Torben Pryds Pedersen. 1991. Non-interactive and information-theoretic secure verifiable secret sharing. In *Crypto*.

[61] Josh Ryan-Collins, Tony Greenham, Richard Werner, and Andrew Jackson. 2012. Where does money come from. *London: New Economics Foundation. Pg* 7 (2012).

[62] Radu Sion and Bogdan Carbunar. 2007. On the computational practicality of private information retrieval. In *Proceedings of the Network and Distributed Systems Security Symposium*. Internet Society, 2006–06.

[63] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM CCS*.

[64] USA.gov. 2020. Credit Reports and Scores. https://www.usa.gov/credit-reports.

[65] Henry de Valence, Cathie Yun, and Oleg Andreev. 2020. Rust Bulletproofs library. https://github.com/dalek-cryptography/bulletproofs.

[66] Karyn R Vanderwarren. 2001. Financial accountability in charitable organizations: Mandating an audit committee function. *Chi.-Kent L. Rev.* 77 (2001), 963.
[67] Surya Viswanathan and Aakash Shah. 2018. The Scalability Trilemma in Blockchain. https://aakash-111.medium.com/the-scalability-trilemma-in-blockchain-75fb57f646df.
[68] Shuhong Wang, Xuhua Ding, Robert H Deng, and Feng Bao. 2006. Private information retrieval using trusted hardware. In *Esorics*.
[69] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *ACM CCS*.
[70] Wikipedia. 2019. 2004 Chinese lottery scandal. https://en.wikipedia.org/wiki/2004_Chinese_lottery_scandal.
[71] Wikipedia. 2020. Hot Lotto fraud scandal. https://en.wikipedia.org/wiki/Hot_Lotto_fraud_scandal.
[72] Wikipedia. 2021. Enron scandal. https://en.wikipedia.org/wiki/Enron_scandal.
[73] Zak Wilcox. 2014. Proving your bitcoin reserves. https://bitcointalk.org/index.php?topic=595180.0.
[74] Gavin Wood. 2014. Ethereum yellow paper. (2014).
[75] Worldmeters. 2020. World Population Clock: 7.8 Billion People (2020). worldometers.info.
[76] Qing Zhang, Thomas Ristenpart, Stefan Savage, and Geoffrey M Voelker. 2011. Got traffic? An evaluation of click traffic providers. In *Proceedings of the 2011 Joint WICOW/AIRWeb Workshop on Web Quality*. 19–26.

A APPLICATIONS

Table 4: *Potential applications requiring PoL.*

DOMAIN	APPLICATIONS
finance	solvency, fundraising, credit-score, sales tax reports, syndicated loans
voting	disapproval voting, negative reviews/ratings
official reports	unemployment rate, work accidents, virus outbreak reports (COVID-19 daily cases)
misc	lottery pots, referral schemes

We summarize applications of PoL from [20] with a few revisions and classify them by domains in table 4.

Proof of Solvency. Proof of solvency [7, 13, 19, 25, 32, 73] is used to prove that a custodial service possesses sufficient assets to settle all clients' accounts. Although Bitcoin is gaining population in using decentralized digital currencies, a large proportion of applications and tradings still happen in centralized exchanges. As clients deposit their assets in exchanges, there is a risk of losing money due to bankruptcy, theft or technical mistakes of exchanges. Clients have lost billions of dollars worth of bitcoins in exchanges [55]. It is important for clients to be aware of the well functioning of exchanges and that their deposits are not lost.

In particular, proof of solvency aims to demonstrate $liabilities \leq reserves$. It consists of two components, i.e., proof of liabilities and proof of assets. Proof of liabilities prevents an exchange from cheating about the total balances of its clients, thus further lower bounding the reported amount of assets when proving solvency. PoL can be implemented independently from the underlying blockchain protocol which the corresponding proof-of-assets mechanism depends on, thus it applies to any blockchain or even traditional banking systems.

As noted in Provisions [25], proof of solvency doesn't prevent bankruptcy or money theft, but is important for clients to estimate the status of a financial institution and their own risks. It also has a potential in mitigating insolvency caused by panic.

Fundraising. Fundraising is the process of aggregating financial contributions from voluntary individuals or entities for a common project or goal. Charitable donation is an example of fundraising. It is critical for the charity to prove that all donations are included in their total reported amount [66] but there is no easy way. Currently charities select auditors by themselves for auditing. A study shows that the willingness of donors are affected by both a charity's reputation and the quality of auditor the charity chooses [45].

PoL enables decentralized auditing without trusting an auditor, which in many donation campaigns doesn't exist anyway. It helps increase transparency while preserving privacy in fundraising activities, thus further enhance trust and participation. The main difference between solvency and donations is that some of the privacy features might be optional in fundraising. For instance, disclosing the total amount raised or how many donors contributed might be considered a feature rather than a privacy leakage.

Credit score and financial obligations. Credit score [64] is a measurement of a person's credit risk, i.e., having a higher credit score indicates that you are more creditworthy, and thus it is easier for you to get a loan or a lower insurance rate, etc. To compute an individual's credit score, we need all relavent credit reports covering financial history records including loans and debts.

Usually centralized credit bureaus collect credit reports and use them to compute credit scores and report to creditors. A hacker successfully accessing the database of a credit bureau can lead to severe consequences. For instance, data breach [39] of a major bureau in 2017 resulted in leakage of hundreds of millions of customers' personal data.

PoL can be used to allow people to maintain credit scores on their own and report to creditors directly without third party intermediaries. This provides better privacy protection of personal data. In addition, the PoL commitment of a credit score can also be combined with other cryptographic primitives, such as MPC, for comparison (i.e., if it meets a threshold) without revealing the actual value.

Sales tax reporting. Enterprises have to report revenues at regular intervals for taxing. PoL can be complementary to the current auditing system and may mitigate tax evasion by allowing customers to automatically or voluntarily contribute on verifying tax liabilities proof for every purchase.

For that to work, each PoL inclusion proof should be considered a decentralized invoice. Briefly, each signed receipt is an entry in merchant's liabilities data set. Then buyers could have an app to automatically and privately check the inclusion of their receipts at the end of the tax year for every purchase they had; and report it to the tax authorities in case of a mismatch.

Among the others, a high participation from citizens would eventually lead to less resources required by the tax authorities to audit companies and as a result IRS for example would no longer need to issue centralized invoices for taxation.

Obviously, motivations to encourage decentralized participation from citizens is an interesting topic of research; some ideas include tax discounts or impacting credit scores for those who contribute.

Moreover, there are cases where dual incentives for applying PoL aligning with each other. One example is charity fundraising where apart from the charity's PoL, tax payers can automatically claim charitable tax deduction on donations they make by using their inclusion proof as proof of expense.

Syndicated loans. Syndicated loans are a form of loan offered by a group of lenders to a large borrower, i.e., a corporation, a project, or a government. The large loan is jointly contributed by lenders in the syndicate and the lending risk is shared among them. Usually there is a lead lender arranging all other lenders and administering the loan. However, this doesn't provide any privacy for lenders. The total loan amount and the amount lent by each lender is at least visible to the arranger. In addition, a malicious arranger might fake the total loan or report inconsistent contribution of each individual lender.

PoL can be applied to this case and solves the problems above. The borrower acts as the prover and has no incentive to exaggerate his total liabilities to lenders. And each lender can check his contribution independently by requesting the corresponding inclusion proof. The contribution amounts can be concealed, and no arranger is needed for managing this.

Disapproval voting. Verifiability, i.e., the ability of voters to check all the cast ballots are correctly tallied in the voting result, is a critical property in e-voting systems. Yet there is no easy way to achieve it and there are known attacks against existing e-voting systems on verifiability, i.e., trash attacks [6] and clash attacks [47].

A disapproval vote is a vote against one or multiple candidates. If the voting system manager is a stakeholder in some of the candidates, e.g., being or owning the candidates, thus has no incentive to increase the amount of negative votes, PoL can be applied for verifiability. In particular, PoL allows the manager to prove inclusion of disapproval votes in the voting result in a distributed manner as all other applications mentioned above. A voter can find out if the manager maliciously discards his disapproval vote. In some cases, the candidates can run the poll for themselves instead of requiring a third party to run the voting system. And each candidate just needs to provide PoL proofs to voters to prove that no disapproval vote is discarded in the final count. The candidates may run an MPC protocol to output an ordering of the amounts of dispproval they receive without revealing the exact values.

Note that voting systems are complex and there are many crucial issues that need to be resolved other than verifiability, e.g., coercion, bribery and sybil attacks. PoL guarantees verifiability under certain incentive assumptions but need to be combined with other cryptographic tools, such as anonymous credentials, for building up a complete solution satisfying other requirements of a voting system.

Negative reviews/ratings. Similarly to disapproval voting, it is critical to guarantee verifiability in negative reviews including reporting illegal behaviors e.g., hate speech, violence promotion, fake news, or negative reviews and complaints on rating platforms for commodities, restaurants, services, etc. PoL can be applied to allow reviewers to check if their negative reviews are properly included in the rating systems without a centralized auditor.

Again, PoL relies on the assumption that there is no incentive for the prover to increase the amount of negative reviews, so it is required that the prover shares similar interests as the reviewed identity. The reviewee or the owner of the reviewee can generate PoL proofs on his own without involving a third party, since there is no incentive for him to maliciously increase the negative rating. PoL doesn't prevent malicious reviewers from rating maliciously negatively and repetitively against some item either, so there is need in preventing this and sybil attacks with additional mechanisms.

Moreover, PoL cannot be applied to rating systems that report average scores because a malicious prover can insert high ratings to increase the average as much as desired.

Official liability reports. During pandemics there is always an urgent need to monitor and track the numbers of infections and deaths because they help epidemiologists learn more about the disease, stop the spread and find a cure. For instance, authorities globally collect COVID-19 data and publish daily reports [28].

Similar examples of official liability reports include the number of work accidents in companies [58] and unemployment cases [57]. The common characteristic of the applications above is that the count represents liabilities and obligations, thus there is no incentive for the counter to overstate the number.

Data accuracy and timeliness are particularly important for public good in these applications. PoL can be applied to provide transparent verifiability while preserving confidentiality of people's personal information. Interestingly, authorities of different districts can also compare the count without leaking their actual values via MPC. In addition, the hierarchical DAPOL+ could be applied here particularly to enable hierarchical proofs, where institutions report to local authorities and in turn local authorities report to the state.

Lottery prizes. Lotteries are a form of gambling and are strictly regulated even in districts where they are not outlawed by the government. Usually there is an auditor making sure fairness and preventing frauds in a lottery. However, there have been reports on lottery frauds all the time [29, 70, 71].

PoL can be applied to lottery audit for ensuring the consistency between the reported total pot and participants' contributions without involving a third party auditor. The lottery organizer is the prover and since the pot depends on the total bet, he has not incentive to increase the amount. Players can check whether their contributions are included individually while actual values can be concealed from the public.

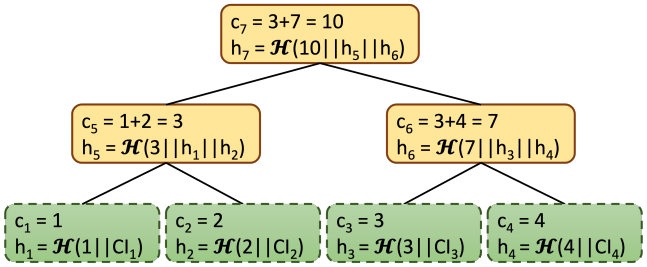
Referral schemes. Referral websites (referrers) are intermediate domains that contain links directing visitors to other sites (referees). Referrals increase the traffic to the referred websites, and can be beneficial because they may invoke business activities by visitors, e.g., purchasing products or depositing money.

Referred websites may need to pay referral websites for referrals based on visitors' activities. Many websites though pay referrals by sign ups (not visits) or amount deposited by the final client (i.e., gambling websites). Thus, it is hard to verify whether the referee intentionally reports a smaller amount of liabilities.

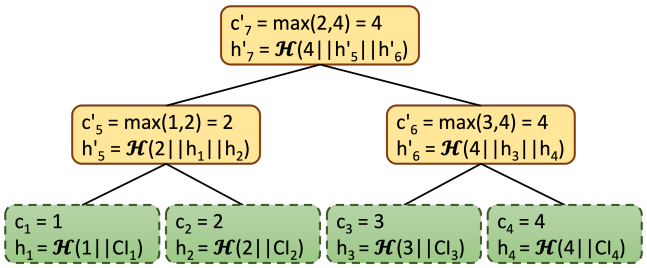
PoL could be a complementary tool for solving the above issue in a distributed way. Briefly, the referee generates a PoL proof for all visiting activities. If visitors are incentivized to verify the inclusion or if there is an automatic scheme, e.g., by the browser to request PoL proofs, the referee won't be able to claim lower traffic/action numbers. Note that PoL on itself doesn't solve the whole problem, however. The issue of how to distinguish between human behaviors and automatic bots when measuring traffic is also concerning and challenging [76].

B VULNERABILITIES IN EXISTING SCHEMES

B.1 Maxwell-Todd



(a) The prover is honest.



(b) The prover is malicious.

Figure 6: *An example of the vulnerability in the Maxwell-Todd scheme with 4 users.*

Maxwell-Todd breaks non-equivocation [43]. Because there's no binding between the hash in a node and the values in its child nodes, provers can cheat even when every user verifies their inclusion proofs.

An example of the vulnerability is shown in fig. 6. When the prover is honest, the sum in the root node should be 10, as demonstrated in fig. 6a. When the prover is adversarial, however, the sum in the root node could be maliciously committed to 4, as shown in fig. 6b. Note that when a user asks for the inclusion proof, the malicious prover is able to serve a proof that passes the verification successfully. For example, for user u_3, the prover generates a Merkle path containing two nodes, $Node_4 = (1, h_4)$ and $Node_5 = (0, h'_5)$. Then u_3 computes $Node_6 = (3 + 1 = 4, h'_6 = \mathcal{H}(4||h_3||h_4))$ and $Node_7 = (0 + 4 = 4, h'_7 = \mathcal{H}(4||h'_5||h'_6))$, which matches the published root node.

B.2 DAPOL

DAPOL may leak information about the number of users because padding nodes are distinguishable from nodes of other types. A semi-honest user can infer which nodes in the Merkle proof are padding nodes by checking if the node hash equals to $\mathcal{H}(\text{“}pad\text{”}||idx)$. As discussed in section 2, assuming an inclusion proof consists of padding nodes at heights $\{x_1, \cdots, x_m\}$ (root at height 0), the user can bound the number of users within the range $[H - m + 1, 2^H - \sum_{i=1}^{m} 2^{H-x_i}]$.

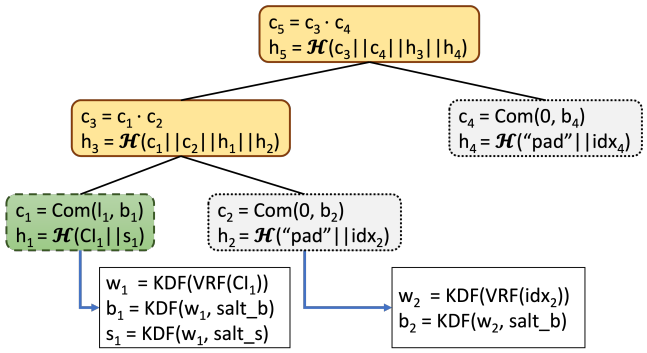


Figure 7: *An example of the DAPOL SMT with 1 single user.*

We demonstrate the leakage by an example. Suppose there is only 1 user as depicted in fig. 7. User u_1 receives an inclusion proof consisting of two padding node at heights $\{1, 2\}$. Thus u_1 can infer that he/she is the single user the prover is liable to.

C TOOLS FOR PROOF OF LIABILITIES

We briefly introduce tools that are potential building blocks when implementing PoL schemes in this section.

C.1 Public Bulletin Board

A bulletin board is a public append-only memory to which anyone has access. People interacting with a PBB are guaranteed to have consistent views of the contents on it. A secure PoL protocol needs a PBB to enable users to have the same view of the prover's claims about the total liabilities and to make sure if their own individual amounts are included. Without a PBB, a malicious prover can always show different total liabilities to different users, excluding some other users' amounts without being detected. To implement a PBB, we either have to place all trust on a single entity for being honest and available all the time, e.g., a trusted hardware which doesn't need to be trusted for confidentiality but only integrity, or use a decentralized scheme [33, 41], such as a blockchain which is believed to be the most practical and promising implementation of a PBB. Note that a forkless blockchain is preferable, otherwise the blockchain need to be resistant to long range attacks [34] to function as a PBB. Due to its low throughput of transactions, committing data on a blockchain is expensive. Therefore a minimized size of commitments on the PBB is desired in a practical PoL protocol.

C.2 Accumulators

Cryptographic accumulators [5] are one-way functions allowing space/time efficient membership and/or non-membership proofs, i.e., whether an element is a member of a set can be proved efficiently. To enable users to check if their amounts are included in the prover's total liabilities while minimizing the public commitment size on a PBB, we can use accumulators for PoL protocols. RSA accumulators [2, 15] are a class of accumulators but don't support summation, thus cannot be used for PoL to prove each user's contribution to the total liabilities. Besides, RSA accumulators are index-free, i.e., the members of a set are not ordered.

In section 4.4.5, we discuss an additional property of PoL protocols - random sampling, which requires indexes of set elements, and RSA accumulators cannot provide this property. In contrast, Merkle trees [14, 53] are a class of tree-based accumulators that can support summation and membership proofs in an ordered set. Therefore, Merkle tree is an ideal accumulator for PoL.

Summation Merkle Trees. Summation Merkle trees are a type of Merkle trees supporting summation of elements in a set, i.e., not only proving the inclusion of an element as a leaf node in the tree, but also that the value of the leaf node is summed in the root node. It was first proposed in [73] followed with a security patch [43], by adding a value field c to each tree node along with the original hash field h. The merging function of each node from its two child nodes lch and rch is $c = c_{lch} + c_{rch}, h = \mathcal{H}(c_{lch}||c_{rch}||h_{lch}||h_{rch})$, where $\mathcal{H}(\cdot)$ is a collision-free hash function. The value field of the root node should equal to the sum of that of all leaf nodes.

Sparse Merkle Trees. Sparse Merkle trees (SMT) can be used to conceal the population while neither a Patricia Merkle tree used in SEEMless [21] nor a full Merkle tree is suitable. In a Patricia Merkle tree of height H with the root at height 0, each user is mapped to a node indexed at height H, but might not reside in that mapped node. Instead, each user moves up along its path to the root as long as the subtree rooted at the current node contains no other users' mapped nodes. Eventually users stop moving and reside in the highest nodes that satisfy the condition above. Although the tree is compact, a PoL based on Patricia Merkle trees leaks the number of users. For example, a user residing in a leaf node at height i can derive partial knowledge $n \leq 2^H - 2^{H-i} + 1$. In the worst case, all users colluding together can have a good estimate of n. A full Merkle tree having all users reside in leaf nodes can conceal n as long as the tree height is large enough, i.e., $H \geq \lceil \log N \rceil$, N being the maximum potential population. However, this is inefficient because there are $2^{H+1} - 1$ nodes in total thus the tree generation complexity is $O(2^H)$, independent of n. Sparse Merkle trees, instead, provide an efficient and practical solution to privacy of population. In an SMT, users are mapped to and reside in nodes at height H. Instead of constructing a full binary tree, only tree nodes that are necessary for Merkle proofs exist, thus the generation complexity is $O(n \cdot H)$. We present this construction in detail in section 4.

C.3 Pedersen Commitments

Pedersen commitments [60] can be used to conceal the prover's liabilities to individual users in PoL. Given a cyclic group G of order q, two public random generators of the group g_1, g_2 whose relative discrete logarithm is unknown to everyone, a Pedersen commitment to a value $l \in \mathbb{Z}_q$ is $\mathsf{Com}(l, b) = g_1^l \cdot g_2^b$, where b, the *blinding factor*, is a randomly selected element in $\mathbb{Z}_q$. Pedersen commitments are *perfectly hiding* (i.e., an unbounded adversary cannot learn anything about a value from its commitment) and *computationally binding* (i.e., a bounded adversary cannot open a commitment for two different values). Additionally, Pedersen commitments are *addition-homomorphic*, i.e., a commitment to the sum of two values l_1, l_2 can be derived from the commitments to them directly: $\mathsf{Com}(l_1, b_1) \cdot \mathsf{Com}(l_2, b_2) = g_1^{l_1+l_2} \cdot g_2^{b_1+b_2} = \mathsf{Com}(l_1 + l_2, b_1 + b_2)$. Therefore, users can audit the total liabilities in Pedersen commitments instead of the plain values.

C.4 Zero-Knowledge Range Proofs

Zero-knowledge range proofs (ZKRFs) allow a prover to prove to a verifier that a number x is within a certain range $[a, b]$ and the verifier is not able to learn x from the proof. ZKRFs are utilized for two purposes in DAPOL+. First, for the security of PoL, the prover proves that each value committed is within a certain range to prevent overflow when summing them up, e.g., if there are at most N users, and in the worst case all users can have the same maximum amount, the prover needs to prove that for each user's committed value v, it holds that $v \in [0, p/N]$, where p is the group order. Second, depending on the particular application, a prover may prove zero-knowledge claims regarding the total liabilities with its commitment. For example, in the solvency case, the prover can prove solvency by showing the total liabilities is no more than the total assets he/she owns with a zero-knowledge range proof. Generic zero-knowledge proofs constructions (e.g., Gro16 [40], Halo [10], Ligero [1], Plonk [31]) offer ZKRFs but we choose Bulletproofs [11] for DAPOL+ due to its aggregation property and no need of a trusted setup.

C.5 Key Derivation Functions

Key derivation functions (KDF) are used to derive one or more secret keys from a low-entropy secret value such as a password or stretch keys to a desired length [46]. In PoL, this allows the prover and the users to generate blinding factors in Pedersen commitments, user masks and mapped tree node indexes in an SMT deterministically from users' unique IDs. Thus, in response to users' verification queries, the prover can send inclusion proofs without including any sensitive information such as the blinding factors. With this feature, the sparse Merkle tree of commitments can be outsourced to a third party without leaking liabilities and users can interact with the third party directly. In this way, the prover has no knowledge of which users verify or not, and prevents the prover from manipulating liabilities to users who seldom check based on the verification patterns he/she learns.

C.6 Private Information Retrieval

Private information retrieval (PIR) protocols allow users storing their data on a database to retrieve data without revealing which piece of data is accessed [23]. It is proven that with a single server to achieve information theoretic privacy, the communication complexity is at least $O(n)$, i.e., the server returns the whole dataset to the user. To get around this lower bound, we can distribute the database to multiple entities assuming they don't collude [3], or use a single server assuming the adversary is computationally bounded [49], or use trusted hardware [68]. To hide verifiers' identities, PIR seems to be a candidate solution.

C.7 Oblivious RAM

An oblivious RAM (ORAM) protocol enables users storing their data on a database to safely read and write data without leaking their access patterns [37, 38]. This can be used in PoL to support updates of the prover's liabilities to users while preserving privacy of update patterns. Viewing the aggregated inclusion proofs from colluding users as the dataset, the users play the role of the adversarial server in ORAM attempting to infer information, while the prover acts as

the data owner in ORAM aiming to conceal update patterns. Circuit ORAM [69] is the first tree-based ORAM protocol that achieves asymptotically optimal circuit size and bandwidth cost. Combining it with zero-knowledge proofs, a new primitive called Publicly-Verifiable Oblivious RAM Machine (PVORM) is proposed [16] to achieve a similar goal as in PoL, i.e., allowing an database manager to verifiably update the publicly visible but encrypted dataset for users without leaking users' update patterns. Adopting PVORM in PoL, users don't even need to verify inclusion proofs after each update but only the update proofs of PVORM, which we discuss further in section 4.3.3.

D DAPOL+ PSEUDOCODE

We present the pseudocode of the DAPOL+ protocol in fig. 8 to help demonstrate how it works.

E $\mathsf{Prot}_{\mathsf{DAPOL+}}$ SECURITY

We prove theorem 4.1 that $\mathsf{Prot}_{\mathsf{DAPOL+}}(N, MaxL)$ satisfies correctness and soundness.

Proof. It is straightforward that $\mathsf{Prot}_{\mathsf{DAPOL+}}(N, MaxL)$ satisfies correctness.

Now we prove $\mathsf{Prot}_{\mathsf{DAPOL+}}(N, MaxL)$ satisfies soundness. Assume for contradiction that soundness is broken, which means there exists a $(N, MaxL)$-valid data set DB, a p.p.t. adversarial prover $\mathcal{A}^*$ corrupting some users in $\mathcal{U}$, and a subset V of non-corrupted users such that with non-negligible probability, the adversary can cheat users in V without being detected. First, because users' IDs are distinct in a valid DB, by collision resistance of $\mathcal{H}(\cdot)$, each user must be mapped to a unique leaf node. Combining Merkle proofs for all users in V, again by collision resistance of $\mathcal{H}(\cdot)$, the intersection nodes of different Merkle paths together with the leaf nodes of V are consistent. Hence a unique sparse Merkle tree SMT can be derived from the Merkle paths and the leaf nodes for users in V. By the computationally binding property of Pedersen commitments, each leaf node mapped to $u \in V$ commits to the prover's liabilities to u. By the additively homomorphic property of Pedersen commitments, the root node should commit to the sum of values in all leaf nodes and padding nodes in SMT if there is no overflow. For each $u \in V$, by the soundness of Bulletproofs, each sibling node in u's Merkle path commits to a value within range $[0, N \cdot MaxL)$. Since $l_u \in [0, MaxL)$ as DB is valid, the internal node at height i on the path from u's leaf node to the root commits to a value within $[0, ((H-i) \cdot N + 1) \cdot MaxL)$, so the root within $[0, (H \cdot N + 1) \cdot MaxL)$. Given that $q \geq (H \cdot N + 1) \cdot MaxL$, there isn't an overflow in the additions. Therefore, $L = \sum_{u \in V} l_u + \sum_{i \in W} v_i$, where W is the set of padding nodes in SMT not mapped to any user in V and v_i is the value each node in W is committed to. Thus $L < \sum_{u \in V} l_u$ indicates that there exists some node $i \in W$ such that $v_i < 0$. This violates the security of Bulletproofs proving the range of v_i. Therefore, $\mathsf{Prot}_{\mathsf{DAPOL+}}(N, MaxL)$ satisfies soundness and thus is secure. □

We now show that $\mathsf{Prot}_{\mathsf{DAPOL+}}(N, MaxL)$ is Φ^{user}-private where $\Phi^{\mathsf{user}} = \emptyset$ as defined in theorem 4.2.

Proof. The real game returns PD, $DB[V]$ and $\{\pi_u\}_{u \in V}$. We construct as follows the simulator $\mathcal{S}$ taking 1^κ and $DB[V]$ as inputs:

$\mathsf{Prot}_{\mathsf{DAPOL+}}(N, MaxL)$

Public parameters :
$N, MaxL, H, G, g_1, g_2, salt\_b, salt\_s$

On init, $\mathcal{P}$ executes $\mathsf{Setup}(1^\kappa, DB)$:
Let $master\_secret \xleftarrow{\$} \{0,1\}^\kappa$;
Randomly map $u \in \mathcal{U}$ to a bottom-layer leaf node in an empty SMT;
For i in $H..1$:
Add padding nodes for non-existing siblings of existing nodes at height i;
Add parent nodes at height $i-1$ for existing nodes at height i;
For j in $H..0$:
For i is an existing node at height j in SMT :
If i is a leaf node of user u :
Let $w_u = KDF(master\_secret, id_u)$;
Let $b_u = KDF(w_u, salt\_b)$, $s_u = KDF(w_u, salt\_s)$;
Let $c_u = Com(l_u, b_u) = g_1^{l_u} \cdot g_2^{b_u}$, $h_u = \mathcal{H}(\text{“}leaf\text{”}||id_u||s_u)$;
If i is a padding node :
Let $w_i = KDF(master\_secret, idx_i)$;
Let $b_i = KDF(w_i, salt\_b)$, $s_i = KDF(w_i, salt\_s)$;
Let $c_i = Com(0, b_i) = g_1^0 \cdot g_2^{b_i}$, $h_i = \mathcal{H}(\text{“}pad\text{”}||idx_i||s_i)$;
If i is an internal node :
Let $c_i = c_{lch_i} \cdot c_{rch_i}$, $h_i = \mathcal{H}(c_{lch_i}||c_{rch_i}||h_{lch_i}||h_{rch_i})$;
Let $SD = (master\_secret$, the mapping);
Publish $PD = (c_{root}, h_{root})$;

On receive “ProveTot” **from a requester, $\mathcal{P}$ executes** $\mathsf{ProveTot}(DB, SD)$:
Let $L = \sum_{u \in \mathcal{U}} l_u$, $\Pi = \sum_{u \in \mathcal{U}} b_u + \sum_{\text{padding node } i} b_i$;
Send the requester (“VerifyTot”, L, Π);

On receive (“VerifyTot”, L, Π) from $\mathcal{P}$, execute $\mathsf{VerifyTot}(PD, L, \Pi)$:
If $PD = (c_{root}, h_{root})$ and $c_{root} = g_1^L \cdot g_2^\Pi$:
return 1;
Else :
return 0;

On receive (“Prove”, id) from a request, $\mathcal{P}$ executes $\mathsf{Prove}(DB, SD, id)$:
If the requester fails to authenticate identity with respect to id :
exit;
Retrieve b and s of the user with id;
Retrieve the Merkle path $\{(c_i, h_i)\}_{i \in [1,H]}$ authenticating the user's leaf node;
Generate π_{range} proving $\{c_i\}_{i \in [1,H]}$ commits to values within $[0, N \cdot MaxL)$;
Send the requester (“Verify”, $\pi = (b, s, \{(c_i, h_i)\}_{i \in [1,H]}, \pi_{range})$);

On receive (“Verify”, π) from $\mathcal{P}$, execute $\mathsf{Verify}(PD, id, l, \pi)$:
Let $c'_H = \mathsf{Com}(l, b)$, $h'_H = \mathcal{H}(\text{“}leaf\text{”}||id||s)$;
For i in $H-1..0$:
Let $c'_i = c'_{i+1} \cdot c_{i+1}$ and similarly compute h'_i;
If $(c'_0, h'_0) = PD$ and π_{range} is valid :
return 1;
Else :
return 0;

Figure 8: *The DAPOL+ protocol.*

If $V = \emptyset$, randomly sample $s \xleftarrow{\$} \{0,1\}^\kappa$ and $b \xleftarrow{\$} \mathbb{Z}_q$, and return $(c = \mathsf{Com}(0, b), h = \mathcal{H}(s))$, $\emptyset$ and $\emptyset$.

Otherwise, when $V \neq \emptyset$,

(1) Randomly map each user in V to a bottom layer node in an empty SMT of height H. For each node mapped to a user $u \in V$, let $c_u = \mathsf{Com}(l_u, b_u)$ and $h_u = \mathcal{H}(\text{“leaf”}||id_u||s_u)$, where $s_u \xleftarrow{\$} \{0,1\}^\kappa$ and $b_u \xleftarrow{\$} \mathbb{Z}_q$. Note that $\mathcal{H}(\cdot)$ and $KDF(\cdot)$ are calls to a random oracle.
(2) Construct a sparse Merkle tree SMT initiated with the bottom layer nodes mapped to users in V. For each padding

node i, randomly sample $s_i \xleftarrow{\$} \{0,1\}^\kappa$ and $b_i \xleftarrow{\$} \mathbb{Z}_q$. Let $c_i = \mathrm{Com}(0, b_i)$ and $h_i = \mathcal{H}(\text{“}pad\text{”}||idx_i||s_i)$. Then for each internal node i with child nodes lch_i and rch_i, let $c_i = c_{lch_i} \cdot c_{rch_i}$ and $h_i = \mathcal{H}(c_{lch_i}||c_{rch_i}||h_{lch_i}||h_{rch_i})$.

(3) Return (c_{root}, h_{root}), $DB[V]$ and inclusion proofs for each user in V generated from SMT.

Now we have defined the simulator, and the proof is straightforward. We introduce a hybrid game in which each user in V is mapped to a bottom layer node in an empty SMT of height H, and the SMT for V only is generated as in the real game. This hybrid game is indistinguishable from the real game. Then by perfect hiding of Pedersen commitments and zero-knowledge of Bulletproofs, the hybrid game is indistinguishable from the simulated game. □

We now show that $\mathrm{Prot}_{\mathrm{DAPOL+}}(N, MaxL)$ is Φ^{auditor}-private where $\Phi^{\text{auditor}} = \emptyset$ as defined in theorem 4.3.

Proof. The real game returns PD, L, Π, $DB[V]$ and $\{\pi_u\}_{u\in V}$. We construct as follows the simulator $\mathcal{S}$ taking 1^κ, L and $DB[V]$ as inputs:

If $V = \emptyset$, randomly sample $s \xleftarrow{\$} \{0,1\}^\kappa$ and $b \xleftarrow{\$} \mathbb{Z}_q$, and return $(c = \mathrm{Com}(L, b), h = \mathcal{H}(s))$, L, b, $\emptyset$ and $\emptyset$.
Otherwise, when $V \neq \emptyset$,

(1) Randomly map each user in V to a bottom layer node in an empty SMT of height H, and compute the commitment and hash for each user by $c_u = \mathrm{Com}(l_u, b_u)$ and $h_u = \mathcal{H}(\text{“leaf”}||id_u||s_u)$, where $s_u \xleftarrow{\$} \{0,1\}^\kappa$ and $b_u \xleftarrow{\$} \mathbb{Z}_q$.

(2) Construct a sparse Merkle tree SMT initiated with the bottom layer nodes mapped to users in V. For each padding node i, randomly sample $s_i \xleftarrow{\$} \{0,1\}^\kappa$ and $b_i \xleftarrow{\$} \mathbb{Z}_q$. Let $c_i = \mathrm{Com}(0, b_i)$ and $h_i = \mathcal{H}(\text{“}pad\text{”}||idx_i||s_i)$ except that for one padding node j, let $c_j = \mathrm{Com}(L - \sum_{u\in V} l_u, b_j)$ if j exists. Note that when there isn't a padding node in SMT, the tree is full and the adversary corrupts all users. Next for each internal node i with child nodes lch_i and rch_i, let $c_i = c_{lch_i} \cdot c_{rch_i}$ and $h_i = \mathcal{H}(c_{lch_i}||c_{rch_i}||h_{lch_i}||h_{rch_i})$.

(3) Return (c_{root}, h_{root}), L, $b_{root} = \sum_{u\in V} b_u + \sum_{\text{padding node } i} b_i$, $DB[V]$ and inclusion proofs for each user in V generated from SMT.

Now we have defined the simulator, and the proof is similar to that of theorem 4.2. Consider a hybrid game in which each user in V is mapped to a bottom layer node in an empty SMT of height H first, and users in $\mathcal{U} - V$ are mapped next, and then the game proceeds as in the real game. This hybrid game is indistinguishable from the real game. By perfect hiding of Pedersen commitments and zero-knowledge of Bulletproofs, the hybrid game is indistinguishable from the simulated game. □

F DISPUTE RESOLUTION

In this section, we take the solvency case as an example and discuss dispute resolution in PoL. Each virtual asset service provider (VASP) [27] plays the role of a prover and clients depositing virtual assets to it need to verify the inclusion of their balances in the VASP's total liabilities. A user may send money to another a transaction via the VASP he/she belongs to. Assuming every transaction consists of one sender and one receiver for simplicity, each transaction involves users either within a single VASP, i.e., the sender and the receiver belong to the same VASP, or across two VASPs. Here we empirically analyze all possible scenarios of dispute resolution for the single-VASP case and the cross-VASP case separately.

F.1 Transaction within a single VASP

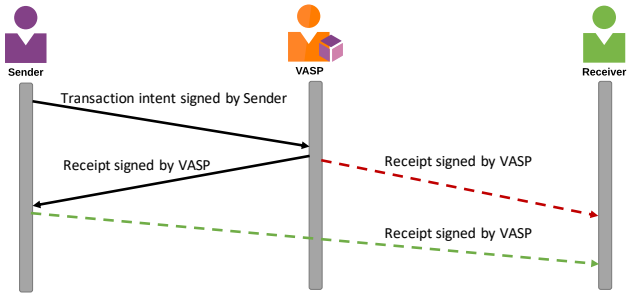


Figure 9: *Transaction within a single VASP.*

We depict in fig. 9 the protocol for a user to send a transaction with only necessary interactions. Note that the green and red dotted arrows are for the same purpose of guaranteeing that the receiver has the receipt of the transaction, and are complementary to each other, which we soon explain in detail. We classify all possible scenarios of disputes by the set of maliciously colluding entities and summarize them in table 5. We denote by S the sender, R the receiver, V the VASP, and H an entity being honest and M being malicious.

Table 5: *Transaction within a single VASP.*

S	V	R	Potential attack	Solution
M	H	H	deny a tx	V keeps the rcpt
H	M	H	deny a tx	S and R keep the rcpt
			replay a tx	use nonce
			not notify R	S has incentive to send rcpt
			not notify S	3 disputes/fair rcpt signing
			DoS	inevitable, hurts V's reput
H	H	M	forge a rcpt	rcpt must be signed by V
M	M	H	deny a tx	R keeps the rcpt
			not notify R	S has incentive to send rcpt
M	H	M	forge a rcpt	rcpt must be signed by V
H	M	M	replay a tx	use nonce
			not notify S	3 disputes/fair rcpt signing

S **being malicious.** S can only try to deny an approved transaction (tx) to claim a higher balance. However, this won't succeed as long as V keeps the receipt (rcpt) signed by both itself and S.

V **being malicious.** There are five possible scenarios. *First*, V may attempt to deny a transaction. But again, S has the receipt signed by V as an evidence. *Second*, V may replay the same transaction even if S doesn't have the intention. To prevent this, we can add a nonce to each transaction like in Ethereum [74], and transactions with the same nonce shall only be executed once. *Third*, usually V

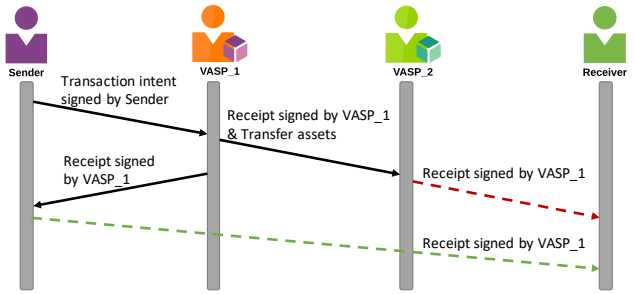


Figure 10: *Transaction across two VASPs.*

Table 6: *Transaction across two VASPs.*

S	V_1	V_2	R	Potential attack	Solution
H	M	H	H	not notify S	3 disputes/fair rcpt signing
				replay a tx	use nonce
				not transfer assets	PoT needs to match rcpt
				DoS	inevitable, hurts V_1's reput
H	H	M	H	deny asset transfer	V_1 provides PoT
				not notify R	S has incentive to send rcpt
M	M	H	H	not transfer assets	PoT needs to match rcpt
M	H	M	H	deny a tx	V_1 keeps the rcpt
				not notify R	S has incentive to send rcpt
				deny asset transfer	V_1 provides PoT
H	M	H	M	not notify S	3 disputes/fair rcpt signing
				replay a tx	use nonce
				not transfer assets	PoT needs to match rcpt
H	H	M	M	deny asset transfer	V_1 provides PoT
M	M	H	M	not transfer assets	PoT needs to match rcpt
M	H	M	M	deny a tx	V_1 keeps the rcpt
				deny asset transfer	V_1 provides PoT

automatically sends receipts to R for usability and efficiency in a trading system, but V may withhold a receipt so that R cannot verify the balance accordingly. In this case, S can forward the receipt to R directly to proceed with the trading, thus this is not an issue. In practice S has the incentive to do so because he/she is either paying for a product or service, or transferring money to R who he/she knows. In either case, S needs to get confirmation of receipt from R. *Fourth*, V may not notify S of the approval of a transaction but secretly decreases S's balance. S only finds out when verifying the balance. When S raises a dispute, however, V is able to provide the receipt. S may use the receipt to raise another dispute with R as R should have received the funds. This is in a different layer of trading protocols between S and R, so we don't go into detail. R disagreeing with the balance can use the receipt for another dispute. There are three potential disputes. To avoid all the complexities, S and V can also run a fair contract signing protocol [4] for receipts on each transaction to eliminate V's advantage here. *Fifth*, V may mount a denial-of-service (DoS) attack, not approving valid transactions, which is inevitable in most centralized applications. However, V cannot claim a lower balance for S, thus the issue is out of the scope of PoL. Meanwhile, DoS hurts V's reputation (reput), so V has no incentive to do so.

R **being malicious.** The capability of R is limited. Without a receipt signed by V, R cannot claim a higher balance than he should possess.

V **colludes with** S. There are two possible disputes. *First*, V and S may execute a transaction but later deny it so R's balance is not properly increased. This won't succeed as long as R keeps the receipt. *Second*, V and S may withhold the receipt from R. As mentioned, S has no incentive to do so in practice.

S **colludes with** R. A valid receipt requires the signature from V. So if only S and R are colluding, there is nothing harmful they can do.

V **colludes with** R. There are two possible disputes in this case. First, V may collude with R and replay a transaction from S. Similarly as before, adding a nonce to each transaction prevents this. Second, V may secretly process a transaction but doesn't send the receipt to S. As mentioned, this can be resolved via potentially three separate disputes or mitigated if S and V run a fair contract signing protocol.

Note that there is no incentive for S, R and V to collude together, because it is in S's and R's interests that their balances are no less than they actually own while V wants its total liabilities to be as low as possible.

Overall, all the scenarios except the DoS case which is inevitable in most centralized applications and hurts V's reputation, can be resolved. However, for an individual user, the cost (including time and effort) for dispute resolution may be higher than the balance in dispute, so the user might not have incentive to raise a dispute. In practice, V may slightly manipulate balances so that users have little motive to dispute. There is a game between these entities when taking such incentives into consideration, which is open for future research.

F.2 2-VASP Dispute Resolution

We depict in fig. 10 the execution of a transaction across two VASPs and analyze all possible dispute scenarios. The difference between transaction execution within a single VASP and that across two VASPs is that in the latter case, V_1 needs to transfer assets to V_2 and V_2 forwards the receipt to R. We can have a bijective mapping from scenarios in the single-VASP case to the two-VASP case by viewing V as the union of V_1 and V_2, i.e., V_1 and V_2 collude as a single entity, and all solutions still work. Therefore, we only analyze additional scenarios when either of V_1 and V_2 is malicious. Still, we classify them by the set of malicious entities and summarize the analysis in table 6.

V_1 **being malicious.** There are four possible disputes. *First*, V_1 may secretly decrease S's balance without sending the signed receipt to him. Similarly to the single-VASP case, this can be resolved by possibly three separate disputes or mitigated by S and V_1 running a fair contract signing protocol on the receipt. *Second*, V_1 may replay a transaction and this can be prevented by using nonce as discussed previously. *Third*, V_1 may approve a cross-VASP transaction but not transfer the corresponding asset to V_2. This can be resolved by requiring proof of transfer (PoT) from V_1. *Fourth*, V_1 may DoS S. As mentioned, this is inevitable but hurts V_1's reputation.

V_2 **being malicious.** There are two possible disputes. *First*, V_2 may deny receiving the asset from V_1. This can be resolved by V_1 providing a PoT. *Second*, V_2 may withhold the receipt from R and

not increase his balance properly. However, S has the incentive to notify R, so R can dispute with the receipt.

V_1 **colludes with** S**.** V_1 may approve a transaction but not send the assets to V_2. However, this won't succeed because S has incentive to send R the receipt, e.g., for service in return, which allows R to dispute. In the dispute, V_2 can require V_1 to provide a PoT that matches with the receipt.

V_2 **colludes with** S**.** There are three possible scenarios. *First*, S may deny an issued transaction, but V_1 keeps the receipt signed S. *Second*, S and V_2 may not notify R about the transaction. However, there is no incentive for S to do so in practice as discussed before. *Third*, V_2 may deny receiving assets from V_1 for the transaction, but V_1 can provide a PoT.

V_1 **colludes with** R**.** There are three possible scenarios. *First*, V_1 may secretly process the transaction but withhold the receipt from S. This can either be resolved via potentially three disputes or mitigated by S and V_1 running a fair signing protocol on the receipt. *Second*, V_1 may replay transactions, which can be prevented by using the nonce. *Third*, V_1 may not transfer the assets to V_2. V_2 can require V_1 to provide a PoT that matches the receipt with which R claims a higher balance.

V_2 **colludes with** R**.** V_2 may deny asset transfer from V_2. This can be resolved by V_1 providing a PoT.

V_1 **colludes with** S **and** R**.** V_1 may not send the assets to V_2 properly. However, once R claims a higher balance with a valid receipt, V_2 can dispute and require from V_1 a PoT that matches the receipt.

V_2 **colludes with** S **and** R**.** There are two possible disputes. *First*, S may deny a transaction he/she issued and approved by V_1. V_1 can use the receipt as a defense in the dispute. *Second*, V_2 may deny receiving assets from V_1 for the approved transaction, but V_1 can provide the corresponding PoT.

Overall, all the scenarios except the DoS case can be resolved. The discussions regarding the cost and game of dispute resolution in the single-VASP case also hold in the cross-VASP case.