Hyperproofs: Aggregating and Maintaining Proofs in Vector Commitments

Sharavan Srinivasan
University of Maryland

Alexander Chepurnoy
Ergo Platform

Charalampos Papamanthou
Yale University

Alin Tomescu
VMware Research

Yupeng Zhang
Texas A&M University

Abstract

We present Hyperproofs, the first vector commitment (VC) scheme that is efficiently *maintainable* and *aggregatable*. Similar to Merkle proofs, our proofs form a tree that can be efficiently maintained: updating all n proofs in the tree after a single leaf change only requires $O(\log n)$ time. Importantly, unlike Merkle proofs, Hyperproofs are *efficiently* aggregatable, anywhere from $10\times$ to $41\times$ faster than SNARK-based aggregation of Merkle proofs. At the same time, an individual Hyperproof consists of only $\log n$ algebraic hashes (e.g., 32-byte elliptic curve points) and an aggregation of b such proofs is only $O(\log(b \log n))$-sized. Hyperproofs are also reasonably fast to update when compared to Merkle trees with SNARK-friendly hash functions.

As another benefit over Merkle trees, Hyperproofs are *homomorphic*: digests (and proofs) for two vectors can be homomorphically combined into a digest (and proofs) for their sum. Homomorphism is very useful in emerging applications such as stateless cryptocurrencies. First, it enables *unstealability*, a novel property that incentivizes proof computation. Second, it makes digests and proofs much more convenient to update.

Finally, Hyperproofs have certain limitations: they are not transparent, have linear-sized public parameters, are slower to verify, and have larger aggregated proofs and slower verification than SNARK-based approaches. Nonetheless, end-to-end, aggregation and verification in Hyperproofs is $10\times$ to $41\times$ faster than in SNARK-based Merkle trees.

1 Introduction

Vector commitment (VC) schemes [21, 39] such as Merkle trees [40] are fundamental building blocks in many protocols. In a VC scheme, a *prover* computes a succinct *digest* d of a vector $\mathbf{a} = [a_1, \ldots, a_n]$ and proofs $\pi_1, \ldots, \pi_n$ for each position. A *verifier* who has the digest d can later verify a proof π_i that a_i is the correct value at position i. Some VCs, such as Merkle trees, are *maintainable*: when the vector changes *all* proofs can be *efficiently* updated in sublinear time, rather than recomputed from scratch in linear time. Other VCs, such as Pointproofs [28], are *aggregatable*: the prover can take several proofs π_i for $i \in I$ and *efficiently* aggregate them into a single, succinct proof π_I.

Unfortunately, no current VC scheme is both maintainable and aggregatable; at least not efficiently. Yet emerging applications such as *stateless cryptocurrencies* [12,22,28,41,52,57, 59] rely on dedicated nodes to efficiently maintain all proofs and also on miners to efficiently aggregate proofs. While generic argument systems (e.g., SNARKs [30, 49]) can be used to add aggregation to maintainable VCs such as Merkle trees, this is too slow in practice (see §5.2). This brings us to this paper's main concern: *Can we build an efficient VC that is both maintainable and aggregatable?* In this paper, we answer this positively and present *Hyperproofs*. Similar to Merkle trees, Hyperproofs are $\log n$-sized and determine a tree. This makes updating all proofs very efficient in logarithmic time. However, Hyperproofs are built from *polynomial commitments* [32,46] rather than hash functions such as SHA-256. This enables a natural aggregation algorithm that is $10\times$ to $41\times$ faster than "SNARKing" multiple Merkle proofs.

In addition to aggregation and maintainability, Hyperproofs have another very useful property: *homomorphism*. Specifically, trees (and digests) for two vectors can be combined into a single tree (and digest) for their sum. This has several applications. First, homomorphism allows us to obtain *unstealability*, a property which incentivizes proof computation in applications such as stateless cryptocurrencies [64]. In a nutshell, unstealability allows a prover to *watermark the proofs she computes with her identity, in an irreversible manner*. This way, honest provers can be rewarded for the proofs they compute while malicious provers cannot *steal* other provers' proofs. Second, homomorphism makes updating digests (and Hyperproofs) more convenient than updating Merkle roots (and proofs), which requires having the proof(s) for the changed position(s) in the vector. Third, homomorphism allows authenticating data in a streaming setting [48].

Challenges. In designing Hyperproofs, we surmount three key challenges. First, computing n proofs in Papamanthou-

Shi-Tamassia (PST) polynomial commitments [46] takes $O(n^2)$ time and is too slow. Second, aggregation of PST proofs is difficult without generic SNARKs [30,49], which would be too expensive. Third, unstealable proofs must remain maintainable and aggregatable. This precludes solutions based on computing SNARKs over proofs which, in addition to being slow (see §5.2), would sacrifice updatability (see *"Strawmen"* in §3.4). Furthermore, unstealable proofs must continue to verify with respect to one global digest. This precludes solutions that embed the identity of the prover inside the vector, which results in as many digests as there are provers (and would only be practical in a small-scale, permissioned setting).

Evaluation. In §5.1, we show Hyperproofs are small (1.44 KiB), they verify quickly (17.4 milliseconds) and are fast to maintain (2.6 milliseconds per update). In §5.2, we show Hyperproof aggregation is much faster than Merkle proof aggregation: 10× faster when using Poseidon hashes [29], which likely need more cryptanalysis, and 41× faster when using provably-secure Pedersen hashes. However, our faster aggregation comes at a cost of slower verification for aggregated proofs and a larger 52 KiB aggregate proof size. Nonetheless, when considering the end-to-end aggregation and verification time in stateless cryptocurrencies, Hyperproofs remain 10× to 41× faster and outperform Merkle trees (see §5.3).

Limitations. To commit to a vector of size n, Hyperproofs requires public parameters consisting of $2n-1$ group elements, which must be generated via a *trusted setup*, typically decentralized via multi-party computation protocols [15]. In future work, we hope to have a *transparent setup* by using assumptions in hidden-order groups. We also do not explore the subtleties of fully-integrating unstealable proofs into a statelessly-validated cryptocurrency. Lastly, our macrobenchmarks only measure the computational overhead of VCs that arises on the critical path to a consensus decision. While our results show Hyperproofs lead to 10× faster decisions, we do not claim this is sufficient to make the stateless setting practical.

1.1 Overview of Techniques

Vectors as multilinear extensions (MLEs). We build upon previous work [68,69] that represents a vector of size $n = 2^\ell$ as a *multilinear extension (MLE)* polynomial. For example, the MLE of $\mathbf{a} = [5,2,8,3]$ is $f(x_2,x_1) = 5(1-x_2)(1-x_1)+2(1-x_2)x_1 + 8x_2(1-x_1)+3x_2x_1$. Note that f correctly "selects" the right a_i given the binary expansion of i as input: $f(0,0) = 5, f(0,1) = 2, f(1,0) = 8$ and $f(1,1) = 3$.

PST commitments to MLEs. To commit to a vector, we compute a Papamanthou-Shi-Tamassia (PST) commitment [46] to its MLE (see §2.2) . For example, the PST commitment to f above is $\mathsf{C} = g_1^{f(s_1,s_2)} \in \mathbb{G}_1$, where $(s_1,s_2) \in \mathbb{Z}_p^2$ are secret points encoded in the *public parameters* of the scheme and g_1 is the generator of $\mathbb{G}_1$. For vectors of size 4, these public parameters consist of $g_1^{s_1}, g_1^{1-s_1}, g_1^{(1-s_2)(1-s_1)}, g_1^{(1-s_2)s_1}, g_1^{s_2(1-s_1)}, g_1^{s_2 s_1}$. Importantly, we show that the selectively-secure variant of PST commitments is actually adaptively-secure when restricted to only proving evaluations on the Boolean hypercube $\{0,1\}^\ell$ (see §2.2). This reduces our proof size compared to previous work based on PST [68, 69].

Multilinear trees. To prove that a_i is the ith value in the vector $\mathbf{a} = [a_0,\dots,a_{n-1}]$, we compute a *PST evaluation proof* for $f(i_\ell,\dots,i_1) = a_i$ w.r.t. the commitment C, where $(i_\ell,\dots,i_1)$ is the binary representation of i. Unfortunately, this takes $O(n)$ time *per position*. Thus, computing all n proofs would take $O(n^2)$ time which is prohibitive. We reduce this to $O(n\log n)$ by computing a novel *multilinear tree (MLT)* of proofs using a divide-and-conquer approach. Importantly, our MLT is *maintainable*: updating all proofs after a change to the vector only requires $O(\log n)$ time.

Proof aggregation. A proof π_i for a_i consists of PST commitments $(w_{i,\ell},\dots,w_{i,1}) \in \mathbb{G}_1^\ell$ defined in Fig. 2, such that the following *pairing equation* holds:

$$e(\mathsf{C}/g_1^{a_i}, g_2) = \prod_{j\in[\ell]} e(w_{i,j}, g_2^{s_j - i_j}), \tag{1}$$

where $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a *pairing* and $g_2^{s_j}$'s are additional $O(\ell)$-sized PST public parameters in $\mathbb{G}_2$. To aggregate b proofs, we prove *knowledge* of $w_{i,j}$'s that pass Eq. 1 for each i, resulting in a succinct $O(\log(b\ell))$ aggregated proof size. Our key ingredient is an *inner-product argument (IPA)* by Bünz et al. [19] for proving several pairing equations hold.

Homomorphism and unstealablity. As we mentioned, Hyperproofs are *homomorphic*: exponentiating a PST evaluation proof $(w_{i,\ell},\dots,w_{i,1})$ by a constant α yields a proof for position i but in a vector whose values are multiplied by α. We observe that if α is the secret key of a *proof-serving node (PSN)*, this makes the proof unstealable by other nodes who do not have α. Importantly, the proof can still be verified against the digest C, except the verifier must also give the node's corresponding public key g_2^α: $e(\mathsf{C}/g_1^{a_i}, g_2^\alpha) = \prod_{j\in[\ell]} e(w_{i,j}^\alpha, g_2^{s_j-i_j})$. As an optimization, proof-serving nodes can exponentiate the PST public parameters by α before computing proofs. This way, when computing a multilinear tree (MLT) with these parameters, all proofs are implicitly unstealable and the MLT remains maintainable.

1.2 Related work

Below, we relate our VC to previous work and summarize in Table 1.

Merkle trees. Our proofs consist of $\log n$ (algebraic) hashes and can be as small as Merkle proofs if using 256-bit elliptic curves [6]. However, Hyperproofs are orders of magnitude

Table 1: Comparison with other VCs, which are *not* simultaneously *aggregatable* and *maintainable* (see "Agg time" and "UpdAllProofs time" columns). n is the size of the vector, π_i is a proof for position i and π_I is an aggregated proof for k positions. Proof sizes and time complexities are in terms of group elements and group exponentiations / field operations, respectively. (In RSA-based VCs [12, 20, 21, 37], we count $O(\ell)$ group operations as an exponentiation, where ℓ is the bit-width of VC elements.) Items in red indicate worse performance than Hyperproofs. All schemes* support UpdDig and UpdProof (see Def. 2.1).

Scheme	$\lvert\pi_i\rvert$	$\lvert\pi_I\rvert$	OpenAll time	Agg time	UpdAllProofs time	Trans-parent?	Homo-morphic?	Gen time	$\lvert$pp$\rvert$
AMT [58]	$\log n$	×	$n \log n$	×	$\log n$	×	✓	n^2	$n \log n$
aSVC [59]	1	1	$n \log n$	$k \log^2 k$	n	×	✓	$n \log n$	n
BBF [12]	1	1	$n \log^2 n$	$k \log n$	$n \log n$	×†	×	1	1
CF-CDH [21, 28, 37]	1	1	n^2	k	n	×	✓	n^2	n^2
CF-RSA [20, 21, 37]	1	1	$n \log n$	$k \log^2 k$	n	×†	✓	1	1
CFG+RSA [20]	1	1	$n \log^2 n$	$k \log k \log n$	n	×†	×	1	1
Lattice [48, 51]	$\log n$	×	n	×	$\log n$	✓	✓	1	$\log n$
Merkle	$\log n$	×	n	×	$\log n$	✓	×	1	1
Merkle SNARK	$\log n$	1	n	$k \log n \log (k \log n)$	$\log n$	×	×	1	1
Pointproofs [28]	1	1	$n \log n$	k	n	×	✓	n	n
Hyperproofs	$\log n$	$\log(k \log n)$	$n \log n$	$k \log n$	$\log n$	×	✓	n	n

†: BBF, CF-RSA and CFG+RSA avoid the trusted setup if instantiated using class groups of imaginary quadratic order, which are known to be slower than RSA groups.
*: Merkle trees, BBF and CFG+RSA require *dynamic* update hints, rather than *static* update keys, for digest and proof updates. Only the *weakly-binding* variant of CFG+RSA supports digest updates. CF-CDH and Pointproofs have $O(n)$-sized update keys, which can be too large for some applications.

slower to compute and update, when compared to normal Merkle trees hashed with SHA-256. Nonetheless, when compared to aggregatable Merkle trees that use SNARK-friendly hash functions (e.g., Poseidon-128 [29]), Hyperproofs are only slightly slower to compute and update (see §5.3) but have faster aggregation, homomorphism and unstealability.

SNARK-based works. Ozdemir et al. [45] explore using SNARKs to prove knowledge of changes that update a vector with digest d into a new vector with digest d'. Lee et al. [38] also use SNARKs to prove correctness of state transitions in replicated state machines, without having to send the state changes. Neither work explores unstealability nor maintaining and aggregating proofs efficiently. Similar to our work, aggregating SNARK proofs [19] and some *proof-carrying data (PCD)* schemes [16] also rely on inner-product arguments.

Algebraic VCs. Zhang et al. [68, 69] were the first to build VCs from PST commitments to MLEs. However, their $O(\log n)$-sized proofs are concretely larger and do not support updates. Some VCs have $O(1)$-sized proofs [12, 20, 21, 28, 35, 37, 59], which inherently require $\Theta(n)$ time to update all proofs after a change. Aggregation and verification in these VCs is concretely, and sometimes asymptotically, faster (see Table 1). They also have smaller aggregated proofs. However, these VCs are not efficiently maintainable (see §5.1), which precludes using them in settings where provers are rewarded to maintain proofs (see §4).

Previous maintainable VCs [48, 51, 58, 60] do not support aggregation; at least not without expensive generic argument systems (e.g., SNARKs). The lattice-based construction from [48, 51] is also homomorphic and additionally transparent, with constant-sized public parameters. However, it is too slow for practice and non-aggregatable. The *authenticated multipoint evaluation tree (AMT)* construction from [58, 60] can be viewed as the dual to our construction, but from univariate polynomials rather than multivariate. Unfortunately, it is non-aggregatable, its trusted setup requires $O(n^2)$ time and it has larger $O(n \log n)$-sized public parameters.

Recent work [3, 12, 61] enhances VCs into *key-value commitments (KVCs)*, where arbitrary keys (rather than vector positions) are mapped to values. Unfortunately, all of these constructions have constant-sized proofs and are thus not maintainable. Some VCs have transparent setup [12, 20, 37], support incremental aggregation [20], have a "specialiazable" CRS [20] and provide time/space trade-offs when computing proofs [12, 20]. Hyperproofs do not have any of these features.

Unstealability. To the best of our knowledge, Katz et al. are the first to observe that (carefully) tying the identity of the prover to a proof she computes allows rewarding the prover for her effort [33]. However, their work focuses on *watermarking zero-knowledge proofs of knowledge* of a secret witness. In contrast, in our work, our proofs need not be zero-knowledge and they need not prove knowledge of secret witnesses. Furthermore, unlike Katz et al.'s result, our notion of unstealability captures the difficulty of extracting useful information from watermarked proofs that might help an adversary steal proofs faster than computing them from scratch. Subsequently, Wesolowski explores such *watermarked proofs* in the context of verifiable delay functions (VDF) [64]. In contrast, we are the first to explore watermarking VC proofs and to give security definitions.

Although no previous VC scheme is unstealable, some can be made so using our pairing-based techniques from §3.4. Specifically, VCs from pairing-based polynomial commitments [28, 59, 60] appear compatible with our techniques. On the other hand, RSA-based VCs [12, 20, 21], which lack pairings, are less amenable to our techniques. While proofs-of-knowledge of exponent (PoKEs) [12] could be used to replace the reliance on pairings, this would come at the cost of losing maintainability of watermarked proofs. Lastly, our pairing-based techniques do not apply to Merkle trees as they are based on hash functions. Instead, we discuss watermarking Merkle proofs via SNARKs and their pitfalls in §3.4, under *"Strawmen"*.

2 Preliminaries

Notation. Let $[0,n) = \{0,1,\ldots,n-1\}$. An ℓ-bit number i has *binary representation* $\mathbf{i} = (i_\ell,\ldots,i_1)$ if, and only if, $i = \sum_{k=0}^{\ell-1} i_{k+1}2^k$. Note that i_ℓ is the MSB of i and i_1 is the LSB. We often use $\mathbf{i}$ as i's binary representation and i_k as its kth bit, without explicit definition. Let $r \in_R S$ denote picking an element from S uniformly at random.

Pairings. $(p,\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,e,g_1,g_2) \leftarrow \mathsf{BilGen}(1^\lambda)$ denotes generating groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ of prime order p, with g_i a generator of $\mathbb{G}_i$, and a *pairing* $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ such that $\forall u \in \mathbb{G}_1, w \in \mathbb{G}_2$ and $a,b \in \mathbb{Z}_p, e(u^a,w^b) = e(u,w)^{ab}$. A useful property of $e(\cdot,\cdot)$ is that $e(u,h)e(v,h) = e(uv,h), \forall u,v,h \in \mathbb{G}_1^2 \times \mathbb{G}_2$. In this paper, we assume *Type III bilinear groups* (i.e., without efficiently-computable homomorphisms between $\mathbb{G}_1$ and $\mathbb{G}_2$ or viceversa), which are needed by the *inner-product argument* from §2.4 and are also more efficient in practice. Let $1_\mathbb{G}$ denote the identity in a group $\mathbb{G}$.

Vectors. Bolded, lower-case symbols such as $\mathbf{a} = [a_0,\ldots,a_{n-1}] \in \mathbb{Z}_p^n$ typically denote vectors of field elements. Bolded, upper-case symbols such as $\mathbf{A} = [A_1,\ldots,A_m] \in \mathbb{G}^m$ typically denote vectors of group elements. $|\mathbf{A}|$ denotes the size of the vector $\mathbf{A}$. $\mathbf{A}^x = [A_1^x,\ldots,A_m^x], x \in \mathbb{Z}_p$, $\mathbf{A} \circ \mathbf{B} = [A_1B_1, A_2B_2,\ldots,A_mB_m]$, and $\langle \mathbf{A},\mathbf{B}\rangle = \prod_{j=1}^m e(A_j,B_j)$ denotes a *pairing product*. Let $\mathbf{A}_L = [A_1,\ldots,A_{m/2}]$ and $\mathbf{A}_R = [A_{m/2+1},\ldots,A_m]$ denote the left and right halves of $\mathbf{A}$. Let $\mathbf{A}||1_\mathbb{G}$ denote a vector of size $2|\mathbf{A}|$ that "extends" $\mathbf{A}$ to the right with the identity of $\mathbb{G}$. (Similarly, $1_\mathbb{G}||\mathbf{A}$ "extends" $\mathbf{A}$ to the left.)

2.1 Multilinear extension (MLE) of a vector

Let $n = 2^\ell$ and $\mathbf{x} = (x_\ell,\ldots,x_1)$. A vector $\mathbf{a} = [a_0,\ldots,a_{n-1}] \in \mathbb{Z}_p^n$ can be represented as a *multilinear extension* polynomial $f : \mathbb{Z}_p^\ell \to \mathbb{Z}_p$ which maps each position i to a_i:

$$f(\mathbf{i}) = f(i_\ell,\ldots,i_2,i_1) = a_i, \forall i \in [0,n) \quad (2)$$

For example, the MLE of $\mathbf{a} = [5,2,8,3]$ is $f(x_2,x_1)$ defined as:

$$5(1-x_2)(1-x_1) + 2(1-x_2)x_1 + 8x_2(1-x_1) + 3x_2x_1 \quad (3)$$

In general, the *unique* multilinear extension f of $\mathbf{a}$ is:

$$f(\mathbf{x}) = f(x_\ell,\ldots,x_1) = \sum_{j=0}^{n-1} a_j\mathcal{S}_{j,\ell}(x_\ell,\ldots,x_1) = \sum_{j=0}^{n-1} a_j\mathcal{S}_{j,\ell}(\mathbf{x}) \quad (4)$$

where $\mathcal{S}_{j,\ell}, j \in [0,2^\ell)$ are *selector multinomials* defined as:

$$\mathcal{S}_{j,\ell}(\mathbf{x}) = \prod_{k=1}^{\ell} \mathsf{sel}_{j_k}(x_k), \text{s.t. } \mathsf{sel}_{j_k}(x_k) = \begin{cases} x_k, & \text{if } j_k = 1 \\ 1-x_k, & \text{if } j_k = 0 \end{cases}, \quad (5)$$

with $S_{0,0}(\mathbf{x}) = 1$. In our example from Eq. 3, we have $\ell = 2$ and so: $\mathcal{S}_{0,2}(\mathbf{x}) = (1-x_2)(1-x_1)$, $\mathcal{S}_{1,2}(\mathbf{x}) = (1-x_2)x_1$, $\mathcal{S}_{2,2}(\mathbf{x}) = x_2(1-x_1)$ and $\mathcal{S}_{3,2}(\mathbf{x}) = x_2x_1$. We often refer to sel_{j_k} as a *selector monomial*. Importantly, note that:

$$\mathcal{S}_{j,\ell}(i_\ell,\ldots,i_1) = \mathcal{S}_{j,\ell}(\mathbf{i}) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}, \forall i \in [0,2^\ell) \quad (6)$$

By these properties above, we can see why Eq. 2 holds for any i:

$$f(\mathbf{i}) = \sum_{j=0}^{n-1} a_j\mathcal{S}_{j,\ell}(\mathbf{i}) = a_i\mathcal{S}_{i,\ell}(\mathbf{i}) + \sum_{j=1,j\neq i}^{n-1} a_j\mathcal{S}_{j,\ell}(\mathbf{i}) = a_i \cdot 1 + 0$$

In other words, an MLE f acts as a "multiplexer", choosing the right a_i based on the input position i, given as $\mathbf{i}$ in binary.

MLE decomposition. An MLE of size $n = 2^\ell$ can be *decomposed* into two MLEs of size $n/2$ [69]. For example, split $\mathbf{a}$ from Eq. 3 into its left and right halves $\mathbf{a}_0 = [5,2]$ and $\mathbf{a}_1 = [8,3]$, with MLEs $f_0 = 5(1-x_1) + 2x_1$ and $f_1 = 8(1-x_1) + 3x_1$, respectively. Then, observe that the MLE f for $\mathbf{a}$ is a combination of f_0 and f_1: i.e., $f = (1-x_2)f_0 + x_2f_1$. In general, the MLE f of any $\mathbf{a}$ decomposes as:

$$f(\mathbf{x}) = (1-x_\ell)f_0(x_{\ell-1},\ldots,x_1) + x_\ell f_1(x_{\ell-1},\ldots,x_1) \quad (7)$$

Note that for $\mathbf{a} = [a_0,a_1]$ of size 2, the MLEs f_0, f_1 are *trivial* (i.e., of size 1) and are simply set to a_0 and a_1, respectively. We use $f_{b_\ell b_{\ell-1}\ldots b_k}$ to denote the MLE of the $\mathbf{a}_{b_\ell b_{\ell-1}\ldots b_k}$ subvector, which is a subvector of all a_i's with $i_\ell = b_\ell, i_{\ell-1} = b_{\ell-1},\ldots,i_k = b_k$. For example, in a vector $\mathbf{a} = [a_0,\ldots,a_7]$, f_{01} is the MLE of $\mathbf{a}_{01}$, which contains all (three bit) positions i whose first two bits are 01: i.e., $\mathbf{a}_{01} = [a_2,a_3]$ because, in binary, 2 and 3 are 010 and 011, respectively.

2.2 PST commitments to MLEs

Papamanthou, Shi and Tamassia [46] extend Kate-Zaverucha-Goldberg (KZG) univariate polynomial commitments [32]

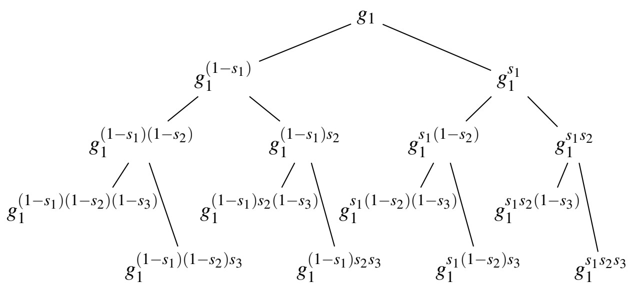


Figure 1: PST (and Hyperproofs) public parameters. The uth path in this tree is actually the update key upk_u from Eq. 16.

to multivariate ones. We refer to their scheme as PST and restrict its use to multilinear extensions, introduced above.

Commitments. PST works over a bilinear group obtained via BilGen. The PST commitment to a multilinear extension f for a vector $\mathbf{a}$ of size $n = 2^\ell$ is a single group element in $\mathbb{G}_1$:

$$\mathsf{pst}(f) = g_1^{f(s_\ell,\ldots,s_1)} = g_1^{\sum_{j=0}^{n-1} a_j \mathcal{S}_{j,\ell}(\mathbf{s})} = \prod_{j=0}^{n-1} \left(g_1^{\mathcal{S}_{j,\ell}(\mathbf{s})}\right)^{a_j} \quad (8)$$

Here, $\mathbf{s} = (s_\ell,\ldots,s_1)$ are *trapdoors* generated via a *trusted setup* that outputs n-sized *public parameters*: $g_1^{\mathcal{S}_{j,\ell}(\mathbf{s})} = g_1^{\mathcal{S}_{j,\ell}(s_\ell,\ldots,s_1)}, \forall j \in [0, 2^\ell)$. Importantly, the setup discards $\mathbf{s}$, since knowledge of it directly breaks PST's security [47]. We stress that $\mathsf{pst}(f)$ can be computed without knowing $\mathbf{s}$, as per Eq. 8. Lastly, PST commitments are *homomorphic*, with $\mathsf{pst}(f+f') = \mathsf{pst}(f)\mathsf{pst}(f')$ for any MLEs f, f'.

Evaluation proofs. Papamanthou, Shi and Tamassia give a way to prove evaluations $f(\mathbf{i})$ against $\mathsf{pst}(f)$ [46], where $\mathbf{i}$ is the binary representation of $i \in [0,n)$. Their key observation, which we refer to as the *PST decomposition*, is that:

$$f(\mathbf{i}) = z \Leftrightarrow \exists q_j\text{'s}, f(\mathbf{x}) - z = \sum_{j\in[\ell]} q_j(x_{j-1},\ldots,x_1)\cdot(x_j - i_j) \quad (9)$$

This yields a *PST evaluation proof* for $f(\mathbf{i}) = z$ consisting of commitments $w_j = g_1^{q_j(\mathbf{s})}$ to the *quotient polynomials* q_j. To compute the q_j's, the prover first divides f by $x_\ell - i_\ell$, obtaining q_ℓ and a remainder r_ℓ. Then, the prover continues recursively on the remainder r_ℓ, which no longer has variable x_ℓ. Specifically, the prover divides r_ℓ by $x_{\ell-1} - i_{\ell-1}$, obtaining $q_{\ell-1}$ and $r_{\ell-1}$. And so on, until he obtains the last quotient q_1 with remainder $r_1 = f(\mathbf{i})$ (see Fig. 2 and [47, Lemma 1]). Overall, this takes $T(n) = O(n) + T(n/2) = O(n)$ time, including the time to commit to the q_j's.

Note that the q_j's are actually MLEs of size $n/2, n/4, \ldots, 1$. As a result, PST's actual public parameters are $g_1^{\mathcal{S}_{j,k}(\mathbf{s})}, \forall k \in [0,\ell], \forall j \in [0, 2^k)$, so as to also be able to commit to these quotient MLEs. Lastly, the parameters form a tree (see Fig. 1) and are thus of size $2n-1$ $\mathbb{G}_1$ elements.

A verifier who has the commitment $\mathsf{pst}(f)$, the claimed evaluation $(i, f(\mathbf{i}) = z)$ and a logarithmic-sized, publicly-known *verification key* $g_2^{s_j}, \forall j \in [\ell]$ can verify the proof using

$\mathsf{PST.Prove}(f, \ell, \mathbf{i} = (i_\ell,\ldots,i_1)) \rightarrow \pi_i$:

1. If $\ell = 0$ (i.e., f is a constant), return $\varnothing$.
2. Otherwise, divide f by $x_\ell - i_\ell$, obtaining quotient $q_\ell(x_{\ell-1},\ldots,x_1)$ and remainder $r_\ell(x_{\ell-1},\ldots,x_1)$ such that $f = q_\ell \cdot (x_\ell - i_\ell) + r_\ell$.
3. Return $\left(g_1^{q_\ell(\mathbf{s})}, \mathsf{PST.Prove}(r_\ell, \ell-1, (i_{\ell-1},\ldots,i_1))\right)$

Figure 2: $O(n)$-time algorithm for computing a single PST evaluation proof π_i for $f(\mathbf{i})$ w.r.t. an MLE f of size $n = 2^\ell$.

$\ell+1$ pairings:

$$e(\mathsf{pst}(f)/g_1^z, g_2) = \prod_{j\in[\ell]} e(w_j, g_2^{s_j - i_j}) \quad (10)$$

The check above ensures Eq. 9 holds when $\mathbf{x} = \mathbf{s}$, which is sufficient for security since $\mathbf{s}$ is random and secret. In constructing our VC, we prove a stronger notion of security for PST commitments (see §6).

2.3 Vector Commitments (VCs)

We formalize VCs below, similar to Catalano and Fiore [21].

Definition 2.1 (VC)**.** *A VC scheme is a set of* PPT *algorithms:*

$\mathsf{Gen}(1^\lambda, n) \rightarrow \mathsf{pp}$: Given security parameter λ and maximum vector size n, outputs randomly-generated public parameters pp.

$\mathsf{Com}_{\mathsf{pp}}(\mathbf{a}) \rightarrow \mathsf{C}$: Outputs digest C of $\mathbf{a} = [a_0,\ldots,a_{n-1}] \in \mathbb{Z}_p^n$.

$\mathsf{Open}_{\mathsf{pp}}(i, \mathbf{a}) \rightarrow \pi_i$: Outputs a proof π_i for position i in $\mathbf{a}$.

$\mathsf{OpenAll}_{\mathsf{pp}}(\boldsymbol{a}) \rightarrow (\pi_0,\ldots,\pi_{n-1})$: Outputs all proofs π_i for $\mathbf{a}$.

$\mathsf{Agg}_{\mathsf{pp}}(I, (a_i, \pi_i)_{i\in I}) \rightarrow \pi_I$: Combines individual proofs π_i for values a_i into an aggregated proof π_I.

$\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, I, (a_i)_{i\in I}, \pi_I) \rightarrow \{0,1\}$: Verifies proof π_I that each position $i \in I$ has value a_i against digest C.

$\mathsf{UpdDig}_{\mathsf{pp}}(u, \delta, \mathsf{C}) \rightarrow \mathsf{C}'$: Updates digest C to C' to reflect position u changing by $\delta \in \mathbb{Z}_p$.

$\mathsf{UpdProof}_{\mathsf{pp}}(u, \delta, \pi_i) \rightarrow \pi_i'$: Updates proof π_i to π_i' to reflect position u changing by $\delta \in \mathbb{Z}_p$.

$\mathsf{UpdAllProofs}_{\mathsf{pp}}(u, \delta, \pi_0,\ldots,\pi_{n-1}) \rightarrow (\pi_0',\ldots,\pi_{n-1}')$: Updates all proofs π_i to π_i' to reflect position u changing by $\delta \in \mathbb{Z}_p$.

Observations: For simplicity, we give our algorithms oracle access to the public parameters pp of the scheme. This way, each algorithm can easily access the subset of the parameters it needs.

We formalize OpenAll and UpdAllProofs since, in some VCs, these algorithms are faster than n calls to Open and UpdProof, respectively. In this sense, we stress that the UpdAllProofs algorithm can work in sublinear time, since it does not necessarily need to read all input or write all output (e.g., in Merkle trees, UpdAllProofs only reads $\log n$ sibling hashes and overwrites another $\log n$ hashes).

Correctness and soundness. We define VC correctness in Def. B.1 and VC soundness in Def. B.2.

2.4 Inner Product Arguments (IPA)

Let CM denote a commitment scheme by Abe et al. [1] for vectors $\mathbf{A},\mathbf{B} \in \mathbb{G}_1^m \times \mathbb{G}_2^m$ and their *pairing product* $Z = \langle \mathbf{A},\mathbf{B}\rangle = \prod_{i=1}^m e(A_i,B_i)$. CM uses a randomly-generated *commitment key* $\mathbf{ck} = (\mathbf{v},\mathbf{w}) \in \mathbb{G}_1^m \times \mathbb{G}_2^m$ to commit to $\mathbf{A},\mathbf{B}$ and Z as:

$$\mathbf{C} = \mathsf{CM}(\mathbf{ck};\mathbf{A},\mathbf{B},Z) = (\langle \mathbf{A},\mathbf{v}\rangle,\ \langle \mathbf{w},\mathbf{B}\rangle,\ Z) \stackrel{\text{def}}{=} (C_1,C_2,C_3) \quad (11)$$

This commitment scheme is not hiding but is binding under *Symmetric-eXternal Diffie-Hellman (SXDH)* [1, 2].

Bünz et al. [19] give a non-interactive *inner-product argument (IPA)* where a *prover* convinces a *verifier*, that the prover *knows* how to open an Abe et al. commitment $\mathbf{C}$ to $(\mathbf{A},\mathbf{B},\langle \mathbf{A},\mathbf{B}\rangle)$; i.e. they give an argument for the language:

$$\mathcal{L}_{\mathsf{IPA}}^m = \{(\mathbf{ck},\mathbf{C}) \mid \exists \mathbf{A} \in \mathbb{G}_1^m, \mathbf{B} \in \mathbb{G}_2^m,\ \text{s.t.}\ \mathbf{C} = \mathsf{CM}(\mathbf{ck};\mathbf{A},\mathbf{B},\langle \mathbf{A},\mathbf{B}\rangle)\}$$

We abstract Bünz et al.'s [19] non-interactive argument for $\mathcal{L}_{\mathsf{IPA}}$ as three algorithms:

$\underline{\mathcal{G}_{\mathsf{IPA}}(1^\lambda,m) \to (PK,VK)}$: Returns $PK = VK = \langle \mathsf{BilGen}(1^\lambda), \mathbf{ck} = (\mathbf{v} \in_R \mathbb{G}_2^m, \mathbf{w} \in_R \mathbb{G}_1^m)\rangle$

$\underline{\mathcal{P}_{\mathsf{IPA}}(PK,\mathbf{A},\mathbf{B}) \to \pi}$: Returns a proof π that $\mathbf{C} = \mathsf{CM}(\mathbf{ck};\mathbf{A},\mathbf{B},\langle \mathbf{A},\mathbf{B}\rangle)$

$\underline{\mathcal{V}_{\mathsf{IPA}}(VK,\mathbf{C},\pi) \to \{0,1\}}$: Verifies proof π that $\mathbf{C} = \mathsf{CM}(\mathbf{ck};\mathbf{A},\mathbf{B},\langle \mathbf{A},\mathbf{B}\rangle)$

IPA complexity. $\mathcal{P}_{\mathsf{IPA}}$ takes $O(m)$ time, $\mathcal{V}_{\mathsf{IPA}}$ takes $O(\log m)$ time and the proof size is $|\pi| = O(\log m)$ (see App. A).

3 Hyperproofs

In this section, we intuitively explain how Hyperproofs work, often referring to a *prover* who computes the vector's *digest*, as well as proofs, and to a *verifier* who verifies proofs against this digest. Without loss of generality, our discussion will assume vectors of size exactly $n = 2^\ell$. Hyperproofs represents a vector $\mathbf{a} = [a_0,\ldots,a_{n-1}]$ as a multilinear extension (MLE):

$$f(\mathbf{x}) = \sum_{i=0}^{n-1} a_i \mathcal{S}_{i,\ell}(\mathbf{x}),$$

where $\mathcal{S}_{i,\ell}$ are selector multinomials as per Eq. 5. The *digest* of the vector $\mathbf{a}$ is a Papamanthou-Shi-Tamassia (PST) commitment to f:

$$\mathsf{pst}(f) = g_1^{f(\mathbf{s})},$$

where $\mathbf{s}$ is the PST trapdoor (see §2.2). Thus, our public parameters are the same as PST's parameters depicted in Fig. 1.

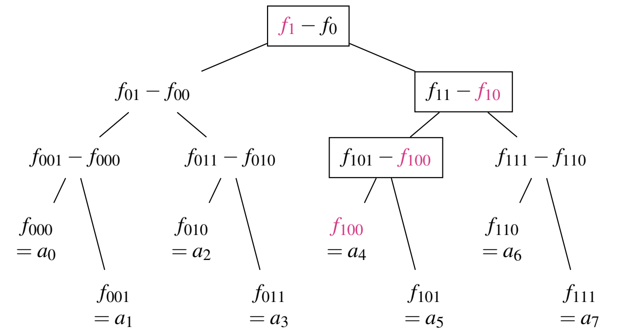


Figure 3: A multilinear tree (MLT) of size 8. Recall from §2 that $f_{b_\ell b_{\ell-1}\ldots b_k}$ denotes the MLE of $\mathbf{a}_{b_\ell b_{\ell-1}\ldots b_k}$. Each node stores a PST commitment to the depicted MLE: e.g., root stores $\mathsf{pst}(f_1 - f_0)$, not $f_1 - f_0$. The proof for a_i consists of all commitments along a_i's path to the root (e.g., for a_4, the boxed nodes). Sibling leaves $[a_{2j}, a_{2j+1}]$ have the same proof. If, say, a_4 changes, all pink-colored MLEs change, and all boxed commitments must be updated.

3.1 Multilinear trees (MLTs)

A Hyperproof for position i is just a PST evaluation proof (see §2.2) for $f(\mathbf{i})$. Unfortunately, if one uses the $\mathsf{PST.Prove}$ algorithm from Fig. 2 to compute *all* PST evaluation proofs, this takes $O(n^2)$ time. Below, we show how to compute all n proofs faster, in $O(n \log n)$ time, by avoiding unnecessary computations (see Fig. 4).

Denote the proof for $f(\mathbf{i})$ as $\pi_i = (\pi_{i,\ell},\ldots,\pi_{i,1})$. Next, observe that if we compute all proofs π_i via n calls to:

$$\mathsf{PST.Prove}(f,\ell,(i_\ell,\ldots,i_1)), \forall i \in [n],$$

they actually all have the same first quotient q_ℓ committed in $\pi_{i,\ell}$! This is because all n $\mathsf{PST.Prove}$ calls initially divide f by $x_\ell - i_\ell$, which actually yields the same quotient, independent of i_ℓ. To see this, recall the MLE decomposition from Eq. 7 and reorganize it in two ways as:

$$\begin{aligned} f &= (1-x_\ell)\cdot f_0 + x_\ell \cdot f_1 \Leftrightarrow \\ f &= (f_1 - f_0)\cdot(x_\ell - 1) + f_1 && (12)\\ &= (f_1 - f_0)\cdot x_\ell + f_0, && (13) \end{aligned}$$

where f_0 is the MLE for the left half $\mathbf{a}_0$ of $\mathbf{a}$ and f_1 is the MLE for the right half $\mathbf{a}_1$ (recall from §2). Since both divisions yield the same $q_\ell = f_1 - f_0$ quotient, all π_i's share the same $\pi_{i,\ell}$ commitment to q_ℓ! We depict this q_ℓ as the root of a *multilinear tree (MLT)* in Fig. 3.

Next, recall that each one of the n $\mathsf{PST.Prove}$ calls recurses on its remainder, which was either f_0 or f_1 (as per Eqs. 12 and 13). Specifically, the first $n/2$ calls for $i \in [0,n/2)$ (i.e., $i_\ell = 0$) recurse on $\mathsf{PST.Prove}(f_0,\ell-1,(i_{\ell-1},\ldots,i_1))$, and the other $n/2$ calls for $i \in [n/2+1,n)$ (i.e., $i_\ell = 1$) recurse on $\mathsf{PST.Prove}(f_1,\ell-1,(i_{\ell-1},\ldots,i_1))$. But by the same argument above, each group of $n/2$ calls returns the same first quotient commitment. For example, for the first group, we have quotient $f_{01} - f_{00}$:

$$f_0 = (f_{01} - f_{00})(x_{\ell-1} - 1) + f_{01} \quad (14)$$

$\mathsf{MLT.Compute}(f,\ell) \to [t_1,\ldots,t_{2^\ell-1}]$:

1. If $\ell = 0$ (i.e., f is a constant), return $\varnothing$.
2. Otherwise, $\forall b \in \{0,1\}$, divide f by $x_\ell - b$, obtaining quotient $f_1 - f_0$ and remainder f_b such that $f = (f_1 - f_0)\cdot(x_\ell - b) + f_b$.
3. Return $\left(g_1^{(f_1-f_0)(\mathbf{s})}, \mathsf{MLT.Compute}(f_0,\ell-1), \mathsf{MLT.Compute}(f_1,\ell-1)\right)$

Figure 4: Computes an MLT in $O(n \log n)$ time consisting of PST evaluation proofs for *all* $f(\mathbf{i})$ w.r.t. an MLE f of $\mathbf{a}$ of size $n = 2^\ell$. In contrast, n naive calls to $\mathsf{PST.Prove}$ would take $O(n^2)$. Recall that f_0 and f_1 are MLEs for the left and right halves of $\mathbf{a}$. Returns the tree stored in preorder in an array.

$$= (f_{01} - f_{00})x_{\ell-1} + f_{00}\,, \tag{15}$$

Similarly, for the second group, the quotient will be $f_{11} - f_{10}$. Both quotients are depicted as the children of the root in Fig. 3. Continuing recursively in this fashion yields our *multilinear tree (MLT)* from Fig. 3. We describe the algorithm for computing it in Fig. 4 and we argue correctness of MLT proofs in the extended version of our paper [56].

3.2 Updates and homomorphism

Updating digests and MLTs. Suppose a_4 changes by δ in our MLT from Fig. 3. Then, by Eq. 4, we know that $\mathbf{a}$'s MLE will change to:

$$f' = f + x_3(1-x_2)(1-x_1)\delta = f + \mathcal{S}_{4,3}(\mathbf{x})\delta$$

But what about the MLT? The following highlighted MLEs from Fig. 3 will be updated to:

$$\begin{aligned} f'_{100} &= f_{100} + \delta \\ f'_{10} &= f_{10} + (1-x_1)\delta \\ f'_{1} &= f_{1} + (1-x_2)(1-x_1)\delta \\ f' &= f + x_3(1-x_2)(1-x_1)\delta \end{aligned}$$

These MLEs changing affect the MLEs along a_4's path. For example, the root MLE $f_1 - f_0$ also changes by the same amount as f_1: i.e., by $+(1-x_2)(1-x_1)\delta$. Furthermore, their corresponding commitments are easy to update via the PST homomorphism. For example, the new root will be $\mathsf{pst}(f_1 - f_0)\cdot g_1^{(1-s_2)(1-s_1)\delta}$. However, note that updating commitments requires knowing $g_1^{(1-s_2)(1-s_1)}$, which is referred to as an *update key*. We delve into this next.

Update keys. Recall that $\mathcal{S}_{u,k}(\mathbf{x})$ is the selector multinomial for position $u \in [0,2^k)$ in an MLE of size 2^k (see Eq. 5). However, to easily reason about updates, it is useful to define $\mathcal{S}_{u,k}$ even when $u \ge 2^k$ as $\mathcal{S}_{u,k} = \mathcal{S}_{u \bmod 2^k,k}$. As explained above, updating the MLT after a_u changes by δ requires some auxiliary information referred to as an *update key* for position u. This consists of commitments to all selector multinomials for u in MLEs of size $1,2,\ldots,2^\ell$:

$$\mathsf{upk}_u = \left\{g_1^{\mathcal{S}_{u,k}(\mathbf{s})} : k \in [0,\ell]\right\} = \left\{\mathsf{upk}_{u,k} : k \in [0,\ell]\right\} \tag{16}$$

Recall that $\mathcal{S}_{u,0}(\mathbf{x}) = 1$, so that $\mathsf{upk}_{u,0} = g_1, \forall u \in [0,2^\ell)$. Then, the MLT commitments $(w_{u,\ell},\ldots,w_{u,1})$ along u's path are updated as:

$$w'_{u,k} = w_{u,k}\cdot(\mathsf{upk}_{u,k-1})^\delta = w_{u,k}\cdot(g_1^{\mathcal{S}_{u,k-1}(\mathbf{s})})^\delta, \forall k \in [\ell] \tag{17}$$

Note that this implies that any proof $\pi_i = (w_{i,\ell},\ldots,w_{i,1})$ can be updated after a change at u: one simply has to identify the "intersection" of u's proof with i's proof and apply the update as above, as if updating a pruned MLT consisting of just π_i. More formally, suppose i and u have the same t most significant bits (i.e., $i_k = u_k, \forall k \in \{\ell,\ell-1,\ldots,\ell-t+1\}$). Then, the updated proof π'_i is initially set to π_i and (partially) updated as:

$$w'_{i,k} = w_{i,k}\cdot(\mathsf{upk}_{u,k-1})^\delta, \forall k \in \{\ell,\ldots,\ell-t\}, 1 \le k \le \ell \tag{18}$$

The digest updates more simply as:

$$\mathsf{pst}(f') = \mathsf{pst}(f)\cdot g_1^{\mathcal{S}_{u,\ell}(\mathbf{s})} = \mathsf{pst}(f)\cdot\left(\mathsf{upk}_{u,\ell}\right)^\delta \tag{19}$$

Lastly, we note that the update keys actually coincide with our public parameters (see Fig. 1).

MLTs are homomorphic. Since our multilinear tree stores an MLE commitment at each node, we observe that the MLT itself is *homomorphic*: the MLT for $\mathbf{a}+\mathbf{b}$ can be obtained by "node-by-node multiplying" $\mathbf{a}$'s MLT with $\mathbf{b}$'s MLT. In other words, every node w in the new MLT is the product of the nodes w in the MLTs for $\mathbf{a}$ and $\mathbf{b}$. Specifically, $\mathsf{pst}(f''_w) = \mathsf{pst}(f_w + f'_w) = \mathsf{pst}(f_w)\mathsf{pst}(f'_w)$, where f_w, f'_w, f''_w denote the MLE stored at node w in the MLT for $\mathbf{a},\mathbf{b}$ and $\mathbf{a}+\mathbf{b}$, respectively. This enables our unstealability construction from §3.4 and has other applications to authenticating data in the *streaming* setting [48].

3.3 Aggregating proofs

Recall that a proof $(w_1,\ldots,w_\ell)$ for a_i in the vector $\mathbf{a}$ of size $n = 2^\ell$ is just an ℓ-sized PST evaluation proof (see §2.2) and verifies as:

$$e(\mathsf{C}/g_1^{a_i}, g_2) = \prod_{k=1}^{\ell} e(w_k, g_2^{s_k - i_k})\,, \tag{20}$$

where C is the digest and $(g_2^{s_k-i_k})_{k\in[\ell]}$ is position i's public *verification key*.

Warm-up: Compressing proofs. It is useful to first discuss compressing a size-ℓ proof for a_i to size $\log \ell$ via the IPA from §2.4. For this, we let $\mathbf{A} = [w_1 \ldots w_\ell]$, $\mathbf{B} = [g_2^{s_1-i_1} \ldots g_2^{s_\ell-i_\ell}]$, $Z = e(\mathsf{C}/g_1^{a_i}, g_2)$ and prove that $(Z,\mathbf{B})$ is in the following language:

$$\mathcal{L}^\ell_{\mathsf{PROD}} = \left\{Z \in \mathbb{G}_T, \mathbf{B} \in \mathbb{G}_2^\ell \;\middle|\; \exists \mathbf{A} \in \mathbb{G}_1^\ell, Z = \langle \mathbf{A},\mathbf{B}\rangle\right\} \tag{21}$$

$\underline{\mathcal{G}_{\mathsf{BATCH}}(1^\lambda, b, \ell) \to (PK, VK)}$: Return $\mathcal{G}_{\mathsf{IPA}}(1^\lambda, b \cdot \ell)$

$\underline{\mathcal{P}_{\mathsf{BATCH}}(PK, (\mathbf{A}_i, \mathbf{B}_i)_{i\in[b]}) \to \pi}$:

1. Let $\mathbf{A} = [\mathbf{A}_1 || \mathbf{A}_2 || \dots || \mathbf{A}_b]$ and $\mathbf{B} = [\mathbf{B}_1 || \mathbf{B}_2 || \dots || \mathbf{B}_b]$
2. Let $C_1 = \langle \mathbf{A}, \mathbf{v} \rangle$ (i.e., 1st component of $\mathsf{CM}(\mathbf{ck}; \mathbf{A}, \mathbf{B}', 1_{\mathbb{G}_T})$)
3. Let $r_i = H(C_1, \mathbf{B}, i) \in \mathbb{Z}_p$ for $i = 1, \dots, b$
4. Let $\mathbf{B}' = [\mathbf{B}_1^{r_1} || \mathbf{B}_2^{r_2} || \dots || \mathbf{B}_b^{r_b}]$
5. Let $\pi^* = \mathcal{P}_{\mathsf{IPA}}(\mathbf{ck}, \mathbf{A}, \mathbf{B}')$ and return $\pi = (C_1, \pi^*)$.

$\underline{\mathcal{V}_{\mathsf{BATCH}}(VK, (\mathbf{B}_i, Z_i)_{i\in[b]}, \pi) \to \{0,1\}}$:

1. Parse the proof $\pi = (C_1, \pi^*)$
2. Let $r_i = H(C_1, \mathbf{B}, i) \in \mathbb{Z}_p$ for $i = 1, \dots, b$
3. Let $\mathbf{B}' = [\mathbf{B}_1^{r_1} || \mathbf{B}_2^{r_2} || \dots || \mathbf{B}_b^{r_b}]$ and $Z' = \prod_{i=1}^{b} Z_i^{r_i}$
4. Let $C_2 = \langle \mathbf{w}, \mathbf{B}' \rangle$ (i.e., 2nd component of $\mathsf{CM}(\mathbf{ck}; \mathbf{A}, \mathbf{B}', 1_{\mathbb{G}_T})$)
5. Let $\mathbf{C} = (C_1, C_2, Z')$ and return $\mathcal{V}_{\mathsf{IPA}}(\mathbf{ck}, \mathbf{C}, \pi^*)$.

Figure 5: Our argument for $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ used to aggregate Hyperproofs. H is a random oracle and $(\mathcal{G}_{\mathsf{IPA}}, \mathcal{P}_{\mathsf{IPA}}, \mathcal{V}_{\mathsf{IPA}})$ is the Bünz et al. IPA from §2.4.

Next, we can use the IPA from §2.4. Specifically, assume our $\mathcal{L}_{\mathsf{PROD}}$ prover and verifier share a commitment key $\mathbf{ck} = (\mathbf{v}, \mathbf{w})$. First, the prover gives $C_1 = \langle \mathbf{A}, \mathbf{v} \rangle$ to the verifier. Second, the verifier computes $C_2 = \langle \mathbf{w}, \mathbf{B} \rangle$ and lets $C_3 = Z$. Thus, the verifier now has a commitment $\mathbf{C} = (C_1, C_2, C_3)$ to $\mathbf{A}, \mathbf{B}$ and Z. Third, the prover simply runs $\mathcal{P}_{\mathsf{IPA}}$ from §2.4 and convinces the verifier that the committed values satisfy $Z = \langle \mathbf{A}, \mathbf{B} \rangle$ and thus that the Hyperproof verifies as per Eq. 20.

Aggregating proofs. Next, we observe that aggregating many proofs $(\pi_1, \dots, \pi_b)$, each for a position p_i in $\mathbf{a}$, reduces to proving membership in $\mathcal{L}^{\ell}_{\mathsf{PROD}}$ *for each* $(Z_i, \mathbf{B}_i)$, where $Z_i = e(\mathsf{C}/g_1^{a_{p_i}}, g_2)$ and $\mathbf{B}_i$ is position p_i's verification key. But doing this naively would result in a large, $O(b \log \ell)$ aggregated proof size. Instead, we seek a more succinct argument for the following new language:

$$\begin{aligned}\mathcal{L}^{b,\ell}_{\mathsf{BATCH}} &= \left\{ (Z_i \in \mathbb{G}_T, \mathbf{B}_i \in \mathbb{G}_2^{\ell})_{i\in[b]} \;\middle|\; ((Z_i, \mathbf{B}_i) \in \mathcal{L}^{\ell}_{\mathsf{PROD}})_{i\in[b]} \right\} \quad (22)\\ &= \left\{ (Z_i \in \mathbb{G}_T, \mathbf{B}_i \in \mathbb{G}_2^{\ell})_{i\in[b]} \;\middle|\; (\exists \mathbf{A}_i \in \mathbb{G}_1^{\ell}, Z_i = \langle \mathbf{A}_i, \mathbf{B}_i \rangle)_{i\in[b]} \right\}\end{aligned}$$

In other words, membership in $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ guarantees that $\forall i \in [b], \exists \mathbf{A}_i$:

$$Z_i = \prod_{j=1}^{\ell} e(A_{i,j}, B_{i,j}), \quad (23)$$

where $A_{i,j}$ is the jth entry of $\mathbf{A}_i$. Note that we cannot use the TIPP argument from [19] to prove membership in $\mathcal{L}_{\mathsf{BATCH}}$, since it can only prove that $\forall i, Z_i = e(X_i, Y_i)$, where $(X_i, Y_i) \in \mathbb{G}_1 \times \mathbb{G}_2$. Instead, we design a new argument for $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ (see Fig. 5) which uses a random linear combination to combine the ℓ-sized equations from above into a single $b\ell$-sized one:

$$\prod_{i=1}^{b} Z_i^{r_i} = \prod_{i=1}^{b} \left(\prod_{j=1}^{\ell} e(A_{i,j}, B_{i,j}) \right)^{r_i} \quad (24)$$

$\underline{\mathsf{Gen}(1^\lambda, n) \to \mathsf{pp}}$: Let $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2) \leftarrow \mathsf{BilGen}(1^\lambda)$. Let $\mathbf{s} = (s_1, \dots, s_\ell) \in_R \mathbb{Z}_p^{\ell}$, where $n = 2^\ell$. Let pp consist of

- $\mathsf{pst}(\mathcal{S}_{j,k}) = g_1^{\mathcal{S}_{j,k}(\mathbf{s})}, \forall k \in [0, \ell], \forall j \in [0, 2^k)$;
- $g_2^{s_k}, \forall k \in [\ell]$;
- $(PK, VK) \leftarrow \mathcal{G}_{\mathsf{BATCH}}(1^\lambda, b, \ell)$.

We refer to $(g_2^{s_k - i_k})_{k\in[\ell]}$ as position i's *verification key*.

$\underline{\mathsf{Com}_{\mathsf{pp}}(\mathbf{a}) \to \mathsf{C}}$: Let $\mathsf{C} = \mathsf{pst}(f) = g_1^{f(\mathbf{s})} = g_1^{f(s_1,\dots,s_\ell)}$, where f is $\mathbf{a}$'s MLE.

$\underline{\mathsf{OpenAll}_{\mathsf{pp}}(\boldsymbol{a}) \to (\pi_0, \dots, \pi_{n-1})}$: Return the MLT as per §3.1.

$\underline{\mathsf{Open}_{\mathsf{pp}}(i, \mathbf{a}) \to \pi_i}$: Compute only the ith path in the MLT and return it.

$\underline{\mathsf{Agg}_{\mathsf{pp}}(I, \{a_i, \pi_i\}_{i\in I}) \to \pi_I}$: Let $m = |I|$ and let $\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_m$ denote proofs $(\pi_i)_{i\in I}$, ordered by i, and $\mathbf{B}_1, \dots, \mathbf{B}_m$ denote their corresponding verification keys. Return $\mathcal{P}_{\mathsf{BATCH}}(PK, (A_k, B_k)_{k\in[m]})$.

$\underline{\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, I, \{a_i\}_{i\in I}, \pi_I) \to \{0,1\}}$: If $I = \{i\}$, parse $\pi_I = (w_1, \dots, w_\ell)$ and ensure that

$$e(\mathsf{C}/g_1^{a_i}, g_2) = \prod_{j=1}^{\ell} e(w_j, g_2^{s_j - i_j}).$$

Otherwise, let $m = |I|$ and $\mathbf{B}_1, \dots, \mathbf{B}_m$ denote the verification keys for the proofs, ordered by their position i. Let $Z_1, Z_2, \dots, Z_m$ be all the $e(\mathsf{C}/g_1^{a_i}, g_2)$'s, also ordered by i. Return $\mathcal{V}_{\mathsf{BATCH}}(VK, (B_k, Z_k)_{k\in[m]}, \pi_I)$.

$\underline{\mathsf{UpdDig}_{\mathsf{pp}}(u, \delta, \mathsf{C}) \to \mathsf{C}'}$: Let $\mathsf{C}' = \mathsf{C} \cdot (g_1^{\mathcal{S}_{u,\ell}(\mathbf{s})})^{\delta}$.

$\underline{\mathsf{UpdProof}_{\mathsf{pp}}(u, \delta, \pi_i) \to \pi_i'}$: Update via $\mathsf{UpdAllProofs}$ (see below) as if π_i was a pruned, single-path MLT (see Eq. 18).

$\underline{\mathsf{UpdAllProofs}_{\mathsf{pp}}(u, \delta, \pi_0, \dots, \pi_{n-1}) \to (\pi_0', \dots, \pi_{n-1}')}$: Assume u's MLT path is $(w_1, \dots, w_\ell)$. Update this path as $w_k' = w_k \cdot (\mathsf{upk}_{u,k-1})^{\delta}$ (for $k = 1, \dots, \ell$) as per Eq. 17.

Figure 6: Algorithms for Hyperproofs, implicitly parameterized by the max number of proofs b that can be aggregated into a single proof.

It is well known that, if the r_i's are uniformly random, verifying the combined equation above is sufficient (see Lemma A.1). As a result, our argument for $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ uses the IPA from §2.4 on this combined equation in a black-box manner. (This is similar to the previous $\mathcal{L}^{\ell}_{\mathsf{PROD}}$ argument, except it involves larger vectors and randomization.) We give a precise description of its $(\mathcal{G}_{\mathsf{BATCH}}, \mathcal{P}_{\mathsf{BATCH}}, \mathcal{V}_{\mathsf{BATCH}})$ algorithms in Fig. 5, show how it fits in our VC construction in Fig. 6, and prove security in App. A.

Aggregation time and proof size. It is easy to see from Fig. 5 that the $\mathcal{P}_{\mathsf{BATCH}}$ time (i.e., the time to aggregate b proofs) is $O(b \cdot \ell)$ and the $\mathcal{V}_{\mathsf{BATCH}}$ time (i.e., the time to verify the aggregated proof) is $O(b \cdot \ell)$. Unfortunately, even though our $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ argument uses the IPA with fast, $O(\log(b \cdot \ell))$-time KZG-based verification (see §2.4), the $\mathcal{V}_{\mathsf{BATCH}}$ verifier still needs to do $O(b \cdot \ell)$ work on the $\mathbf{B}_i$ vectors. (Note that this $O(b \cdot \ell)$ verifier work seems inherent for processing the b verification keys.) Lastly, the argument size (i.e., the aggregated proof size) is $O(\log(b \cdot \ell)) = O(\log b + \log \ell)$.

Cross-aggregation. In addition to aggregating proofs w.r.t. the some digest C, we can also *cross-aggregate* proofs w.r.t. different digests [28]. Specifically, suppose we have b proofs

π_i for positions p_i, each w.r.t. a (potentially-different) digest C_i for a vector with MLE f_i. Then, we can use the same $\mathcal{P}_{\mathsf{BATCH}}$ prover from Fig. 5 to cross-aggregate these proofs. To verify, the verifier now computes the Z_i's given to $\mathcal{V}_{\mathsf{BATCH}}$ by using the right digest and evaluation point: i.e., $Z_i = e(C_i/g_1^{f_i(p_i)}, g_2)$.

3.4 Unstealable proofs

In this subsection, we show how to incentivize proof computation by allowing provers, who store the vector and maintain proofs, to *watermark* the proofs they compute. Such watermarked proofs are cryptographically-bound to their prover's identity, which means the prover can be monetarily rewarded for having computed them (e.g., in cryptocurrencies). Importantly, this cryptographic binding cannot be undone by adversaries. In other words, "stealing" a proof by replacing its watermark with your own, is no easier than computing the proof from scratch like everyone else. We call such watermarked proofs *unstealable*, formalize and prove their security and make Hyperproofs unstealable. We show why and how unstealability is helpful in the cryptocurrency setting in §4. We also envision other applications could benefit from it.

Unstealability goals. First, any vector $\mathbf{a}$ should continue to have a single digest C against which all correct proofs verify, whether proofs are watermarked or not. Put differently, unstealability must work in our previous setting where there is a single Com algorithm for everyone, which does not take the identity of the prover as input. Specifically, only the Open and Ver algorithms are given the identity of the prover to watermark proofs and verify them. This ensures compatibility with stateless cryptocurrencies, where the state must have a single (prover-independent) digest against which (prover-dependent) watermarked proofs can be verified. Second, a prover should still be able to precompute all its (now) unstealable proofs and efficiently maintain them over time as the vector changes. In particular, solutions that require provers to watermark proofs "on the fly" would be too expensive. Third, unstealable proofs should remain aggregatable.

Strawmen. One idea for unstealability is to have each prover commit to the original vector $\mathbf{a}$ but "extended" with its identity id as $\mathbf{a}_{\mathsf{id}} = (a_i||\mathsf{id})_{i\in[n]}$. Unfortunately, this results in having multiple, prover-specific digests C_{id} for $\mathbf{a}$. Another idea is to add a digital signature on the VC proof. However, the signature can simply be removed by the adversary and replaced with their own. A last attempt would be to use a non-malleable SNARK [5] to augment a VC proof with a proof of knowledge of (1) the committed vector and (2) a secret associated with the prover's identity. This would require a stealing adversary to maul the SNARK proof so as to verify for their identity. However, this approach would be too slow and would not preserve maintainability due to the non-malleability of the SNARK.

Unstealability via exponentiations. We make a proof $\boldsymbol{\pi}_i = (w_{i,\ell}, \ldots, w_{i,1})$ unstealable by exponentiating it with α as:

$$\boldsymbol{\pi}_i^{\alpha} = (w_{i,\ell}^{\alpha}, \ldots, w_{i,1}^{\alpha}) \stackrel{\text{def}}{=} (\hat{w}_{i,\ell}, \ldots, \hat{w}_{i,1}), \tag{25}$$

where α is the prover's *watermarking secret key (WSK)*. The corresponding *watermarking public key (WPK)* is g_2^{α} together with a zero-knowledge proof of knowledge (ZKPoK) of α (e.g., a Schnorr proof [53]). To verify a proof watermarked with g_2^{α}, one first checks that the ZKPoK of α verifies and that $\alpha \neq 0$. Second, one checks the proof as normal as per Eq. 20, but accounts for the WPK g_2^{α}:

$$e(\mathsf{C}/g_1^{a_i}, g_2^{\alpha}) \stackrel{?}{=} \prod_{k\in[\ell]} e(\hat{w}_{i,k}, g_2^{s_k - i_k}) \tag{26}$$

The ZKPoK of α is used to prevent stealing by exponentiating $\boldsymbol{\pi}_i^{\alpha}$ with a δ known by the adversary, since the adversary would have to prove knowledge of $\alpha \cdot \delta$. As a result, the adversary's only recourse is to remove α from the watermarked proof, but this seems to require exponentiating by α^{-1}, which the adversary does not know. We prove security in the *algebraic group model (AGM)* [26] in the extended version of our paper [56].

Aggregation-preserving unstealability. One important property of our unstealable proofs is that they remain aggregatable via a call to Agg from Fig. 6. Intuitively, this is because the right-hand side of the watermarked verification from Eq. 26 remains the same as for normal verification in Eq. 20. However, the left-hand side changes. Thus, when verifying an aggregated proof via Ver in Fig. 6, the verifier has to account for the WPKs when computing the Z_i's given to $\mathcal{V}_{\mathsf{BATCH}}$ in Fig. 5. In other words, the verifier needs to have these WPKs. In our application setting from §4, we anticipate the verifier will already have all of the WPKs, instead of receiving them with the aggregated proof.

Homomorphism-preserving unstealability. Our approach to watermarking proofs preserves the PST and MLT homomorphisms. This has a few advantages. First, watermarked proofs can still be updated. Specifically, assuming position u changed by δ, the watermarked proof $\boldsymbol{\pi}_i^{\alpha}$ from Eq. 25 can be updated as before (see Eq. 18) if one uses *watermarked update keys* $(\mathsf{upk}_{u,k})^{\alpha}$. Second, an MLT of watermarked proofs can be computed directly, if the prover uses *watermarked public parameters*. The prover can obtain these in a one-time pre-processing step that exponentiates all parameters from Fig. 1 with the WSK α:

$$\hat{g}_1^{S_{u,k}(\mathbf{s})} \stackrel{\text{def}}{=} \left(g_1^{S_{u,k}(\mathbf{s})}\right)^{\alpha} = g_1^{\alpha S_{u,k}(\mathbf{s})}, \forall k \in [0,\ell], \forall u \in [0, 2^k) \tag{27}$$

Importantly, these are still valid Hyperproofs parameters, except under a new base $\hat{g}_1 = g_1^{\alpha}$. As a result, the prover can directly compute a *watermarked MLT* using these new parameters. This is important, as it allows precomputing watermarked proofs, ensuring that serving such proofs is as efficient as serving normal proofs. Third, a watermarked MLT is efficiently maintainable, just like a normal MLT. This follows from the updatability of watermarked proofs argued above.

New UVC algorithms. We must slightly change our VC API from Def. 2.1 into an *unstealable VC (UVC)* API that accounts for watermarked proofs and watermarking key-pairs. First, we introduce two new algorithms:

1. $\mathsf{WtrmkGen}(1^\lambda) \to (\mathsf{wsk}, \mathsf{wpk})$: Generates a random $(\mathsf{wsk}, \mathsf{wpk})$ watermarking key pair.
2. $\mathsf{WtrmkParams}(\mathsf{pp}, \mathsf{wsk}) \to \mathsf{wpp}$: Returns watermarked public parameters wpp, under wsk, as per Eq. 27.

Second, the algorithm $\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, I, (a_i, \mathsf{wpk}_i)_{i \in I}, \pi_I)$ additionally takes as input the watermarking PK wpk_i that each proof π_i is watermarked with. Third, the algorithms for creating and updating proofs now operate on *watermarked* public parameters. In the interest of brevity, we give the full UVC API, with a new correctness definition, in the extended version of our paper [56].

UVC soundness. We model UVC soundness similar to VC soundness, except we account for watermarked proofs. Informally, we prevent adversaries from creating two inconsistent proofs for the same position k, even if those proofs are watermarked with different, adversarially-generated WPKs (see the extended version [56]).

UVC unstealability. In the extended version of our paper [56], we formalize our notion of unstealability which captures that an adversary cannot compute a watermarked proof on a WPK it knows asymptotically any faster than the Open algorithm, despite having adaptive access to a *watermarking oracle* on arbitrary choices of WPK. We prove that Hyperproofs is unstealable in the *algebraic group model (AGM)* [26].

In particular, we show that an adversary who outputs a new watermarked proof (after having access to the watermarking oracle) and runs asymptotically faster than the time it takes to run Open can neither explicitly include the oracle proofs in the output watermarked proof (or otherwise discrete log is broken) nor use the oracle proofs in any other way to speed up computation (or otherwise q-SDH is broken).

4 Hyperproofs for Cryptocurrencies

In this section, we discuss how Hyperproofs can be used to speed up validation in *payment-only* stateless cryptocurrencies.

Stateless validation. In account-based cryptocurrencies [65], *validators* such as miners and P2P nodes store a large amount of *state* to validate transactions and blocks in the consensus protocol. This state consists of each user's account balance and can be represented as a vector that maps each user's public key to their balance. Recent work [12, 22, 28, 41, 52, 57, 59] trades off storage of the state with additional bandwidth and computation. This approach, known as *stateless validation*, commits to the state using a vector commitment (VC) scheme and allows validators to store only the digest rather than the full state. Next, transactions and blocks are augmented with proofs for the accessed state, so validators can check validity against the digest, instead of storing the full state.

Practical relevance. We believe stateless validation addresses two important problems in cryptocurrencies. First, in smart-contract-based cryptocurrencies, every block validator in the network has to store the full state in order to validate. This leads to a *state explosion* problem [17, 43], which could be ameliorated by having validators store succinct digests of the state. Then, each smart contract owner could store its own state and maintain its VC proofs, as proposed in previous work [28].

Second, in sharded cryptocurrencies, validators have to be frequently shuffled between shards to prevent adversaries from corrupting a majority of validators within a shard [34]. However, shuffling a validator from shard A to shard B requires that validator to download shard B's state. This worsens performance, but could be ameliorated by statelessly validating against a digest of the shard's state. As a result, when moving to shard B, a validator need only download that shard's digest, which is very small.

Challenges. There are several challenges in stateless validation. First, when creating a transaction, the sending user needs to include a proof that they have enough balance. In this sense, users should be able to fetch their proof from *proof-serving nodes (PSNs)* [52, 59], who maintain (a subset of) all proofs. Thus, *PSNs should be able to efficiently update all proofs*, as new blocks are confirmed. Second, *PSNs should be incentivized to maintain proofs*. Third, a miner must now include each transaction's proof in a proposed block, so that other miners can statelessly validate this block. This calls for *proofs to be efficiently aggregatable*, to save block space. Finally, when validating a block, miners must now verify such an aggregated proof. Thus, *aggregated proofs should verify fast*.

Why rely on proof-serving nodes? In theory, each user can maintain their proof locally by keeping up with all confirmed transactions and updating their proof (e.g., as per Eq. 18). However, this overwhelms users with large computation (i.e., updating proofs) and large communication (i.e., downloading new blocks). This is why well-incentivized, efficient proof-serving nodes (PSNs) are important: they eliminate this burden from users by allowing them to fetch their latest proof. We discuss below how unstealability helps implement well-incentivized PSNs.

Hyperproofs for stateless validation. As described above, in the stateless validation setting, it is important to minimize the time for (1) PSNs to update all proofs to reflect

Table 2: Single-threaded microbenchmarks for Hyperproofs. Running times with an asterisk symbol (*) are too long and have been interpolated. We measure aggregation of 1024 proofs. OpenAll and Com are only measured once. UpdDig and UpdAllProofs times are averages after a batch of 1024 changes to the vector. All algorithms are parallelizable.

$\ell = \log_2 n$	**22**	**24**	**26**	**28**	**30**
Com (min)	3.1	12.6	50	201^*	807^*
OpenAll (hrs)	0.7	2.7	12^*	52^*	225^*
UpdDig	47.76 μs				
UpdAllProofs (ms)	1.74	1.96	2.15	2.37	2.58
Indiv. Ver (ms)	8.15	8.22	9.10	10.09	10.93
Agg (s)	105	109	114	118	123
Aggr. Ver (s)	13	14	15	16	17
Indiv. proof size (KiB)	1.06	1.15	1.25	1.34	1.44
Aggr. proof size (KiB)	51.6				

the latest block, (2) miners to propose a new block, with aggregated proofs and (3) validators (i.e., miners and P2P nodes) to verify this block, including its aggregated proof. In §5.3, we show experimentally that Hyperproofs outperforms other VCs in this task. This is because VCs with $O(1)$-sized proofs [20, 21, 28, 37, 59] require $O(n)$ time to update all proofs, while Hyperproofs only requires $O(\log n)$. Furthermore, when compared to Merkle trees, aggregation is $10\times$ to $41\times$ faster in Hyperproofs (see §5.2).

How unstealable proofs help. As highlighted above, proof-serving nodes should be rewarded for the proofs they serve. One approach would be for users to simply pay the PSN *before* they receive their proof. Unfortunately, this is vulnerable to a fair-exchange problem: a malicious PSN will take the payment but not send the proof. An alternative approach would be for PSNs to first serve the proof and expect payment *after*. This approach poses two challenges.

First, we must ensure the payment always goes through. Fortunately, this can be achieved via the cryptocurrency's consensus mechanism. Specifically, the miners can enforce a *PSN fee* whenever a valid PSN proof is included in a block. Second, and most importantly, we must ensure that the payment goes only to the PSN who served the proof. This requires that a proof served by PSN A cannot be mauled to appear as a proof served by (a malicious) PSN B. In other words, PSN B should have no recourse but to compute a proof from scratch like everyone else. Our unstealability design from §3.4 guarantees exactly this property.

5 Evaluation

In this section, we measure the performance of Hyperproofs and explore their applicability for stateless validation. We do not directly compare to VCs with constant-sized proofs due to their impractical $O(n)$ cost to update all proofs (see §5.1). Instead, we focus on Merkle trees with SNARK-based aggregation. We use the Golang bindings of the `mcl` library [42] to implement Hyperproofs[1]. We use BLS12-381, a pairing-friendly elliptic curve, which offers 128 bits of security. A serialized $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ element in `mcl` takes 48, 96, and 576 bytes, respectively. A single exponentiation takes 106 μs in $\mathbb{G}_1$ and 250 μs in $\mathbb{G}_2$. Each experiment ran *single-threaded* on an Intel Core i7-4770 CPU @ 3.40GHz with 8 cores and 32 GiB of RAM. Unless stated otherwise, we perform 4 runs of each experiment and report their average. Also, vectors in this section are of size $n = 2^\ell$.

Table 3: The size of the public parameters from Fig. 1 for various values of $\ell = \log_2 n$. Recall that the *verification key* consists of all selector monomial commitments $g_2^{s_k}, \forall k \in [\ell]$, while the *proving key* consists of all selector multinomial commitents $g_1^{S_{j,k}(\mathbf{s})}, \forall k \in [0,\ell], j \in [0, 2^k)$ (see Fig. 1).

$\ell = \log_2 n$	Verification key	Proving key
22	2.11 KiB	384 MiB
24	2.3 KiB	1.5 GiB
26	2.49 KiB	6 GiB
28	2.68 KiB	24 GiB
30	2.88 KiB	96 GiB

5.1 Microbenchmarks

We microbenchmark Hyperproofs in Table 2. All microbenchmarks pick vectors and updates randomly and are *single-threaded*, but trivially parallelizable.

Public parameters. To commit to vectors of size n, Hyperproofs needs a large *proving key* consisting of $2n-1$ $\mathbb{G}_1$ elements depicted in Fig. 1. For $\ell = 28$, this requires around 24 GiB of space (see Table 3). *Verification keys* are all derived from $(g_2^{s_k})_{k\in[\ell]}$. Furthermore, to aggregate b proofs, Abe et al. commitment keys [2] are needed consisting of $\ell \cdot b$ $\mathbb{G}_1$ and $\ell \cdot b$ $\mathbb{G}_2$ elements. For $\ell = 28$ and $b = 1024$, this only adds 3.94 MiB. Watermarking the public parameters as per §3.4 requires $2n-1$ exponentiations in $\mathbb{G}_1$. For $\ell = 28$, this takes 15.87 hours. However, this is a one-time cost.

Committing and computing multilinear trees. We commit to a vector of size n via an $O(n)$-sized multi-exponentiation. For $\ell = 28$, this takes 202 minutes. Computing a multilinear tree (MLT) involves committing to the MLEs in each node via a multi-exponentiation (see Fig. 3). For $\ell = 28$, this takes 52.2 hours (or 1.63 hours with 32 threads). We expect to at least double performance via faster multi-exponentiation algorithms, which `mcl` lacks.

Updating the digest and the multilinear tree. For updating the digest, we measure the time to apply a batch of 1024 updates via a multi-exponentiation, divide this time by 1024

[1] Our code is available at: https://github.com/hyperproofs/hyperproofs

and obtain an average time of 48 μs per update. For the MLT, we also measure the time to apply a batch of 1024 updates. This way, we can use multi-exponentiations when updating nodes in the tree. Dividing the total time by 1024, gives us an average time of 1.74 ($\ell = 22$) to 2.58 milliseconds ($\ell = 30$) per update. Recall from §3.4 that updates will be just as fast for watermarked multilinear trees.

Proof size and verification time. Individual proof size is ℓ $\mathbb{G}_1$ elements and is competitive with Merkle trees (e.g., for $\ell = 30$, 1.44 KiB in MLTs versus 960 bytes in MHTs). Verifying a proof requires $\ell + 1$ pairings, which we optimize into a multi-pairing (i.e., first compute $\ell + 1$ Miller loops and then compute a single final exponentiation). This way, verifying a proof ranges from 8.2 ($\ell = 22$) to 11 milliseconds ($\ell = 30$). If the proof is watermarked, we discount the WPK from the proof size, since the verifier could already have the WPK, depending on the application. Furthermore, this overhead would be acceptable: 224 bytes. Lastly, verifying the ZKPoK for the WPK requires two $\mathbb{G}_2$ exponentiations, which adds around 500 μs to the proof verification time.

Proof aggregation. Let I be the set of transactions to be aggregated via $\mathsf{Agg}_{\mathsf{pp}}$, which calls $\mathcal{P}_{\mathsf{BATCH}}$ from Fig. 5. In our benchmarks, $b = |I| = 1024$. As shown in Table 2, aggregating 1024 transactions takes between 105 ($\ell = 22$) to 123 seconds ($\ell = 30$). Verifying such an aggregated proof takes between 13 ($\ell = 22$) to 17.5 ($\ell = 30$) seconds. These times are not affected by watermarking. In §5.2, we show our aggregation is 10× to 41× faster than SNARKs.

Aggregated proof size. Our aggregated proof size is 52 KiB for any $\ell = 22,\ldots,30$. This is an artifact of the IPA proof size depending on the smallest power of two $\geq \log(b \cdot \ell)$, which is the same for all ℓ's above when $b = 1024$. As with individual proofs, watermarking does not affect proof size when the verifier has the WPKs.

Comparison with Pointproofs. One of the main advantages over aggregatable VCs with constant-sized proofs such as Pointproofs is that Hyperproofs are *maintainable*. For example, in Pointproofs, updating all $n = 2^{\ell}$ proofs involves n exponentiations, taking 31.7 hours for $\ell = 30$. Importantly, multi-exponentiations cannot be used here. In contrast, Hyperproofs only takes 3.2 milliseconds. (Unlike the numbers from Table 2, these numbers assume no batching.)

5.2 Comparison with SNARKs

In this subsection, we show that Hyperproof aggregation is anywhere from 10× to 41× faster than Merkle proof aggregation via SNARKs (see Fig. 7), depending on the choice of hash function. This comes at the cost of larger proofs (52 KiB versus 192 bytes) and slower verification. Nonetheless, the end-to-end time to aggregate-and-verify remains around 10× to 41× faster in Hyperproofs. We find this design trade-off to be a good one for stateless cryptocurrencies where, although fast verification is important, aggregation times cannot be too high (see §5.3).

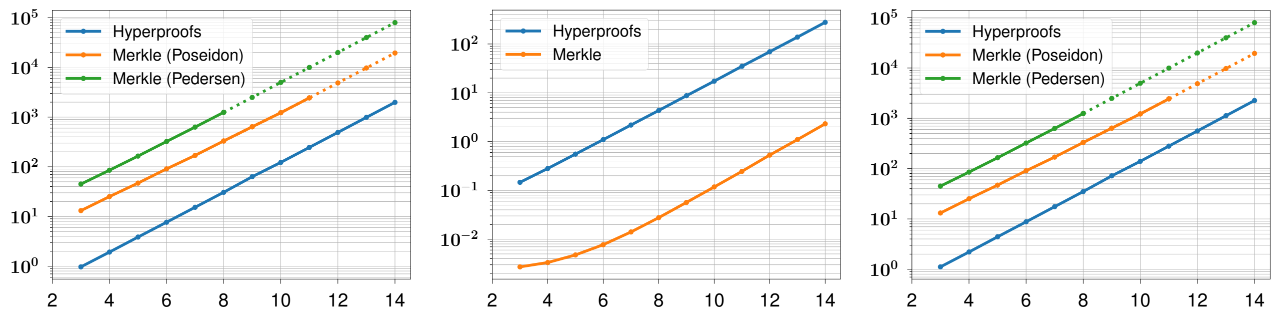
Experimental setting. We fix the height of both the Merkle tree and our MLT to $\ell = 30$, and measure performance when aggregating $b \in \{2^2, 2^4, \ldots, 2^{14}\}$ proofs. We compare to an implementation by Ozdemir *et al.* [45] in Rust [44] which uses the state-of-the-art SNARK by Groth [30] to prove *knowledge* of changes to a Merkle tree, updating it from digest d to digest d'. To benchmark proof aggregation, we notice that proof aggregation would involve half of the work done by the Ozdemir *et al.* prover, and directly use their code. This is because proving knowledge of k changes involves first verifying k Merkle proofs for the original values "inside the SNARK" and then updating the Merkle root with the changes, which also involves k Merkle path verifications. For the SNARK verifier, we directly measure its work, which involves an $O(b)$-sized $\mathbb{G}_1$ multi-exponentiation and 3 pairings.

Choice of Merkle hash function. Choosing a "SNARK-friendly" hash function for the Merkle tree can significantly reduce the prover time. In this sense, we use the recently-proposed Poseidon-128 hash function [29], which only requires 316 R1CS constraints per invocation inside the SNARK, but lacks sufficient cryptanalysis. As a more secure choice, we also use the Pedersen hash function [66] used in Zcash [7], which is collision-resistant under the hardness of discrete log, but requires 2753 constraints per invocation [45].

Proving time. The SNARK prover time is dominated by several multi-exponentiations and Fast Fourier Transforms (FFTs) of size linear in the number of R1CS constraints. For example, aggregating $b = 2^{10}$ proofs in a Poseidon-hashed Merkle tree of height $\ell = 30$, involves 10 million constraints. As a result, SNARK aggregation is very slow, taking 1224 seconds. In contrast, when aggregating b Hyperproofs, also in a height ℓ MLT, our IPA-based prover from Fig. 5 only computes $O(b\ell)$ pairings and $O(b\ell)$ $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ exponentiations. This only takes 123 seconds. On average, as shown in Fig. 7(a), aggregating Hyperproofs is 10× faster than aggregating Merkle-Poseidon proofs and 41× faster than Merkle-Pedersen.

Prover memory. The SNARK prover also requires memory at least linear in the number of constraints. As a result, on our machine with 32 GiB of RAM, SNARK aggregation runs out of memory when aggregating $\geq 2^{11}$ proofs with Poseidon hashing (20 million constraints) or $\geq 2^8$ proofs with Pedersen (23 million constraints). Nonetheless, we extrapolate the proving times in Fig. 7. In contrast, our IPA-based aggregation from Fig. 5 has a much lower memory footprint and never runs out of memory.

Verification time. In general, verifying a SNARK proof requires 3 pairings and a $\mathbb{G}_1$ multi-exponentiation of size equal to the number of verifier-provided inputs [30]. In particular, when aggregating b Merkle proofs, this multi-exponentiation



(a) Aggregation time (seconds) (b) Verification time (seconds) (c) Aggregation plus verification time (seconds)

Figure 7: SNARK-based Merkle proof aggregation versus Hyperproof aggregation. The x-axis is $\log_2$(# of proofs being aggregated). Dotted lines are extrapolated, due to the SNARK prover running out of memory. We use the 128-bit secure variant of Poseidon.

will be of size $2b+1$, since the verifier must input the digest and the b leaves $(i, v_i)_{i \in I}$ being verified. We implement verification in Golang using `mcl` [42] and report the times in Fig. 7(b). (We cannot use the Ozdemir et al. code, since the verifier only inputs two digests and checks *knowledge* of b changes to the Merkle tree.) When aggregating $b = 2^{10}$ proofs, it takes 0.11 seconds to verify a SNARK proof and 17.4 seconds to verify an aggregated Hyperproof. While verification is slower in Hyperproofs, when accounting for both the time to prove and verify in Fig. 7(c), Hyperproofs are faster.

SNARKs without trusted setup. Recent SNARKs [18, 55, 67] are *transparent* (i.e., do not need a trusted setup). Even better, these SNARKs often have faster provers than pairing-based SNARKs. However, compared to Hyperproofs, they are still too slow, have larger proof sizes and consume too much memory. For example, aggregating $b = 2^{14}$ Merkle proofs requires 2^{28} R1CS constraints if using Poseidon hashes. The prover time would be around 2.58 hours using Spartan [55, Figure 7] and 1.53 hours using Virgo [67]. This is close to $5\times$ and $3\times$ slower than Hyperproofs, respectively. The proof size would be around 1.83 MiB using Spartan and 350 KiB using Virgo (estimated using the open-source code of [67]). This is around $36\times$ and $7\times$ bigger than Hyperproofs, respectively. The performance is even worse with Pedersen hashes. Moreover, these transparent SNARKs are not as memory-efficient as Hyperproofs: Virgo scales to 2^{24} constraints, similar to pairing-based SNARKs (i.e., fails aggregating when $b \geq 2^{11}$ proofs) while Spartan scales to 2^{26} constraints (i.e., fails for $b \geq 2^{13}$). Lastly, other transparent arguments (e.g., STARKs [9], Aurora [10], Hyrax [62], Ligero [4]) have similar drawbacks. We defer to [55, 67] for a detailed discussion on trade-offs.

5.3 Macrobenchmarks

Our *single-threaded* experiments measure the VC-induced overheads of statelessly reaching consensus on a new block, as discussed in §4. This consists of three measurements. First, the *block proposal* time (**P**) to verify individual proofs, aggregate them into a new block and update the digest. Second, the *block validation* time (**V**) to verify the aggregated proof and the updated digest in this new block, as it propagates through the P2P network. In particular, we assume the P2P network has diameter $h = 20$. Third, the *proof maintenance* time (**M**) for a proof-serving node (PSN) to update all proofs after applying the updates from this new block, so that the next proposed block can use these proofs.

We estimate the VC overhead as $\mathbf{P} + (h \cdot \mathbf{V}) + \mathbf{M}$ and summarize our results in Table 4. Note that we account for P2P nodes not forwarding a block before validating it by multiplying **V** by h. Overall, Hyperproofs' overhead is $10\times$ smaller than Poseidon-hashed Merkle trees and $41\times$ smaller than Pedersen-hashed. This is because Merkle-based stateless validation involves a slower, more complex SNARK prover (discussed below). Furthermore, Hyperproofs remain competitive in terms of proof maintenance cost (**M**).

Setting: We assume MLTs and Merkle trees of height $\ell = 30$ and blocks of 1024 transactions. We do not compare to VCs with $O(1)$-sized proofs, due to their large proof maintenance cost (i.e., 2^ℓ $\mathbb{G}_1$ exponentiations, or 31.7 hours).

Limitations: Our macrobenchmarks do not account for all the subtleties that would arise in a full prototype, such as communication overheads, or miners needing to update the proofs in the current block they are working on due to another competing block. They also do not account for the overhead of signature verification, which is not affected by the chosen VC scheme. Instead, they focus on the three key operations whose overheads should be minimized: block proposal, block validation and proof maintenance. Lastly, while we show Hyperproofs are faster than other VCs for stateless validation, we do not claim they make the stateless setting practical.

Block transitions with Hyperproofs versus Merkle trees. In a stateless cryptocurrency, the ith block stores the digest d_i of all users' balances at that point in time. When block $i+1$ arrives, it must prove that its new digest d_{i+1} correctly reflects the updated balances, after applying its transactions. With Hyperproofs, the block includes an aggregated proof for the balance of each user sending money. This way, a validator can ensure that a block is spending valid coins and then can compute d_{i+1} from d_i via UpdDig, subtracting coins from each sending user's account and adding coins to each receiving user.

With SNARK-based Merkle trees, it is not possible to update the digest d_{i+1} given the old digest d_i, the SNARK aggregation proof, and the changes in balances: the Merkle proofs for all the changed leaves are also needed as auxiliary information. But including these Merkle proofs in the block would defeat the point of aggregating them via SNARKs! Therefore, the SNARK circuit must be extended to also verify the transition between d_i and d_{i+1}. Specifically, the circuit additionally proves that d_{i+1} is obtained by applying the changes in the block to d_i. A block of b transactions involves $2b$ changes to the Merkle tree, and each change requires two Merkle path verifications inside the circuit. Therefore, the circuit involves $4 \cdot b$ Merkle path verifications ($4\times$ more expensive than the aggregation circuit from §5.2).

Block overhead. As described above, stateless cryptocurrency blocks additionally store the digest of the state and an aggregated proof for all transactions. Both Merkle trees and Hyperproofs have similar digest sizes (i.e., 32 bytes versus 48 bytes). However, aggregated Hyperproofs are 52 KiB, whereas SNARK-aggregated Merkle proofs are only 192 bytes. Nonetheless, relative to the size of the full block, Hyperproof overhead is modest and only decreases with larger blocks. Furthermore, we foresee optimizing the IPA from Fig. 8 to reduce the proof size. Lastly, using unstealability to incentivize proof-serving nodes (which Merkle trees do not support) adds 224 bytes of WPK overhead for each PSN involved in the block. As an alternative, if the set of PSNs is fixed or grows slowly, then WPKs can be stored as part of the public parameters of the system.

Transaction overhead. Transactions propagating through the P2P network in a stateless cryptocurrency need to include proofs. With Hyperproofs, this only requires a 1.44 KiB proof for the sender's balance. With Merkle trees, whether Poseidon- or Pedersen-hashed, this requires two 960 byte proofs, or 1.875 KiB, one for the sender and one for the receiver. This is because, to update the Merkle root, the miner also needs the receiver's Merkle proof as auxiliary information, whereas in Hyperproofs the digest can be updated homomorphically.

Block proposal. With Hyperproofs, a miner proposing a block with 1024 transactions has to (1) verify 1024 individual Hyperproofs, (2) aggregate these proofs, (3) and update the digest. With Merkle trees, this remains the same, except steps (2) and (3) are done in the SNARK prover. Table 4 shows block proposal is $36\times$ (Poseidon) to $149\times$ (Pedersen) faster in Hyperproofs than in SNARKs due to faster aggregation/digest updates.

Block validation. To validate an incoming block, a miner has to verify its aggregated proof and check its digest was computed correctly via UpdDig. In Table 4, we see that SNARKs are $97\times$ faster to verify than an aggregated Hyperproof of $b = 1024$ proofs, which require $O(b\ell)$ $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ exponentiations to verify. While SNARK verification also incurs $O(b)$ cost, this only involves a fast $\mathbb{G}_1$ multi-exponentiation. Nonetheless, when considering the time to propose *and* validate a block ($\mathbf{P} + h \cdot \mathbf{V}$), Hyperproofs remains $10\times$ (Poseidon) to $41\times$ (Pedersen) faster.

Table 4: Single-threaded, stateless cryptocurrency macrobenchmarks that measure the time to prepare a block for proposal (**P**), to validate a proposed block (**V**) and to update all proofs (**M**) after a new block is seen. A block propagates through a P2P network of diameter $h = 20$. Trees have height $\ell = 30$ and blocks have 1024 transactions. A Poseidon-128 hash takes 113 μs using the `go-iden3-crypto` library [27]. A Pedersen hash takes 37 μs using the `sapling-crypto` library [50].

Scheme	Hyperproofs	Merkle w/ Poseidon	Merkle w/ Pedersen
Block proposal (**P**)	2.23 min	81 min	332 min
Block validation (**V**)	17.5 sec	0.18 sec	0.18 sec
Proof maintenance (**M**)	5.14 sec	4.7 sec	1.54 sec
Total ($\mathbf{P} + (h \cdot \mathbf{V}) + \mathbf{M}$)	8 min	81 min	332 min

Proof maintenance. Recall that having updated proofs ready to be served is important in stateless cryptocurrencies, since users need to fetch and include their proofs when sending a transaction to a miner. Fortunately, a PSN can update all proofs in $O(\ell)$ time in both Hypeproofs and Merkle trees. Table 4 gives the concrete *batch* update time after 1024 transactions (or 2048 changes to the tree). Batch-updating Merkle trees is slightly faster than applying each update sequentially, because each node in the Merkle tree need only be updated once, by recomputing a hash (i.e., 113 μs for Poseidon and 37 μs for Pedersen). In contrast, when batch-updating MLTs, each node still needs to be updated several times to account for all the leaves that changed underneath it, as per Eq. 18. While we optimize this using a multi-exponentiation, MLTs will be slightly slower.

6 Discussion

Selective versus adaptive security for PST commitments. Papamanthou et al. prove security under ℓ-SDH (see Assum. A.1), but only in a *selective* sense. Specifically, the (selective) adversary, whose goal is to equivocate about $f(\mathbf{i})$, must first decide on an $\mathbf{i}$ and reveal it to the challenger [47, Appendix C.1]. In contrast, we prove adaptive security for any point on the Boolean hypercube. Specifically, the (adaptive) adversary reveals nothing to the challenger about the point $\mathbf{i}$ it equivocates on and, in our security proof, the reduction simply "guesses" this $\mathbf{i}$ (see Thm. B.1). One negative consequence of this guessing is a security loss of $\log_2 n$ bits, which we hope to address in future work.

Large public parameters. Hyperproofs for vectors of size n require $O(n)$-sized public parameters (see §2.2) which must be generated via a *trusted setup*. In practice, this setup would have to be implemented securely as a multi-party computation (MPC) protocol [8, 14, 15]. Recently, cryptocurrency projects have demonstrated the viability of this approach at the scale of $n \approx 2^{27}$ [13, 25, 63]. We hope to scale these techniques to $n \approx 2^{30}$ in future work. Alternatively, large public parameters can be avoided by splitting a large vector up into k chunks and committing to each chunk. This saves a factor of k in the size of the public parameters but leads to a k-sized digest. Importantly, one can still aggregate proofs in such a chunked vector since Hyperproofs support cross-aggregation. Lastly, Hyperproofs can be modified to work in a decentralized setting where the trapdoor $\mathbf{s} = (s_1, \ldots, s_\ell)$ is secret-shared among a set of servers, similar to recent work for bilinear accumulators [31]. This precludes the need to generate $O(n)$ public parameters and could be useful for applications in the permissioned setting.

Choice of public parameters. The most popular account-based cryptocurrency, Ethereum, currently has less than 185 million accounts. Thus, it would be sufficient to set $\ell = 28$ in Hyperproofs. Furthermore, once Ethereum's consensus layer will partition accounts over 64 shards [23], a smaller $\ell = 22$ would suffice. To determine the maximum number of aggregated proofs b, we need only consider the maximum number of transactions in a block. For example, to handle $2\times$ more than the average number of transactions in a Bitcoin block, setting $b = 4000$ is more than adequate.

Future work. It would be interesting to apply our aggregation and unstealability techniques to *Verkle trees* [36, 39], which are q-ary rather than binary Merkle trees. This would also help extend Hypeproofs into a key-value commitment (KVC) scheme that maps arbitrary keys to values. Building Hyperproofs from assumptions in hidden-order groups would eliminate the large public parameters and, potentially, the trusted setup. Using more malleable inner-product arguments would allow us to update aggregated proofs too. Lastly, optimizing the arguments from Figs. 5 and 8 for our Hyperproof setting could speed up aggregation and verification times as well as reduce proof size.

Acknowledgements

The authors are incredibly thankful to Julian Loss for helping us understand the subtle differences between the generic group model and the algebraic group model. This research was supported in part by Ergo Platform, the National Science Foundation, VMware, the Ethereum Foundation and Protocol Labs.

References

[1] Masayuki Abe, Georg Fuchsbauer, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. Structure-Preserving Signatures and Commitments to Group Elements. In *Advances in Cryptology – CRYPTO 2010*, 2010.

[2] Masayuki Abe, Georg Fuchsbauer, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. Structure-Preserving Signatures and Commitments to Group Elements. *Journal of Cryptology*, 29(2), 2016.

[3] Shashank Agrawal and Srinivasan Raghuraman. KVaC: Key-Value Commitments for Blockchains and Beyond. In *Advances in Cryptology – ASIACRYPT 2020*, 2020.

[4] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[5] Shahla Atapoor and Karim Baghery. Simulation Extractability in Groth's zk-SNARK. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, 2019.

[6] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. In *Selected Areas in Cryptography*, 2006.

[7] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, 2014.

[8] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza. Secure Sampling of Public Parameters for Succinct Zero Knowledge Proofs. In *2015 IEEE Symposium on Security and Privacy*, 2015.

[9] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. https://ia.cr/2018/046.

[10] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent Succinct Arguments for R1CS. In *Advances in Cryptology – EUROCRYPT 2019*, 2019.

[11] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *International Conference on the Theory and Applications of Cryptographic Techniques*, 2004.

[12] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. In *CRYPTO'19*, 2019.

[13] Sean Bowe and Jason Davies. Conclusion of the Powers of Tau Ceremony, 2018. https://www.zfnd.org/blog/conclusion-of-powers-of-tau/.

[14] Sean Bowe, Ariel Gabizon, and Matthew D. Green. A Multi-party Protocol for Constructing the Public Parameters of the Pinocchio zk-SNARK. In *Financial Cryptography and Data Security*, 2019.

[15] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model, 2017. https://ia.cr/2017/1050.

[16] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Recursive Proof Composition from Accumulation Schemes. In *Theory of Cryptography*, 2020.

[17] Vitalik Buterin. A Theory of Ethereum State Size Management, 2021. https://hackmd.io/@vbuterin/state_size_management.

[18] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *Proceedings of the Symposium on Security and Privacy (SP), 2018*, volume 00, 2018.

[19] Benedikt Bünz, Mary Maller, Pratyush Mishra, Nirvan Tyagi, and Psi Vesely. Proofs for Inner Pairing Products and Applications. Cryptology ePrint Archive, Report 2019/1177, 2019. https://ia.cr/2019/1177.

[20] Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. Incrementally Aggregatable Vector Commitments and Applications to Verifiable Decentralized Storage. In *Advances in Cryptology – ASIACRYPT 2020*, 2020.

[21] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *Public-Key Cryptography - PKC 2013*, 2013.

[22] Thaddeus Dryja. Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set, 2019. https://ia.cr/2019/611.

[23] Ethereum. Shard chains, 2022. https://ethereum.org/en/upgrades/shard-chains/.

[24] Amos Fiat and Adi Shamir. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology — CRYPTO' 86*, 1987.

[25] Filecoin. Trusted Setup Complete!, 2020. https://filecoin.io/blog/posts/trusted-setup-complete/.

[26] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The Algebraic Group Model and its Applications. In *Advances in Cryptology – CRYPTO 2018*, 2018.

[27] go-iden3 crypto. go-iden3-crypto, 2020. https://github.com/iden3/go-iden3-crypto/.

[28] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating Proofs for Multiple Vector Commitments. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, 2020.

[29] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[30] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2016.

[31] Lukas Helminger, Daniel Kales, Sebastian Ramacher, and Roman Walch. Multi-party Revocation in Sovrin: Performance through Distributed Trust. In *Topics in Cryptology – CT-RSA 2021*, 2021.

[32] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In *ASIACRYPT'10*, 2010.

[33] Jonathan Katz, Rafail Ostrovsky, and Michael O. Rabin. Identity-Based Zero-Knowledge. In *Security in Communication Networks*, 2005.

[34] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *IEEE S&P'18*, 2018.

[35] Johannes Krupp, Dominique Schröder, Mark Simkin, Dario Fiore, Giuseppe Ateniese, and Stefan Nürnberger. Nearly optimal verifiable data streaming. In *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography*, 2016.

[36] John Kuszmaul. Verkle Trees: V(ery short M)erkle Trees. In *MIT PRIMES Conference '18*, 2018.

[37] Russell W. F. Lai and Giulio Malavolta. Subvector commitments with application to succinct arguments. In *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference*, 2019.

[38] J. Lee, K. Nikitin, and S. Setty. Replicated state machines without replicated execution. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[39] Benoît Libert and Moti Yung. Concise Mercurial Vector Commitments and Independent Zero-Knowledge Sets with Short Proofs. In *TCC'10*, 2010.

[40] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *CRYPTO '87*, 1988.

[41] Andrew Miller. Storing UTXOs in a balanced Merkle tree (zero-trust nodes with O(1)-storage), 2012. https://bitcointalk.org/index.php?topic=101734.msg1117428.

[42] Mitsunari Shigeo. mcl: a portable and fast pairing-based cryptography library. https://github.com/herumi/mcl/, 2015. Accessed: 2020-10-14.

[43] Nervos Network. How Nervos is Tackling the State Explosion Problem Facing Smart Contract Blockchains, 2021. https://medium.com/nervosnetwork/how-nervos-is-tackling-the-state-explosion-problem-facing-smart-contract-blockchains-a9acc4c5708e.

[44] Alex Ozdemir. bellman-bignat, 2020. https://github.com/alex-ozdemir/bellman-bignat.

[45] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. Scaling Verifiable Computation Using Efficient Set Accumulators. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[46] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of Correct Computation. In *TCC'13*.

[47] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of Correct Computation, 2011. https://ia.cr/2011/587.

[48] Charalampos Papamanthou, Elaine Shi, Roberto Tamassia, and Ke Yi. Streaming Authenticated Data Structures. In *EUROCRYPT 2013*, 2013.

[49] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013*, 2013.

[50] Electric Coin Company Prototypes and Experiments. sapling-crypto, 2021. https://github.com/zcash-hackworks/sapling-crypto.

[51] Yi Qian, Yupeng Zhang, Xi Chen, and Charalampos Papamanthou. Streaming Authenticated Data Structures: Abstraction and Implementation. In *ACM CCSW'14*, 2014.

[52] Leonid Reyzin, Dmitry Meshkov, Alexander Chepurnoy, and Sasha Ivanov. Improving Authenticated Dynamic Dictionaries, with Applications to Cryptocurrencies. In *FC'17*, 2017.

[53] C. P. Schnorr. Efficient Identification and Signatures for Smart Cards. In *Advances in Cryptology — CRYPTO' 89 Proceedings*, 1990.

[54] Jacob T Schwartz. Probabilistic algorithms for verification of polynomial identities. In *International Symposium on Symbolic and Algebraic Manipulation*, 1979.

[55] Srinath Setty. Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup. In *Advances in Cryptology – CRYPTO 2020*, 2020.

[56] Shravan Srinivasan, Alex Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang. Hyperproofs: Aggregating and Maintaining Proofs in Vector Commitments. Cryptology ePrint Archive, Report 2021/599, 2021. https://ia.cr/2021/599.

[57] Peter Todd. Making UTXO set growth irrelevant with low-latency delayed TXO commitments, 2016. https://petertodd.org/2016/delayed-txo-commitments.

[58] Alin Tomescu. *How to Keep a Secret and Share a Public Key (Using Polynomial Commitments)*. PhD thesis, Massachusetts Institute of Technology, 2020.

[59] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable Subvector Commitments for Stateless Cryptocurrencies. In *Security and Cryptography for Networks*, 2020.

[60] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan Gueta, and Srinivas Devadas. Towards Scalable Threshold Cryptosystems. In *IEEE S&P'20*, 2020.

[61] Alin Tomescu, Yu Xia, and Zachary Newman. Authenticated Dictionaries with Cross-Incremental Proof (Dis)aggregation. Cryptology ePrint Archive, Report 2020/1239, 2020. https://ia.cr/2020/1239.

[62] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. Doubly-Efficient zkSNARKs Without Trusted Setup. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[63] Thomas Walton-Pocock. AZTEC CRS: The Biggest MPC Setup in History has Successfully Finished, 2020. https://medium.com/aztec-protocol/aztec-crs-the-biggest-mpc-setup-in-history-has-successfully-finished-74c6909cd0c4.

[64] Benjamin Wesolowski. Efficient Verifiable Delay Functions. In *Advances in Cryptology – EUROCRYPT 2019*, 2019.

[65] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. https://gavwood.com/paper.pdf, 2014.

[66] Zcash. What is jubjub?, 2017. https://z.cash/technology/jubjub/.

[67] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *IEEE S&P 2020*, 2020.

[68] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vRAM: Faster verifiable ram with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00.

[69] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vSQL: Verifying arbitrary sql queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[70] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *International Symposium on Symbolic and Algebraic Manipulation*, 1979.

A Assumptions, definitions and primitives

We prove Hyperproofs satisfy soundness, as per Def. B.2, under *q-Strong Diffie-Hellman (q-SDH)* assumption, defined below.

Assumption A.1 (q-SDH [11])**.** *For any* PPT *adversary* $\mathcal{A}$,

$$\Pr\left[\begin{array}{c}(p,\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,e,g_1,g_2)\leftarrow \mathsf{BilGen}(1^\lambda), s\in_R\mathbb{Z}_p^*,\\ \mathsf{pp}=((p,\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,e,g_1,g_2),g_2^s,g_1^s,\dots,g_1^{s^q}):\\ (a,g_1^{\frac{1}{s+a}})\leftarrow\mathcal{A}(1^\lambda,\mathsf{pp})\end{array}\right]\le \mathsf{negl}(\lambda)$$

Inner Product Arguments (IPA). We give the *interactive* variant of the Bünz et al. [19] IPA in Fig. 8. Here, the prover interacts with the verifier over $\log m$ rounds. This interactive IPA is knowledge-sound assuming Abe et al. commitments are binding [19, Theorem 1]. It is made non-interactive via the Fiat-Shamir transform [24] and proved secure in a new *algebraic commitment model* and in the *random oracle model (ROM)* [19, Appendix D.2].

Faster verifier. As described Fig. 8, the prover and verifier take $O(m)$ time and the proof size is $|\pi| = O(\log m)$, if the argument is made non-interactive via the Fiat-Shamir transform [24]. However, Bünz et al. show how to reduce the verifier time by using a "structured" commitment key $\mathbf{ck} = (\mathbf{v},\mathbf{w})$, similar to a q-SDH common reference string (see Assum. A.1). This allows the verifier to outsource the computations

$\mathcal{P}_{\mathsf{IPA}}^{\leftrightarrow}(\mathbf{ck}=(\mathbf{v},\mathbf{w});\mathbf{A},\mathbf{B})$: $\quad$ $\mathcal{V}_{\mathsf{IPA}}^{\leftrightarrow}(\mathbf{ck}=(\mathbf{v},\mathbf{w});\mathbf{C})\rightarrow\{0,1\}$:

If $m=1$

$\xrightarrow{\mathbf{A}=[A_1],\mathbf{B}=[B_1]}$

Return 1 iff.
$\mathbf{C}\stackrel{?}{=}\mathsf{CM}(\mathbf{ck};[A_1],[B_1],e(A_1,B_1))$

Else (i.e., **if** $m\geq 2$)

$Z_L=\langle\mathbf{A}_R,\mathbf{B}_L\rangle$
$\mathbf{C}_L=\mathsf{CM}(\mathbf{ck};\mathbf{A}_R||1_{\mathbb{G}_1},1_{\mathbb{G}_2}||\mathbf{B}_L,Z_L)$
$Z_R=\langle\mathbf{A}_L,\mathbf{B}_R\rangle$
$\mathbf{C}_R=\mathsf{CM}(\mathbf{ck};1_{\mathbb{G}_1}||\mathbf{A}_L,\mathbf{B}_R||1_{\mathbb{G}_2},Z_R)$

$\xrightarrow{\mathbf{C}_L,\mathbf{C}_R}$

$\xleftarrow{x\in_R\mathbb{Z}_p}$

$\mathbf{A}'=\mathbf{A}_L\circ(\mathbf{A}_R)^x$
$\mathbf{v}'=\mathbf{v}_L\circ(\mathbf{v}_R)^{(x^{-1})}$
$\mathbf{B}'=\mathbf{B}_L\circ(\mathbf{B}_R)^{(x^{-1})}$ $\quad$ Computes $\mathbf{v}',\mathbf{w}'$ just like the prover
$\mathbf{w}'=\mathbf{w}_L\circ(\mathbf{w}_R)^x$ $\quad$ $\mathbf{C}'=(\mathbf{C}_L)^x\circ\mathbf{C}\circ(\mathbf{C}_R)^{(x^{-1})}$
Recurse on $(\mathbf{ck}'=(\mathbf{v}',\mathbf{w}'),\mathbf{A}',\mathbf{B}')$ $\quad$ **Recurse** on $(\mathbf{ck}'=(\mathbf{v}',\mathbf{w}'),\mathbf{C}')$

Figure 8: The *interactive* IPA by Bünz *et al.* for $m=2^k$ (wlog). The prover convinces the verifier that he knows $(\mathbf{A},\mathbf{B})\in\mathbb{G}_1^m\times\mathbb{G}_2^m$ such that $\mathbf{A},\mathbf{B},Z$ are committed in $\mathbf{C}$ (under commitment key $\mathbf{ck}$) and that $Z=\langle\mathbf{A},\mathbf{B}\rangle$. See §2 for IPA-specific notation such as $1_{\mathbb{G}_b},\mathbf{A}_L,\mathbf{A}\circ\mathbf{B},\mathsf{CM},\mathbf{A}_R||1_{\mathbb{G}}$ or $\mathbf{A}_L^x$.

of $\mathbf{v}'$ and $\mathbf{w}'$ to the untrusted prover and reduces verification time from $O(m)$ to $O(\log m)$. However, this comes at the cost of additionally relying on the q-SDH assumption (see Assum. A.1) and the q-ASDBP assumption [19]. Our work implicitly assumes this optimized verifier, which we later implement in §5. We refer the reader to [19, Section 5] for the details of this optimization, which is beyond the scope of this paper.

Lemma A.1 (Random linear combinations lemma)**.** *Let* $Z_i\in\mathbb{G}_T$*,* $\mathbf{A}_i\in\mathbb{G}_1^m$ *and* $\mathbf{B}_i\in\mathbb{G}_2^m$ *for* $i=1,\dots,N$*. Assume each* r_i *is chosen uniformly at random from* $\mathbb{Z}_p$*. Then, with probability at least* $1-1/p$*, all Eq. 23 are satisfied iff. Eq. 24 is satisfied.*

Proof (sketch) for Lemma A.1. Clearly if Eq. 23 are satisfied then Eq. 24 is also satisfied. For the other direction, by the Schwartz-Zippel lemma [54, 70], if at least one equation from Eq. 23 does not hold, Eq. 24 holds at randomly selected values r_i with probability $1/p$, completing the proof. □

Theorem A.1. $(\mathcal{G}_{\mathsf{BATCH}},\mathcal{P}_{\mathsf{BATCH}},\mathcal{V}_{\mathsf{BATCH}})$ *from Fig. 5 is a non-interactive argument of knowledge for* $\mathcal{L}_{\mathsf{BATCH}}^{b,\ell}$ *from Eq. 22 that has knowledge soundness under the same assumptions as the non-interactive IPA from §2.4 (i.e., algebraic commitment model [19], the random oracle model,* $(2b\ell)$*-SDH,* $(b\ell)$*-ASDBP).*

Proof (sketch) for Thm. A.1. This follows from Lemma A.1 and the knowledge soundness of the Bünz et al. IPA, which is used in a black box fashion. □

B VCs and Hyperproofs

Definition B.1 (VC Correctness)**.** *A VC is correct, if for all* $\lambda \in \mathbb{N}$ *and* $n = \mathsf{poly}(\lambda)$*, for all* $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, n)$*, for all vectors* $\mathbf{a} = [a_0, \ldots, a_{n-1}]$*, if* $\mathsf{C} = \mathsf{Com}_{\mathsf{pp}}(\mathbf{a})$ *and* $\pi_i = \mathsf{Open}_{\mathsf{pp}}(i, \mathbf{a}), \forall i \in [0, n)$ *(or from* $\mathsf{OpenAll}_{\mathsf{pp}}(\mathbf{a})$*), then, for any polynomial number of updates* (u, δ) *resulting in a new vector* $\mathbf{a}'$*, if* C' *and* π'_i*, for all* i*, are the updated digest and proofs obtained via calls to* $\mathsf{UpdDig}_{\mathsf{pp}}$ *and* $\mathsf{UpdProof}_{\mathsf{pp}}$ *(or to* $\mathsf{UpdAllProofs}_{\mathsf{pp}}$*) respectively, then (1)* $\Pr[1 \leftarrow \mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}', \{i\}, a'_i, \pi'_i)] = 1$ *for all* i*; (2)* $\forall I \subseteq [n], \Pr[1 \leftarrow \mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}', I, (a'_i)_{i \in I}, \mathsf{Agg}_{\mathsf{pp}}(I, (a'_i, \pi'_i)_{i \in I}))] = 1$.

Observation: At a high-level, correctness says that proofs created via Open or OpenAll verify successfully via Ver, even in the presence of updates and aggregated proofs.

Definition B.2 (VC Soundness)**.** $\forall$ PPT *adversaries* $\mathcal{A}$,

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, n), \\ (\mathsf{C}, I, J, (a_i)_{i \in I}, (a'_j)_{j \in J}, \pi_I, \pi'_J) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}) : \\ 1 \leftarrow \mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, I, (a_i)_{i \in I}, \pi_I) \wedge \\ 1 \leftarrow \mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, J, (a'_j)_{j \in J}, \pi'_J) \wedge \\ \exists k \in I \cap J \text{ s.t. } a_k \neq a'_k \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Observation: Soundness says that no adversary can output two *inconsistent proofs* for different values $a_k \neq a'_k$ at position k with respect to an adversarially-produced digest d. Note that such a definition allows the digest C to be produced adversarially. This is stronger than what is required in our cryptocurrency setting from §4, where the digest is produced correctly from the agreed transactions. Nonetheless, having a stronger definition makes our VC from §3 more widely useful.

Theorem B.1 (Individual Hyperproofs are sound)**.** *Our individual* $\log n$*-sized (non-aggregated) proofs from §3.1 are sound as per Def. B.2 under q-SDH (see Assum. A.1).*

Proof for Thm. B.1. Suppose there exists an adversary $\mathcal{A}$ that breaks Def. B.2. We show how to build another adversary $\mathcal{B}$ that breaks the ℓ-SDH assumption (see Assum. A.1). We first assume $\mathcal{A}$ returns individual (non-aggregated) proofs and then generalize to $\mathcal{A}$ returning aggregated proofs.

$\mathcal{B}$ is given ℓ-SDH public parameters $\mathsf{pp} = ((p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2), g_2^s, g_1^s, \ldots, g_1^{s^\ell})$, and must (somehow) break ℓ-SDH by outputting $(a, g_1^{\frac{1}{s-a}})$ for some $a \neq s$. For this, $\mathcal{B}$ will leverage $\mathcal{A}$ into helping him.

First, $\mathcal{B}$ "guesses" the position i on which $\mathcal{A}$ will forge, which he can do with probability $1/\mathsf{poly}(\lambda)$, where λ is our security parameter. Second, $\mathcal{B}$ "tweaks" the ℓ-SDH public parameters into the Hyperproofs public parameters from Fig. 1, which he then calls $\mathcal{A}$ with. Specifically, $\mathcal{B}$ sets $s_k - i_k = r_k(s - i_1), \forall k \in [\ell]$, where $r_1 = 1$, the rest of the r_k's are random, and $i_\ell, \ldots, i_1$ is the binary representation of i. Importantly, note that $\mathcal{B}$ can do this without knowledge of s, since $\mathcal{B}$ can compute any product $g_1^{\prod_{i \in S} s_i}, S \in 2^{\{1,2,\ldots,\ell\}}$ from the $g_1^{s^i}$'s. Similarly, $\mathcal{B}$ can compute any $g_2^{s_k}$ from g_2^s. Third, $\mathcal{B}$ calls $\mathcal{A}$ with the "tweaked" public parameters as input and obtains a digest C and two *inconsistent* proofs $\pi = (w_k)_{k \in [\ell]}$, $\pi' = (w'_k)_{k \in [\ell]}$ for position i having values both v and v'. (If $\mathcal{A}$ outputs proofs for a different position $i' \neq i$, $\mathcal{B}$ retries.)

Since both proofs verify, the following equations hold, where $i_\ell, \ldots, i_1$ is the binary expansion of the position i:

$$e(\mathsf{C}/g_1^v, g_2) = \prod_{k \in [\ell]} e(w_k, g_2^{s_k - i_k}) \tag{28}$$

$$e(\mathsf{C}/g_1^{v'}, g_2) = \prod_{k \in [\ell]} e(w'_k, g_2^{s_k - i_k}) \tag{29}$$

Next, dividing the top equation by the bottom one and substitute $s_k - i_k = r_k(s - i_1), \forall k \in [\ell]$:

$$e(g_1^{v'}/g_1^v, g_2) = \prod_{k \in [\ell]} e(w_k/w'_k, g_2^{s_k - i_k}) \Leftrightarrow \tag{30}$$

$$e(g_1^{v'-v}, g_2) = \prod_{k \in [\ell]} e(w_k/w'_k, g_2^{r_k(s-i_1)}) \Leftrightarrow \tag{31}$$

$$e(g_1, g_2)^{v'-v} = \left(\prod_{k \in [\ell]} e(w_k/w'_k, g_2^{r_k})\right)^{s-i_1} \Leftrightarrow \tag{32}$$

$$e(g_1, g_2)^{\frac{1}{s-i_1}} = \left(\prod_{k \in [\ell]} e(w_k/w'_k, g_2^{r_k})\right)^{\frac{1}{v'-v}} \Leftrightarrow \tag{33}$$

$$e(g_1^{\frac{1}{s-i_1}}, g_2) = \prod_{k \in [\ell]} e\left(\left(w_k/w'_k\right)^{\frac{r_k}{v'-v}}, g_2\right) \Leftrightarrow \tag{34}$$

$$e(g_1^{\frac{1}{s-i_1}}, g_2) = e\left(\prod_{k \in [\ell]} \left(w_k/w'_k\right)^{\frac{r_k}{v'-v}}, g_2\right) \tag{35}$$

Thus, $g_1^{\frac{1}{s-i_1}} = \prod_{k \in [\ell]} \left(w_k/w'_k\right)^{\frac{r_k}{v'-v}}$ and $\mathcal{B}$ can output $(i_1, g_1^{\frac{1}{s-i_1}})$ and break ℓ-SDH. □

Theorem B.2 (Aggregated Hyperproofs are sound)**.** *Our aggregated proofs from §3.3 are sound as per Def. B.2 under the knowledge-soundness of the* $\mathcal{L}_{\mathsf{BATCH}}$ *argument (see Thm. A.1).*

Proof. See the extended version of this paper [56] □