

Faster Incremental Authenticated Dictionaries: Removing the Query Bottleneck

Michael A. Bender*

John Kuszmaul[†]

William Kuszmaul[‡]

Abstract

Known lower bounds prohibit the construction of very-efficient authenticated dictionaries that support both insertions and deletions. It remains an open question, however, whether it may be possible to construct very efficient authenticated dictionaries that are *insertion-only*. The current paper answers this question in the affirmative, providing the constructions that strictly dominate the classic Merkle tree in the incremental setting.

Our first result is an authenticated incremental dictionary, called the exponential Verkle forest, that supports retrieval queries with time and proof size $O(\log \log n)$ and that supports insertions with time and proof size $O(\log \log \log n)$. The data structure also offers an *amortized* bound of $O(1)$ for insertions.

Building on this, we also present a full-query version of the dictionary, supporting both retrieval and exclusion queries. This data structure continues to support retrieval queries in time and proof size $O(\log \log n)$, while supporting all other operations in time and proof size $O(\log n)$.

Our results take a purely data-structural perspective on authenticated data structures, seeking to achieve the best bounds possible with the cryptographic primitives (collision-resistant hashing and position-binding vector commitments) that already exist. We also give partial evidence, in the form of a lower bound that applies to a natural class of data structures, that our $O(\log \log n)$ query bound is optimal for any solution with linear construction time.

1 Introduction

An ***authenticated data structure*** [1, 6, 8, 9, 12, 17–20, 23–25, 27–29, 32–35, 37–40, 43, 45] allows a client to offload the maintenance and storage of a data structure to some ***untrusted server***, while also allowing the client to efficiently verify the correctness of operations that are performed on the data structure.

Formally, an authenticated data structure can be defined using the following two-party model [32–34]. A ***client*** wishes to perform operations on a data structure that resides on an ***untrusted server***. The client cannot afford to store the entire data structure, but is able to store a constant-sized ***digest*** τ (e.g., a hash of the data structure). Whenever the client sends an operation to the server (either an update or a query), the server must perform that operation, return any answer α that the operation produces, and present a short ***proof*** π certifying that the operation was performed correctly. Given α, π, τ , the client should be able to both cryptographically verify the

*Stony Brook University. bender@cs.stonybrook.edu

[†]MIT CSAIL. Email: john.kuszmaul@gmail.com

[‡]Carnegie Mellon University. Email: kuszmaul@cmu.edu

correctness of the operation and (if the operation changed the data structure) also update the digest τ to whatever new value τ' is appropriate. Neither party may possess any secrets [36].

In this paper, we will be interested, in particular, in *authenticated dictionaries*. These are data structures that support the operations typically associated with, say, a hash table: they store a dynamic set of key-value pairs (k, v) , while supporting insertions, deletions, and queries. Note that, when discussing *queries*, it is typically the case that both query time and proof size are asymptotically the same—thus, we will typically refer just to proof size for queries, while discussing both time and proof size for other operations.

Merkle trees, Verkle trees, and the query bottleneck. The classic way to implement an authenticated dictionary is with a Merkle tree [26, 28]. Merkle trees have served as the default authenticated dictionary, both in theory and practice, for over four decades [9, 28, 32], achieving $O(\log n)$ time and proof size for every operation.

In recent years, we have seen the emergence of so-called “planet-scale” authenticated dictionaries [45], in which the authenticated dictionaries are so large that even $O(\log n)$ operations become a bottleneck.¹ These planet-scale authenticated dictionaries are typically query-heavy, serving a huge number of queries to third-party users all over the world. This has led practitioners to seek out solutions that support queries in time and proof size better than $\log n$.

The main alternative that practitioners have turned to is the so-called Verkle tree [21, 23, 34, 35]. Verkle trees offer a tradeoff between queries and updates: Update time increases by a factor of k , while query time (and all proof sizes) decrease by a factor of $\log k$. A priori, this might seem like a bad trade. And yet, queries are so important that this tradeoff has increasingly become the default choice in practice (see, e.g., the scaling roadmap of Ethereum [16]).²

The problem: memory checking is (pretty) hard. Given the relatively poor tradeoff curve that Verkle trees offer, one might wonder: Can we do better? Can we, for example, hope to support updates in $O(\log n)$ time and proof size, while supporting queries much more efficiently?

Perhaps surprisingly, the answer is *no*. In a celebrated sequence of results, several quite strong lower bounds have been proven [8, 10, 15], the most notable of which for us is the following: in general, if the client-side space usage is $n^{1-\Omega(1)}$, and the server-side query time is $o(\log n / \log \log n)$, then the server-side *update time* must be extremely large—at least $(\log n)^{\omega(1)}$. This lower bound, which was recently extended [10] to both randomized and adaptive data structures, applies not just to the authenticated dictionary problem, but also more generally to the so-called *memory-checking model* [8], which is the problem of constructing an authenticated data structure that supports reads and writes to some memory of size n .

This places us in an unfortunate situation. On one hand, the $O(\log n)$ bounds of Merkle trees are no longer sufficient for many of the planet-scale authenticated dictionaries that exist in the real world. On the other hand, there is a clear obstacle to doing better. If we wish to have significantly better than $O(\log n)$ -time queries, then we must significantly compromise on the cost of updates.

¹For example, if we consider even a (moderately large) Merkle tree with 2^{32} keys, then membership proofs will require approximately 32 hashes, each of size 32 bytes (assuming a smaller cryptographic hash function like SHA-256 [31] is used). This results in a proof size of 1024 bytes, or a proof size of 2048 bytes assuming 64 byte hashes are used (such as SHA-512 [31]). Regardless of the hash function used, Merkle proofs will be in the range of kilobytes.

²In the case of Ethereum, the decrease in proof size that the transition from a Merkle tree to a Verkle tree yields is “a reduction sufficient to make stateless clients finally viable in practice” [11].

But what about *incremental* dictionaries? This paper begins with the following simple observation: many real-world authenticated dictionaries store data sets that are *growing* over time, but not shrinking. Such a dictionary requires efficient *insertions*, but does not necessarily require efficient *deletions*. We call a dictionary of this type an *incremental dictionary*.

The prototypical example of such a dictionary is a certificate transparency log, which offers a public record of all certificates that have ever been issued—these are commonly built on a type of incremental Merkle tree known as a *history Merkle tree* [13]. Other examples of dictionaries that (depending on the setting) are either fully or mostly incremental include dictionaries that store transactions on a blockchain and dictionaries that store sets of registered public keys [9].

What makes incremental dictionaries interesting is that they are not subject to the memory-checking lower bounds in [8, 10, 15]. This prompts a natural question: is it possible to build incremental dictionaries that support $O(\log n)$ -time insertions, while supporting much more efficient queries? Or, do incremental dictionaries ultimately suffer from the same query bottleneck as their dynamic counterparts?

Despite the remarkable simplicity of this question, it has, to the best of our knowledge, remained open.

This paper: incremental dictionaries with very fast queries. When discussing incremental dictionaries, we will find it helpful to separately consider four types of operations:

- **Construct(S):** Initializes the dictionary from a finite sequence S of key-value records $(k_1, v_1), (k_2, v_2), \dots$ provided in sorted order.
- **Insert(k, v):** Adds a key-value pair (k, v) to the dictionary.
- **RetrievalQuery(k):** Returns a value v , and a proof that (k, v) is in the dictionary.
- **ExclusionQuery(k):** Returns a proof that the given key k is not present.

In most applications, it would be natural to assume that the client never inserts the same key multiple times. Our formal model, however, will take a more relaxed view: If the client inserts the same key multiple times with different values, then the data structure’s behavior continues to be well-defined, and the **RetrievalQuery** operation on that key may return any of the inserted values. In applications where the **RetrievalQuery** function is being used to verify a key’s membership (and where values are not used), this distinction does not matter.

There are two types of incremental dictionaries that one typically considers: **retrieval dictionaries** [9, 17, 37, 41], which support all of the operations except for **ExclusionQuery**; and **full-query dictionaries** [21, 23, 34, 35], which support all four operations. We remark that, in practice, even in applications that use full-query dictionaries, it is typically retrieval queries, rather than exclusion queries, that form a performance bottleneck. The current states of the art for both retrieval dictionaries and full-query dictionaries are given in Table 1.

The main technical result of this paper is that it is possible to build an authenticated retrieval dictionary, which we call the **Verkle forest**, that supports *all* of its operations significantly more efficiently than does a classic Merkle tree. Specifically, this data structure supports **Construct** in linear time; **Insert** with worst-case time and proof size $O(\log \log \log n)$ and amortized time and proof size $O(1)$; and **Query** with worst-case time and proof size $O(\log \log n)$.

	Construction Time	Insertion Time	Insertion Proof Size	Query Time/Proof-Size	Query Type
Merkle Tree [26, 28]	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	Full-Query
Merkle Mountain Range [9, 17, 37, 41]	$O(n)$	$O(\log n)$ $O(1)$ <i>amortized</i>	$O(\log n)$ $O(1)$ <i>amortized</i>	$O(\log n)$	Retrieval
Vector Commitment [12] (See also [6, 20])	$O(n^2)$	$O(n)$	$O(1)$	$O(1)$	Full-Query
Verkle Tree [21, 23, 34, 35]	$O(kn)$	$O(k \log_k n)$	$O(\log_k n)$	$O(\log_k n)$	Full-Query

Figure 1: Summary of the asymptotic performance of incremental authenticated dictionaries from past work. All of the data structures except for the Merkle Mountain Range can also be implemented to support exclusion queries by storing keys in sorted order. These data structures are discussed in more detail in Section 7.

Building on this, we also construct an authenticated full-query dictionary, which we call the **augmented Verkle forest**, that *matches* the Merkle tree on construction, insertion, and exclusion queries, while supporting retrieval queries in $O(\log \log n)$ time and proof size.

In the process of constructing these data structures, we also introduce two other intermediate data structures that may be of practical interest. The bounds for all of our data structures are summarized in Table 2. An interesting feature of all of our results is that they are achieved through purely *data-structural* contributions, applying data-structural ideas on top of the cryptographic primitives that already exist.

At a technical level, the progression of the paper is as follows. After introducing security and correctness definitions in Section 2, we continue to Section 3, where we introduce an authenticated retrieval dictionary, called the **exponential Verkle tree**,³ which achieves $O(n)$ construction time with a query proof size of $O(\log \log n)$. This data structure is very efficient for static settings (i.e., where **Construct** and **Query** are the main two operations), but comes with the drawback that its insertions are relatively slow, taking time $O(n^\varepsilon)$ for some $\varepsilon \in (0, 1/4)$.

We then show how to build on the exponential Verkle tree in order to construct what we call the **linear-time Verkle tree**. This authenticated retrieval dictionary has *almost the same* query/insert bounds as does the standard Verkle tree, but with linear construction time. This means that the succinct proofs that a Verkle tree enjoys do not need to come at any cost to construction time. This distinction, although seemingly small at first glance, ends up being critical in both of our final two constructions.

Finally, our main technical result—as already noted above—is our construction of the Verkle forest (Sections 4 and 5). This data structure is able to match the $O(\log \log n)$ queries that exponential Verkle trees get, while also getting $O(\log \log \log n)$ bounds for insertions. Thus the Verkle forest has exponentially better bounds for queries, and *doubly*-exponentially better bounds for insertions, than does the classical Merkle tree. Building on the Verkle forest, we are able to construct the augmented Verkle forest as an almost immediate consequence (Section 5.1).

Based on these results, a natural question is whether one might be able to reduce the query

³We remark that the name of the exponential Verkle tree comes from the fact that, at a high level, its recursive structure is similar to that of an *exponential tree* [2, 3], a classic (non-authenticated) dictionary that can be used to store n $\Theta(\log n)$ -bit elements with deterministic update/query time $O((\log \log n)^2 / \log \log n)$.

	Construction Time	Insert Time	Insertion Proof-Size	Retrieval Queries	Exclusion Queries
Exponential Verkle tree	$O(n)$	$O(n^\epsilon)$	$O(\epsilon^{-1} \log \log n)$	$O(\epsilon^{-1} \log \log n)$	$\cdot$
Linear-Time Verkle tree	$O(n)$	$O(k \log_k n)$	$O(\log_k n + \log \log k)$	$O(\log_k n + \log \log k)$	$\cdot$
Verkle Forest	$O(n)$	$O(\log \log \log n)$ $O(1)$ <i>amortized</i>	$O(\log \log \log n)$ $O(1)$ <i>amortized</i>	$O(\log \log n)$	$\cdot$
Augmented Verkle Forest	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log \log n)$	$O(\log n)$

Figure 2: Summary of the asymptotic performance of the incremental authenticated dictionaries in this paper.

proof-size further, getting a bound of the form $o(\log \log n)$ but without compromising insertion time and construction time. Our final result, in Section 6, is a lower bound showing that, for a natural class of data structures (including those in this paper), no such improvement is possible. Specifically, we show that any generalized Verkle tree with costs as defined in Section 2 cannot achieve construction time of $O(n)$ with a query proof-size of $o(\log \log n)$.

Combined, our data structures reveal that there *is no fundamental tension* between retrieval queries and other operations in an incremental authenticated dictionary.

2 Preliminaries

2.1 Correctness and Security Definitions

In this section, we make precise the correctness and security guarantees used throughout the paper.

Basic notation and the computational model. Following the standard cryptographic convention used in authenticated-data-structure and vector-commitment definitions, the ***security parameter*** is denoted by λ , and adversaries run in probabilistic polynomial time in λ [12, 35]. A function $\mu(\lambda)$ is ***negligible*** if, for every positive polynomial p , we have $\mu(\lambda) < 1/p(\lambda)$ for all sufficiently large λ .

Each instantiation also has a ***maximum size*** $n_{\max}$, which is the largest number of stored records, counting repetitions, that the dictionary is allowed to contain. So that the security guarantees in the paper are meaningful across the entire history of the data structure, we require

$$1 \leq n_{\max} \leq \text{poly}(\lambda).$$

We work in the word-RAM model with w -bit words, where w is large enough to store key-value pairs, cryptographic hashes, and commitments in $O(1)$ words. Unless stated otherwise, space bounds are measured in machine words.

Throughout the paper, we will assume that all public parameters (e.g., for the vector commitment scheme) are honestly precomputed.⁴ After setup, neither the client nor the server possesses a secret.

⁴Note that, without loss of generality, we only need public parameters for vector commitments with power-of-two

We begin by first formally defining (non-authenticated) incremental dictionaries.

Incremental Dictionaries. Let $\mathcal{K}$ be the key universe and $\mathcal{V}$ be the value universe, and let $\perp \notin \mathcal{V}$. An incremental dictionary maintains a finite *set* M of key-value pairs in $\mathcal{K} \times \mathcal{V}$, and we use $\text{Vals}_M(k)$ to denote the set of values $v \in \mathcal{V}$ such that $(k, v) \in M$. Records may be inserted repeatedly, and an implementation is free to store one record per insertion; but a query for a key k is answered by any value ever inserted for k , so the abstract state need only track which pairs are present. We write n for the number of records supplied to the dictionary so far, counting repetitions, and require $n \leq n_{\max}$ at all times.

Definition 1 (Incremental dictionary functionality). *An incremental dictionary supports the following operations.*

- **Construct**(S), where $S = ((k_1, v_1), \dots, (k_s, v_s))$ is a sequence of at most $n_{\max}$ records, initializes the state to the set of pairs occurring in S .
- **Insert**(k, v), when $n < n_{\max}$, changes the state from M to $M \cup \{(k, v)\}$.
- **RetrievalQuery**(k), when $\text{Vals}_M(k) \neq \emptyset$, may return any value $v \in \text{Vals}_M(k)$.
- **ExclusionQuery**(k), when $\text{Vals}_M(k) = \emptyset$, returns $\perp$.

A **retrieval dictionary** supports the first three operations. A **full-query dictionary** additionally supports exclusion queries.

Although, in principle, a dictionary may use super-linear space, in this paper we will be interested only in dictionaries that use linear space, meaning that the total size of the data structure is $O(n)$ machine words. Having defined incremental dictionaries, we now turn our attention to defining their authenticated counterparts.

Authenticated incremental dictionaries. The authenticated part of the server's state is denoted by X , and the digest by $\tau = \text{Root}_{\text{pp}}(X)$. The complete server state σ may additionally contain arbitrary auxiliary information, such as a hash table, to help with time efficiency.

Definition 2 (Authenticated incremental dictionary). *An authenticated incremental dictionary consists of the following algorithms.*

- $\text{pp} \leftarrow \text{Setup}(1^\lambda, 1^{n_{\max}})$, for $1 \leq n_{\max} \leq \text{poly}(\lambda)$, generates public parameters for dictionaries of maximum size $n_{\max}$.
- $(\sigma, \tau) \leftarrow \text{Construct}_{\text{pp}}(S)$, for a sorted sequence $S \in (\mathcal{K} \times \mathcal{V})^*$ of length at most $n_{\max}$, constructs the server state σ and digest τ for S .
- $(\sigma', \widehat{\tau}, \pi_{\text{ins}}) \leftarrow \text{ProveInsert}_{\text{pp}}(\sigma, k, v)$ produces a new server state σ' , a proposed new digest $\widehat{\tau}$, and an insertion proof π_{ins} .
- $b \in \{0, 1\} \leftarrow \text{VerifyInsert}_{\text{pp}}(\tau, k, v, \widehat{\tau}, \pi_{\text{ins}})$ verifies the proposed digest transition for an insertion.

arities. In each of our constructions that largest arity is $O(n_{\max}^\varepsilon)$, so the $O(\log n_{\max})$ parameter sets together occupy $O(n_{\max}^{2\varepsilon})$ space and can be precomputed in time $O(n_{\max}^{2\varepsilon})$ [12].

- $(v, \pi_{\text{ret}}) \leftarrow \text{ProveRetrieve}_{\text{pp}}(\sigma, k)$ produces a value and a retrieval proof.
- $b \in \{0, 1\} \leftarrow \text{VerifyRetrieve}_{\text{pp}}(\tau, k, v, \pi_{\text{ret}})$ verifies the claimed retrieval key-value pair (k, v) , given a proof π_{ret} .
- $\pi_{\text{exc}} \leftarrow \text{ProveExclude}_{\text{pp}}(\sigma, k)$ produces an exclusion proof.
- $b \in \{0, 1\} \leftarrow \text{VerifyExclude}_{\text{pp}}(\tau, k, \pi_{\text{exc}})$ verifies the exclusion proof.

A retrieval dictionary need not include the final two algorithms.

For an authenticated dictionary with authenticated state X , we use $\text{Dict}(X)$ to denote the set M of pairs represented by the dictionary, and $\text{Next}_{\text{pp}}(X, k, v)$ to denote the updated authenticated state X' that one obtains after an insertion (k, v) if the insertion is implemented honestly.

Definition 3 (Correctness). *An authenticated incremental dictionary is **correct** if the following holds. Construction, when performed honestly, produces an authenticated state X such that $\text{Dict}(X)$ is the set of pairs occurring in the requested initial sequence. Provided fewer than n_{max} records have been supplied so far, insertion of a key-value pair (k, v) updates X to $X' = \text{Next}_{\text{pp}}(X, k, v)$ such that $\text{Dict}(X') = \text{Dict}(X) \cup \{(k, v)\}$, and produces a proof that (if honestly generated) is accepted. For a key k with $\text{Vals}_{\text{Dict}(X)}(k) \neq \emptyset$, a retrieval query returns some $v \in \text{Vals}_{\text{Dict}(X)}(k)$, and produces a proof that (if honestly generated) is accepted. And, for a key k with $\text{Vals}_{\text{Dict}(X)}(k) = \emptyset$, an exclusion query produces a proof that (if honestly generated) is accepted.*

Definition 4 (Security). *Fix λ and n_{max} with $1 \leq n_{\text{max}} \leq \text{poly}(\lambda)$. Consider the following experiment with a PPT adversary $\mathcal{A}$.*

1. *The challenger samples $\text{pp} \leftarrow \text{Setup}(1^\lambda, 1^{n_{\text{max}}})$ and gives pp to the adversary. The adversary chooses an initial sequence $S_0 \in (\mathcal{K} \times \mathcal{V})^*$ containing at most n_{max} records, and the challenger computes $(\sigma_0, \tau_0) \leftarrow \text{Construct}_{\text{pp}}(S_0)$. Let X_0 be the authenticated representation inside σ_0 . The adversary receives (σ_0, τ_0) , and the challenger records the state (X_0, τ_0) .*
2. *While the current recorded state contains fewer than n_{max} records, the adversary may make insertion attempts adaptively. In an attempt made from the current recorded state (X_i, τ_i) , it chooses any requested pair $(k, v) \in \mathcal{K} \times \mathcal{V}$ and outputs a proposed server response $(\hat{\tau}, \pi)$. The challenger computes*

$$b = \text{VerifyInsert}_{\text{pp}}(\tau_i, k, v, \hat{\tau}, \pi).$$

If $b = 0$, the state is unchanged and the challenger returns (reject, τ_i) . If $b = 1$, the challenger computes the reference state $X^* = \text{Next}_{\text{pp}}(X_i, k, v)$. It then records the next state $(X_{i+1}, \tau_{i+1}) = (X^*, \hat{\tau})$ and returns $(\text{accept}, \hat{\tau})$. The experiment records only states reached by accepted insertions.

3. *Finally, the adversary outputs an index t of a recorded state (X_t, τ_t) and either a retrieval tuple (k, v, π) or an exclusion tuple (k, π) . The adversary wins if either*
 - (a) *it outputs a retrieval tuple (k, v, π) such that $\text{VerifyRetrieve}_{\text{pp}}(\tau_t, k, v, \pi) = 1$ but $(k, v) \notin \text{Dict}(X_t)$, or*
 - (b) *the dictionary is a full-query dictionary, and the adversary outputs an exclusion tuple (k, π) such that $\text{VerifyExclude}_{\text{pp}}(\tau_t, k, \pi) = 1$ but $\text{Vals}_{\text{Dict}(X_t)}(k) \neq \emptyset$.*

Let $\text{Adv}_{\mathcal{A}}^{\text{ADS}}(\lambda, n_{\max})$ denote the adversary's winning probability. The authenticated dictionary is **secure** if, for every PPT adversary $\mathcal{A}$ (i.e., an adversary that completes the entire experiment in time polynomial in the security parameter λ), there is a negligible function $\mu_{\mathcal{A}}$ such that

$$\text{Adv}_{\mathcal{A}}^{\text{ADS}}(\lambda, n_{\max}) \leq \mu_{\mathcal{A}}(\lambda).$$

We remark that the definition above allows for an adaptive adversary, where the word *adaptive* means that every proposed server response, including the final query proof, may depend on all earlier public parameters, proofs, digests, and accept/reject outcomes.

We further note that the security of an authenticated data structure does not (and cannot) serve as any form of liveness guarantee; a malicious server may simply refuse to answer queries.

2.2 Cryptographic Building Blocks

The two cryptographic building blocks used throughout this paper are collision-resistant hashes and position-binding vector commitments.

Collision-resistant hashing. A hash family consists of a randomized setup algorithm that samples a public hash key s and a function $\mathcal{H}_s : \{0, 1\}^* \rightarrow \mathcal{D}$, where $\mathcal{D}$ is the output space. For a PPT adversary $\mathcal{A}$, define

$$\begin{aligned} \text{Adv}_{\mathcal{H}, \mathcal{A}}^{\text{coll}}(\lambda) = \Pr[& x \neq x' \wedge \mathcal{H}_s(x) = \mathcal{H}_s(x') : \\ & s \leftarrow \text{HashSetup}(1^\lambda), (x, x') \leftarrow \mathcal{A}(s)]. \end{aligned}$$

The family is **collision resistant** if this advantage is negligible in λ for every PPT adversary. We assume that hash outputs in $\mathcal{D}$ can be stored in $O(1)$ machine words.

Vector commitments. A vector-commitment scheme [12] with arity q is an authenticated data structure that implements a dynamic vector of length q . It consists of a randomized setup algorithm $\text{pp}_q \leftarrow \text{VC.KeyGen}(1^\lambda, q)$ and algorithms

$$\text{VC.Commit}, \quad \text{VC.Open}, \quad \text{VC.Verify}, \quad \text{VC.Update}, \quad \text{and} \quad \text{VC.ProofUpdate}.$$

These algorithms play the following roles. Given public parameters pp_q and a vector $\mathbf{m} = (m_1, \dots, m_q)$, the algorithm $\text{VC.Commit}_{\text{pp}_q}(\mathbf{m})$ returns a short commitment C to $\mathbf{m}$, together with auxiliary information aux that whoever holds the vector keeps in order to answer opening requests later; the commitment C plays the role of a digest in an authenticated data structure. $\text{VC.Open}_{\text{pp}_q}(i, \mathbf{m}, \text{aux})$ produces an **opening** π_i for a single position i , which is the proof that a client checks; and $\text{VC.Verify}_{\text{pp}_q}(C, i, m, \pi)$ takes a commitment, a position, a claimed message, and an opening, and outputs 1 exactly when π certifies that position i of the vector committed by C contains m . The remaining two algorithms make it possible to maintain a commitment and its openings after a coordinate changes, instead of recomputing them from scratch. When position i changes from m_i to m'_i , the algorithm $\text{VC.Update}_{\text{pp}_q}(C, i, m_i, m'_i)$ returns the commitment C' of the modified vector along with a small piece of update information U that describes the change, and $\text{VC.ProofUpdate}_{\text{pp}_q}(C, \pi_j, j, U)$ takes an opening π_j that was valid against C for a position j (possibly $j = i$) and returns the corresponding opening that is valid against C' . We assume that each

	Construction time	Update time	Update proof size	Query time/proof size
Cryptographic hash of q messages	$O(q)$	$O(q)$	$O(q)$	$O(q)$
Vector commitment to q messages [12]	$O(q^2)$	$O(q)$	$O(1)$	$O(1)$

Figure 3: The black-box costs of the two node-authentication primitives used in this paper. Sizes are measured in machine words. An update changes a single coordinate of the vector. Update/query times both refer to server time. Client-side computation is always proportional to proof size.

commitment C , each opening π_i , and each piece of update information U occupy $O(1)$ machine words. We also assume the usual correctness property: honestly generated and honestly updated openings verify.

The security property we use is the standard **position-binding** property of Catalano and Fiore [12]. For a PPT adversary $\mathcal{A}$ and an arity q , define

$$\text{Adv}_{\text{VC}, \mathcal{A}, q}^{\text{pb}}(\lambda) = \Pr \left[\begin{array}{l} m \neq m' \wedge \text{VC.Verify}_{\text{pp}_q}(C, i, m, \pi) = 1 \wedge \\ \text{VC.Verify}_{\text{pp}_q}(C, i, m', \pi') = 1 : \\ \text{pp}_q \leftarrow \text{VC.KeyGen}(1^\lambda, q), \\ (C, i, m, m', \pi, \pi') \leftarrow \mathcal{A}(\text{pp}_q) \end{array} \right].$$

The scheme is position binding if, for every PPT adversary $\mathcal{A}$, there is a negligible function $\mu_{\mathcal{A}}$ such that

$$\text{Adv}_{\text{VC}, \mathcal{A}, q}^{\text{pb}}(\lambda) \leq \mu_{\mathcal{A}}(\lambda)$$

for every λ and every arity $1 \leq q \leq \text{poly}(\lambda)$. In particular, this covers every $q \leq n_{\max}$ used by our data structures.

We assume throughout the paper the black-box performance of the Catalano–Fiore construction. Committing to a vector of length q and precomputing all q openings takes $O(q^2)$ time. Changing the commitment and updating any one opening each take $O(1)$ time. Since the server stores an opening for every position, refreshing all of them after one coordinate changes takes $O(q)$ total server time. The update information needed by the client and an opening of one coordinate both have size $O(1)$ words.

The time/space costs are summarized in Table 3. Here, an update refers to a single coordinate change. We assume that the server precomputes all openings for the vector commitment, and recomputes all of them on an update (hence the server update time is $O(q)$). The time bounds in Table 3 refer to server-side computation time, while the client-side time bounds are all proportional to proof size (so $O(1)$ for vector commitments).

Empty vector commitments in $O(1)$ time. In the Catalano–Fiore construction, the commitment to an all-zero vector, as well as every opening of that vector, is the group identity. An empty vector-commitment node of capacity q can therefore be instantiated and represented implicitly in $O(1)$ time and space. On the first update, the server explicitly initializes the empty vector commitment in $O(q)$ time, which is no more than the ordinary $O(q)$ cost of an update. Thus, an empty vector commitment can be initialized in $O(1)$ time.

Standing assumptions. Our theorems assume correct and secure instantiations of the above cryptographic building blocks. We refer collectively to these assumptions as our *standing cryptographic assumptions*. In particular, we assume a collision-resistant hash family and a correct, position-binding vector commitment with the costs of Table 3, whose `VC.Update` returns constant-size update information from which `VC.ProofUpdate` can bring any one opening that was valid against the old commitment up to date against the new one in $O(1)$ time. Note that the Catalano–Fiore construction (itself based on Computational Diffie–Hellman) naturally satisfies these assumptions [12].

2.3 Generalized Verkle Trees

All authenticated data structures in this paper are instances of what we call a *generalized Verkle tree*, which is a rooted tree where each internal node is a vector commitment to its children.

A generalized Verkle tree uses one fixed hash key and one set of vector-commitment parameters, both drawn from the public parameters `pp`. Concretely, `pp` contains a hash key $s \leftarrow \text{HashSetup}(1^\lambda)$, and we write $\mathcal{H}_s$ for the corresponding hash function; every node of the tree is hashed with this same s . For each arity q used by the tree, `pp` also provides honestly generated parameters $\text{pp}_q \leftarrow \text{VC.KeyGen}(1^\lambda, q)$ used by nodes of arity q .

We assume throughout that generalized Verkle trees are implemented using the following convention, in which each (non-empty) node comes with a tag to indicate whether it is a leaf or internal node.

Definition 5 (Generalized Verkle tree). *A **generalized Verkle tree** is a rooted tree in which every node is of one of the following three kinds. Each node has a **digest** as defined below.*

- An **internal node** is a vector commitment to its children. An internal node of **arity** q has q ordered child slots, and it stores a commitment C to the vector $(d_1, \dots, d_q)$, where d_j is the digest of the child in slot j . It also stores a constant-size authenticated **header** h , which contains any constant-size information that the construction needs in order to parse or update the node. In particular, in our constructions, the header will include the node’s **capacity**, the number of records that its subtree holds when full, and its **occupancy**, the number of its q slots that are currently non-null. The digest of the node is

$$d = \mathcal{H}_s(\langle \text{vc-node}, h, q, C \rangle).$$

- A **leaf** stores a single key-value record (k, v) , and has digest

$$d = \mathcal{H}_s(\langle \text{leaf}, k, v \rangle).$$

- An **empty leaf** stores nothing at all: it is simply a **null** slot in its parent, meaning that the corresponding message d_j committed by the parent is the reserved value 0.

The digest of the entire authenticated representation is the digest of the root, and this is the digest that the client stores.

As a practical convenience, we allow an internal node of constant arity to be implemented with a hash rather than a vector commitment. Such a **hash node** lists its children’s digests explicitly, using the canonical digest $\mathcal{H}_s(\langle \text{empty} \rangle)$ in place of a null slot, and has digest

$$d = \mathcal{H}_s(\langle \text{hash-node}, h, q, d_1, \dots, d_q \rangle).$$

One could equivalently use a vector commitment of arity $O(1)$ at every such node.

Unauthenticated server-side data. All of our data structures will keep track of additional unauthenticated server-side data that is used to improve time efficiency of server-side operations (but that does not affect security). This includes both the use of unauthenticated hash tables (which we discuss in more detail below) in all of our constructions; and in Section 5 it also includes cached vector-commitment openings, parent pointers, and cursors needed for scheduling work in the data structure. As we will see, these parts of the data structure need not be authenticated to achieve security.

The use of hash tables, and the probabilistic nature of insertion-time bounds. All of our constructions will maintain a hash table that records where each key-value pair resides in the data structure (i.e., for each key k such that (k, v) is present for some value v , it records where (k, v) is stored). We assume implicitly that any such hash table is implemented to use linear space, to support worst-case constant time queries, and to support constant-time insertions/deletions with probability $1 - 1/\text{poly}(n)$ (see, e.g., [5, 14, 22], for constructions of such hash tables). Note that the use of such a hash table means that all insertion-time bounds in this paper are probabilistic, holding with probability $1 - 1/\text{poly}(n)$. This is essentially unavoidable in the sense that, even for *unauthenticated* dictionaries, the only known solutions to support $O(\log \log n)$ time operations require probabilistic time guarantees.

2.4 Security for Retrieval Dictionaries

In this section, we describe the basic framework of security for generalized Verkle trees. In particular, we show how both retrieval and insertion proofs are generated and checked for these data structures, and we show that the resulting retrieval dictionary is guaranteed to be secure. We focus on the security of retrieval dictionaries; see Section 5.1 for a discussion of proving security for the augmented Verkle forest, a full-query authenticated dictionary.

At a high level, proofs will work as follows. Retrieval proofs simply authenticate a root-to-leaf path, showing that the leaf in question is present. Insertion proofs, on the other hand, must demonstrate that the only key-value pair added to the tree is the one being inserted. This is achieved with the double-envelope structure discussed next.

Encodings and envelopes. Fix a generalized Verkle tree X . An **address** in X is a sequence $a = (i_1, \dots, i_t)$ of child slots followed from the root, whose own address is $\langle \rangle$; an address either holds a node of X , meaning an internal node or a leaf, or is a null slot. For an address $a = (i_1, \dots, i_t)$ and an index j , we let $(a \parallel j)$ denote the address $(i_1, \dots, i_t, j)$, i.e., the address of the j th child of the node at address a . The **encoding** of a node is the string that its digest hashes, as prescribed by Definition 5: $\langle \text{vc-node}, h, q, C \rangle$, $\langle \text{hash-node}, h, q, d_1, \dots, d_q \rangle$, or $\langle \text{leaf}, k, v \rangle$. The header metadata for each encoding ensures that leaves and internal nodes cannot be confused for one another. We write $\text{enc}_X(a)$ for the encoding of the node that X holds at address a , using $\emptyset$ when a holds no node, and $\text{dig}(\rho) = \mathcal{H}_s(\rho)$ for the digest of an encoding ρ , with $\text{dig}(\emptyset)$ the canonical null value used by the parent (the reserved message 0 at a vector-commitment node, and $\mathcal{H}_s(\langle \text{empty} \rangle)$ at a hash node). Write $L(X)$ for the *set* of key-value pairs occurring in the leaves of X ; our retrieval dictionaries will satisfy $\text{Dict}(X) = L(X)$. Finally, call a set A of addresses an **envelope** if it contains every prefix of each of its members, or equivalently if it is a union of paths from the root.

We now describe an envelope proof, which proves what encodings reside in some specific envelope of a tree.

Definition 6 (Envelope proof). *Let A be an envelope. An **envelope proof** for A consists of a claimed encoding ρ_a for every $a \in A$, together with an opening $\pi_{a,i}$ for every $a \in A$ such that ρ_a is a vector-commitment encoding and $(a \parallel i) \in A$. To verify the proof against a digest τ , the client checks that*

1. $\text{dig}(\rho_{\langle \rangle}) = \tau$;
2. A is an envelope, i.e., every prefix of each $a \in A$ is contained in A ;
3. for every $a \in A$ with $\rho_a = \langle \text{vc-node}, h, q, C \rangle$ and every $(a \parallel i) \in A$, we have $i \leq q$ and

$$\text{VC.Verify}_{\text{pp}_q}(C, i, \text{dig}(\rho_{(a \parallel i)}), \pi_{a,i}) = 1;$$

4. for every $a \in A$ such that ρ_a is a hash-node encoding and every $(a \parallel i) \in A$, the i -th child digest listed in ρ_a equals $\text{dig}(\rho_{(a \parallel i)})$; and
5. no $a \in A$ with $\rho_a \in \{\emptyset\} \cup \{\text{leaf encodings}\}$ has a child in A .

The proof is **consistent with** a tree X if $\rho_a = \text{enc}_X(a)$ for every $a \in A$.

We also introduce two pieces of terminology to be used later, both of which are relative to a fixed envelope proof for an envelope A . We call an address $a \in A$ a **node** when its claimed encoding ρ_a is not $\emptyset$; in that case the proof claims that a holds either an internal node or a leaf of the generalized Verkle tree, and we say that a **stores** the encoding ρ_a . Additionally, we say that a coordinate of a node a is **opened** when the corresponding child address is again in A (meaning that the proof exhibits an encoding, and hence a digest, for that child). When the envelope proof is clear from context, we will speak of the nodes in the envelope A itself.

Retrieval proofs. A retrieval proof for key-value pair (k, v) must convince the client that some leaf ℓ of the tree has encoding $\langle \text{leaf}, k, v \rangle$. Letting E be the envelope for leaf ℓ (i.e., the root-to-leaf path), the proof is simply the envelope proof for E . The client verifies the envelope proof against its digest in order to verify that the leaf really is in the tree and therefore that the returned pair (k, v) really is in $L(X)$. Because, for retrieval proofs, the envelope is just a root-to-leaf path, we refer to this as a **root-to-leaf-path proof**. The size of the proof, and the time needed for the client to verify it, are both asymptotically the same as the length of the path.

Insertion proofs. Now consider the insertion of a pair (k, v) , and consider what a proof must accomplish in order to convince the client that the old digest τ should be replaced with some new digest $\hat{\tau}$. The proof must demonstrate that no key-value pair besides (k, v) has been added to the tree. This is accomplished by sending two envelope proofs, one that verifies a **before envelope** E_1 against the old digest τ and one that verifies an **after envelope** E_2 against the new digest $\hat{\tau}$. Intuitively, the envelopes E_1 and E_2 must capture the parts of the tree that change during the insertion. The proof also includes a map φ matching nodes u in the before envelope E_1 with nodes $\varphi(u)$ in the after envelope E_2 , such that u and $\varphi(u)$ have the same opened coordinates as each other in E_1 and E_2 , respectively. To confirm that the server transformed envelope E_1 into envelope E_2 without changing any other parts of the tree, the client not only verifies the envelope proofs for E_1 and E_2 , but also checks that the claimed updates to each $u \in E_1$ (i.e., the VC.Update calls needed

to transform the opened coordinates of u into those of $\varphi(u) \in E_2$) transform the commitment for u into the commitment for $\varphi(u)$. This allows the client to confirm that—unless a cryptographic primitive has been broken—the only changes to the tree are within the differences between the envelopes. To confirm the correctness of the insertion, the client then verifies that the only new pair added by these changes is (k, v) . This proof approach is formalized by the following definition:

Definition 7 (Double-envelope insertion proof). *A double-envelope insertion proof certifies that the insertion of a key-value pair (k, v) into a generalized Verkle tree with digest τ results in a new generalized Verkle tree with digest $\hat{\tau}$. The proof consists of a **before envelope** E_1 with an envelope proof P_1 , an **after envelope** E_2 with an envelope proof P_2 , and a map φ from the nodes of E_1 to the nodes of E_2 . The proof is valid if and only if:*

1. **Verified envelopes.** *The envelope proofs P_1 and P_2 verify E_1 against τ and E_2 against $\hat{\tau}$, respectively, and φ is injective.*
2. **Matched nodes.** *Every node $u \in E_1$ is a non-leaf and maps to a node $\varphi(u) \in E_2$ that has the same type, the same arity, and the same set of opened coordinates. Moreover, if u is a vector-commitment node, then the commitment for $\varphi(u)$ must be the commitment one obtains from the commitment of u by applying VC.Update at the opened coordinates; and if u is a hash node, then the child-digest list of $\varphi(u)$ must agree with that of u at every unopened coordinate.*
3. **Fresh nodes.** *Every node of E_2 outside of the image of φ has all of its unopened coordinates null: its encoding is obtained from that of the canonical empty node of its type and arity by filling in its opened coordinates.*
4. **Revealed leaves.** *Every leaf in the after envelope stores record (k, v) , and at least one such leaf exists.*

The client verifies the double-envelope proof by checking each of the above properties.

Note that the time needed to verify a double-envelope proof is $O(|E_1| + |E_2|)$, since the time spent on each node in either envelope is $O(1)$, plus $O(1)$ for each opened coordinate of that node.

Establishing security. Any retrieval dictionary whose proofs take the two forms just described is secure. We prove this in Appendix A.

Theorem 1 (Security of generalized Verkle trees). *Let $\mathcal{D}$ be an authenticated incremental retrieval dictionary with $n_{\max} \leq \text{poly}(\lambda)$, implemented by a generalized Verkle tree, whose retrieval verifier accepts only root-to-leaf path proofs and whose insertion verifier accepts only double-envelope insertion proofs. If $\mathcal{H}$ is collision resistant and VC is correct and position binding, then $\mathcal{D}$ is secure: specifically, for every PPT adversary $\mathcal{A}$ there are a polynomial $p_{\mathcal{A}}$, PPT adversaries $\mathcal{B}_{\text{hash}}$ and $\mathcal{B}_{\text{vc}}$, and an arity $q_{\mathcal{B}} \leq n_{\max}$ with*

$$\text{Adv}_{\mathcal{A}}^{\text{ADS}}(\lambda, n_{\max}) \leq p_{\mathcal{A}}(\lambda) \left(\text{Adv}_{\mathcal{H}, \mathcal{B}_{\text{hash}}}^{\text{coll}}(\lambda) + \text{Adv}_{\text{VC}, \mathcal{B}_{\text{vc}}, q_{\mathcal{B}}}^{\text{pb}}(\lambda) \right).$$

Conventions for establishing security and proof-size/time bounds. The above theorem immediately implies that each of the retrieval dictionaries built in subsequent sections, which use root-to-leaf path proofs for retrieval queries and double-envelope insertion proofs for insertions, is secure. Both retrieval and insertion proofs can be checked by the client in time proportional to the proof size. Thus, in all of our retrieval-dictionary constructions, the task of bounding client time will be the same as the task of bounding proof size.

3 Exponential Verkle Trees

In this section, we develop the *exponential Verkle tree*, an authenticated retrieval dictionary with $O(n)$ construction time that supports queries in time and proof size $O(\log \log n)$. The main role of the exponential Verkle tree in this paper will be to serve as a building block for the more sophisticated data structures that we construct in later sections. Nonetheless, we will see in this section that, even on its own, the exponential Verkle tree comes with several consequences that are of independent interest.

A (standard) Verkle tree is a balanced k -ary tree, where each internal node is implemented as a vector commitment of size k . Queries to such a tree take time/proof-size $O(\log_k n)$, insertions take time/proof-size $O(k \log_k n)$, and the construction time is $\Theta(nk)$. The main purpose of exponential Verkle trees is to bring down the construction time to $O(n)$.

To improve upon the construction time, in this section, we make use of the following simple (but, as we will see, surprisingly powerful) observation: If we want to construct a Verkle-tree variant that has linear construction time, then we should have low-degree nodes near the leaves and higher-degree nodes near the root. This is because the *total* number of nodes near the root is very small, so we can afford for them to each incur high construction times.

Warmup: Linear-time Verkle trees for $k = 2^{O(\sqrt{\log n})}$. As an example of this paradigm, consider the following simple modification to a Verkle tree in the parameter regime where $k = 2^{O(\sqrt{\log n})}$.

The new data structure is an amalgamation of a Verkle tree and a binary Merkle tree; on top, we have a Verkle tree with n/k total leaves. Each of these leaves is the root of a binary Merkle tree with k leaves. Thus the total fanout of the entire data structure is n . The construction time for the top-layer Verkle tree is $O((n/k)k) = O(n)$, and the total construction time for the n/k different Merkle trees is $O((n/k)k) = O(n)$. Thus the overall construction time for this amalgamated data structure is linear, i.e., $O(n)$.

A retrieval proof consists of a proof both for the Verkle tree with n/k leaves (i.e., a vector commitment proof for each node on the root-to-leaf path), and for a Merkle tree with k leaves. Thus a retrieval proof is of size $O(\log_k n + \log k)$. Recalling that $k = 2^{O(\sqrt{\log n})}$, we have $\log_k n = (\log n)/(\log k) = \Omega(\log k)$, so the retrieval proof size can be rewritten simply as $O(\log_k n)$.

Thus, even with this very simple data structure, we arrive at the following warmup result:

Lemma 1. *For $k = 2^{O(\sqrt{\log n})}$, it is possible to construct a variation of the standard Verkle tree that offers $O(n)$ construction time, while preserving the insertion/query bounds.*

Later in the section, we will see a generalization of Lemma 1 that supports arbitrary $k \leq O(n)$ —indeed, this will actually be a straightforward *application* of the exponential Verkle tree.

The (static) exponential Verkle tree. The above warmup suggests a high-level paradigm for achieving linear-time constructions: as we go down the tree, the fanouts of nodes (i.e., of VCs) should decrease. The exponential Verkle tree takes this idea farther: it carefully selects different fanouts at every layer on the tree in order to achieve linear construction time while *also* achieving an optimal query time of $O(\log \log n)$ for any generalized Verkle tree (for the proof of optimality, see the lower bound in Section 6).

We begin by describing the tree as a static structure. The *exponential Verkle tree* is parameterized by some $\varepsilon \in (0, 0.25)$. It can be viewed as a Verkle tree in which each level of the tree is assigned a different branching factor. The root has branching factor $\lceil n^\varepsilon \rceil$, and each child of the root is itself an exponential Verkle tree with $n/\lceil n^\varepsilon \rceil \pm 1 = \Theta(n^{1-\varepsilon})$ leaves (here ± 1 means an integer within 1 of). For the base case of $n = 1$, the exponential Verkle tree is just a single leaf. Note that by definition, the branching factor at each internal node is at least 2.

Formally, to use the terminology laid out in Section 2, we implement the exponential Verkle tree as a generalized Verkle tree with the branching factors above; and we implement retrieval proofs as root-to-leaf-path proofs.

To analyze the exponential Verkle tree, we begin by bounding its depth.

Lemma 2. *The depth of the exponential Verkle tree is $O(\varepsilon^{-1} \log \log n)$.*

Proof. Fix a leaf with a deepest root-to-leaf path in the tree, and let $n_1, n_2, n_3, \dots, n_d$ be the sequence of subtree sizes in that element's root-to-leaf path.

Some care must be taken to handle rounding issues in the bottom layers of the tree. Specifically, let i be the smallest integer such that $n_i^\varepsilon < 2$. We have then that $n_i < 2^{\varepsilon^{-1}}$, which means that the number of layers $d - i$ after layer i is at most $O(\varepsilon^{-1})$ (in particular, those layers form a binary tree on at most $2^{\varepsilon^{-1}}$ elements). Thus, to complete the proof, it suffices to bound i rather than d .

If we examine the sequence $\log n_1, \log n_2, \log n_3, \dots, \log n_i$, we observe that it is a decreasing geometric sequence with ratio at most $1 - \Omega(\varepsilon)$. Thus we have $i = O(-\log_{1-\Omega(\varepsilon)}(\log n)) = O(\varepsilon^{-1} \log \log n)$, where the equality follows via the inequality $-\ln(1 - x) \geq x$ for $x < 1$. This gives a bound on the total depth of the tree of $O(\varepsilon^{-1} \log \log n)$. $\square$

The bound of $O(\log \log n)$ on depth immediately implies a query time/proof-size bound of $O(\log \log n)$. Next, we show that the construction time is $O(n)$.

Lemma 3. *The exponential Verkle tree has construction time $O(n)$.*

Proof. We bound the construction time with an amortization argument. For a subtree of size m , it costs $\Theta(m^{2\varepsilon})$ to construct the first level of the subtree, and we charge that cost in equal parts to each of its elements, charging $m^{2\varepsilon-1}$ to each leaf. To prove a construction time of $O(m)$, it suffices to show that each individual element incurs a total charge of $O(1)$ across all layers of the tree.

Fix an element, and let $n_1, n_2, n_3, \dots, n_d$ be the sequence of subtree sizes in that element's root-to-leaf path, so that $n_d = 1$ is the leaf holding the element. The element gets charged a total of

$$\sum_{i=1}^d \Theta(n_i^{2\varepsilon-1}). \quad (1)$$

The n_i s form a decreasing geometric sequence with $n_{i+1} \leq \lceil n_i/2 \rceil$, since every internal node has branching factor at least 2. Specifically, we have $n_{i+1} \leq n_i/1.5$, or equivalently $n_i \geq 1.5 n_{i+1}$. Since $n_d \geq 1$, we may therefore conclude for $t \in \{0, 1, \dots, d-1\}$ that $n_{d-t} \geq 1.5^t$, and thus that

$$n_{d-t}^{2\varepsilon-1} \leq \frac{1}{1.5^{(1-2\varepsilon)t}}.$$

Summing over $t \geq 0$, and using that $\varepsilon < 1/4$, we conclude that (1) is a geometric series summing to $O(1)$. $\square$

Supporting insertions in the exponential Verkle tree. As described so far, exponential trees are a static data structure. However, one can straightforwardly support insertions as follows: We fill in the leaves of the tree from left to right, adding new internal nodes when necessary. Additionally, when the tree gets large enough, we increase its depth (by creating a new root), at which point we can continue to fill in leaves left to right.

The insertion proof follows the double-envelope insertion framework laid out in Section 2.4. For this construction, the before and after envelopes are both the root-to-leaf path to the position that the new leaf occupies, and the map φ matches each node of the old path with the corresponding node of the new path. One feature of the path deserves comment: when the insertion opens a fresh part of the tree, the before envelope stops at the null slot where the new subtree will hang, and the internal nodes that the insertion creates below that slot are *fresh* nodes of the after envelope, lying outside the image of φ ; each is built from the canonical empty node of its arity, so all of its unopened coordinates are null. Overall, the insertion proof has size $O(\varepsilon^{-1} \log \log n)$ and security follows from Theorem 1.

Lemma 4. *The worst-case server time for an insertion is at most $O(n^{(1+2\varepsilon)\cdot\varepsilon} + \varepsilon^{-1} \log \log n) = O(n^{2\varepsilon} + \varepsilon^{-1} \log \log n)$. The insertion proof size and client verification time are $O(\varepsilon^{-1} \log \log n)$.*

Proof. Since each vector-commitment proof has size $O(1)$, the insertion proof size (and client verification time) is proportional to the depth of the tree, which is $O(\varepsilon^{-1} \log \log n)$. For the rest of the proof, we focus on bounding insertion time.

The time for an insertion is at most the time to update a root-to-leaf path. If the nodes along the root-to-leaf path have capacities $n_1, n_2, \dots$, then their branching factors are $b_i = \Theta(n_i^\varepsilon + 1)$, and the insertion time is at most

$$\sum_{i \leq O(\varepsilon^{-1} \log \log n)} O(b_i) = O(\varepsilon^{-1} \log \log n) + \sum_{i \leq O(\varepsilon^{-1} \log \log n)} \Theta(n_i^\varepsilon).$$

By the analysis in Lemma 2, we know that the n_i s form a decreasing geometric sequence with $n_{i+1} \leq \lceil n_i/2 \rceil$. So the insertion time is at most

$$\Theta(\varepsilon^{-1} \log \log n + n_1^\varepsilon).$$

Since the left-most subtree of the root is necessarily at capacity, we know that $n \geq \Omega(n_1^{(1-\varepsilon)})$, which implies $n_1 \leq n^{1+2\varepsilon}$. It follows that $n_1^\varepsilon \leq O(n^{(1+2\varepsilon)\cdot\varepsilon} + 1)$, which completes the proof. $\square$

Putting the pieces together, we get the following theorem:

Theorem 2. *Fix a constant $\varepsilon \in (0, 1/4)$. Under the standing cryptographic assumptions of Section 2.2, the exponential Verkle tree is a correct authenticated retrieval dictionary that is secure. It has $O(n)$ construction time. A retrieval query has server time, proof size, and client verification time $O(\varepsilon^{-1} \log \log n)$. An insertion has server time $O(n^{2\varepsilon} + \varepsilon^{-1} \log \log n)$ and proof size and client verification time $O(\varepsilon^{-1} \log \log n)$.*

Proof. The construction time is $O(n)$ by Lemma 3, and the claimed insertion times and proof size follow from Lemma 4. Since a retrieval proof is a root-to-leaf-path proof in a tree of depth $O(\varepsilon^{-1} \log \log n)$, the proof size and client verification time are $O(\varepsilon^{-1} \log \log n)$. The server can use an unauthenticated hash table to find the appropriate leaf for a given retrieval query in $O(1)$ time, allowing the server to also achieve $O(\varepsilon^{-1} \log \log n)$ time for a retrieval query. Finally, security follows from Theorem 1. $\square$

An application: Optimal linear-time Verkle trees. We conclude the section with an application of exponential Verkle trees, in which we show how to construct a Verkle-tree alternative, which we call the *linear-time Verkle tree*. Although the linear-time Verkle tree is ultimately dominated by the data structure in the next section (the exponential Verkle forest), the simplicity of its design, and the fact that, in most parameter regimes, it is a strict improvement on Verkle trees, suggest that it may be of independent interest.

Theorem 3. *Fix $2 \leq k \leq n$. Under the standing cryptographic assumptions of Section 2.2, the linear-time Verkle tree is a correct authenticated retrieval dictionary that is secure. It has $O(n)$ construction time. A retrieval query has server time, proof size, and client verification time $O(\log_k n + \log \log k)$. An insertion has server time $O(k \log_k n)$ and proof size and client verification time $O(\log_k n + \log \log k)$.*

Note that, when $\log_k n \geq \Omega(\log \log n)$ —equivalently, when $\log \log k \leq O(\log_k n)$, so that the second term is absorbed by the first—the insertion/query bound above matches the bound for a standard Verkle tree, *but* with linear construction time. On the other hand, when $\log_k n \leq \log \log n$, the $\log \log k$ term in the insertion/query proof sizes is unavoidable by the lower bound that we prove in Section 6.

The proof of Theorem 3 is achieved by constructing a Verkle tree that converts to an exponential Verkle tree in its final layers:

of Theorem 3. The data structure is an amalgamation of a vanilla Verkle tree and an exponential Verkle tree. On top, we have a Verkle tree with n/k total leaves. Each of these leaves is the root of an exponential Verkle tree with k leaves, and with the parameter ε set to be a small positive constant. The total fanout of the entire data structure is n .

The construction time for the top-layer Verkle tree is $O((n/k)k) = O(n)$, and the total construction time for the n/k different exponential Verkle trees is $O((n/k) \cdot k) = O(n)$, since each has linear construction time. Thus the overall construction time for this amalgamated data structure is linear, i.e., $O(n)$.

A retrieval proof is a root-to-leaf-path proof in the tree. Since the tree has depth $O(\log_k n + \log \log k)$, the proof size and client verification time are both given by this bound. The server uses an unauthenticated hash table to identify the leaf used for the retrieval proof, and is therefore also able to achieve retrieval time $O(\log_k n + \log \log k)$.

Finally, insertion proofs follow the same approach as for the exponential Verkle tree—each insertion is verified using before and after envelopes which are themselves just root-to-leaf paths. This means that the client verification time and proof size are both equal to the depth of the tree, $O(\log_k n + \log \log k)$. Security follows from Theorem 1.

Finally, the server time for an insertion is $O(k \log_k n + k^{2\varepsilon} + \log \log k) = O(k \log_k n)$. □

4 The Amortized Verkle Forest

Using exponential Verkle trees as a building block, we now describe how to implement the *exponential Verkle forest* (called the *Verkle forest* for short). We begin in this section with an amortized version of the data structure. Then, in the next section, we show how to also achieve good worst-case insertion bounds.

Throughout the section, let $\varepsilon \in (0, 1/4)$ be the positive constant that we use to parameterize our exponential Verkle trees. Define $m_1, m_2, \dots$ and $b_1, b_2, \dots$ so that m_i is the maximum capacity

of an exponential Verkle tree with depth i , and $b_i = \lceil m_i^\varepsilon \rceil$ is the branching factor for the root of the tree.

The basic data structure. The Verkle forest consists of $t = O(\log \log n)$ exponential Verkle trees, which we label $V_1, \dots, V_t$. We use $\text{Digest}(V_i)$ to refer to the digest of V_i and $\text{Root}(V_i)$ to refer to the root node of V_i . Each Verkle tree V_i has depth i and has capacity m_i . The root node $\text{Root}(V_i)$ is a vector commitment of size b_i .

Each V_i will interact with V_{i+1} as follows. Whenever V_i **fills up**, meaning that it reaches its capacity of m_i , we do the following:

- We take the tree rooted at $\text{Root}(V_i)$, and we add it to V_{i+1} as a child of $\text{Root}(V_{i+1})$. This is referred to as **updating** V_{i+1} , and takes time $O(b_{i+1})$.
- We replace V_i with an empty tree. This is referred to as **clearing** V_i and takes time $O(1)$ (since empty vector commitments can be created in time $O(1)$).

Insertions are then implemented as follows. To insert an element x , we first insert it into V_1 . We then check if V_1 fills up, and if so we update V_2 and clear V_1 . If this, in turn, fills up V_2 , then we update V_3 and clear V_2 , and so on.

We remark that there are two ways to think about the condition of V_i filling up. One is that V_i has m_i elements, and the other is that V_i has b_i distinct children subtrees. These two conditions are equivalent because, in our algorithm, each of V_i 's subtrees has size m_{i-1} , and $m_i = b_i \cdot m_{i-1}$.

Bridging the V_i s. Notice that the description above is not quite complete, because, to serve as an authenticated data structure, the Verkle forest must produce a single digest, rather than $O(\log \log n)$ different digests $\text{Digest}(V_1), \text{Digest}(V_2), \dots$. Thus, the final step is to maintain what we call a **bridging tree**, which is itself a Merkle tree storing the digests for the V_i s.

It is tempting to implement the bridging tree as a balanced Merkle tree of depth $O(\log \log \log n)$, with the digests of the V_i s in the leaves. But such a solution would come with a drawback: the cost to navigate the bridging tree would end up dominating the (amortized) cost of insertions, increasing it to $\Theta(\log \log \log n)$.

To avoid this issue, we implement the bridging tree in a way that, a priori, might seem worse, but that allows us to achieve good amortized bounds for insertions: we store the digests $\text{Digest}(V_1), \text{Digest}(V_2), \dots$ in a simple hash chain, where the root is $\mathcal{H}(\mathcal{H}(V_1), \mathcal{H}(\mathcal{H}(V_2), \mathcal{H}(\dots, \mathcal{H}(V_t))))$, and the i -th node in the chain is $\mathcal{H}(\mathcal{H}(V_i), \mathcal{H}(\mathcal{H}(V_{i+1}), \mathcal{H}(\dots, \mathcal{H}(V_t))))$. What is nice about this chain is that the small V_i s (which are the ones we access most often during insertions) are close to the start of the chain, whereas the big V_i s (which insertions access much less often) are the ones that are further down the chain. As we will see, this structure allows the Verkle forest to achieve amortized $O(1)$ time insertions, while maintaining $O(\log \log n)$ -sized proofs.

Retrieval proofs. The entire Verkle forest (including both the bridging tree and the V_i s) can be viewed as a generalized Verkle tree as defined in Section 2.

Retrieval proofs are root-to-leaf path proofs in the Verkle forest. Since the bridging tree has depth $O(\log \log n)$ and each V_i also has depth $O(\log \log n)$, the total depth of the generalized Verkle tree is $O(\log \log n)$. Thus retrieval proofs have size and client verification time $O(\log \log n)$. The

server uses an unauthenticated hash table to identify the leaf used for the retrieval proof, and is therefore also able to achieve retrieval time $O(\log \log n)$.

Insertion proofs. Insertion proofs follow the double-envelope insertion proof framework laid out in Section 2.4. We describe the before and after envelopes, along with the injection φ , in detail below.

First, consider what happens on a given insertion. Write B_i for the i -th node of the bridging hash chain, so that B_i is a node of arity 2 whose slots hold $\text{Digest}(V_i)$ and $\text{Digest}(B_{i+1})$, and note that B_1 is the root of the entire forest. Consider an insertion of (k, v) , and let j be the index of the last component that the insertion touches: the components $V_1, \dots, V_{j-1}$ all fill up and are cleared, and V_j is updated. (In the common case, V_1 does not fill up, and $j = 1$.) The subtree that gets added as a child of $\text{Root}(V_j)$ is the old V_{j-1} , which by that point has the old V_{j-2} as a child, which has the old V_{j-3} as a child, and so on down to the old V_1 , which holds the new pair (k, v) in one of its leaves. So a carry chain of length j takes a comb of j components hanging off the bridge and nests them into a single path inside V_j .

With this in mind, the *before envelope* consists of the bridge prefix $B_1, \dots, B_j$; the roots $\text{Root}(V_1), \dots, \text{Root}(V_j)$ hanging off of them; and, at each of those roots, the one slot that the insertion is about to fill (which is null). For $\text{Root}(V_i)$ with $i \geq 2$, that slot is the free coordinate that will receive the carried component V_{i-1} ; for $\text{Root}(V_1)$, it is the empty leaf slot that will receive (k, v) .

The *after envelope* consists of the same bridge prefix $B_1, \dots, B_j$, now with their new digests; for each $i < j$, the canonical empty root that replaces the cleared component V_i , hanging off B_i ; and, hanging off B_j , the new $\text{Root}(V_j)$, whose newly filled coordinate holds the relocated $\text{Root}(V_{j-1})$, whose newly filled coordinate holds the relocated $\text{Root}(V_{j-2})$, and so on down to the relocated $\text{Root}(V_1)$ and, at the end of the path, the new leaf $\langle \text{leaf}, k, v \rangle$.

Finally, the injection φ of Definition 7 sends each B_i to B_i , sends $\text{Root}(V_j)$ to $\text{Root}(V_j)$, and sends each $\text{Root}(V_i)$ with $i < j$ to its relocated copy inside V_j .

Since the before envelope, the after envelope, and the injection φ all satisfy the conditions of Definition 7, the insertion proof is a double-envelope insertion proof, and security follows from Theorem 1.

Bounding amortized insertion time. Having specified the data structure, along with how its retrieval and insertion proofs are constructed, we now argue that the amortized insertion time is $O(1)$.

Lemma 5. *Starting from an empty, the total server time, proof size, and client verification time of the amortized Verkle forest over n insertions are each $O(n)$.*

Proof. We begin by focusing on server time. For $i > 1$, a logical update to V_i occurs when V_{i-1} fills. Adding the old V_{i-1} as a child of V_i takes server time $O(b_i)$; clearing V_{i-1} takes server time $O(1)$; and authenticating and updating the affected locations in the hash-chain bridge takes $O(i)$ server time. Thus the cost of one update to V_i is

$$O(b_i + i) \leq O(ib_i).$$

Over any sequence of n insertions, V_i is updated at most n/m_{i-1} times, because each such update consumes a full component of capacity m_{i-1} . Including the $O(n)$ work at V_1 , the total cost

is therefore

$$O\left(n + \sum_{i \geq 2} \frac{nb_i}{m_{i-1}}\right) = n \cdot O\left(1 + \sum_{i \geq 2} \frac{i}{m_i^{1-2\varepsilon}}\right),$$

where the equality uses that $m_i = b_i m_{i-1}$ and $b_i = \Theta(m_i^\varepsilon + 1)$. Since $m_i \geq 2^{i-1}$ and $\varepsilon < 1/4$,

$$\sum_{i \geq 2} \frac{i}{m_i^{1-2\varepsilon}} \leq \sum_{i \geq 2} \frac{i}{2^{(1-2\varepsilon)(i-1)}} = O(1).$$

Thus the total server time over all insertions is $O(n)$. Finally, for any given insertion, the proof size and client verification time are both (asymptotically) at most the server time, so the total proof size and client verification time over all insertions are also $O(n)$. $\square$

The full result. Putting the pieces together, we have the following analysis of the amortized Verkle forest. Note that the result need not explicitly discuss construction time since the $O(1)$ amortized insertion time immediately implies an $O(n)$ bound on construction time.

Theorem 4. *Under the standing cryptographic assumptions of Section 2.2, the amortized Verkle forest is a correct authenticated retrieval dictionary that is secure. A retrieval query has worst-case server time, proof size, and client verification time $O(\log \log n)$. Starting from the empty dictionary, every sequence of m insertions has total server time, total proof length, and total client verification time $O(m)$; equivalently, each of these insertion costs is $O(1)$ amortized.*

Proof. The $O(\log \log n)$ bound on retrieval proof size and client verification time follows from the fact that the entire Verkle forest, when interpreted as a generalized Verkle tree, has depth $O(\log \log n)$. The server uses an unauthenticated hash table to identify the leaf used for the retrieval proof, so it achieves retrieval server time $O(\log \log n)$ as well. The claim about amortized insertion time and proof size follows from Lemma 5. Finally, security follows from Theorem 1. $\square$

5 The (Deamortized) Verkle Forest

Finally, in this section, we show how to deamortize the exponential Verkle forest to achieve worst-case insertion time/proof-size $O(\log \log \log n)$, while preserving the $O(\log \log n)$ query time/proof-size and the $O(1)$ amortized insertion time/proof-size. We describe the construction in three steps.

Step 1: Flattening the bridging tree. We begin by tackling the following issue. In order to achieve our $O(1)$ amortized insertion bound in the previous section, we implemented the bridging tree (i.e., the tree containing the digests for $V_1, \dots, V_{O(\log \log n)}$) as a hash chain—but, this alone contributes $\Omega(\log \log n)$ to the worst-case insertion time/proof-size.

As we discussed in the previous section, if we were willing to give up on our amortized insertion bound, then we could replace the hash chain with a Merkle tree containing $\text{Digest}(V_1), \dots, \text{Digest}(V_{O(\log \log n)})$ as leaves. This, however, would force every insertion to have a proof of size at least $\Omega(\log \log \log n)$.

Fortunately, with a slightly more careful layout, we can get the best of both worlds. We restructure the bridging tree as a perfect binary Merkle tree Q that stores elements not just in its leaves, but also in its *internal nodes*. The i -th layer of the tree stores $\text{Digest}(V_{2^{i-1}}), \dots, \text{Digest}(V_{2^i-1})$. If a

node $x \in Q$ stores some $\text{Digest}(V_j)$, then the hash of x is defined by $\mathcal{H}(\text{Digest}(V_j), \mathcal{H}_a, \mathcal{H}_b)$, where $\mathcal{H}_a$ and $\mathcal{H}_b$ are the hashes of x 's children in Q (or null if no such children exist).

This restructuring places each $\text{Digest}(V_i)$ at depth $O(\log i)$, which is strictly better (asymptotically) than the depth of i that we got from our hash chain in the previous construction. Thus we are able to preserve the amortized benefits of the hash chain while achieving a maximum depth of $O(\log \log \log n)$ for Q .

Step 2: Deamortizing updates to a given V_i . Recall that when V_{i-1} reaches its capacity m_{i-1} , it gets cleared, and V_i gets updated. The update to V_i sets a previously null coordinate of $\text{Root}(V_i)$ to $\text{Digest}(V_{i-1})$, so that the old V_{i-1} becomes a child of $\text{Root}(V_i)$. Since $\text{Root}(V_i)$ is a vector commitment of size b_i , this update takes time $\Theta(b_i)$. To deamortize the data structure, we must be able to spread this work across multiple operations.

The good news is that the vector commitment naturally splits this work into $\Theta(b_i)$ steps that each take $O(1)$ time: (1) updating the commitment itself, via the **VC.Update** algorithm, and (2) updating each of the b_i cached openings, via the **VC.ProofUpdate** algorithm (see Section 2.2). Of these steps, only the first changes a digest, and so only the first is needed for the insertion proof; the b_i opening refreshes are invisible to the client. Furthermore, it is okay to perform those refreshes lazily, after the insertion, since the cached openings are not used until some query needs them; and, if a query needs an opening that has not yet been refreshed, that opening can be brought up to date, during the query, in $O(1)$ time.

Thus, in principle, we can spread the work of updating V_i out across multiple operations. However, to actually do this, there are two challenges we must overcome.

- **Challenge 1: Scheduling work across the V_j s.** If we spread the work to update a given V_j across multiple operations, then we will end up with several components that are simultaneously mid-update. We cannot afford to work on all $O(\log \log n)$ of the components at once, so we need to schedule when we will work on each one.
- **Challenge 2: The multiple-update problem.** When we are mid-update for a given V_j (i.e., adding a new child to V_j), we do not want to begin an *additional* update to V_j . Thus, we maintain the following **single-update property**: Whenever an update to V_j gets performed, any previous updates to V_j must already be completed.

It is worth remarking on *why* we need to maintain the single-update property. In principle, one could imagine allowing V_j to have multiple updates in flight. However, if we did this, then we would end up with a situation where some cached opening is multiple updates out of date. If a query then needs that opening, it could require $\omega(1)$ time to refresh, which could potentially break the time efficiency of the query. By processing only one update per V_j at a time, we ensure that a query can always refresh any out-of-date cached opening in $O(1)$ time.

Step 3: A scheduling algorithm. We now present a simple scheduling algorithm that addresses both Challenges 1 and 2. The algorithm is parameterized by a constant c that we take to be a sufficiently large positive constant.

At any given moment there may be multiple V_j s that are mid-update. During a given insertion, we spend (up to) $c \log \log \log n$ work on updating the V_j s, where we prioritize working on the smallest V_j s that require our attention (i.e., we prioritize V_j s in the order $V_1, V_2, \dots$). This $c \log \log \log n$ work

includes the time needed to traverse the $O(\log j)$ -long path in the bridging tree to get to a given V_j . Finally, there is also the question of when to *begin* a new update on a given V_j . Whenever we finish an update on some V_{j-1} , we check *at that point in time* whether $|V_{j-1}|$ has reached its capacity m_{j-1} , and only then do we begin an update to V_j (meaning that we replace V_{j-1} with an empty Verkle tree, and we begin the process of adding the old V_{j-1} in as a child in V_j). As above, the carry itself happens all at once and is authenticated all at once; what gets spread across later insertions is the refresh of V_j 's cached openings.

Finally, we add one small modification to our algorithm to handle a certain edge case. During a given insertion, our budget of performing $c \log \log \log n$ work may expire right when we have just started working on some V_j (or perhaps when we are still just traversing the path to V_j in the bridging tree Q). In this case, we spend up to an additional $\Theta(\log j) \leq O(\log \log \log n)$ work on V_j to make sure we amortize the $\Theta(\log j)$ cost of visiting V_j in Q . This ensures what we call the **worthwhile-work property**, which says that whenever we work on V_j , we allocate at least $\Omega(\log j)$ time to it (unless we finish the update in less than that).

Verification and security. Retrieval proofs are root-to-leaf-path proofs, as before. Insertion proofs are the double-envelope proofs of Section 4, with two differences. First, the bridging paths are now root-to-node paths in the bridging tree Q rather than prefixes of a hash chain; since an envelope is just a union of root-to-node paths, this changes the lengths of those paths and nothing else. Second, the V_i s that an insertion touches need not form a prefix $V_1, \dots, V_j$, because the schedule begins a carry into V_j only once the update to V_{j-1} finishes. Nonetheless, the before and after envelopes can still be implemented in the same way as before. For the insertion into V_1 , they contain the path in Q to V_1 , followed by the path inside V_1 down to the leaf slot that receives (k, v) . For each carry of a full V_{j-1} into V_j , they contain the paths in Q to V_{j-1} and to V_j , the roots $\text{Root}(V_{j-1})$ and $\text{Root}(V_j)$, and the coordinate of $\text{Root}(V_j)$ that receives the carry; that coordinate is null in the before envelope, whereas in the after envelope it holds the relocated $\text{Root}(V_{j-1})$, and the slot in Q that V_{j-1} vacates holds the canonical empty root of the cleared component. When several carries nest together during a single insertion, the after envelope reaches down the whole nest that they create, exactly as in Section 4. The injection φ matches the nodes up one carry at a time.

Note that each carry is authenticated at the moment it happens, so what the deamortization changes is only *when* carries occur but not how they are authenticated. The opening refreshes that follow a carry do not change any encodings or digests, so they do not appear in any proofs and the schedule that spreads them out is invisible to the client. An insertion proof is therefore a double-envelope insertion proof, and security follows from Theorem 1.

Analysis. To establish the time bounds claimed for this algorithm, we must prove the *single-update property*.

Lemma 6. *Whenever an update to V_j gets performed, any previous updates to V_j will already be completed, meaning that their cached-opening refreshes have finished.*

Proof. The lemma is trivial for $j = O(1)$, because in this case the update is guaranteed to be completed during a single insertion (i.e., the same insertion during which the update begins). For the rest of the proof, suppose that j is at least a large positive constant.

Consider the time t at which we begin some update to V_j . By the design of the scheduling algorithm, we know that there are no outstanding updates to any of $V_1, \dots, V_{j-1}$ at time t . Additionally, immediately after the update begins, V_{j-1} is empty.

Collectively, the trees $V_1, \dots, V_{j-2}$ contain at most

$$m_1 + \dots + m_{j-2} = O(m_{j-2}) = O(m_{j-1}^{1-\varepsilon}) \leq m_{j-1}/2 \quad (2)$$

elements, where the final inequality uses the fact that j is at least a large positive constant. The next update to V_j will not occur until V_{j-1} reaches size m_{j-1} , which by (2) will not occur for at least $m_{j-1}/2$ insertions.

To complete the proof, it suffices to show that, during the next $m_{j-1}/2$ insertions, at least, say, $\frac{c}{4} \cdot m_j^\varepsilon$ time is allocated to implementing V_j 's update. By the worthwhile-work property, at least a constant fraction of this time will actually be spent on V_j 's update (as opposed to being spent on traversing the path in Q to V_j)—and this is enough to ensure that V_j completes its update.

During the $m_{j-1}/2$ insertions (and since by (2) there are at most $m_{j-1}/2$ elements in the $V_1, \dots, V_{j-1}$ before the insertions), the number of updates to a given V_k , $k < j$, is at most m_{j-1}/m_{k-1} . As each such update takes $O(m_k^\varepsilon + \log k) = O(m_k^\varepsilon)$ work (including the work spent traversing Q to get to V_k each time we work on it), the total work spent on updating V_k is

$$O\left(\frac{m_{j-1}m_k^\varepsilon}{m_{k-1}}\right) = O\left(\frac{m_{j-1}}{m_k^{1-2\varepsilon}}\right).$$

Summing over k , this is at most

$$\sum_{k=1}^{j-1} O\left(\frac{m_{j-1}}{m_k^{1-2\varepsilon}}\right) = O(m_{j-1}).$$

Since the total work budget for the $m_{j-1}/2$ operations is $\Theta(m_{j-1} \cdot c \log \log \log n)$, this leaves much more than $\frac{c}{4} \cdot m_j^\varepsilon$ budget to be spent on V_j —here we are using that $m_j^\varepsilon = O(m_{j-1}^{\varepsilon/(1-\varepsilon)}) = o(m_{j-1})$, since $\varepsilon < 1/2$ —as desired. $\square$

We must also reestablish our amortized analysis from the previous section.

Lemma 7. *The amortized insertion time and proof size is $O(1)$.*

Proof. To bound the server's amortized insertion time, it suffices to show that we spend the same asymptotic time on each update to V_j as we did in the previous (amortized) algorithm. By the worthwhile-work property, the total time spent on a given update to V_j (including time spent traversing Q to get to V_j each time we work on it) is at most $O(m_j^\varepsilon + \log j) = O(m_j^\varepsilon)$, which is the same amount of time that the amortized algorithm would have spent updating V_j .

As in the analysis of the amortized algorithm, the proof size and the client verification time for a given insertion are (asymptotically) at most the server time for that insertion. The lemma therefore follows. $\square$

Putting the pieces together, we prove the main theorem of the paper. Note that, as for Theorem 4, we need not bother including any explicit construction-time bound, since $O(n)$ construction time follows immediately from the $O(1)$ amortized insertion time.

Theorem 5. *Under the standing cryptographic assumptions of Section 2.2, the deamortized Verkle forest is a correct authenticated retrieval dictionary that is secure. A retrieval query has worst-case server time, proof size, and client verification time $O(\log \log n)$, and an insertion has worst-case server time, proof size, and client verification time $O(\log \log \log n)$. Moreover, starting from the empty dictionary, every sequence of m insertions has total server time, total proof length, and total client verification time $O(m)$.*

Proof. The query time/proof-size analysis is bottlenecked by the maximum depth of any V_j , which as in the previous section is $O(\log \log n)$, plus the depth of the bridging tree Q , which is $O(\log \log \log n)$; if the opening that the query needs has not been refreshed yet, the server first applies the necessary update at $O(1)$ cost (there is at most one such update needed by the single-update property). The amortized $O(1)$ insertion analysis follows from Lemma 7. The worst-case $O(\log \log \log n)$ server insertion-time bound follows from the specification of the work-scheduling algorithm. The proof size for an insertion is at most the server time for that insertion, which is at most $O(\log \log \log n)$; and client verification time matches proof size, as discussed in Section 2.4, meaning it is also $O(\log \log \log n)$. Finally, security follows from Theorem 1. $\square$

5.1 Augmenting the Verkle Forest to Support Negative Queries

Finally, in this section, we show how to extend the Verkle forest to be a full-query dictionary (supporting both retrieval and exclusion queries). The basic idea is to maintain a second authenticated data structure (a balanced Merkle tree) whose only job is to support exclusion queries. The final data structure matches the Merkle tree on construction, insertions, and exclusion queries, while still answering retrieval queries in $O(\log \log n)$ time and proof size.

Before continuing, we state the following standard result about balanced Merkle trees [25, 30].

Lemma 8 (Balanced Merkle Trees [25, 30]). *Assuming access to a collision-resistant hash family, it is possible to implement a full-query authenticated dictionary that is secure; that has $O(n)$ construction time; and that guarantees $O(\log n)$ server time, proof size, and client verification time for every operation.*

Define the **augmented Verkle forest** to be an authenticated full-query dictionary implemented as follows. The server stores both a deamortized Verkle forest (as given by Theorem 5) and a balanced Merkle tree (as given by Lemma 8). If τ_F and τ_M are the digests for the Verkle forest and Merkle tree, respectively, then the client stores the single digest

$$\tau = \mathcal{H}_s(\langle \text{augmented-forest}, \tau_F, \tau_M \rangle).$$

Insertions insert into both data structures; retrieval queries certify that a key-value pair is present using the deamortized Verkle forest; and exclusion queries certify that a key-value pair is *not* present using the balanced Merkle tree.

By combining the guarantees of the deamortized Verkle forest and the balanced Merkle tree, we arrive at the following theorem:

Theorem 6. *Under the standing cryptographic assumptions of Section 2.2, the augmented Verkle forest is a correct authenticated full-query dictionary that is secure. It has $O(n)$ construction time, and it supports server time, proof size, and client verification time $O(\log \log n)$ for retrieval queries, and $O(\log n)$ for insertions and exclusion queries.*

Proof. The claimed time and proof size bounds follow immediately by construction; and the claimed correctness follows from the correctness of the constituent data structures. To prove security, observe that a false retrieval proof would break the security of the deamortized Verkle forest, and that a false exclusion proof would break the security of the balanced Merkle tree. It follows by Theorem 5 and Lemma 8 that the data structure is secure. $\square$

6 A Lower Bound

Every retrieval dictionary in this paper is a generalized Verkle tree (Definition 5), and every retrieval proof has size equal to the depth of the leaf where the desired key/value pair resides. We now argue that, if one wishes for $O(n)$ construction time, and if the time to build a VC of arity q is $\Theta(q^2)$ (as in Section 2.2), then the depth of the tree (and therefore the retrieval proof size) must be $\Omega(\log \log n)$. This gives a sense in which the constructions in this paper are optimal.

Theorem 7. *Consider an authenticated retrieval dictionary implemented as a generalized Verkle tree. Suppose that the proof size for a retrieval query is equal to the depth of the corresponding leaf, and suppose that the time to construct a vector commitment with q non-empty entries is $\Omega(q^2)$. If the construction time is $O(n)$, then some retrieval query has proof size $\Omega(\log \log n)$ words.*

Proof. We assume without loss of generality that the tree does not contain any null slots (since the $\Omega(q^2)$ bound on construction time is a function of the number of non-null slots only).

Number the levels of the tree from the root, which is the only node on level 1, and let ℓ_i denote the number of nodes in the first i levels. Focus on some i , and let $q_1, \dots, q_{\ell_i}$ be the arities of those nodes. Building them takes time at least $\Omega(\sum_j q_j^2)$, which by the Cauchy–Schwarz inequality is at least $\Omega(\ell_i \cdot \bar{q}^2)$, where $\bar{q} := \sum_j q_j / \ell_i$ is the average arity.

Since the construction time is $O(n)$, it must be that $\ell_i \cdot \bar{q}^2 \leq O(n)$, which means that $\bar{q} \leq O(\sqrt{n/\ell_i})$. On the other hand, every node at level $i+1$ is a child of a node at level i , so $\ell_{i+1} \leq \ell_i \cdot (1 + \bar{q})$, and we can conclude that

$$\ell_{i+1} = O\left(\ell_i + \sqrt{n\ell_i}\right). \quad (3)$$

Now consider the quantities $r_i := n/\ell_i$. Since $\ell_1 = 1$, we have $r_1 = n$. For $i > 1$, (3) gives

$$r_i = \frac{n}{\ell_i} \geq \Omega\left(\frac{n}{\ell_{i-1} + \sqrt{n\ell_{i-1}}}\right) = \Omega\left(\min(r_{i-1}, \sqrt{r_{i-1}})\right). \quad (4)$$

In particular, $r_i = \Omega(\sqrt{r_{i-1}})$ for every i such that $r_{i-1} \geq 1$.

If the tree has depth d , then all n records sit in leaves within the first d levels, so we must have $\ell_d \geq n$, and thus $r_d \leq 1$. Let $j \leq d$ be the first level with $r_j \leq 1$, so that $r_i > 1$, and hence $r_{i+1} = \Omega(\sqrt{r_i})$, for every $i < j$. Starting from $r_1 = n$, the only way to reach $r_j \leq 1$ is if $j = \Omega(\log \log n)$. Some record therefore sits at depth $d \geq j = \Omega(\log \log n)$, and the retrieval proof for that record is that large, as desired. $\square$

7 Related Work on Authenticated Dictionaries

In this section, we give a more detailed overview of past work on authenticated dictionaries. See, also, [24] for a recent survey on the topic.

Merkle Trees. Merkle trees [26, 28] are a classic cryptographic tool for authenticating data structures, i.e., allowing a client to verify operations that are performed by an untrusted server on a data structure. A patent for the eponymous Merkle tree was filed in 1979, and the corresponding paper appeared at CRYPTO in 1989, having been originally submitted 10 years prior.⁵ Merkle trees are the canonical authenticated data structure.

Merkle trees use a cryptographic hash function $\mathcal{H}$ as a black-boxed primitive. For example, variants of SHA-2 [31] are widely used in practice. Merkle trees rely on the strong collision-resistance property of cryptographic hash functions; forging a false proof for a Merkle tree requires finding a collision in the underlying cryptographic hash function [28].

In its most basic form, a Merkle tree can provide authentication to an arbitrary array of key-value pairs (k_i, v_i) . These key-value pairs are stored in the leaves of the tree. Each leaf is hashed once by itself to produce the second layer of this tree. A binary tree is then built over these initial hashes, where each of the $n - 1$ internal nodes is itself the hash of the concatenation of its two children. The root of this tree, which is itself a hash, serves as a $O(1)$ -sized digest of all of the data in the Merkle tree. Some care must be taken to prevent second preimage attacks (see [4]) in which the server could claim the tree is of a lower than actual depth, and attempt to present the concatenation of two internal hashes as being itself a key value pair. One simple solution is to affix different prefix bits to leaves and internal nodes.

Merkle trees have been extended to authenticated data structures such as 2-3 trees [30], balanced binary search trees, multi-dimensional range trees, tries, skip lists, and I/O-efficient B-tree variants [25].

Vector Commitment Schemes. Vector commitment schemes, introduced by [12] in 2013, allow for a constant-sized commitment C to be computed over a vector of messages m_i , along with a membership proof π_i for each message. Critically, the proofs π_i are constant-sized, allowing for constant query time/proof-size even as the vector grows in length. Updates of elements of the vector take $O(1)$ time to update the commitment (and have constant insertion proof-size), but require $O(q)$ time to update all of the q membership proofs. The discovery of vector commitment schemes continues a long line of cryptographic primitives, including accumulators [7], RSA accumulators [6], and polynomial commitments [20]. See also [44] for lattice-based vector commitments and [42] for vector commitments with aggregatable proofs.

Verkle Trees. Verkle trees⁶ [21, 23, 34, 35] are an authenticated data structure, analogous to Merkle trees, with vector commitments used in the place of cryptographic hash functions. Each node in a Verkle tree is a vector commitment over its children, whereas in a Merkle tree, each node is a cryptographic hash of its children. Verkle trees offer a tradeoff, parameterized by the branching factor k , between proof size and computation time. In particular, Verkle trees have $O(kn)$ construction time, $O(k \log n / \log k)$ insertion/update time, and $O(\log n / \log k)$ sized proofs. Critically, as the branching factor k increases, Verkle membership proofs still only consist of one, constant-sized vector commitment proof per level of the tree. In contrast, when the branching factor k of a Merkle tree is increased, the proof must then contain all $k - 1$ siblings at each level in

⁵The paper was published with the subtitle “That Antique Paper from 1979.”

⁶Etymologically, the Verkle tree is a portmanteau of “V(ector commitment M)erkle tree” or, from a more tongue-in-cheek perspective, “V(ery short M)erkle tree.”

the root-to-leaf path.⁷

In 2008, Papamanthou, Tamassia, and Triandopoulos presented the RSA tree [34], an authenticated data structure based on RSA accumulators [6]. This data structure achieves the same asymptotic bounds as the Verkle tree, with the parameterization that $k = n^\varepsilon$, for some $\varepsilon = \Theta(1)$. In 2010, Libert and Yung [23] discussed using trapdoor q -mercurial commitments to build a Merkle-like tree—essentially the key concept of the Verkle tree. Kuszmaul [21] explicitly analyzed Verkle trees in theory and practice, with k as an arbitrary parameter, and using the vector commitment scheme of Catalano and Fiore [12] as the central building block.

Merkle Mountain Ranges. Merkle mountain ranges [9, 17, 37, 41] are authenticated dictionaries that store $O(\log n)$ Merkle trees $T_1, T_2, \dots$, where each T_i is either empty or contains exactly 2^{i-1} keys. If, at any given moment, there are n keys, then the set of T_i s that are non-empty corresponds to the binary representation of n . A priori, the Merkle mountain range has a digest of size $\Theta(\log n)$, but this can be reduced to a single digest by placing the roots of the trees $T_1, T_2, \dots$ into their own Merkle tree (see, e.g., how we handle the V_i s in Section 4). With this modification, Merkle mountain ranges are an authenticated data structure with $O(1)$ -sized digest that support $O(1)$ amortized insertion time/proof-size and $O(\log n)$ query time/proof-size.

Merkle mountain ranges were first introduced informally by Todd [41], and then also independently by [37] and [17]. See [9] for an in-depth discussion of their history.

8 Acknowledgements

M. A. Bender was supported by NSF grants CCF 2247577, IIS 2551616, Broadcom, Sandia National Laboratories, and the John L. Hennessy Chaired Professorship. W. Kuszmaul was supported by NSF grants CNS-2504471 and CCF-254216, and a Jane Street Research Grant. J. Kuszmaul was supported by NSF Grant CCF-2330048, BSF Grant 2024233, a Simons Investigator Award, and an MIT Akamai Presidential Fellowship.

All of the algorithmic and analytical contributions in this paper are human generated. AIs (GPT 5.6 pro and Claude 5.0) were used to help draft the formal definitions and proofs of security, as well as to help perform editing passes on the paper. All writing contributions by AI were subsequently heavily edited by the authors.

References

- [1] Aris Anagnostopoulos, Michael T Goodrich, and Roberto Tamassia. Persistent authenticated dictionaries and their applications. In *International Conference on Information Security*, pages 379–393. Springer, 2001.
- [2] Arne Andersson. Faster deterministic sorting and searching in linear space. In *Proc. 37th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 135–141, 1996.
- [3] Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13, jun 2007.

⁷Indeed, Merkle trees have minimal proof size with a branching factor of $k = 2$. Thus, we only consider binary Merkle trees.

- [4] Elena Andreeva, Charles Bouillaguet, Orr Dunkelman, Pierre-Alain Fouque, Jonathan Hoch, John Kelsey, Adi Shamir, and Sébastien Zimmer. New second-preimage attacks on hash functions. *Journal of Cryptology*, 29(4):657–696, 2016.
- [5] Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *International Colloquium on Automata, Languages, and Programming*, pages 107–118. Springer, 2009.
- [6] Niko Barić and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *International conference on the theory and applications of cryptographic techniques*, pages 480–494. Springer, 1997.
- [7] Josh Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Workshop on the Theory and Application of Cryptographic Techniques*, pages 274–285. Springer, 1993.
- [8] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.
- [9] Joseph Bonneau, Jessica Chen, Miranda Christ, and Ioanna Karantaidou. Merkle mountain ranges are optimal: On witness update frequency for cryptographic accumulators. In *Advances in Cryptology – CRYPTO 2025*, Lecture Notes in Computer Science, pages 170–202. Springer, 2025.
- [10] Elette Boyle, Ilan Komargodski, and Neekon Vafa. Memory checking requires logarithmic overhead. *Journal of the ACM*, 72(2):1–43, 2025.
- [11] Vitalik Buterin. Verkle trees. <https://vitalik.eth.limo/general/2021/06/18/verkle.html>, June 2021. Blog post.
- [12] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *International Conference on Theory and Practice of Public Key Cryptography*, volume 7778 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2013.
- [13] Scott A Crosby and Dan S Wallach. Efficient data structures for tamper-evident logging. In *USENIX security symposium*, pages 317–334, 2009.
- [14] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *International Colloquium on Automata, Languages, and Programming*, pages 6–19. Springer, 1990.
- [15] Cynthia Dwork, Moni Naor, Guy N Rothblum, and Vinod Vaikuntanathan. How efficient can memory checking be? In *Theory of Cryptography Conference*, pages 503–520. Springer, 2009.
- [16] Ethereum. Verkle trees. <https://ethereum.org/en/roadmap/verkle-trees/>, May 24, 2023.
- [17] Sanjam Garg, Mohammad Hajiabadi, Mohammad Mahmoody, and Ahmadreza Rahimi. Registration-based encryption: removing private-key generator from IBE. In *Theory of Cryptography Conference*, pages 689–718. Springer, 2018.

- [18] Michael T Goodrich, Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *International Conference on Information Security*, pages 80–96. Springer, 2008.
- [19] Michael T Goodrich, Roberto Tamassia, and Andrew Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, volume 2, pages 68–82. IEEE, 2001.
- [20] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *International conference on the theory and application of cryptography and information security*, pages 177–194. Springer, 2010.
- [21] John Kuszmaul. Verkle trees. *MIT PRIMES Report*, 2019. <https://math.mit.edu/research/highschool/primes/materials/2018/Kuszmaul.pdf>.
- [22] William Kuszmaul. A hash table without hash functions, and how to get the most out of your random bits. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 991–1001. IEEE, 2022.
- [23] Benoît Libert and Moti Yung. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In *Proc. 7th Theory of Cryptography Conference (TCC)*, volume 5978 of *Lecture Notes in Computer Science*, pages 499–517. Springer, 2010.
- [24] Harjasleen Malvai, Francesca Falzon, Andrew Zitek-Estrada, Sarah Meiklejohn, and Joseph Bonneau. SoK: Cryptographic authenticated dictionaries. In *Network and Distributed System Security (NDSS) Symposium 2026*. Internet Society, 2026.
- [25] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.
- [26] Ralph C Merkle. Method of providing digital signatures, January 5 1982. US Patent 4,309,569.
- [27] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology (CRYPTO '87)*, pages 369–378. Springer Berlin Heidelberg, 1987.
- [28] Ralph C Merkle. A certified digital signature. In *Advances in Cryptology (CRYPTO)*, pages 218–238. Springer, 1989.
- [29] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 411–423, 2014.
- [30] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. *IEEE Journal on selected areas in communications*, 18(4):561–570, 2000.
- [31] National Institute of Standards and Technology. Secure Hash Standard (SHS). Technical Report FIPS PUB 180-4, National Institute of Standards and Technology, 2012. Updated August 2015.

- [32] Charalampos Papamanthou. *Cryptography for Efficiency: New Directions in Authenticated Data Structures*. PhD thesis, Brown University, USA, 2011.
- [33] Charalampos Papamanthou and Roberto Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *International conference on information and communications security*, pages 1–15. Springer, 2007.
- [34] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 437–448, 2008.
- [35] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables based on cryptographic accumulators. *Algorithmica*, 74(2):664–712, 2016.
- [36] Leonid Reyzin, Dmitry Meshkov, Alexander Chepuronoy, and Sasha Ivanov. Improving authenticated dynamic dictionaries, with applications to cryptocurrencies. In *International Conference on Financial Cryptography and Data Security*, pages 376–392. Springer, 2017.
- [37] Leonid Reyzin and Sophia Yakubov. Efficient asynchronous accumulators for distributed PKI. In *International Conference on Security and Cryptography for Networks*, pages 292–309. Springer, 2016.
- [38] Roberto Tamassia. Authenticated data structures. In *Algorithms – ESA 2003, 11th Annual European Symposium*, pages 2–5. Springer, 2003.
- [39] Roberto Tamassia and Nikos Triandopoulos. On the cost of authenticated data structures. Technical report, Center for Geometric Computing, Brown University, 2003.
- [40] Roberto Tamassia and Nikos Triandopoulos. Certification and authentication of data structures. *AMW*, 619, 2010.
- [41] Peter Todd. Merkle mountain ranges. <https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md>, 2012. GitHub documentation.
- [42] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In *International Conference on Security and Cryptography for Networks*, pages 45–64. Springer, 2020.
- [43] Alin Tomescu, Yu Xia, and Zachary Newman. Authenticated dictionaries with cross-incremental proof (dis)aggregation. *Cryptology ePrint Archive, Report 2020/1239*, 2020.
- [44] Hoeteck Wee and David J Wu. Succinct vector, polynomial, and functional commitments from lattices. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 385–416. Springer, 2023.
- [45] Isaac Zhang, Ryan Zarick, Daniel Wong, Thomas Kim, Bryan Pellegrino, Mignon Li, and Kelvin Wong. QMDB: Quick merkle database. *arXiv preprint arXiv:2501.05262*, 2025.

A Proof of Theorem 1

This appendix provides, for completeness, the full proof of Theorem 1:

Theorem 1 (Security of generalized Verkle trees). *Let $\mathcal{D}$ be an authenticated incremental retrieval dictionary with $n_{\max} \leq \text{poly}(\lambda)$, implemented by a generalized Verkle tree, whose retrieval verifier accepts only root-to-leaf path proofs and whose insertion verifier accepts only double-envelope insertion proofs. If $\mathcal{H}$ is collision resistant and VC is correct and position binding, then $\mathcal{D}$ is secure: specifically, for every PPT adversary $\mathcal{A}$ there are a polynomial $p_{\mathcal{A}}$, PPT adversaries $\mathcal{B}_{\text{hash}}$ and $\mathcal{B}_{\text{vc}}$, and an arity $q_{\mathcal{B}} \leq n_{\max}$ with*

$$\text{Adv}_{\mathcal{A}}^{\text{ADS}}(\lambda, n_{\max}) \leq p_{\mathcal{A}}(\lambda) \left(\text{Adv}_{\mathcal{H}, \mathcal{B}_{\text{hash}}}^{\text{coll}}(\lambda) + \text{Adv}_{\text{VC}, \mathcal{B}_{\text{vc}}, q_{\mathcal{B}}}^{\text{pb}}(\lambda) \right).$$

Throughout, the public parameters pp are honestly generated, and every node is encoded by the canonical typed string prescribed in Definition 5, so that nodes of different kinds never share an encoding. By the *server-side representation* of a generalized Verkle tree X we mean X together with the auxiliary information that VC.Commit produces at each of its vector-commitment nodes; a tree presented in this form can be honestly opened at every coordinate.

We begin with a lemma that gives an explicit algorithm turning any envelope proof that verifies against a tree’s root digest, yet misreports the tree’s contents, into either a hash collision or a position-binding violation.

Lemma 9 (Envelope binding). *There is an algorithm Ext , running in time polynomial in the length of its input, with the following property. Let X be a generalized Verkle tree with root digest τ and with every arity at most $n_{\max}$, and let P be an envelope proof that verifies against τ but is inconsistent with X . Given pp , the server-side representation of X , and P , the algorithm Ext outputs one of the following two witnesses.*

1. Two distinct strings $\rho \neq \rho'$ with $\mathcal{H}_s(\rho) = \mathcal{H}_s(\rho')$.
2. An arity $q \leq n_{\max}$, a commitment C , a coordinate i , two distinct messages $m \neq m'$, and openings π, π' such that

$$\text{VC.Verify}_{\text{pp}_q}(C, i, m, \pi) = \text{VC.Verify}_{\text{pp}_q}(C, i, m', \pi') = 1.$$

Proof. The algorithm is the standard first-divergence argument [28, 34, 35], adapted to a tree that mixes hash nodes with vector-commitment nodes of many different arities. Write A for the envelope of P and ρ_a for the encoding that P claims at $a \in A$. Since P is inconsistent with X , the set $D = \{a \in A : \rho_a \neq \text{enc}_X(a)\}$ of addresses at which P misreports X is nonempty. In polynomial time the algorithm computes D and selects an address $a = (i_1, \dots, i_t) \in D$ of minimum depth, the depth of an address being the number of child slots listed in it; call a the *first divergence*.

Consider any proper prefix $b = (i_1, \dots, i_r)$ of a , where $r < t$, so that b is the address of an ancestor of a . Then $b \in A$, because an envelope contains every prefix of each of its members, and $b \notin D$, because b has depth $r < t$ while t is the least depth of any address in D . Hence $\rho_b = \text{enc}_X(b)$: the proof reports X correctly at every proper prefix of a , and in particular at the address of its parent. We complete the proof by considering three cases.

The first divergence is the root. If $t = 0$, so that $a = \langle \rangle$, then check 1 of Definition 6 gives $\text{dig}(\rho_{\langle \rangle}) = \tau = \text{dig}(\text{enc}_X(\langle \rangle))$, while $\rho_{\langle \rangle} \neq \text{enc}_X(\langle \rangle)$ because $\langle \rangle \in D$. The algorithm outputs these two encodings, a collision for $\mathcal{H}_s$.

The first divergence lies below a hash node. Otherwise $t \geq 1$; write $a = (b \parallel i)$, so that b is the address of the parent and i is the slot of b that a occupies, and recall that $\rho_b = \text{enc}_X(b)$. Suppose ρ_b is a hash-node encoding. Check 4 forces the i -th child digest listed in ρ_b to equal $\text{dig}(\rho_a)$, and that same listed digest equals $\text{dig}(\text{enc}_X(a))$, because $\rho_b = \text{enc}_X(b)$ and a hash node lists exactly the digests of its children. So ρ_a and $\text{enc}_X(a)$ are two distinct encodings with equal digests, and the algorithm outputs the pair.

The first divergence lies below a vector-commitment node. Suppose instead $\rho_b = \langle \text{vc-node}, h, q, C \rangle$. Check 3 supplies an opening π of (C, i) to the message $\text{dig}(\rho_a)$, and it forces $q \leq n_{\max}$, since running that check calls $\text{VC.Verify}_{\text{pp}_q}$ and pp provides parameters only for arities up to $n_{\max}$. Because $\rho_b = \text{enc}_X(b)$, the commitment C is the one that X stores at b , so the algorithm can run VC.Open on the server-side representation of X to obtain an honest opening π' of the same pair (C, i) , this one to the message $\text{dig}(\text{enc}_X(a))$. If $\text{dig}(\rho_a) \neq \text{dig}(\text{enc}_X(a))$, the algorithm outputs $(q, C, i, \text{dig}(\rho_a), \text{dig}(\text{enc}_X(a)), \pi, \pi')$, a witness of kind 2. If the two digests are equal, then ρ_a and $\text{enc}_X(a)$ are two distinct encodings with equal digests, and the algorithm outputs that pair instead.

These three cases are exhaustive. By check 5, an address whose claimed encoding is a leaf encoding or $\emptyset$ has no child in A ; since $a = (b \parallel i)$ does lie in A , the encoding ρ_b is neither of these, and so it is a hash-node or a vector-commitment encoding.

One step above needs care when ρ_a or $\text{enc}_X(a)$ is $\emptyset$, that is, when the proof or the tree declares a to be a null slot. They are not both $\emptyset$, since $a \in D$, but $\emptyset$ is not a string and so cannot serve as half of a collision. At the root the situation does not arise: X has a root node, so $\text{enc}_X(\langle \rangle) \neq \emptyset$, and a proof claiming $\rho_{\langle \rangle} = \emptyset$ fails check 1, because $\text{dig}(\emptyset)$ is a reserved null value while τ is the digest of a genuine encoding. When ρ_b is a hash-node encoding, the algorithm substitutes the string $\langle \text{empty} \rangle$ for whichever of the two is $\emptyset$, which is legitimate because $\text{dig}(\emptyset) = \mathcal{H}_s(\langle \text{empty} \rangle)$ at a hash node. When ρ_b is a vector-commitment encoding, $\text{dig}(\emptyset)$ is instead the reserved message 0, which is not equal to any digest of a genuine encoding; the two digests compared above therefore differ, and the algorithm outputs a witness of kind 2 rather than a collision. $\square$

The next lemma takes a valid double-envelope insertion proof whose before envelope correctly reports the current tree X and builds, in polynomial time, a tree whose root digest is the proposed new digest and whose leaf set is contained in that of X together with the requested pair.

Lemma 10 (Soundness of a double-envelope insertion proof). *Let X be a generalized Verkle tree with root digest τ , let P be a valid double-envelope insertion proof for the insertion of (k, v) from τ to a digest $\hat{\tau}$, and suppose that the before envelope of P is consistent with X . Then there is a generalized Verkle tree X' such that*

1. *the root digest of X' is $\hat{\tau}$; and*
2. *$L(X') \subseteq L(X) \cup \{(k, v)\}$.*

Moreover, the server-side representation of X' is computed from the server-side representation of X and from P in time polynomial in the length of those inputs.

Proof. Write φ for the map from the nodes of the before envelope to the nodes of the after envelope, which the verified-envelopes condition of Definition 7 makes injective, and write ρ'_c for the encoding that the after envelope claims at an address c in it.

Constructing X' . At each address c of the after envelope, let X' hold the encoding ρ'_c , leaving c a null slot when $\rho'_c = \emptyset$. This determines all of X' except the subtrees hanging at the coordinates

that the after envelope leaves unopened, which are filled in as follows. If $c = \varphi(u)$ for a node u of the before envelope, then by the matched-nodes condition of Definition 7 the coordinates left unopened at c are exactly those left unopened at u , and at each such coordinate j we hang the subtree that X has at address $(u \parallel j)$. If instead c lies outside the image of φ , then there is nothing to hang, since the fresh-nodes condition makes every coordinate left unopened at c null. No subtree of X is used twice: distinct matched nodes have distinct preimages under φ , and distinct unopened coordinates j of one node u give distinct addresses $(u \parallel j)$. So X' is a tree, and we can write it down in polynomial time from X and P .

Conclusion 1. The encoding that X' holds at an address c of the after envelope is ρ'_c by construction, provided that ρ'_c describes the children that c actually received: a hash node must list the digests of its children, and a vector-commitment node must commit to them. At a coordinate that c opens, the child received is the one the after envelope names there, whose digest ρ'_c lists or commits to by check 4 or check 3. At a coordinate j that a matched node $c = \varphi(u)$ leaves unopened, the child received is the one X had at $(u \parallel j)$, so its digest is the one that $\text{enc}_X(u)$ listed or committed to at j ; and $\text{enc}_X(u)$ is the before envelope's claim at u , since that envelope is consistent with X . The matched-nodes condition compares ρ'_c with exactly that claim: at a hash node the two child lists agree at every unopened coordinate, and at a vector-commitment node the commitment in ρ'_c is obtained by applying VC.Update at opened coordinates alone, so by the correctness of VC.Update it commits to the digests just described. At a coordinate that a fresh node leaves unopened, the fresh-nodes condition supplies the same conclusion with the canonical empty node of that type and arity in place of u . Hence the after envelope is consistent with X' , and since it verifies against $\hat{\tau}$, check 1 of Definition 6 makes $\hat{\tau}$ the root digest of X' .

Conclusion 2. Let (k', v') be a pair stored by a leaf of X' . By construction that leaf either sits at an address of the after envelope or lies in a subtree hung at an unopened coordinate of some $\varphi(u)$. In the first case the revealed-leaves condition of Definition 7 makes every leaf of the after envelope store (k, v) , so $(k', v') = (k, v)$. In the second case the subtree hung there is a subtree of X , so $(k', v') \in L(X)$. Either way $(k', v') \in L(X) \cup \{(k, v)\}$.

The server-side representation. Finally, the auxiliary information for X' is computed as follows. At a matched vector-commitment node $\varphi(u)$, apply VC.ProofUpdate to the auxiliary information that the server-side representation of X carries at u ; at a fresh node, compute it from the canonical empty node of that arity together with the coordinates the node opens. A node of arity q may leave $\Theta(q)$ of its coordinates unopened, and P exhibits nothing at those coordinates, but VC.ProofUpdate still yields openings there, to the digests that X held at the corresponding coordinates of u . $\square$

We can now prove the theorem, iterating Lemma 10 along the transcript and applying Lemma 9 to the first proof that misreports the tree it verifies against.

of Theorem 1. Fix a PPT adversary $\mathcal{A}$ and consider the security experiment of Definition 4. Alongside the challenger's reference states $X_0, X_1, \dots$, we maintain a sequence of *extracted* trees $\tilde{X}_0, \tilde{X}_1, \dots$, each with its server-side representation. Put $\tilde{X}_0 = X_0$, the tree that honest construction produces from the adversary's initial sequence. If the attempt made from recorded state i requests (k, v) , is accepted with proof P , and has a before envelope consistent with $\tilde{X}_i$, let $\tilde{X}_{i+1}$ be the tree that Lemma 10 supplies for $\tilde{X}_i$ and P ; otherwise put $\tilde{X}_{i+1} = \tilde{X}_i$, a value that will not matter, because a run reaching this case is one of the runs whose probability we bound below. Rejected attempts leave both sequences unchanged. The two sequences need not agree: X_{i+1} is the tree an

honest server would hold, whereas $\tilde{X}_{i+1}$ is built from the proof the adversary actually supplied, and part 1 below shows that it is $\tilde{X}_i$, not necessarily X_i , whose root digest is the recorded τ_i .

Call the run *bad* if either of the following happens.

- Some accepted insertion attempt, made from a recorded state i , carries a proof whose before envelope is inconsistent with $\tilde{X}_i$.
- The adversary's final output is a retrieval tuple (k, v, π) at a recorded state t with $\text{VerifyRetrieve}_{\text{pp}}(\tau_t, k, v, \pi) = 1$, and π is inconsistent with $\tilde{X}_t$.

We show first that $\mathcal{A}$ cannot win a run that is not bad, and then that a run is bad only with negligible probability.

An invariant for runs that are not bad. Suppose the run is not bad. We claim that at every recorded state i ,

1. τ_i is the root digest of $\tilde{X}_i$; and
2. $L(\tilde{X}_i) \subseteq \text{Dict}(X_i)$.

Both hold at $i = 0$: honest construction makes τ_0 the root digest of $X_0 = \tilde{X}_0$, and $\text{Dict}(X_0) = L(X_0)$ because the retrieval dictionaries of Section 2.4 store their pairs in the leaves of the tree, so that Dict and L agree on every tree. Suppose both hold at i and consider an accepted attempt requesting (k, v) , with proof P . Since the run is not bad, the before envelope of P is consistent with $\tilde{X}_i$; it verifies against τ_i , which by part 1 is the root digest of $\tilde{X}_i$. So Lemma 10 applies with $X = \tilde{X}_i$, and the tree it supplies is by definition $\tilde{X}_{i+1}$. Conclusion 1 of that lemma says that $\hat{\tau} = \tau_{i+1}$ is the root digest of $\tilde{X}_{i+1}$, which is part 1 at $i + 1$, and Conclusion 2 gives

$$L(\tilde{X}_{i+1}) \subseteq L(\tilde{X}_i) \cup \{(k, v)\} \subseteq \text{Dict}(X_i) \cup \{(k, v)\} = \text{Dict}(X_{i+1}),$$

where the second containment is part 2 at i and the equality is correctness of the honest rewrite Next_{pp} . This is part 2 at $i + 1$.

No win without a bad run. Suppose the run is not bad and $\mathcal{A}$ wins. Since $\mathcal{D}$ is a retrieval dictionary, the only available win is the retrieval case: $\mathcal{A}$ outputs an index t of a recorded state and a tuple (k, v, π) with $\text{VerifyRetrieve}_{\text{pp}}(\tau_t, k, v, \pi) = 1$ while $(k, v) \notin \text{Dict}(X_t)$. By hypothesis the retrieval verifier accepts only root-to-leaf-path proofs, so π is an envelope proof for a single root-to-leaf path that verifies against τ_t and whose terminal address stores the encoding $\langle \text{leaf}, k, v \rangle$. Because the run is not bad, π is consistent with $\tilde{X}_t$, so $\langle \text{leaf}, k, v \rangle$ is the encoding that $\tilde{X}_t$ genuinely holds at the address ending that path; that is, $\tilde{X}_t$ has a leaf there storing (k, v) . Hence $(k, v) \in L(\tilde{X}_t) \subseteq \text{Dict}(X_t)$ by part 2 of the invariant, contradicting $(k, v) \notin \text{Dict}(X_t)$. Therefore $\text{Adv}_{\mathcal{A}}^{\text{ADS}}(\lambda, n_{\max}) \leq \Pr[\text{the run is bad}]$.

Bad runs break a primitive. Suppose the run is bad, and consider the earlier of the two bad events to occur. Let P^* be the proof it involves: the before envelope of the first accepted insertion attempt whose before envelope is inconsistent with the extracted tree at that state, or else the final retrieval proof. Let $\tilde{X}^*$ and τ^* be the extracted tree and the recorded digest at the state against which P^* is verified. Neither bad event occurred before that state, so the induction above establishes part 1 there, making τ^* the root digest of $\tilde{X}^*$. Thus P^* verifies against the root digest of $\tilde{X}^*$ but is inconsistent with it, which is exactly the hypothesis of Lemma 9. Its remaining hypothesis holds too: every arity occurring in $\tilde{X}^*$ is at most $n_{\max}$. This holds in $\tilde{X}_0$, and each

tree that Lemma 10 builds consists of nodes of an accepted after envelope together with subtrees of the previous extracted tree; check 3 runs $\text{VC.Verify}_{\text{pp}_q}$ at every vector-commitment node of an accepted envelope, and pp provides parameters only for arities up to $n_{\max}$. So Ext , applied to pp , the server-side representation of $\tilde{X}^*$, and P^* , returns either a collision for $\mathcal{H}_s$ or a position-binding violation at some arity $q \leq n_{\max}$. Let Bad_{coll} and Bad_{pb} be the events that the run is bad and Ext returns, respectively, a collision or a position-binding violation. These two events are disjoint and together exhaust badness, so $\Pr[\text{the run is bad}] = \Pr[\text{Bad}_{\text{coll}}] + \Pr[\text{Bad}_{\text{pb}}]$.

The reductions. We now build the two adversaries $\mathcal{B}_{\text{hash}}$ and $\mathcal{B}_{\text{vc}}$. Both assemble public parameters, simulate the whole experiment for $\mathcal{A}$ while playing the challenger themselves, and then test whether the run was bad and, if so, locate P^* and run Ext . The simulation is polynomial-time: holding pp , they can run Construct and every verifier, compute each reference state as $\text{Next}_{\text{pp}}(X_i, k, v)$, and maintain the extracted trees and their server-side representations by the algorithm of Lemma 10; those trees stay of polynomial size, because $\tilde{X}_0$ is of polynomial size, each accepted proof is of polynomial size, and there are polynomially many of them.

The adversary $\mathcal{B}_{\text{hash}}$ takes the hash key s from the collision challenger, samples $\text{pp}_q \leftarrow \text{VC.KeyGen}(1^\lambda, q)$ itself for every arity $q \leq n_{\max}$, and outputs the colliding pair when Ext returns one. The adversary $\mathcal{B}_{\text{vc}}$ cannot proceed the same way, because its challenger hands it pp_q for one arity only; it therefore draws $q_{\mathcal{B}}$ uniformly from the at most $n_{\max}$ arities for which pp provides parameters, takes $\text{pp}_{q_{\mathcal{B}}}$ from the position-binding challenger, samples s and the parameters for every other arity honestly, and outputs the violation when Ext returns one at arity $q_{\mathcal{B}}$. Both sets of parameters have exactly the honest distribution, and $q_{\mathcal{B}}$ is independent of the run, so

$$\text{Adv}_{\mathcal{H}, \mathcal{B}_{\text{hash}}}^{\text{coll}}(\lambda) \geq \Pr[\text{Bad}_{\text{coll}}], \quad \text{Adv}_{\text{VC}, \mathcal{B}_{\text{vc}}, q_{\mathcal{B}}}^{\text{pb}}(\lambda) \geq \Pr[\text{Bad}_{\text{pb}}]/n_{\max}.$$

The arity is the only thing either adversary must commit to in advance: because each simulates the run in full, neither has to guess *where* in the transcript the failure occurs. Combining the bounds above gives

$$\begin{aligned} \text{Adv}_{\mathcal{A}}^{\text{ADS}}(\lambda, n_{\max}) &\leq \text{Adv}_{\mathcal{H}, \mathcal{B}_{\text{hash}}}^{\text{coll}}(\lambda) + n_{\max} \cdot \text{Adv}_{\text{VC}, \mathcal{B}_{\text{vc}}, q_{\mathcal{B}}}^{\text{pb}}(\lambda) \\ &\leq n_{\max} \left(\text{Adv}_{\mathcal{H}, \mathcal{B}_{\text{hash}}}^{\text{coll}}(\lambda) + \text{Adv}_{\text{VC}, \mathcal{B}_{\text{vc}}, q_{\mathcal{B}}}^{\text{pb}}(\lambda) \right). \end{aligned}$$

This is the bound claimed by the theorem, with $p_{\mathcal{A}}(\lambda) = n_{\max}$, which is polynomial in λ because $n_{\max} \leq \text{poly}(\lambda)$; collision resistance and position binding then make the right-hand side negligible. $\square$