Transparency Logs via Append-Only Authenticated Dictionaries

Alin Tomescu*, Vivek Bhupatiraju†, Dimitrios Papadopoulos‡,
Charalampos Papamanthou§, Nikos Triandopoulos¶, Srinivas Devadas*

*MIT CSAIL, Cambridge, MA
†Lexington High School, Lexington, MA
‡Hong Kong University of Science and Technology, Hong Kong
§University of Maryland, College Park, MD
¶Stevens Institute of Technology, Hoboken, NJ

***Abstract*—Transparency logs allow users to audit a potentially malicious service, paving the way towards a more accountable Internet. For example, Certificate Transparency (CT) enables domain owners to audit Certificate Authorities (CAs) and detect impersonation attacks. Yet to achieve their full potential, transparency logs must be *efficiently* auditable. Specifically, everyone should be able to verify both *(non)membership* of log entries *and* that the log remains *append-only*. Unfortunately, current transparency logs either provide small-sized (non)membership proofs or small-sized append-only proofs, but never both. In fact, one of the proofs always requires bandwidth linear in the size of the log, making it expensive for everyone to audit the log and resulting in a few "opaque" *trusted* auditors. In this paper, we address this gap with a new primitive called an *append-only authenticated dictionary* (AAD). Our construction is the first to achieve (poly)logarithmic size for both proof types. Moreover, our experimental evaluation is very encouraging: for reasonable application scenarios, our AAD reduces the total communication bandwidth in transparency schemes by more than $200\times$, compared to previous approaches.**

I. Introduction

Security is often bootstrapped from a *public-key infrastructure (PKI)*. For example, on the web, *Certificate Authorities (CAs)* digitally sign *certificates* that bind a website to its public key. This way, a user who successfully verifies the certificate can set up a secure channel. In general, many systems (e.g., WhatsApp [1]) require a PKI or assume one exists [2]–[5].

Despite their necessity, PKIs have proven difficult to secure as evidenced by past CA compromises [6]–[8]. To address such attacks, *transparency logs* [9]–[11] have been proposed as a way of building *accountable* (and thus more secure) PKIs. A transparency log is managed by a *log server*, who periodically appends *key-value pairs*, and is queried by *users*, who want to know certain keys' values. For example, in *key transparency* [10], [12]–[18], CAs are required to publicly log certificates they issue (i.e., values) for each domain (i.e., keys). Fake certificates can thus be detected in the (honest) log, holding CAs accountable for their misbehavior.

Transparency logging is becoming increasingly important in today's Internet. This is evident with the widespread deployment of Google's Certificate Transparency (CT) [10] project. Since its initial March 2013 deployment, CT has publicly logged over 2.1 billion certificates [19]. Furthermore, since April 2018, Google's Chrome browser requires all new certificates to be published in a CT log [20]. In the same spirit, there has been increased research effort into *software transparency* schemes [21]–[26] for securing software updates [27], [28]. Furthermore, Google is prototyping *general transparency* logs [11], [29] via their Trillian project [29]. Therefore, it is not far-fetched to imagine generalized transparency improving our census system, our elections, and perhaps our government.

To realize their full potential, transparency logs must operate correctly or be easily caught otherwise. Specifically, transparency logs must have three key properties:

Logs should remain append-only. In a log-based PKI, a devastating attack is still possible: a malicious CA can publish a fake certificate in the log but later collude with the log server to have it removed, which prevents the victim from ever detecting the attack. Transparency logs should therefore be able to prove that they remain *append-only*, i.e., the new version of the log still contains all entries of the old version. One trivial way to provide such a proof is to return the newly-added entries to the user and have the user enforce a subset relation. But this is terribly inefficient. Instead, in a real-world system we want a client with a "short" digest h_{old} to accept a new digest h_{new} only if it comes with a small *append-only proof* computed by the log. This proof should convince the client that the old log with digest h_{old} is a subset of the new log with digest h_{new}.

Logs should not equivocate. When users have access to digests (instead of whole logs), the central question becomes: How can the client check *against their digest* which values are registered for a certain key k in the log? This is done via a small *(non)membership proof*, which we also refer to as a *lookup proof*. Importantly, this proof should convince a client that the server has returned *nothing more or less than all values* of key k. Otherwise, the server could *equivocate* and present one set of values V for k to a user and a different set V' to some other user.

Logs should remain fork-consistent. An unavoidable issue in this setting is that the log server can *fork* users [3], [9]. In

TABLE I
ASYMPTOTIC COSTS OF OUR CONSTRUCTION VERSUS PREVIOUS WORK

Time & Bandwidth	Space	Append time	(Non)membership proof size	Append-only proof size
Prefix tree [13], [32]	$n \log n$	$\log n$	$\log n$	n
History tree [9], [10]	n	$\log n$	n	$\log n$
AAD (this work)	$\lambda n \log n$	$\lambda \log^3 n$	$\log^2 n$	$\log n$

other words, although each user keeps their view of the log append-only by verifying proofs, users' logs could still differ. For example, the server can append (k, v) to one user's log while withholding it from another user's view of the log and instead appending (k, v'). To counter this, logs can provide a property called *fork consistency* [3], [30]. Intuitively, fork consistency guarantees that once two users have been forked, their digests will diverge forever afterwards. As a result, users can easily detect forks by periodically checking if they have the same digest at some version i, e.g., via *gossiping* [25], [31].

Limitations of existing approaches. Unfortunately, current transparency logs have to pick between low-bandwidth append-only proofs [9], [10] or low-bandwidth lookup proofs [13], [32]. For example, in CT [10], users (i.e., domain owners) must periodically *monitor* the log by looking up their own certificates to detect impersonation. However, because CT lacks efficient lookups, the only way for users to monitor is to download every certificate after it is appended. Unfortunately, this blows up the log server's communication cost: assuming 30 million users who monitor, the log server would need 99.5 GBps of bandwidth (see Section VI-E3). Due to this, domain owners tend to outsource monitoring to *opaque* trusted third-parties, which defeats the purpose of transparency.

Similarly, CONIKS [13] periodically publishes a new version of the log that *should* include all past entries. However, CONIKS lacks efficient append-only proofs: there is no efficient way for users to check no entries were removed. Instead, users only check that their own public key has not been changed or removed, but must do so *in every updated version of the log*. Unfortunately, this blows up the CONIKS server's communication cost. For example, suppose WhatsApp [1] used CONIKS for key transparency, with 0.001% of its 1 billion users updating their key daily (mild assumption). With a 960-byte membership proof, the total server bandwidth would be 111 GBps—without even accounting for key revocations.

Our contribution. In this paper, we show how to build *efficiently-auditable* transparency logs using a novel cryptographic primitive called an *append-only authenticated dictionary (AAD)* which maps a key to one or more values in an append-only fashion. We present an AAD construction from *bilinear accumulators* [34] (see Sections III to V). Our construction *is the first* to offer logarithmic-sized append-only proofs, polylogarithmic-sized lookup proofs and polylogarithmic worst-case time appends. (See Table I for comparison with previous works: n is the number of key-value pairs in the dictionary and λ is the bit-size of one key-value pair.) We prove its security under the q-SBDH and q-PKE assumptions.

We also implement and evaluate our construction in Section VI. We show that our lookup and append-only proofs are in the order of KBs and our verification time is in the order of seconds. For example, a proof for a key with 32 values in a dictionary of 10^6 entries is 94 KB and verifies in 2.5 seconds. While our membership proof sizes are much larger than in previous work [9], [13], our small-sized append-only proofs easily make up for it. In fact, in reasonable scenarios, we vastly decrease the total bandwidth consumption of CT from 99.5 GBps to 400 MBps ($254\times$) and of CONIKS from 111 GBps to 544 MBps ($208\times$) (see Section VI-E3).

A. Overview of our techniques

Our starting point is to first build an *append-only authenticated set* (AAS) by enhancing *bilinear accumulators* (see Section II-A). We later modify our AAS into an append-only authenticated dictionary (AAD). A bilinear accumulator [34] can be used to represent a set $\mathcal{Y} = \{y_1, \ldots, y_n\}$ with a secure digest $d(\mathcal{Y}) = g^{\mathcal{Y}(s)}$, where $\mathcal{Y}(x) = \prod_{i=1}^{n}(x - y_i)$ is the characteristic polynomial of $\mathcal{Y}$ and s is a trapdoor. The accumulator naturally supports append-only proofs for $\mathcal{Y} \subseteq \mathcal{Z}$: given $d(\mathcal{Y})$ and $d(\mathcal{Z})$, the proof is $d(\mathcal{W})$, where $\mathcal{W}(x) = \mathcal{Z}(x)/\mathcal{Y}(x)$. Moreover, one can naturally define (non)membership proofs too. In principle, the bilinear-map accumulator *as is* gives us a secure AAS with *short proofs* but, unfortunately, *computing these proofs* requires time at least linear in the set sizes, which is not practical.

Precomputing membership proofs. We reduce proof computation cost in bilinear accumulators by *precomputing* all proofs. Naively, it is already possible to precompute membership proofs, since there are only n of them. Unfortunately, this requires $O(n^2)$ time which is prohibitive for large sets. Our first contribution is to design a new bilinear accumulator called a *bilinear tree* where elements are stored at the leaves of a binary tree and every tree node stores an accumulator of its subtree. A bilinear tree can be computed in $O(n \log^2 n)$ time and, once ready, it has all n membership proofs precomputed. This leads to an amortized $O(\log^2 n)$ cost for computing a membership proof, which will be crucial later when we efficiently dynamize our data structure [35], [42].

Precomputing non-membership proofs. Unlike membership proofs, it seems that non-membership proofs are inherently-hard to precompute, since the elements that are not in the accumulator can be exponentially-many. To overcome this problem, for every element y_i in our set, we include in the accumulator all prefixes of y_i's binary representation. In this manner, non-membership of an element can be demonstrated by proving non-membership for one of its prefixes. Importantly, a certain prefix ρ can be shared by a potentially-large subset S of elements that are not present. Thus a non-membership for ρ can be used as a non-membership proof for every element in S! This technique will allow us to

precompute non-membership proofs for $O(2^\lambda)$ elements in $O(\lambda n \log^2(\lambda n))$ time.

Efficient appends and append-only proofs. Our discussion so far does not address how to update a bilinear tree with a new element. The naive approach would be to simply *recompute it after every append* in $O(\lambda n \log^2(\lambda n))$ time, but this is prohibitive. Instead, we use standard techniques from Overmars [35], [42] to dynamize any static data structure and achieve appends in $O(\lambda \log^3 n)$ worst-case time. This comes at the expense of a multiplicative logarithmic factor in our append time and our proof sizes. Finally, after applying the Overmars technique, our resulting data structure lends itself easily to append-only proofs. Our append-only proof is $O(\log n)$-sized and reduces to proving that a certain number of bilinear trees are subsets of a new bilinear tree.

B. Related work

Previous work on authenticated data structures (ADS) falls into two categories. First, some have succinct append-only proofs but lack succinct lookup proofs [9], [10], [36]–[38]. Second, other work has succinct lookup proofs but lacks succinct append-only proofs [17], [32], [39], [40]. Our work extends these works with efficient proofs for both operations. Our work can also be regarded as an extension of bilinear accumulators [34], [41]. As discussed in Section I-A, we enhance bilinear accumulators with precomputed proofs of non-membership, membership and append-only (at the expense of slightly larger proofs). We also improve update times in accumulators from linear time to *worst-case* polylogarithmic time using de-amortization techniques [35], [42].

Many transparency log designs were proposed after CT [12], [13], [15], [16], [18], [24], [43]. However, due to the limitations of ADSs explained above, all proposed designs provide efficient lookup proofs at the cost of inefficient append-only proofs. Some designs work around this by relying on trusted third parties to maintain the append-only property [15], [16], [24], [43]. Other designs require users to collectively monitor the log to ensure it remains append-only [12], [13], [18], [44]. Unfortunately, similar to CONIKS, these designs also require too much bandwidth from auditors and log servers. In contrast, our design allows everybody to audit efficiently via logarithmic-sized lookup and append-only proofs.

Previous work formalizes Certificate Transparency (CT) [45], [46] and general transparency logs [45]. In contrast, our work formalizes append-only authenticated dictionaries (AAD) and sets (AAS), which can be used directly as a transparency log. Our AAD abstraction is more expressive than the *dynamic list commitment (DLC)* abstraction introduced in previous work [45]. Specifically, DLCs are append-only lists with non-membership by insertion time, while AADs are append-only dictionaries with non-membership by arbitrary keys. Furthermore, AADs can easily be extended to support non-membership by insertion time as well. Perhaps most importantly, AAD-based transparency logs do not need to be monitored by trusted third parties [45], [46]. Instead, AADs can be audited by everybody via efficient lookup proofs and efficient append-only proofs. Finally, previous work carefully formalizes proofs of misbehavior for transparency logs [45], [46]. Although AADs provide proofs of misbehavior, we do not formalize them in this paper.

Lastly, previous work improves or extends transparency logging in various ways but does not tackle the append-only problem [47]–[49]. Also, neither our work nor previous work adequately models the network connectivity assumptions needed to detect forks in a gossip protocol.

II. Preliminaries

In this section we define some notation used throughout Sections IV and V, and we introduce our cryptographic assumptions and bilinear accumulators [34], [50].

Notation. Let λ denote our security parameter. Let $\mathcal{H}$ denote a collision-resistant hash function (CRHF) with 2λ-bits output. We use multiplicative notation for all algebraic groups in this paper. Let $\mathbb{F}_p$ denote the finite field "in the exponent" associated with a group $\mathbb{G}$ of prime order p. Let $\mathsf{poly}(\cdot)$ denote any function upper-bounded by some univariate polynomial. Let $\varepsilon(\cdot)$ denote any negligible function and $\eta(\lambda) = 1 - \varepsilon(\lambda)$. Let $\log x$ be shorthand for $\log_2 x$. Let $[n] = \{1, 2, \ldots, n\}$ and $[i, j] = \{i, i+1, \ldots, j-1, j\}$. Let $\mathcal{PP}_q(s) = \langle g^s, g^{s^2}, \ldots, g^{s^q} \rangle$ denote public parameters used in the q-SBDH assumption and let $\mathcal{PP}_q(s, \tau) = \langle g^s, g^{s^2}, \ldots, g^{s^q}, g^{\tau s}, g^{\tau s^2}, \ldots, g^{\tau s^q} \rangle$ denote public parameters used in the q-PKE assumption (see Section II). Let ε denote the empty string.

Cryptographic assumptions. Our work relies on the use of *pairings* or *bilinear maps*, first introduced by Menezes et al [51] and used in many later works [34], [50], [52]–[54]. Recall that a bilinear map $e(\cdot, \cdot)$ has useful algebraic properties: $e(g^a, g^b) = e(g^a, g)^b = e(g, g^b)^a = e(g, g)^{ab}$. To simplify exposition, throughout the paper we assume symmetric (Type I) pairings. Our assumptions can be restated in the setting of the more efficient asymmetric (Type II and III) pairings in a straight-forward manner. Indeed our implementation in Section VI uses asymmetric pairings.

Definition II.1 (Bilinear pairing parameters)**.** Let $\mathcal{G}(\cdot)$ be a randomized polynomial algorithm with input a security parameter λ. Then, $\langle \mathbb{G}, \mathbb{G}_T, p, g, e \rangle \leftarrow \mathcal{G}(1^\lambda)$ are called bilinear pairing parameters if $\mathbb{G}$ and $\mathbb{G}_T$ are cyclic groups of prime order p where discrete log is hard, $\mathbb{G} = \langle g \rangle$ (i.e., $\mathbb{G}$ has generator g) and if e is a bilinear map, $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ such that $\mathbb{G}_T = \langle e(g, g) \rangle$.

Our AAD construction from Section V is provably secure under the q-SBDH [55] and q-PKE assumptions [56] over elliptic curve groups with bilinear pairings, which we define in Appendix A.

A. Bilinear Accumulators

A *bilinear accumulator* is a cryptographic commitment to a set $T = \{e_1, e_2, \ldots, e_n\}$ that supports (non)membership, subset and disjointness proofs [34], [41]. We refer to T as the *accumulated set*.

Committing to a set. Let $\mathcal{C}_T(x) = (x-e_1)(x-e_2)\cdots(x-e_n)$ denote the *characteristic polynomial* of T and s denote a trapdoor that nobody knows. The accumulator $\mathsf{acc}(T)$ of T is computed as $\mathsf{acc}(T) = g^{\mathcal{C}_T(s)} = g^{(s-e_1)(s-e_2)\cdots(s-e_n)}$. The trapdoor s is generated during a *trusted setup phase* after which nobody knows s. Specifically, given an upper-bound q on the set size, this phase returns q-SDH public parameters $\mathcal{PP}_q(s) = \langle g^s, g^{s^2}, \ldots, g^{s^q}\rangle$. Given coefficients $c_0, c_1, \ldots, c_n$ of $\mathcal{C}_T(\cdot)$ where $n \leq q$, the accumulator is computed using the q-SDH public parameters, without knowledge of s:

$$\begin{aligned}\mathsf{acc}(T) &= g^{c_0}(g^s)^{c_1}(g^{s^2})^{c_2}\cdots(g^{s^n})^{c_n} \\ &= g^{c_0+c_1 s+c_2 s^2\cdots c_n s^n} = g^{\mathcal{C}_T(s)}\end{aligned}$$

In other words, the server computes a *polynomial commitment* [34], [50] to the characteristic polynomial of T. Note that the bilinear accumulator supports sets of elements from $\mathbb{F}_p$. To avoid ambiguity, given a domain $\mathcal{D}$ we define a function $\mathcal{H}_{\mathbb{F}} : \mathcal{D} \to \mathbb{F}_p$ that maps elements to be accumulated to values in $\mathbb{F}_p$. To accumulate sets with elements of larger size (i.e., $|\mathcal{D}| > |\mathbb{F}_p|$), one can always reduce each element down to $\log_p$ bits first using a collision-resistant hash function.

Membership proofs. A *prover* who has T can convince a *verifier* who has $\mathsf{acc}(T)$ that an *element* e_i is in the set T. The prover simply convinces the verifier that $x - e_i \mid \mathcal{C}_T(x)$ by presenting a commitment $\pi = g^{q(s)}$ to a quotient polynomial $q(\cdot)$ such that $\mathcal{C}_T(x) = (x-e_i)q(x)$. Using a bilinear map, the verifier checks the property above holds at $x = s$, which under q-SDH is enough for security [50]:

$$e(g, \mathsf{acc}(T)) \stackrel{?}{=} e(\pi, g^s/g^{e_i}) \Leftrightarrow e(g,g)^{\mathcal{C}_T(s)} \stackrel{?}{=} e(g,g)^{q(s)(s-e_i)}$$

Non-membership proofs. To prove that some element u is *not* in the set T, the prover convinces the verifier that $x - u \mid \mathcal{C}_T(x) - y$ for some $y \neq 0$. Specifically, the prover presents a commitment $\pi = g^{q(s)}$ to a quotient polynomial $q(\cdot)$ such that $\mathcal{C}_T(x) - y = (x-u)q(x)$ [41], [50]. The verifier checks that $y \neq 0$ and that $e(g, \mathsf{acc}(T)/g^y) \stackrel{?}{=} e(\pi, g^s/g^u)$.

Subset proofs. To prove that $A \subseteq B$, the prover shows that $\mathcal{C}_A(x) \mid \mathcal{C}_B(x)$. Specifically, the prover presents a commitment $\pi = g^{q(s)}$ of a quotient polynomial $q(\cdot)$ such that $\mathcal{C}_B(x) = q(x)\mathcal{C}_A(x)$. The verifier checks that $e(g, \mathsf{acc}(B)) \stackrel{?}{=} e(\pi, \mathsf{acc}(A))$.

Disjointness proofs. Suppose we have sets A and B such that $A \cap B = \varnothing$. To prove disjointness, the prover uses the Extend Euclidean Algorithm (EEA) [57]. Specifically, the prover computes Bézout coefficients $y(\cdot)$ and $z(\cdot)$ such that $y(x)\mathcal{C}_A(x) + z(x)\mathcal{C}_B(x) = 1$. The proof consists of commitments to the Bézout coefficients $\gamma = g^{y(s)}$ and $\zeta = g^{z(s)}$. The verifier checks that $e(\gamma, \mathsf{acc}(A))e(\zeta, \mathsf{acc}(B)) \stackrel{?}{=} e(g,g)$.

Fast Fourier Transform (FFT). We use FFT [58] to speed up polynomial interpolation, multiplication and division. In particular, for polynomials of degree-bound n, we divide and multiply them in $O(n \log n)$ field operations [59], [60]. We interpolate a polynomial from its n roots in $O(n \log^2 n)$ field operations [61]. We compute Bézout coefficients for two polynomials of degree-bound n using the Extended Euclidean Algorithm (EEA) in $O(n \log^2 n)$ field operations [57].

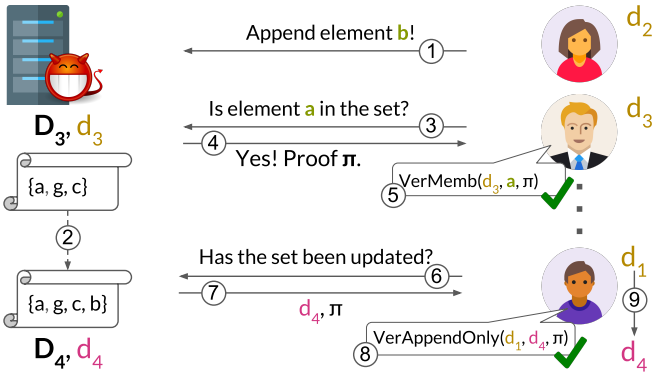


Fig. 1. Our model consists of a single malicious *server* who manages a *set* and an unbounded number of *clients* who query the set. The server stores the full set while clients only store a small-sized *digest*. Clients will not necessarily have the latest digest associated with the latest set. The clients can (1) append a new element to the set, (2) query for an element and (3) update their digest of the set. Clients query the server concurrently and an honest server should serialize all appends. Clients verify a (non)membership proof when querying for an element. Clients will also verify an append-only proof when updating their digests to ensure no elements were removed.

III. Append-only Authenticated Set (AAS)

We begin by introducing a new primitive called an *append-only authenticated set* (AAS). An AAS can be used for *revocation transparency* as proposed by Google [62]. We later modify our AAS into an *append-only authenticated dictionary* (AAD), which can be used for generalized transparency [11].

Overview. An AAS is a set of *elements* that can only be appended to and is efficiently auditable. An AAS is managed by an *untrusted server* and queried by many *mutually-distrusting clients* who *append* and look up elements (see Figure 1). Initially, the set starts out empty at *version* zero, with new appends increasing its size and version by one. Importantly, once an element has been appended to the set, it remains there forever: an adversary cannot remove the element.

Clients append elements to the set concurrently (e.g., Step 1 in Figure 1). The server serializes appends from clients and publishes a new, small-sized *digest* of the set after each append (e.g., Step 2). Clients periodically update their view of the set by requesting a new digest from the server (e.g., Step 6 and 7). We stress that the new digest could be for an arbitrary version $j > i$, where i is the previous version of the set (not just for $j = i+1$). Importantly, clients always ensure the set remains *append-only* by verifying an *append-only proof* between the old and new digest (e.g., Step 8). This way, clients can be certain the malicious server has not removed any elements from the set. Finally, clients securely check if an element k is in the set via a *(non)membership proof* (e.g., Steps 3-5 in Figure 1).

A malicious server can *fork* clients' views [3], preventing them from seeing each other's appends. To deal with this, clients maintain a *fork consistent* view [3], [30] of the set

by checking append-only proofs. This is crucial for security in transparency logs such as Revocation Transparency [62], CT [10] or CONIKS [13]. Specifically, if the server ever withholds an append from one client, that client's digest will forever diverge from other clients' digests. As a consequence, if that client *gossips* [25], [31] with other clients and learns his digest differs from their digest (at the same version i), the client detects the forking attack.

Other models. Our model is stronger when compared to the more common 2- and 3-party models [39], [43], [63] of authenticated data structures. Specifically, there is no *trusted source* that updates the set in our model. The clients and, possibly, the malicious server are the untrusted source of updates. In fact, having a trusted source trivially solves the AAS problem because the trusted source can simply verify that the set remained append-only and vouch for it with a digital signature. Unfortunately, this kind of solution is useless for applications such as Certificate Transparency (CT) [10] where there is no trusted source. Finally, our model arises in important applications such as public-key distribution [10], [12], [15]–[17], [64], [65], encrypted web applications [2], [4], [66], software transparency schemes [22], [23] and tamper-evident logging [9].

A. Append-only Authenticated Set API

Server-side API. The untrusted server implements:

$\mathsf{Setup}(1^\lambda, \beta) \to pp, VK$. Randomized algorithm that returns public parameters pp for the AAS scheme and a *verification key* VK used by clients. Here, λ is a security parameter and β is an upper-bound on the number of elements n in the set (i.e., $n \leq \beta$). If the scheme has a trapdoor, anyone in possession of the trapdoor can break the security of the scheme (defined in Section III-B). In that case, $\mathsf{Setup}(\cdot)$ has to be run by a trusted entity who promises to "forget" the trapdoor.

$\mathsf{Append}(pp, \mathcal{S}_i, d_i, k) \to \mathcal{S}_{i+1}, d_{i+1}$. Deterministic algorithm that appends a new element k to the version i set, creating a new version $i+1$ set. Succeeds only if the set is not full (i.e., $i+1 \leq \beta$). Returns the new authenticated set $\mathcal{S}_{i+1}$ and its digest d_{i+1}.

$\mathsf{ProveMemb}(pp, \mathcal{S}_i, k) \to b, \pi$. Deterministic algorithm that proves (non)membership for element k. When k is in the set, generates a membership proof π and sets $b = 1$. Otherwise, generates a non-membership proof π and sets $b = 0$.

$\mathsf{ProveAppendOnly}(pp, \mathcal{S}_i, \mathcal{S}_j) \to \pi_{i,j}$. Deterministic algorithm that proves $\mathcal{S}_i \subseteq \mathcal{S}_j$. In other words, generates an *append-only proof* $\pi_{i,j}$ that all elements in $\mathcal{S}_i$ are also present in $\mathcal{S}_j$. Importantly, a malicious server who removed elements from $\mathcal{S}_j$ that were present in $\mathcal{S}_i$ cannot construct a valid append-only proof (see Section III-B).

Client-side API. Clients implement:

$\mathsf{VerMemb}(VK, d_i, k, b, \pi) \to \{T, F\}$. Deterministic algorithm that verifies proofs returned by $\mathsf{ProveMemb}(\cdot)$ against the digest d_i at version i of the set. When $b = 1$, verifies k is in the set via the membership proof π. When $b = 0$, verifies k is *not* in the set via the non-membership proof π. (We formalize security in Section III-B.)

$\mathsf{VerAppendOnly}(VK, d_i, i, d_j, j, \pi_{i,j}) \to \{T, F\}$. Deterministic algorithm that ensures a set remains append-only. Verifies that $\pi_{i,j}$ correctly proves that the set with digest d_j is a superset of the set with digest d_i (see Section III-B). Also, verifies that d_i and d_j are digests of sets at version i and j respectively, enforcing fork consistency.

Using the API. To use an AAS scheme, first public parameters need to be computed using a call to $\mathsf{Setup}(\cdot)$. If the AAS scheme is trapdoored, a trusted party runs $\mathsf{Setup}(\cdot)$ and forgets the trapdoor. Once computed, the parameters can be reused by different servers for different append-only sets. $\mathsf{Setup}(\cdot)$ also returns a *public* verification key VK used by clients.

Clients first need to obtain the verification key VK returned by $\mathsf{Setup}(\cdot)$. Then, the server digitally signs and broadcasts the initial digest d_0 of the empty set $\mathcal{S}_0$ to its many clients. Clients can start appending elements using $\mathsf{Append}(\cdot)$ calls. If the server is honest, it serializes $\mathsf{Append}(\cdot)$ calls and returns the new digest d_i, digitally-signed, to its clients along with an append-only proof $\pi_{0,i}$ computed using $\mathsf{ProveAppendOnly}(\cdot)$. Some clients might be offline but eventually they'll receive either d_i or a newer $d_j, j > i$ later on. Importantly, whenever they transition from version i to j, all clients check an append-only proof $\pi_{i,j}$ using $\mathsf{VerAppendOnly}(VK, d_i, i, d_j, j, \pi_{i,j})$.

Clients can look up elements in the set. The server proves (non)membership of an element using $\mathsf{ProveMemb}(\cdot)$. A client verifies the proof using $\mathsf{VerMemb}(\cdot)$ against their digest. As more elements are added by clients, the server continues to publish a new digest d_j and proves it's a superset of the previous digest d_i using $\mathsf{ProveAppendOnly}(\cdot)$.

B. AAS Correctness and Security Definitions

We first introduce some helpful notation, before presenting our correctness definition. Consider an ordered sequence of n elements ($k_i \in \mathcal{K}$). Let $\mathcal{S}', d' \leftarrow \mathsf{Append}^+(pp, \mathcal{S}, d, (k_i)_{i\in[n]})$ denote a sequence of $\mathsf{Append}(\cdot)$ calls arbitrarily interleaved with other $\mathsf{ProveMemb}(\cdot)$ and $\mathsf{ProveAppendOnly}(\cdot)$ calls such that $\mathcal{S}', d' \leftarrow \mathsf{Append}(pp, \mathcal{S}_{n-1}, d_{n-1}, k_n)$, $\mathcal{S}_{n-1}, d_{n-1} \leftarrow \mathsf{Append}(pp, \mathcal{S}_{n-2}, d_{n-2}, k_{n-1}), \ldots, \mathcal{S}_1, d_1 \leftarrow \mathsf{Append}(pp, \mathcal{S}, d, k_1)$. Finally, let $\mathcal{S}_0$ denote an empty AAS with empty digest d_0.

Definition III.1 (Append-only Authenticated Set)**.** (Setup, Append, $\mathsf{ProveMemb}$, $\mathsf{ProveAppendOnly}$, $\mathsf{VerMemb}$, $\mathsf{VerAppendOnly}$) is a secure append-only authenticated set (AAS) if, $\forall$ security parameters λ, $\forall$ upper-bounds $\beta = \mathsf{poly}(\lambda)$ and $\forall n \leq \beta$ it satisfies the following properties:

Membership correctness. $\forall$ ordered sequences of elements $(k_i)_{i\in[n]}$, for all elements k, where $b = 1$ if $k \in (k_i)_{i\in[n]}$ and

$b = 0$ otherwise,

$$\Pr\left[\begin{array}{c}(pp, VK) \leftarrow \mathsf{Setup}(1^\lambda, \beta),\\ (\mathcal{S}, d) \leftarrow \mathsf{Append}^+(pp, \mathcal{S}_0, d_0, (k_i)_{i\in[n]}),\\ (b', \pi) \leftarrow \mathsf{ProveMemb}(pp, \mathcal{S}, k) :\\ b = b' \wedge \mathsf{VerMemb}(VK, d, k, b, \pi) = T\end{array}\right] = \eta(\lambda)$$

Observation: Note that this definition compares the returned bit b' with the "ground truth" in $(k_i)_{i\in[n]}$ and thus provides membership correctness. Also, it handles non-membership correctness since b' can be zero. Finally, the definition handles all possible orders of inserting elements.

Membership security. $\forall$ adversaries A running in time $\mathsf{poly}(\lambda)$,

$$\Pr\left[\begin{array}{c}(pp, VK) \leftarrow \mathsf{Setup}(1^\lambda, \beta),\\ (d, k, \pi, \pi') \leftarrow \mathsf{A}(pp) :\\ \mathsf{VerMemb}(VK, d, k, 0, \pi,) = T \wedge\\ \mathsf{VerMemb}(VK, d, k, 1, \pi',) = T\end{array}\right] = \varepsilon(\lambda)$$

Observation: This definition captures the lack of any "ground truth" about what was inserted in the set, since there is no trusted source in our model. Nonetheless, given a fixed digest d, our definition prevents *all* equivocation attacks about the membership of an element in the set.

Append-only correctness. $\forall m < n, \forall$ sequences $(k_i)_{i\in[n]}$ where $n \geq 2$,

$$\Pr\left[\begin{array}{c}(pp, VK) \leftarrow \mathsf{Setup}(1^\lambda, \beta)\\ (\mathcal{S}_m, d_m) \leftarrow \mathsf{Append}^+(pp, \mathcal{S}_0, d_0, (k_i)_{i\in[m]}),\\ (\mathcal{S}_n, d_n) \leftarrow \mathsf{Append}^+(pp, \mathcal{S}_m, d_m, (k_i)_{i\in[m+1,n]}),\\ \pi \leftarrow \mathsf{ProveAppendOnly}(pp, \mathcal{S}_m, \mathcal{S}_n) :\\ \mathsf{VerAppendOnly}(VK, d_m, m, d_n, n, \pi) = T\end{array}\right] = \eta(\lambda)$$

Append-only security. $\forall$ adversaries A running in time $\mathsf{poly}(\lambda)$,

$$\Pr\left[\begin{array}{c}(pp, VK) \leftarrow \mathsf{Setup}(1^\lambda, \beta)\\ (d_i, d_j, i < j, \pi_a, k, \pi, \pi') \leftarrow \mathsf{A}(pp) :\\ \mathsf{VerAppendOnly}(VK, d_i, i, d_j, j, \pi_a) = T \wedge\\ \mathsf{VerMemb}(VK, d_i, k, 1, \pi) = T \wedge\\ \mathsf{VerMemb}(VK, d_j, k, 0, \pi') = T\end{array}\right] = \varepsilon(\lambda)$$

Observation: This definition ensures that elements can only be added to an AAS.

Fork consistency. $\forall$ adversaries A running in time $\mathsf{poly}(\lambda)$,

$$\Pr\left[\begin{array}{c}(pp, VK) \leftarrow \mathsf{Setup}(1^\lambda, \beta)\\ (d_i \neq d_i', d_j, i < j, \pi_i, \pi_i') \leftarrow \mathsf{A}(pp) :\\ \mathsf{VerAppendOnly}(VK, d_i, i, d_j, j, \pi_i) = T \wedge\\ \mathsf{VerAppendOnly}(VK, d_i', i, d_j, j, \pi_i') = T\end{array}\right] = \varepsilon(\lambda)$$

Observation: This is our own version of fork consistency for append-only sets that captures what is known in the literature about fork consistency [3], [9]. Specifically, it allows a server to fork the set at version i by presenting two different digests d_i and d_i'. Importantly, it prevents the server from forging append-only proofs that "join" the two forks into some common digest d_j at a later version j.

IV. AAS from Bilinear Accumulators

Here, we present our AAS construction based on a bilinear accumulator and we give an algorithmic description in Section IV-A. As discussed in Section II, proving (non)membership with a bilinear accumulator requires a polynomial division which takes $O(n)$ time where n is the number of elements in the set. Thus, all n membership proofs can be precomputed (naively) in time $O(n^2)$. Unfortunately, this quadratic cost would be prohibitive for most use cases, even if only incurred once. Even worse, for non-membership, there is no tractable way to precompute proofs for accumulators over exponential-sized universes: the non-membership procedure from Section II-A would have to be called an exponential number of times. Therefore, we need new techniques to achieve our desired polylogarithmic time complexity for computing both types of proofs in our AAS.

A bilinear-tree accumulator. Our first technique is to deploy the bilinear accumulator in a tree structure, as follows. We start with the elements e_i as leaves of a binary tree (see Figure 2). Specifically, each leaf will store an accumulator over (the singleton set) e_i. Every internal node in the tree will then store an accumulator over the set defined as the union of the sets corresponding to its two children. For example, the parent node of the two leaves corresponding to e_i and e_{i+1} stores the accumulator of the set $\{e_i, e_{i+1}\}$. In this way, the root is the accumulator over the full set $S = \{e_i, \ldots, e_n\}$ (see Figure 2). We stress that all these accumulators use the same public parameters. The time to compute all the accumulators in the tree is $T(n) = 2T(n/2) + O(n \log n)$ where $O(n \log n)$ is the time to multiply the characteristic polynomials of two children sets of size n in the tree, thus $T(n) = O(n \log^2 n)$. We call the resulting structure a *bilinear tree* over set S.

Membership proofs in bilinear trees. A membership proof for element e_i will leverage the fact that sets along the path from e_i's leaf to the root of the bilinear tree are subsets of each other. The proof will consist of a sequence of *subset proofs* that validate this (computed as explained in Section II-A). Specifically, the proof contains the accumulators along the path from e_i's leaf to the root, as well as the accumulators of all sibling nodes along this path (see Figure 2). The client simply verifies all these subset proofs, starting from the singleton set e_i in the leaf. This convinces him that e_i is contained in the parent's accumulated set, which in turn is contained in its parent's accumulated set and so on, until the root.

Our bilinear tree approach gives us membership proofs of logarithmic size and thus logarithmic verification time. Importantly, computing a bilinear tree in $O(n \log^2 n)$ time implicitly computes all membership proofs "for free"! In contrast, building a standard billinear accumulator over S would yield constant-size proofs but in $O(n^2)$ time for all n proofs. Unfortunately, our bilinear tree structure does not (yet) support precomputing non-membership proofs. We devise new techniques that address this next.

Bilinear prefix trees to the rescue. To efficiently precompute non-membership proofs, we slightly modify our bilinear tree.

Instead of storing an element $e_i \in S$, the ith leaf will store the *set of prefixes* of the binary representation of e_i. (We assume this representation is λ bits or can be made λ bits using a collision-resistant hash function.) For example, a leaf that previously stored element e_1 with binary representation 0001, will now store the set $P(e_1) = \{\varepsilon, 0, 00, 000, 0001\}$ (i.e., all the prefixes of the binary representation of e_1, including the empty string ε). In general, for each element e_i, $P(e_i)$ is the set of all $\lambda + 1$ prefixes of e_i. Also, for any set $S = \{e_1, \ldots, e_n\}$, we define its *prefix set* as $P(S) = P(e_1) \cup \cdots \cup P(e_n)$. For example, let $S = \{a = 0001, b = 0101, c = 1110\}$. The root of S's bilinear tree will contain an accumulator over $P(S) = P(a) \cup P(b) \cup P(c) = \{\varepsilon, 0, 1, 00, 01, 11, 000, 010, 111, 0001, 0101, 1110\}$.

We refer to such a bilinear tree as a *bilinear prefix tree (BPT)* over S. The time to build a BPT for S is $O(\lambda n \log^2 n)$ since there are $O(\lambda n)$ prefixes across all leaves. Note that membership proofs in a BPT are the same as in bilinear trees, with a minor change. The internal nodes of the tree still store accumulators over the union of their children. However, the children now have common prefixes, which will only appear once in the parent. For example, two children sets have the empty string ε while their parent set only has ε once (because of the union). As a result, it is no longer the case that multiplying the characteristic polynomials of the children gives us the parent's polynomial. Therefore, we can no longer rely on the sibling as subset proofs: we have to explicitly compute subset proofs for each child w.r.t. its parent. We stress that this does not affect the asymptotic time complexity of computing the BPT. As before, the verifier starts the proof verification from the leaf, which now stores a prefix set $P(e_i)$ rather than a singleton set e_i.

Efficient non-membership proofs. But how does a BPT help with precomputing non-membership proofs for any element $e' \notin S$? First, note that, because of our use of prefixes, to prove $e' \notin S$ it suffices to show that *any one prefix ρ of e' is not contained in $P(S)$*. Second, note that there exist other keys e'' who share ρ as a prefix. As a result, the non-membership proof for e' can also be "reused" as a non-membership proof for e''. This is best illustrated in Figure 2 using our previous example where $S = \{a, b, c\}$. Consider elements $d = \underline{0111}$ and $f = \underline{0110}$ that are not in S. To prove non-membership for both elements, it suffices to prove the same statement: $011 \notin P(S)$. Thus, if we can identify all such *shared prefixes*, we can use them to prove the non-membership of (exponentially) many elements. This begs the question how many such shared prefixes are there? And can we efficiently precompute non-membership for all of them? We answer these questions next.

Suppose, we insert all elements from S in a trie as depicted in Figure 2 (by inserting a node in the trie for each prefix, as usual). Next, we keep track of the prefixes at the "frontier" of the trie depicted in red in Figure 2. We immediately notice that to prove non-membership of any element not in S, it suffices to prove non-membership of one of these *frontier prefixes*! In other words, elements that are not in S will have one of these

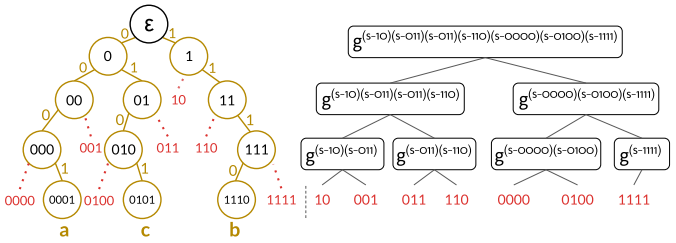


Fig. 2. On the left side, we depict a trie over set $S = \{a, b, c\}$. Each element is mapped to a unique path of length 4 in the trie. Nodes that are not in the trie but are at its *frontier* are depicted in **red**. On the right side, we depict a *bilinear frontier tree* (BFT) corresponding to the set S. To prove that an element is not in S, we prove one of its prefixes is in the BFT.

as a prefix. Thus, we formally define the *frontier set* of S as:

$$F(S) = \{\rho \in \{0,1\}^{\leq \lambda} : \rho \notin P(S) \wedge \mathsf{parent}(\rho) \in P(S)\},$$

where $\mathsf{parent}(\rho)$ is ρ without its last bit (e.g., $\mathsf{parent}(011) = 01$). Note that the size of $F(S)$ is $O(\lambda n)$, same as $P(S)$.

Most importantly, from the way $P(S)$ and $F(S)$ are defined, for any element e' it holds that $e' \notin S$ if, and only if, some prefix of e' is in $F(S)$. Therefore, proving non-membership of e' boils down to proving two statements: (i) some prefix of e' belongs to $F(S)$, and (ii) $P(S) \cap F(S) = \varnothing$. We stress that the latter is necessary as a malicious server may try to craft $F(S)$ in a false way (e.g., by adding some prefixes both in $P(S)$ and in $F(S)$). To prove (i), we build a bilinear tree over $F(S)$ which gives us precomputed membership proofs for all $s \in F(s)$. We refer to this tree as the *bilinear frontier tree (BFT)* for set S and to the proofs as *frontier proofs*. To prove (ii), we compute a *disjointness proof* between sets $P(S)$ and $F(S)$, as described in Section II-A (i.e., between the root accumulators of the BFT and the BPT of S). The time to build a BFT for S is $O(\lambda n \log^2 n)$ since $F(S)$ has $O(\lambda n)$ elements. The disjointness proof can be computed in $O(\lambda n \log^2 n)$ time.

Core (static) AAS construction. Combining all the above techniques, our core (static) AAS construction consists of: (a) a BPT for S, (b) a BFT for S, and (c) a proof of disjointness between $P(S)$ and $F(S)$ (i.e., between the root BPT and BFT accumulators). The height of the BPT is $O(\log n)$ and the height of the BFT is $O(\log(\lambda n))$ so the size and verification time of a (non)membership proof is $O(\log n)$. The digest that users store locally is just the root accumulator of the BPT.

Handling appends efficiently. So far, we only discussed the case of a static set S. However, our AAS should support appending new elements to S. The main challenge here is *efficiency* since updating the BPT and BFT as well as the disjointness proof after each update is very expensive (at least linear). To address this we use a classic "amortization" trick from Overmars [35].

Specifically, our AAS will consist not of one BPT for the entire set S, but will be partitioned into a *forest* of BPTs and their corresponding BFTs. Initially, we start with an empty set S. When the first element e_1 is appended, we build its *tree-pair*: a BPT, BFT and disjointness proof for the singleton set $\{e_1\}$. When the second element e_2 is appended, we "merge":

we build a tree-pair over $\{e_1, e_2\}$. The thumb rule is we always merge equal-sized subsets of S! When e_3 is appended, we cannot merge it because there's no other subset of size 1. Instead, we create a tree-pair over $\{e_1\}$. In general, after $2^i - 1$ appends, we will have built i separate tree-pairs corresponding to disjoint sets $S_1, \ldots, S_i$ where $S = \bigcup_{j=1}^{i} S_j$ and $|S_j| = 2^j$. The evolution of such a *forest* is depicted in Figure 3.

When merging two subsets S_1 and S_2 of S in the forest, we have to compute a new tree-pair over $S_1 \cup S_2$. To compute the new BPT, we need to (i) compute its root accumulator, (ii) set its children to the "old" roots of S_1 and S_2 and (iii) compute subset proofs $S_1 \subset S$ and $S_2 \subset S$. Since $|S_1| = |S_2| = n$, operations (i), (ii) and (iii) take time $O(\lambda n \log^2 n)$. Finally, we recompute the BFT for $S_1 \cup S_2$ from scratch which also takes time $O(\lambda n \log^2 n)$.

To analyze the append time, consider the time to $T(n)$ to create an AAS over a set S with $n = 2^i$ elements (without loss of generality). Then, $T(n)$ is just the time to create a tree-pair over S and can be broken into (i) the time to create a tree-pair over the children of S of size $n/2$ (i.e., $2T(n/2)$) (ii) the time to merge these two children BPTs (including computing subset proofs) and (iii) the time to compute the BFT of S. More formally, $T(n) = 2T(n/2) + O(\lambda n \log^2 n)$ which simplifies to $T(n) = O(\lambda n \log^3 n)$ time for n appends. Thus, the *amortized* time for 1 append is $O(\lambda \log^3 n)$ and can be de-amortized into *worst-case* time using generic techniques [35], [42].

The downside of our de-amortized approach is that proving non-membership becomes slightly more expensive. Now the server needs to prove non-membership in each tree-pair separately, requiring an $O(\log n)$ frontier proof in each of the $O(\log n)$ BFTs. This increases the non-membership proof size to $O(\log^2 n)$. On a good note, membership proofs remain unaffected: the server just sends a path to a leaf in *one* of the BPTs where the element is found. Finally, the AAS digest is just the root accumulators of all BPTs and has size $O(\log n)$. We analyze the AAS overheads from Table I in Appendix C.

Handling appends securely. Ensuring the set remains append-only is rather straight-forward, given the structure of bilinear prefix trees. In fact, our technique is similar to append-only proofs in history trees [9]. In particular, recall that when we merge the BPTs for S_1, S_2 and build a new BPT, (i) we compute its new root as the accumulator of $P(S_1) \cup P(S_2)$, (ii) we set the two old roots as the new root's children and (iii) we compute subset proofs between the old roots and the new root. Thus, the old roots become children nodes in the new BPT. In fact, because every append to the AAS triggers a sequence of merges, we can generalize the above statement: after a sequence of appends, *some* of the old roots in the AAS will become children of a new BPT. The other old roots, if any, will remain as (new) roots in the new forest (e.g., root 0 from F_4 to F_5 in Figure 3).

Our append-only proof leverages the above invariant. The proof asks that every old root should either (i) remain as a (new) root in the new AAS or (ii) have a path to a new root with valid subset proofs at every level. The path is verified by checking the subset proofs between every child and its parent, exactly as in a membership proof. At the same time, note that there might be new roots that are neither old roots nor do they have paths to old roots (e.g., root 111 in F_5 from Figure 3). The proof simply ignores such roots since they securely add new elements to the set. Overall, ensuring that the set remains append-only boils down to checking that every old root is either a descendant of a new root or has remained a (new) root in the new AAS.

Ensuring fork-consistency. Finally, we need to ensure that the construction is fork-consistent. Interestingly, append-only proofs do not imply fork-consistency. For example, consider a server who computes an AAS for set $\{e_1\}$ and another one for the set $\{e_2\}$. The server gives the first set's digest to user A and the second digest to user B. Afterwards, he appends e_2 to the first set and e_1 to the second one, which "joins" the two sets into a common set $\{e_1, e_2\}$. The append-only property was not violated (as the two users can deduce independently) but fork-consistency has: the two users had diverging views that were subsequently merged.

To avoid this, we will "Merkle-ize" each BPT using a collision-resistant hash function in the standard manner. Our AAS digest now consists of Merkle roots for all BPTs, which implicitly commit to all accumulators in the BPTs. In our previous example, after merging BPTs for elements e_1 and e_2, the root hash of the merged BPT will differ based on how leafs are ordered: (e_1, e_2), or (e_2, e_1). (In contrast, the root accumulator of the new BPT will be the same: $g^{(s-\mathcal{H}_{\mathbb{F}}(e_1))(s-\mathcal{H}_{\mathbb{F}}(e_2))}$, where $\mathcal{H}_{\mathbb{F}}$ maps elements to be accumulated to values in $\mathbb{F}_p$.) While verifying the append-only proof, user A with a digest for $\{e_1\}$ will explicitly check that the old root BPT is a left child of the new root BPT (using the Merkle hashes). Similarly, user B with a digest for $\{e_2\}$ will ensure the old root BPT is a left child of the new BPT as well. As a result of this, the two users' digests will diverge. Thus, violating fork-consistency becomes as hard as finding a collision in the hash function. To conclude, users enforce fork-consistency by checking that (i) old root BPTs are descendants of a new root BPT and (ii) old roots are *at the right position* in the new BPT (see Algorithm 4).

A. AAS Algorithms

In this subsection, we describe in detail the algorithms that implement our AAS (as originally defined in Section III-A). Recall from Section III that our AAS is just a forest of BPTs with corresponding BFTs. In particular, observe that each forest node has a BPT accumulator associated with it, while root nodes in the forest have BFTs associated with them. Our algorithms described below will operate on this forest, adding new leaves with new elements and their BPT accumulators, merging nodes in the forest and computing BFTs in the roots.

Trees notation. The $|$ symbol denotes string concatenation. A *tree* is a set of nodes denoted by binary strings in a canonical way. The root of a tree is denoted by the empty string ε and the left and right children of a node w are denoted by $w|0$ and $w|1$ respectively. If $b \in \{0, 1\}$, then the sibling of $w = v|b$ is

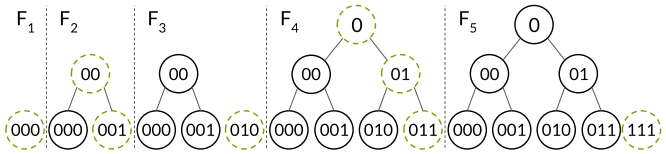


Fig. 3. A forest starting empty and going through a sequence of five appends. A forest only has trees of exact size 2^j for distinct j's. Note that there are *at most* $\log n$ trees in a forest of n leaves.

denoted by $\mathsf{sibling}(w) = v|\bar{b}$, where $\bar{b} = 1 - b$. A *path* from one node v to its ancestor node w is denoted by $\mathsf{path}[v, w] = \{u_1 = v, u_2 = \mathsf{parent}(u_1), \ldots, u_\ell = \mathsf{parent}(u_{\ell-1}) = w\}$. The parent node of $w = v|b$ is denoted by $\mathsf{parent}(w) = \mathsf{parent}(v|b) = v$. We also use $\mathsf{path}[v, w) = \mathsf{path}[v, w] - \{w\}$.

Forest notation. Let F_i denote a forest of up to β leaves that only has i leaves in it (e.g., see Figure 3). Intuitively, a forest is a set of nodes that might contain multiple trees in it. For our purposes, in F_i trees of the same size ≥ 1 are merged recursively, resulting in a number of trees logarithmic in i where each tree's size is 2^k for some unique k (e.g., see F_5 in Figure 3). Let $\mathsf{bin}^\beta(x)$ denote the $\lceil \log \beta \rceil$-bit binary expansion of a number x. Note that $\mathsf{bin}^1(x) = \varepsilon, \forall x$. Let $\mathsf{bin}^\beta(i)$ denote the ith inserted leaf, where i starts at 0 (e.g., see leafs 000 through 111 in F_5 in Figure 3). Let $\mathsf{roots}(F_i)$ denote all the roots of all the trees in the forest (e.g., $\mathsf{roots}(F_5) = \{0, 111\}$ in Figure 3). Let $\mathsf{leaves}(F_i)$ denote all the leaves in the forest (e.g., $\mathsf{leaves}(F_3) = \{000, 001, 010\}$ in Figure 3).

AAS notation. Note that **assert**$(\cdot)$ ensures a condition is true or fails the calling function otherwise. Let $\mathsf{Dom}(f)$ be the domain of a function f. We use $f(x) = \perp$ to indicate $x \notin \mathsf{Dom}(f)$. Let $\mathcal{S}_i$ denote our AAS with i elements. Each node w in the forest stores extractable accumulators $\mathbf{a}_w, \hat{\mathbf{a}}_w$ of its BPT together with a Merkle hash $\mathbf{h}_w$. Internal nodes (i.e., non-roots) store a subset proof π_w between $\mathbf{a}_w$ and $\mathbf{a}_{\mathsf{parent}(w)}$. The digest d_i of $\mathcal{S}_i$ maps each root r to $\mathbf{h}_r$ Every root r stores a disjointness proof $\boldsymbol{\psi}_r$ between its BPT and BFT. For simplicity, we give the server algorithms *global access* to many of the variables above and indicate this by **bolding** the global variable names in the algorithms.

Server algorithms. $\mathsf{Setup}(\cdot)$ generates large enough q-PKE public parameters $\mathcal{PP}_q(s, \tau)$ (see Section II), given an upper bound β on the number of elements. Importantly, forgets the trapdoors s and τ used to generate the public parameters. In other words, this is a *trusted setup* phase.

Algorithm 1 Computes public parameters (trusted setup)

1: **function** $\mathsf{Setup}(1^\lambda, \beta) \to (pp, VK)$
2: $\quad \ell \leftarrow 2^{\lfloor \log \beta \rfloor} - 1 \quad q \leftarrow \lambda\ell \quad (\mathbb{G}, \mathbb{G}_T, p, g, e(\cdot,\cdot)) \leftarrow \mathcal{G}(1^\lambda)$
3: $\quad s \xleftarrow{\$} \mathbb{F}_p \quad \tau \xleftarrow{\$} \mathbb{F}_p \quad VK = ((g^{s^i})_{i=0}^{\lambda+1}, g^\tau)$ ▷ Picks trapdoors
4: $\quad$ **return** $(((\mathbb{G}, \mathbb{G}_T, p, g, e(\cdot,\cdot)), \beta, q, \mathcal{PP}_q(s, \tau)), VK)$

$\mathsf{Append}(\cdot)$ creates a new leaf ℓ for the element k (Lines 2 to 3). Recursively merges equal-sized BPTs in the forest, as described in Section IV (Lines 5 to 9). In this process, computes subset proofs between old BPT roots and the new BPT. Merging ends when the newly created BPT w has no equal-sized BPT to be merged with.

Algorithm 2 Appends a new element to the AAS

1: **function** $\mathsf{Append}(pp, \mathcal{S}_i, d_i, k) \to (\mathcal{S}_{i+1}, d_{i+1})$
2: $\quad w \leftarrow \mathsf{bin}^\beta(i) \quad \mathbf{S}_w \leftarrow \{k\}$ ▷ Create new leaf w for element k
3: $\quad (\alpha_w, \mathbf{a}_w, \cdot) \leftarrow \mathsf{Accum}(P(\mathbf{S}_w)) \quad \mathbf{h}_w \leftarrow \mathcal{H}(w|\perp|\mathbf{a}_w|\perp)$
4: $\quad$ ▷ "Merge" old BPT roots with new BPT root (recursively)
5: $\quad$ **while** $\mathsf{sibling}(w) \in \mathsf{roots}(F_i)$ **do**
6: $\quad\quad \ell \leftarrow \mathsf{sibling}(w) \quad p \leftarrow \mathsf{parent}(w) \quad \mathbf{S}_p \leftarrow \mathbf{S}_\ell \cup \mathbf{S}_w$
7: $\quad\quad (\alpha_p, \mathbf{a}_p, \hat{\mathbf{a}}_p) \leftarrow \mathsf{Accum}(P(\mathbf{S}_p)) \quad \mathbf{h}_p = \mathcal{H}(p|\mathbf{h}_\ell|\mathbf{a}_p|\mathbf{h}_w)$
8: $\quad\quad (\pi_\ell, \cdot) \leftarrow \mathsf{Accum}(P(\mathbf{S}_p \setminus \mathbf{S}_\ell))$
9: $\quad\quad (\pi_w, \cdot) \leftarrow \mathsf{Accum}(P(\mathbf{S}_p \setminus \mathbf{S}_w)) \quad w \leftarrow p$
10: $\quad$ ▷ Invariant: w is a new root in F_{i+1}. Next, sets w's accumulator.
11: $\quad (\phi_w, \boldsymbol{\sigma}_w) \leftarrow \mathsf{CreateFrontier}(F(\mathbf{S}_w))$
12: $\quad (y, z) \leftarrow \mathsf{ExtEuclideanAlg}(\alpha_w, \phi_w) \quad \boldsymbol{\psi}_w \leftarrow (g^{y(s)}, g^{z(s)})$
13: $\quad$ Store AAS state into $\mathcal{S}_{i+1}$ ▷ i.e., persist **bolded** variables in $\mathcal{S}_{i+1}$
14: $\quad d_{i+1}(r) \leftarrow \mathbf{h}_r, \forall r \in \mathsf{roots}(F_{i+1})$ ▷ Set new digest
15: $\quad$ **return** $\mathcal{S}_{i+1}, d_{i+1}$
16: **function** $\mathsf{Accum}(T)$
17: $\quad \alpha(x) \leftarrow \prod_{w \in T} (x - \mathcal{H}_\mathbb{F}(w))$
18: $\quad$ **return** $(\alpha, g^{\alpha(s)}, g^{\tau\alpha(s)})$

If k is in the set, $\mathsf{ProveMemb}(\cdot)$ sends a Merkle path to k's leaf in some tree with root r (Lines 3 to 5) via $\mathsf{ProvePath}(\cdot)$ (see Algorithm 3) . This path contains subset proofs between every node's accumulator and its parent node's accumulator. If k is not in the set, then sends frontier proofs in each BFT in the forest (Lines 6 to 8) via $\mathsf{ProveFrontier}(\cdot)$ (see Algorithm 6).

Algorithm 3 Constructs a (non)membership proof

1: **function** $\mathsf{ProveMemb}(pp, \mathcal{S}_i, k) \to (b, \pi)$
2: $\quad$ Let $\ell \in \mathsf{leaves}(F_i)$ be the leaf where k is stored or $\perp$ if $k \notin \mathcal{S}_i$
3: $\quad$ **if** $k \in \mathcal{S}_i$ **then** ▷ Construct Merkle path to element
4: $\quad\quad$ Let $r \in \mathsf{roots}(F_i)$ be the root of the tree where k is stored
5: $\quad\quad \pi \leftarrow \mathsf{ProvePath}(\mathcal{S}_i, \ell, r, \perp) \quad b \leftarrow 1$
6: $\quad$ **else** ▷ Prove non-membership in all BFTs
7: $\quad\quad \chi_r \leftarrow \mathsf{ProveFrontier}(\mathcal{S}_i, r, k), \forall r \in \mathsf{roots}(F_i) \quad b \leftarrow 0$
8: $\quad\quad \forall r \in \mathsf{roots}(F_i), b \in \{0, 1\}, \pi(r) \leftarrow (\mathbf{a}_r, \cdot) \quad \pi(r|b) \leftarrow \mathbf{h}_{r|b}$
9: $\quad$ **return** $b, (\ell, \pi, (\chi_r)_{r \in \mathsf{roots}(F_i)}, (\boldsymbol{\psi}_r)_{r \in \mathsf{roots}(F_i)})$
10: **function** $\mathsf{ProvePath}(\mathcal{S}_i, u, r, \pi) \to \pi$ ▷ Precondition: r is a root in F_i
11: $\quad$ ▷ Note: Will re-set $\pi(w)$ set by previous ProvePath call (if any)
12: $\quad \pi(r) \leftarrow (\perp, \mathbf{a}_r, \hat{\mathbf{a}}_r, \perp)$
13: $\quad \pi(w) \leftarrow (\perp, \mathbf{a}_w, \hat{\mathbf{a}}_w, \pi_w), \forall w \in \mathsf{path}[u, r)$
14: $\quad$ ▷ Note: Will not set $\pi(\mathsf{sibling}(w))$ if already set from previous call!
15: $\quad$ **for** $w \in \mathsf{path}[u, r)$ s.t. $\mathsf{sibling}(w) \notin \mathsf{Dom}(\pi)$ **do**
16: $\quad\quad \pi(\mathsf{sibling}(w)) \leftarrow (\mathbf{h}_{\mathsf{sibling}(w)}, \perp, \perp)$
17: $\quad$ **return** π

For each root r in F_i, $\mathsf{ProveAppendOnly}(\cdot)$ sends a Merkle path to an ancestor root in F_j, if any. To verify the proof, $\mathsf{VerAppendOnly}(\cdot)$ checks that each root r from F_i is a subset of some root in F_j by checking subset proofs (Line 12) via $\mathsf{VerPath}(\cdot)$ (see Algorithm 5). $\mathsf{VerAppendOnly}(\cdot)$ enforces fork-consistency implicitly when verifying Merkle hashes.

Algorithm 4 Creates and verifies append-only proofs

1: **function** ProveAppendOnly($pp, \mathcal{S}_i, \mathcal{S}_j) \to \pi$
2: **if** roots(F_i) $\subset$ roots(F_j) **then return** $\perp$
3: Let $r' \in$ roots(F_j) be the ancestor root of old roots in F_i not in F_j
4: $\pi \leftarrow$ ProvePath($\mathcal{S}_j, r, r', \pi), \forall r \in$ roots(F_i)
5: **return** π
6: **function** VerAppendOnly($VK, d_i, i, d_j, j, \pi_{i,j}) \to \{T, F\}$
7: **assert** $d_i(r) \neq \perp \Leftrightarrow r \in$ roots(F_i) ▷ Is valid version i digest?
8: **assert** $d_j(r) \neq \perp \Leftrightarrow r \in$ roots(F_j) ▷ Is valid version j digest?
9: Let $R =$ roots(F_i) $-$ roots(F_j) be old roots with paths to new root
10: $\forall r \in R$, fetch h_r from $d_i(r)$ and update $\pi_{i,j}(r)$ with it
11: **assert** $\pi_{i,j}$ is well-formed Merkle proof for all roots in R
12: **assert** VerPath($d_j, r, \pi_{i,j}), \forall r \in R$,

Client algorithms. If k is stored at leaf ℓ in the AAS, VerMemb($\cdot$) reconstructs ℓ's accumulator from k. Then, checks if there's a valid Merkle path from ℓ to some root, verifying subset proofs along the path (Lines 3 to 6) via VerPath($\cdot$) (see Algorithm 5). If k is not in the AAS, verifies frontier proofs for k in each BFT in the forest (Lines 7 to 12) via VerFrontier($\cdot$) (see Algorithm 6).

Algorithm 5 Verifies a (non)membership proof

1: **function** VerMemb($VK, d_i, k, b, \pi_k) \to \{T, F\}$
2: Parse π_k as $(\ell, \pi, (\chi_r)_{r \in \text{roots}(F_i)}, (\boldsymbol{\psi}_r)_{r \in \text{roots}(F_i)})$
3: **if** $b = 1$ **then** ▷ Membership proof
4: $(\cdot, a_\ell, \hat{a}_\ell) \leftarrow$ Accum($P(\{k\})$) $h_\ell \leftarrow H(\ell|\perp|a_\ell|\perp)$
5: Update $\pi(\ell)$ with h_ℓ and accumulators a_ℓ and $\hat{a}_\ell$
6: **assert** π is Merkle path to leaf $\ell \wedge$ VerPath(d_i, ℓ, π)
7: **else** ▷ Non-membership proof
8: **for all** $r \in$ roots(F_i) **do** ▷ Check BFTs
9: $(a_r, \cdot) \leftarrow \pi(r)$ $h_{r|b} \leftarrow \pi(r|b), \forall b \in \{0, 1\}$
10: $(o_r, \cdot) \leftarrow \chi_r(\varepsilon)$ $(y, z) \leftarrow \psi_r$
11: **assert** $d_i(r) = \mathcal{H}(r|h_{r|0}|a_r|h_{r|1})$
12: **assert** $e(a_r, y)e(o_r, z) = e(g, g) \wedge$ VerFrontier(k, χ_r)
13: **function** VerPath($d_k, w, \pi) \to \{T, F\}$
14: Let $r \in$ roots(F_k) denote the ancestor root of w
15: ▷ Walk path invariant: u is *not* root node (but parent(u) might be)
16: **for** $u \leftarrow w; d_k(u) = \perp; u \leftarrow$ parent(u) **do**
17: $p \leftarrow$ parent(u) ▷ Check subset proof and extractability (below)
18: $(\cdot, a_u, \hat{a}_u, \pi_u) \leftarrow \pi(u)$ $(\cdot, a_p, \hat{a}_p, \cdot) \leftarrow \pi(p)$
19: **assert** $e(a_u, \pi_u) = e(a_p, g) \wedge e(a_u, g^\tau) = e(\hat{a}_u, g)$
20: **assert** $d_k(u) =$ MerkleHash(π, u) ▷ Invariant: u is a root node
21: **assert** $e(a_u, g^\tau) = e(\hat{a}_u, g)$ ▷ Is root accumulator extractable?
22: **function** MerkleHash($\pi, w) \to h_w$
23: $(h_w, a_w, \cdot, \cdot) \leftarrow \pi(w)$
24: **if** $h_w = \perp$ **then**
25: **return** $\mathcal{H}(w|$MerkleHash$(\pi, w|0)|a_w|$MerkleHash$(\pi, w|1))$
26: **else**
27: **return** h_w

Frontier algorithms. Given a frontier set F, CreateFrontier($\cdot$) creates its BFT level by level starting from the leaves. Given a key $k \notin \mathcal{S}_i$ and a root r, ProveFrontier($\cdot$) returns a frontier proof for k in the BFT at root r.

Algorithm 6 Manages BFT of a set

1: **function** CreateFrontier($F) \to (\phi, \sigma)$
2: ▷ Create leaves with $g^{s-\mathcal{H}_\mathbb{F}(\rho)}$ for each prefix ρ
3: $i \leftarrow 0$ $S_w \leftarrow \varnothing, \forall w$
4: **for** $\rho \in F$ **do**
5: $w \leftarrow \text{bin}^{|F|}(i)$ $S_w \leftarrow \rho$ $i \leftarrow i + 1$
6: $(\phi_w, o, \hat{o}) \leftarrow$ Accum(S_w) $\sigma(w) \leftarrow (o, \hat{o})$
7: ▷ "Merge" children accumulators into a parent accumulator
8: **for** $i \leftarrow \lceil \log |F| \rceil; i \neq 0; i \leftarrow i - 1$ **do**
9: $j \leftarrow 0$ levelSize $\leftarrow 2^i$ $u \leftarrow \text{bin}^{\text{levelSize}}(0)$
10: **while** $S_u \neq \varnothing$ **do**
11: $p \leftarrow$ parent(u) $S_p \leftarrow S_u \cup S_{\text{sibling}(u)}$ $j \leftarrow j + 2$
12: $(\phi_p, o, \hat{o}) \leftarrow$ Accum(S_p) $\sigma(p) \leftarrow (o, \hat{o})$ $u \leftarrow \text{bin}^{2^i}(j)$
13: **return** $(\phi_\varepsilon, \sigma)$
14: **function** ProveFrontier($\mathcal{S}_i, r, k) \to \chi$
15: Let ρ be the smallest prefix of k that is not in $P(\mathbf{S}_r)$
16: Let ℓ denote the leaf where $\boldsymbol{\sigma}_r(\ell) = g^{(s-\mathcal{H}_\mathbb{F}(\rho))}$
17: $\chi(\varepsilon) \leftarrow \boldsymbol{\sigma}_r(\varepsilon)$ ▷ Copy root BFT accumulator
18: **for** $w \in$ path$[\ell, \varepsilon)$ **do**
19: $\chi(w) \leftarrow \boldsymbol{\sigma}_r(w)$
20: **if** $\boldsymbol{\sigma}_r($sibling$(w)) \neq \perp$ **then**
21: $\chi($sibling$(w)) \leftarrow \boldsymbol{\sigma}_r($sibling$(w))$
22: **else**
23: $\chi($sibling$(w)) \leftarrow (g, g^\tau)$
24: **return** χ
25: **function** VerFrontier($k, \chi) \to \{T, F\}$
26: Let ℓ denote the leaf in χ with a prefix ρ for k
27: **assert** $\rho \in P(\{k\}) \wedge g^{(s-\mathcal{H}_\mathbb{F}(\rho))} = \chi(\ell)$
28: **assert** $e(o, g^\tau) = e(\hat{o}_w, g)$ where $(o, \hat{o}) \leftarrow \chi(\varepsilon)$
29: **for** $w \in$ path$[\ell, \varepsilon)$ **do**
30: $(c_w, \hat{c}_w) \leftarrow \chi(w)$ $(s_w, \cdot) \leftarrow \chi($sibling$(w))$
31: $(p_w, \cdot) \leftarrow \chi($parent$(w))$
32: **assert** $e(c_w, s_w) = e(p_w, g) \wedge e(c_w, g^\tau) = e(\hat{c}_w, g)$

VerFrontier($\cdot$) verifies a frontier proof for one of k's prefixes against a specific root BFT accumulator.

Theorem IV.1. Under the q-SBDH and q-PKE assumptions, and assuming that $\mathcal{H}$ is a secure collision-resistant hash function, our construction is a secure AAS as per Definition III.1.

We prove Theorem IV.1 in Appendix B and analyze AAS overheads from Table I asymptotically in Appendix C.

V. From Append-Only Authenticated Sets to Append-Only Authenticated Dictionaries

In this section, we turn our attention to constructing an append-only authenticated dictionary (AAD). An AAD maps a *key* k to a multiset of *values* V, but does so in an append-only fashion. Specifically, once a value has been added to a key, it cannot be removed nor changed. Compared to an AAS, an AAD provides additional functionality: instead of simple (non)membership queries for an element, AADs support more elaborate queries of the form *"return all the values associated with key k."* This type of query occurs in many real-world applications such as public-key directories where k is the identity of a user and V are all the public keys of that user.

Our construction has great similarities with the AAS of Section IV. However, the different functionality calls for modifications. Indeed, even the security notion for AADs is different. In an AAS, no malicious server can simultaneously produce accepting proofs of membership and non-membership for the same element e with respect to the same digest. In

contrast, in an AAD, no malicious server can simultaneously produce accepting proofs for two different sets of values V, V' for a key k with respect to the same digest. This captures the related notion for an AAS since one of the sets of values may be empty (indicating k has never been registered in the dictionary) and the other non-empty. We give full security definitions for AADs in Appendix D. Next we describe how we modify our construction from Section IV to get an AAD.

Encoding key-value pairs. An AAS construction can trivially support key-value pairs (k, v) by increasing the size of the domain of the underlying AAS from λ bits to 2λ bits so as to account for the value v. That is, (k, v) would be inserted in the AAS as $k|v$, using the same algorithms from Section IV-A.

Proving complete membership. Non-membership of a key k remains the same as in the AAS: for each BFT in the AAD, the server gives a frontier proof for a prefix of k (see Section IV). In contrast, membership proofs change: in addition to giving a membership proof for each value in some BPT, the server must prove *completeness* by convincing the client it is not leaving out any values of key k. For this, the server uses the same frontier technique: it proves specific prefixes for the *missing values* of k are not in the BPTs (and thus are not maliciously being left out). This is best illustrated with an example.

Assume we have an AAD of size 2^i with just *one* tree-pair. Suppose the server wants to prove $k = 0000$ has *complete* set of values $V = \{v_1 = 0001, v_2 = 0011\}$. Consider a trie over $k|v_1$ and $k|v_2$ and note that $F^{[k]} = \{(0000|1), (0000|01), (0000|0000), (0000|0010)\}$ is the set of all frontier prefixes for the missing values of k. We call this set the *lower frontier* of k relative to V. The key idea to prove completeness is to prove all lower frontier prefixes are in the BFT via frontier proofs (as discussed in Section IV). Note that $|F^k| = O(\lambda)$ and each frontier proof is $O(\log n)$-sized, resulting in an $O(\lambda \log n)$-sized proof.

This idea can be generalized to AADs of arbitrary size which have (i) BPTs with no values for k, where the server proves non-membership of k and (ii) BPTs with values for k, where the server uses the above technique to prove BPT i has complete set of values V_i. In that case, a complete membership proof for a key k with a single value consists of (1) an $O(\log n)$-sized membership proof in some BPT, (2) an $O(\lambda \log n)$-sized completeness proof in its corresponding BFT (as discussed above) and (3) an $O(\log n)$-sized non-membership proof for k in all other $O(\log n)$ BFTs.

Smaller completeness proofs. When k has one value, it follows from above that a complete membership proof for k is $O(\lambda \log n)$-sized. However, we can easily decrease its size to $O(\log^2 n)$. Note that the main overhead comes from having to prove that all $O(\lambda)$ lower frontier prefixes of k are in a BFT. The key idea is to group all of k's lower frontier prefixes into a single BFT leaf, creating an accumulator over all of them. As a result, instead of having to send $O(\lambda)$ frontier proofs (one for each lower frontier prefix), we send a single $O(\log n)$-sized frontier proof for a single leaf (which contains all lower frontier prefixes). We can generalize this idea: when k has $|V_i|$ values in the ith BFT in the forest, k's lower frontier relative to V_i has $O(|V_i|\lambda)$ prefixes. Then, for each BFT i, we split the lower frontier prefixes of k associated with V_i into separate BFT leaves each of size at most $2\lambda+1$. Importantly, we stress that clients have enough public parameters to reconstruct the accumulators in these BFT leaves (i.e., they have $(g^{s^i})_{i=0}^{2\lambda+1}$).

Supporting large domains and multisets. To handle keys and values longer than λ bits, we store $\mathcal{H}(k)|\mathcal{H}(v)$ in the AAS (rather than $k|v$), where $\mathcal{H}$ is a collision-resistant hash and we can retrieve the actual value v from another repository. To support multisets (same v can be inserted twice for a k), the server can insert $\mathcal{H}(\mathcal{H}(v)|i)$ for the i-th occurrence of (k, v).

VI. Evaluation

Our evaluation answers the following questions: How expensive is it to append to an AAD? Does batching appends help? How large and expensive-to-verify are complete membership proofs? What about append-only proofs?

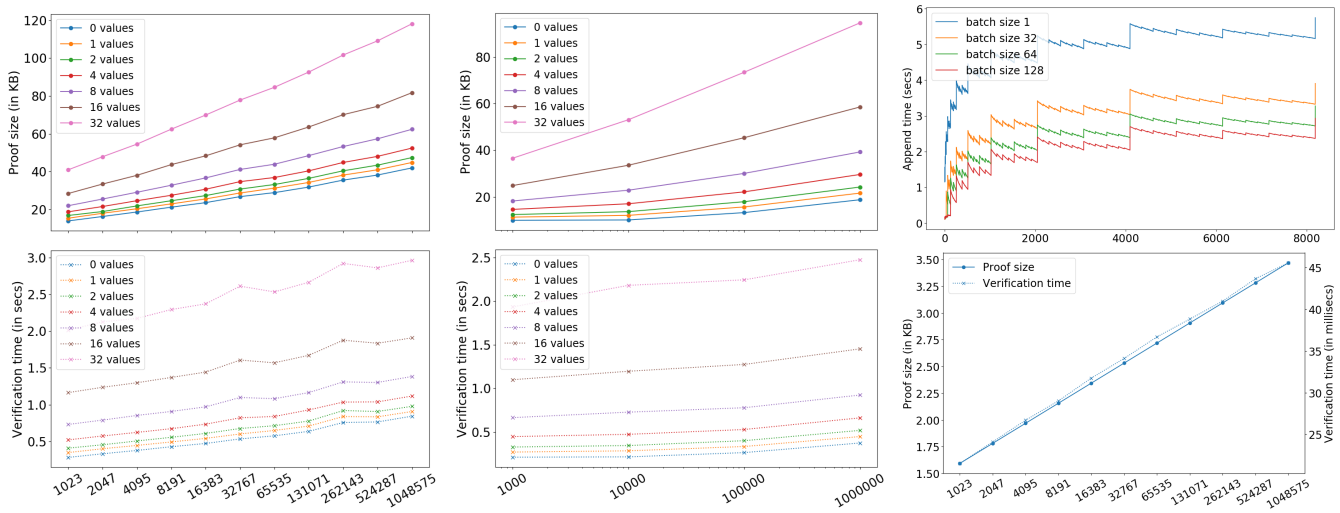
A. Codebase and experimental setup

We implemented our append-only authenticated dictionary (AAD) construction in 5700 lines of C++. Specifically, we implemented the amortized version of our construction whose *worst-case* append time is $O(\lambda n \log^3(n))$ while its amortized append time is $O(\lambda \log^3(n))$. We used Zcash's `libff` [67] as our elliptic curve library with support for pairings, which internally uses the `ate-pairing` library [68]. We use a 254-bit Barretto-Naehrig curve with a Type III pairing [69]. We used `libfqfft` [70] to multiply polynomials. We used Victor Shoup's `libntl` [71] to interpolate polynomials, divide polynomials and compute Bezout coefficients.

We ran our evaluation in the cloud on Amazon Web Services (AWS). All experiments were run on an a r4.16xlarge instance type with 488 GB of RAM and 64 VCPUs, running Ubuntu 16.04.4 (64-bit version). This instance type is "memory-optimized" which, according to AWS, means it is "designed to deliver fast performance for workloads that process large data sets in memory." Because we implement BPTs and BFTs using pointers, we believe we require more memory than other implementations (e.g. hash table-based implementations).

B. Append times

We start with an empty AAD, we append key-value pairs to it and keep track of the *cumulative average append-time* after every append. Recall that appends are amortized in our construction (but can be de-amortized using known techniques [35], [42]). As a result, in our benchmark some appends are very fast (e.g., 25 milliseconds) while others are painfully slow (e.g., 1.5 hours). To keep the running time of our benchmark reasonable, we only benchmarked $2^{13} = 8192$ appends. We also investigate the effect of batching on append times. When we batch $k = 2^i$ appends together, we only compute one BFT for the full tree of size k created after inserting the batch. In contrast, without batching, we compute k BFTs for k appends (one for each new forest root created



(a) AAD complete membership (worst-case sizes) (b) AAD complete membership (average-case sizes) (c) AAD append times and append-only proofs

Fig. 4. These figures illustrate complete membership proof size and verification time for (a) worst-case dictionary sizes of $2^i - 1$, (b) for average-case dictionary sizes of 10^i, (c, down) append-only proof sizes and verification time between dictionaries of size $2^i - 1$ and $2^{i+1} - 1$, and (c, up) average append time. We measured append times by inserting $2^{13} = 8192$ key-value pairs. Each line corresponds to a different experiment with a different batch size. A larger batch size means BFTs are computed more rarely, which speeds up appends. Spikes in the graph occur every time two large trees in the forest are merged, resulting in the (expensive) computation of a new BFT. The last spike corresponds to the last merge of the largest two trees, both of size 4096.

after an append). Figure 4c shows that the average append time is 6 seconds, while batching can reduce it to 3 seconds.

The bottleneck for appends is computing the BFTs. To speed this up, our implementation uses `libff`'s multi-threaded multi-exponentiation to compute accumulators. However, there are other opportunities for parallelization that our implementation does not explore. The first, is to parallelize computing the polynomials on the same level in a BFT. The second, is to parallelize computing the smaller accumulators at lower levels of the BFT, where multi-threaded multi-exponentiation does not help as much. The third, is to parallelize computing subset proofs between a BPT and its children in the forest. Finally, we can leverage techniques for distributed FFT to speed up polynomial operations [72]. We believe these optimizations can reduce the average append time to less than a second.

C. Complete membership proofs

We investigate three factors that affect membership proof size (and thus verification time): (1) the dictionary size, (2) the number of trees in the forest and (3) the number of values of a key. Our results are summarized in Figures 4a and 4b.

1) Benchmark implementation: We take as input a dictionary size n and create an AAD of that size. To speed up the benchmark, instead of computing accumulators, we simply pick them uniformly at random. (Note that this does not affect the proof verification time.) We measure *average* proof sizes for keys with ℓ values in an AAD of size n, where $\ell \in \{0, 1, 2, 4, 8, 16, 32\}$. To get an average, we make sure we insert 10 different keys with ℓ values, for each number of values ℓ. We call these keys *target* keys. The rest of the inserted keys are random (and simply ignored by the benchmark). Importantly, we randomly disperse the target key-value pairs throughout the forest. We do this to avoid having all the values of a key end up in consecutive forest leaves, which would artificially decrease the proof size.

Once the dictionary reaches size n, we go through every target key with ℓ values, compute its complete membership proof, and measure the size and verification time. (Note that, when $\ell = 0$, we are measuring non-membership proofs.) Since for each ℓ the AAD contains 10 different target keys with ℓ values, we obtain an average proof size and verification time. We repeat the experiment for increasing dictionary sizes n and summarize the numbers in Figures 4a and 4b. We stress that proof verification is single-threaded in these benchmarks.

2) Worst-case versus best-case dictionary sizes: Recall that some dictionary sizes will be "better" than others because they have fewer trees in the forest. Membership proofs will be smaller and will verify faster in these dictionaries. For example, a dictionary of (worst-case) size $2^i - 1$ will have i trees in the forest and thus i BFTs. Thus, when proving membership of a key, the complete membership proof will include i frontier proofs. In contrast, a dictionary of size 2^i only has a single tree in the forest, so a membership proof needs only one frontier proof. Our evaluation shows that AADs of size 10^i (see Figure 4b.) have slightly smaller proof sizes than AADs of size $2^i - 1$ (see Figure 4a). For example, for size 1,000,000, the proof for a key with 32 values averages 95KB, while for size $2^{20} - 1$ the average is 118KB.

3) Memory consumption: Our complete membership proof benchmark was the most memory-hungry: it consumed 263 GB of RAM for an AAD of size $2^{20} - 1$. Fortunately, this benchmark did not require q-PKE public parameters, which would have added another ~68 GBs of RAM. While our prototype's memory consumption could be improved, we

stress that the main overhead comes from the size of the BFTs and the accumulators stored at each node. For example, in an AAD of $n = 2^{20}-1$ key-value pairs, the BFT will have around $400n$ nodes. Since we are using Type III pairings, each node stores three accumulators (two in $\mathbb{G}_1$ and one in $\mathbb{G}_2$) and the memory consumption reaches $400n(32 \cdot 2 + 64)$ bytes, which is around 53 GBs. The rest of the overhead comes from our own implementation's use of pointers to implement trees and other bookkeeping.

D. Append-only proofs

This benchmark appends random key-value pairs until it reaches a target size $n = 2^{i+1} - 1$. Then, it measures the size and verification time of the append-only proof between the AAD of size n and an earlier one of size $m = 2^i - 1$. Then, it repeats for the next target size $n' = 2^{i+2} - 1$. We plot the results in Figure 4c. We benchmarked on $n = 2^i - 1$ AAD sizes because the proof size is logarithmic in n (i.e. $\Theta(i)$) so it illustrates worst-case append-only proof sizes. Append-only proof verification is single-threaded. To speed up the benchmark, we randomly pick accumulators in the forest. Unlike the membership proof benchmark, this benchmark does not need to compute BFTs and consumes only 12.5 GBs of memory.

Our results show append-only proofs are reasonably small and fast to verify. For example, the biggest proof between AAD sizes $2^{19} - 1$ and $2^{20} - 1$ is 3.5 KB and takes a little over 45 milliseconds to verify.

E. Comparison to Merkle tree approaches

How do AADs compare to Merkle prefix trees or History Trees (HTs), which are used in CONIKS and Certificate Transparency (CT) respectively? First of all, appends in AADs are orders of magnitude slower because of the overheads of cryptographic accumulators. However, we believe append times can be made practical using parallelization (see Section VI-B) and de-amortization techniques [35], [42].

1) Prefix trees: Complete membership proofs in prefix trees are much smaller than in AADs. In a prefix tree of size 2^{20}, a proof consisting of a Merkle path would be around 640 bytes. In comparison, our proofs for a key with 32 values are 152 times to 189 times more expensive (depending on the number of forests in the tree). On the other hand, append-only proofs in AADs are much smaller than in prefix trees. (Asymptotically, the difference is logarithmic versus linear.) To illustrate this, we implemented append-only proofs in prefix trees in Golang [73] and benchmarked them. Our results show that the append-only proof between an old prefix tree of size 2^{19} and a new one of size 2^{20} is 32 MB (as opposed to 3.5 KB in AADs). The proof gets a bit smaller when the size gap between the dictionaries is larger but not by much. For example, the proof between 10^5 and 10^6 is 14.6 MB.

2) History trees (HTs): Complete membership proofs in history trees are $O(n)$-sized for an HT over n key-value pairs. This is because, to guarantee completeness, the proof must consist of all key-value pairs in the HT. Thus, our complete membership proofs are orders of magnitude smaller. On the other hand, append-only proofs in AADs are slightly larger than in HTs. AAD append-only proofs have approximately the same number of nodes as HT append-only proofs, but our nodes store two additional accumulators: two BPT accumulators in $\mathbb{G}_1$ and a subset proof in $\mathbb{G}_2$. As a result, the per-node proof size increases from 32 bytes to 32 + 64 + 64 = 160 bytes, making our proofs 5 times larger.

3) Are AADs ever worth it?: Asymptotically, AADs outperform previous work because, unlike previous constructions based on prefix or history trees, **both** append-only proofs and complete membership proofs are polylogarithmic in the dictionary size. But in practice, our evaluation shows AAD proof sizes are still larger than ideal, especially complete membership proofs. This begs the question: *Is it ever worth using AADs?* We believe the answer is "yes": even with large complete membership proofs, the much-improved append-only proof substantially decreases the practical bandwidth of previous approaches such as CT logs and CONIKS.

Consider a messaging service for 1 billion users that uses a CONIKS transparency log to deter impersonation attacks. Suppose, a very small percentage of 0.001% of users reset their PK in a day (i.e., 10,000 users). That means the directory will have to create (more-or-less) 10,000 new Merkle roots in one day. For a user to check his PK in the last version of the directory, the CONIKS server will have to push a $30 \cdot 32 = 960$-byte membership proof *per Merkle root* to each one of its 1 billion users. To support this, the CONIKS server needs 111.11 GBps of bandwidth.

Consider the same scenario as above but replace the CONIKS log with an AAD so users no longer have to monitor in every version of the directory. Instead users can safely fetch the latest version of the directory via an append-only proof and only check their PK in this latest version. This dramatically reduces bandwidth for the server by decoupling the update frequency of the directory from the monitoring frequency of the users. Suppose that users checks their PK once per day and, on average, a membership proof is 40 KB and an append-only proof is 7 KB. Note that the membership proof of 40 KB is even smaller than the one previously reported because the log only needs to convince users no new PKs have been added for them. Therefore, the proof consists of only frontier proofs. Thus, the server needs to only push 47 KBs to each user every day, resulting in only 544 MBps of bandwidth.

What about CT? The current ecosystem tracks over 2.1 billion certificates [19] across many different CT logs. Because the same certificate might be logged multiple times in different logs, CT monitors must check every log to detect impersonation. Recall that a CT monitor will download each new certificate appended to a log, which we argue puts too much bandwidth pressure on logs. This, in turn, limits the number of CT monitors. For example, let us assume 30 million websites want to monitor their own certificates in the system. At the current rate of 12.37 certificates per second being added to CT [19], with a mean size of 1.4 KB [33], this would require 99.5 GBps of combined bandwidth from CT logs. In contrast,

with AADs, if a membership proof is 40 KBs, an append-only proof is 7KB, and domains monitor once an hour, then the bandwidth is only 400 MBps.

VII. Conclusion

In this work, we introduced the first authenticated append-only dictionary (AAD) that achieves polylogarithmic size for all proofs. This allows users to audit the dictionary themselves without resorting to third-party trusted auditors, a limitation of previous work [10], [13], which defeats the purpose of transparency. Our evaluation shows that AADs can help reduce the bandwidth of CT and CONIKS by over $200\times$, from hundreds of GBps down to hundreds of MBps. Finally, we also introduced the first authenticated append-only set (AAS), which is even more efficient than an AAD, and can be used to implement Google's Revocation Transparency (RT) [62].

Open problems. Our work leaves open a number of problems. First, our construction requires a trusted setup phase. While this can be securely executed in a distributed manner (e.g., via multi-party computation protocols [74], [75], as previously used to bootstrap the Zcash cryptocurrency [76]), it would be interesting to explore whether the same asymptotic performance can be achieved without trusted setup. Second, can we build efficient AADs with polylogarithmic proof sizes from standard assumptions, such as the existence of collision-resistant hash functions? Finally, an interesting direction would be to develop AADs with a "zero-knowledge" property which guarantees that no information about the dictionary is leaked during verification, other than the query response itself.

REFERENCES

[1] WhatsApp, "WhatsApp," https://www.whatsapp.com/features/, Accessed: 2018-04-13.

[2] A. J. Feldman, A. Blankstein, M. J. Freedman, and E. W. Felten, "Social Networking with Frientegrity: Privacy and Integrity with an Untrusted Provider," in *21st USENIX Security Symposium (USENIX Security '12)*. Berkeley, CA, USA: USENIX Association, 2012, pp. 647–662.

[3] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure Untrusted Data Repository (SUNDR)," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation (OSDI '04)*. Berkeley, CA, USA: USENIX Association, 2004.

[4] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan, "Building Web Applications on Top of Encrypted Data Using Mylar," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 157–172.

[5] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "SPORC: Group Collaboration Using Untrusted Cloud Resources," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2010, pp. 337–350.

[6] H. Adkins, "An update on attempted man-in-the-middle attacks," http://googleonlinesecurity.blogspot.com/2011/08/update-on-attempted-man-in-middle.html, Accessed: 2015-08-22.

[7] R. Mandalia, "Security breach in CA networks - Comodo, DigiNotar, GlobalSign," http://blog.isc2.org/isc2_blog/2012/04/test.html, Accessed: 2015-08-22.

[8] A. Niemann and J. Brendel, "A Survey on CA Compromises," https://www.cdc.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_CDC/Documents/Lehre/SS13/Seminar/CPS/cps2014_submission_8.pdf, Accessed: 2016-05-15.

[9] S. A. Crosby and D. S. Wallach, "Efficient Data Structures for Tamper-evident Logging," in *18th USENIX Security Symposium (USENIX Security '09)*. Berkeley, CA, USA: USENIX Association, 2009, pp. 317–334.

[10] B. Laurie, A. Langley, and E. Kasper, "RFC: Certificate Transparency," http://tools.ietf.org/html/rfc6962, Accessed: 2015-5-13.

[11] A. Eijdenberg, B. Laurie, and A. Cutter, "Verifiable Data Structures," https://github.com/google/trillian/blob/master/docs/VerifiableDataStructures.pdf, Accessed: 2018-04-12.

[12] M. D. Ryan, "Enhanced certificate transparency and end-to-end encrypted mail," in *21st Annual Network and Distributed System Security Symposium (NDSS 2014)*, Feb. 2014.

[13] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "Bringing Deployable Key Transparency to End Users," in *24th USENIX Security Symposium (USENIX Security '15)*. Berkeley, CA, USA: USENIX Association, Aug. 2015, pp. 383–398.

[14] J. Bonneau, "EthIKS: Using Ethereum to audit a CONIKS key transparency log," BITCOIN'16, 2016, http://www.jbonneau.com/doc/B16b-BITCOIN-ethiks.pdf.

[15] T. H.-J. Kim, L.-S. Huang, A. Perring, C. Jackson, and V. Gligor, "Accountable Key Infrastructure (AKI): A Proposal for a Public-key Validation Infrastructure," in *Proceedings of the 22nd International Conference on World Wide Web (WWW '13)*. New York, NY, USA: ACM, 2013, pp. 679–690.

[16] D. Basin, C. Cremers, T. H.-J. Kim, A. Perrig, R. Sasse, and P. Szalachowski, "ARPKI: Attack Resilient Public-Key Infrastructure," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. New York, NY, USA: ACM, 2014, pp. 382–393.

[17] A. Buldas, P. Laud, and H. Lipmaa, "Accountable Certificate Management using Undeniable Attestations," in *Proceedings of the 7th ACM Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2000, pp. 9–17.

[18] J. Yu, V. Cheval, and M. Ryan, "DTKI: A New Formalized PKI with Verifiable Trusted Parties," *The Computer Journal*, vol. 59, no. 11, pp. 1695–1713, 2016.

[19] Google, "HTTPS encryption on the web: Certificate transparency," https://transparencyreport.google.com/https/certificates, Accessed: 2018-04-12.

[20] R. Sleevi, "Certificate Transparency in Chrome - Change to Enforcement Date," https://groups.google.com/a/chromium.org/forum/#!msg/ct-policy/sz_3W_xKBNY/6jq2ghJXBAAJ, Accessed: 2018-04-20.

[21] M. Al-Bassam and S. Meiklejohn, "Contour: A Practical System for Binary Transparency," Tech. Rep., 2017. [Online]. Available: http://arxiv.org/abs/1712.08427

[22] S. Fahl, S. Dechand, H. Perl, F. Fischer, J. Smrcek, and M. Smith, "Hey, nsa: Stay away from my market! future proofing app markets against powerful attackers," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1143–1155.

[23] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford, "CHAINIAC: Proactive software-update transparency via collectively signed skipchains and verified builds," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 1271–1287.

[24] B. Hof and G. Carle, "Software distribution transparency and auditability," *CoRR*, vol. abs/1711.07278, 2017. [Online]. Available: http://arxiv.org/abs/1711.07278

[25] A. Tomescu and S. Devadas, "Catena: Efficient non-equivocation via bitcoin," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 393–409.

[26] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, "Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning," in *2016 IEEE Symposium on Security and Privacy*, May 2016, pp. 526–545.

[27] bitcoin.org, "0.13.0 Binary Safety Warning," https://bitcoin.org/en/alert/2016-08-17-binary-safety, Accessed: 2018-04-11.

[28] B. Ford, "Apple, FBI, and Software Transparency," https://freedom-to-tinker.com/2016/03/10/apple-fbi-and-software-transparency/, Mar. 2016, Accessed: 2017-03-08.

[29] Google, "Trillian: General Transparency," https://github.com/google/trillian, Accessed: 2018-04-12.

[30] J. Li and D. Maziéres, "Beyond one-third faulty replicas in byzantine fault tolerant systems," in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, ser. NSDI'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1973430.1973440

[31] L. Chuat, P. Szalachowski, A. Perrig, B. Laurie, and E. Messeri, "Efficient gossip protocols for verifying the consistency of Certificate logs," in *Communications and Network Security (CNS), 2015 IEEE Conference on*, Sept 2015, pp. 415–423.

[32] A. Oprea and K. D. Bowers, *Authentic Time-Stamps for Archival Storage*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 136–151.

[33] G. Edgecombe, "Compressing X.509 certificates," https://www.grahamedgecombe.com/blog/2016/12/22/compressing-x509-certificates, Accessed: 2018-04-12.

[34] L. Nguyen, *Accumulators from Bilinear Pairings and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 275–292.

[35] M. H. Overmars, *Design of Dynamic Data Structures*. Berlin, Heidelberg: Springer-Verlag, 1987.

[36] J. van den Hooff, M. F. Kaashoek, and N. Zeldovich, "VerSum: Verifiable Computations over Large Public Logs," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2014, pp. 1304–1316.

[37] L. Reyzin and S. Yakoubov, *Efficient Asynchronous Accumulators for Distributed PKI*. Cham, Switzerland: Springer International Publishing, 2016, pp. 292–309. [Online]. Available: https://doi.org/10.1007/978-3-319-44618-9_16

[38] P. Maniatis and M. Baker, "Authenticated append-only skip lists," *CoRR*, vol. cs.CR/0302010, 2003. [Online]. Available: http://arxiv.org/abs/cs.CR/0302010

[39] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia, *Persistent Authenticated Dictionaries and Their Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 379–393.

[40] S. A. Crosby and D. S. Wallach, "Authenticated Dictionaries: Real-World Costs and Trade-Offs," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 2, Sep. 2011.

[41] I. Damgård and N. Triandopoulos, "Supporting Non-membership Proofs with Bilinear-map Accumulators," Cryptology ePrint Archive, Report 2008/538, 2008, http://eprint.iacr.org/2008/538.

[42] M. H. Overmars and J. van Leeuwen, "Worst-case optimal insertion and deletion methods for decomposable searching problems," *Information Processing Letters*, vol. 12, no. 4, pp. 168 – 173, 1981.
[43] T. Pulls and R. Peeters, *Balloon: A Forward-Secure Append-Only Persistent Authenticated Data Structure*. Cham, Switzerland: Springer International Publishing, 2015, pp. 622–641.
[44] M. Chase, A. Deshpande, and E. Ghosh, "Privacy preserving verifiable key directories," Cryptology ePrint Archive, Report 2018/607, 2018, https://eprint.iacr.org/2018/607.
[45] M. Chase and S. Meiklejohn, "Transparency Overlays and Applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2016, pp. 168–179.
[46] B. Dowling, F. Günther, U. Herath, and D. Stebila, "Secure Logging Schemes and Certificate Transparency," in *Computer Security - ESORICS 2016*, I. Askoxylakis, S. Ioannidis, S. Katsikas, and C. Meadows, Eds. Cham, Switzerland: Springer International Publishing, 2016, pp. 140–158.
[47] S. Eskandarian, E. Messeri, J. Bonneau, and D. Boneh, "Certificate Transparency with Privacy," *Proceedings on Privacy Enhancing Technologies*, vol. 2017, no. 4, pp. 329–344, 2017.
[48] R. Peeters and T. Pulls, "Insynd: Improved Privacy-Preserving Transparency Logging," in *Computer Security – ESORICS 2016*, I. Askoxylakis, S. Ioannidis, S. Katsikas, and C. Meadows, Eds. Cham, Switzerland: Springer International Publishing, 2016, pp. 121–139.
[49] R. Dahlberg and T. Pulls, "Verifiable light-weight monitoring for certificate transparency logs," *CoRR*, vol. abs/1711.03952, 2017. [Online]. Available: http://arxiv.org/abs/1711.03952
[50] A. Kate, G. M. Zaverucha, and I. Goldberg, *Constant-Size Commitments to Polynomials and Their Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 177–194.
[51] A. Menezes, S. Vanstone, and T. Okamoto, "Reducing elliptic curve logarithms to logarithms in a finite field," in *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, ser. STOC '91. New York, NY, USA: ACM, 1991, pp. 80–89.
[52] A. Joux, *A One Round Protocol for Tripartite Diffie–Hellman*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 385–393.
[53] D. Boneh, B. Lynn, and H. Shacham, *Short Signatures from the Weil Pairing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 514–532.
[54] C. Papamanthou, R. Tamassia, and N. Triandopoulos, *Optimal Verification of Operations on Dynamic Sets*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 91–110.
[55] V. Goyal, *Reducing Trust in the PKG in Identity Based Cryptosystems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 430–447. [Online]. Available: https://doi.org/10.1007/978-3-540-74143-5_24
[56] J. Groth, "Short pairing-based non-interactive zero-knowledge arguments," in *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, 2010, pp. 321–340. [Online]. Available: https://doi.org/10.1007/978-3-642-17373-8_19
[57] J. von zur Gathen and J. Gerhard, "Fast Euclidean Algorithm," in *Modern Computer Algebra*, 3rd ed. New York, NY, USA: Cambridge University Press, 2013, ch. 11, pp. 313–333.
[58] ——, "Fast Multiplication," in *Modern Computer Algebra*, 3rd ed. New York, NY, USA: Cambridge University Press, 2013, ch. 8, pp. 221–254.
[59] F. P. Preparata and D. V. Sarwate, "Computational Fourier Transforms Complexity of Over Finite Fields," *Mathematics of Computation*, vol. 31, no. 139, pp. 740–751, 1977.
[60] A. V. Aho and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing, 1974.
[61] J. von zur Gathen and J. Gerhard, "Fast polynomial evaluation and interpolation," in *Modern Computer Algebra*, 3rd ed. New York, NY, USA: Cambridge University Press, 2013, ch. 10, pp. 295–310.
[62] B. Laurie, "Revocation Transparency," https://www.links.org/files/RevocationTransparency.pdf, Accessed: 2018-07-31.
[63] C. Papamanthou and R. Tamassia, *Time and Space Efficient Algorithms for Two-Party Authenticated Data Structures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1–15.
[64] M. Naor and K. Nissim, "Certificate Revocation and Certificate Update," in *7th USENIX Security Symposium (USENIX Security '98)*. Berkeley, CA, USA: USENIX Association, 1998.
[65] Keybase.io, "Keybase," http://keybase.io, Accessed: 2016-05-15.
[66] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun, "Verena: End-to-end integrity protection for web applications," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016.
[67] SCIPR Lab, "libff," https://github.com/scipr-lab/libff, 2016, Accessed: 2018-07-28.
[68] S. Mitsunari, "A Fast Implementation of the Optimal Ate Pairing over BN curve on Intel Haswell Processor," Cryptology ePrint Archive, Report 2013/362, 2013, https://eprint.iacr.org/2013/362.
[69] P. S. L. M. Barreto and M. Naehrig, "Pairing-friendly elliptic curves of prime order," in *Selected Areas in Cryptography*, B. Preneel and S. Tavares, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 319–331.
[70] SCIPR Lab, "libfqfft," https://github.com/scipr-lab/libfqfft, 2016, Accessed: 2018-07-28.
[71] V. Shoup, "libntl," https://www.shoup.net/ntl/, 2016, Accessed: 2018-07-28.
[72] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica, "DIZK: A distributed zero knowledge proof system," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/wu
[73] "The go programming language," https://golang.org/#, Accessed: 2018-07-30.
[74] S. Bowe, A. Gabizon, and M. D. Green, "A multi-party protocol for constructing the public parameters of the pinocchio zk-snark," Cryptology ePrint Archive, Report 2017/602, 2017, https://eprint.iacr.org/2017/602.
[75] S. Bowe, A. Gabizon, and I. Miers, "Scalable multi-party computation for zk-snark parameters in the random beacon model," Cryptology ePrint Archive, Report 2017/1050, 2017, https://eprint.iacr.org/2017/1050.
[76] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, "Zcash Protocol Specification," https://github.com/zcash/zips/blob/master/protocol/protocol.pdf, Accessed: 2017-11-17.
[77] N. Bitansky, R. Canetti, O. Paneth, and A. Rosen, "On the existence of extractable one-way functions," *SIAM J. Comput.*, vol. 45, no. 5, pp. 1910–1952, 2016. [Online]. Available: https://doi.org/10.1137/140975048
[78] E. Boyle and R. Pass, "Limits of extractability assumptions with distributional auxiliary input," in *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, 2015, pp. 236–261. [Online]. Available: https://doi.org/10.1007/978-3-662-48800-3_10

APPENDIX

A. Cryptographic Assumptions

Our security analysis utilizes the following two cryptographic assumptions over elliptic curve groups with bilinear pairings.

Definition A.1 (q-SBDH Assumption)**.** Given security parameter λ, bilinear pairing parameters $\langle \mathbb{G}, \mathbb{G}_T, p, g, e \rangle \leftarrow \mathcal{G}(1^\lambda)$, public parameters $\langle g, g^s, g^{s^2}, \ldots, g^{s^q} \rangle$ for some $q = \mathsf{poly}(\lambda)$ and some s chosen uniformly at random from $\mathbb{Z}_p^*$, no probabilistic polynomial-time adversary can output a pair $\langle c, e(g,g)^{\frac{1}{s+c}} \rangle$ for some $c \in \mathbb{Z}_p$, except with probability negligible in λ.

Definition A.2 (q-PKE Assumption)**.** The q-power knowledge of exponent assumption holds for $\mathcal{G}$ if for all probabilistic polynomial-time adversaries A, there exists a probabilistic polynomial time extractor χ_A such that for all *benign* auxiliary

inputs $z \in \{0,1\}^{poly(\lambda)}$

$$\Pr\left[\begin{array}{c} \langle \mathbb{G}, \mathbb{G}_T, p, g, e\rangle \leftarrow \mathcal{G}(1^\lambda); \langle s, \tau\rangle \leftarrow \mathbb{Z}_p^*; \\ \sigma \leftarrow \langle \mathbb{G}, \mathbb{G}_T, p, g, e, \mathcal{PP}_q(s,\tau)\rangle; \\ \langle c, \hat{c}; a_0, a_1, \ldots, a_q\rangle \leftarrow (A||\chi_A)(\sigma, z) : \\ \hat{c} = c^\tau \wedge c \neq g^{\prod_{i=0}^{q} a_i s^i} \end{array}\right] = \mathsf{negl}(\lambda)$$

where $\langle y_1; y_2\rangle \leftarrow (A||\chi_A)(x)$ means A returns y_1 on input x and χ_A returns y_2 given the same input x and A's random tape. Auxiliary input z is required to be drawn from a benign distribution to avoid known negative results associated with knowledge-type assumptions [77], [78].

B. AAS Membership Security Proof Sketches

Membership and append-only correctness follow from close inspection of the algorithms. Here, we provide proof sketches for membership and append-only security, as well as fork-consistency. Full proofs are delegated to the extended version of the paper.

Membership security. Assume there exists polynomial-time adversary A that produces digest d, element e and proofs π, π', of membership and non-membership respectively, such that $\mathsf{VerMemb}(VK, d, e, 1, \pi)$ and $\mathsf{VerMemb}(VK, d, e, 0, \pi')$ both accept. We will now describe how A can either find a collision in $\mathcal{H}$ or break the q-SBDH assumption.

Let k be the index of the BPT from the digest d involved in the membership proof π. First, observe that π contains accumulator values $\sigma_1, \ldots, \sigma_k$ corresponding to the path from e's leaf to the root of the kth BPT. Next, let p_k be the prefix of e that is included in the frontier proof π' for the k-th BFT. Second, π' contains accumulators $a_1, \ldots, a_{k'}$ corresponding to the path from p_k's leaf to the root of the kth BFT. (k' may be different than k because the BFT has different height than the BPT.) Let σ_k^* be the root accumulator for the kth BPT, as contained in π'.

When verifying π and π', both σ_k and σ_k^* are hashed (together with two other hash values, mapping to their corresponding claimed children), and the result is checked against the common hash $h_k \in d$ corresponding to the root of the kth BPT. Since verification of π and π' succeeds, if $\sigma_k \neq \sigma_k^*$ this would imply a collision in $\mathcal{H}$ has been found.

Else, we argue as follows. Each accumulator $\sigma_1, \ldots, \sigma_k$ is accompanied by an extractability term $\sigma_1', \ldots, \sigma_k'$. As part of membership verification, the client checks for each such pair that $e(\sigma_j, g^\tau) = e(\sigma_j', g)$ for $j = 1, \ldots, k$. Hence, from the q-PKE assumption, it follows that for each σ_j there exists a polynomial time algorithm that, upon receiving the same input as A, outputs a polynomial $O_j(x)$ (in coefficient form) such that $g^{O_j(s)} = \sigma_j$ with all but negligible probability. The same holds for all accumulators $a_1, \ldots, a_{k'}$ and terms $a_1', \ldots, a_{k'}'$ included in π', and let $Q_{j'}(x)$ denote the polynomial corresponding to $a_{j'}$ for $j' = 1, \ldots, \log(F(S_k))$ where S_k denotes the set of elements in the kth BPT. Finally, let $O_0(x) = \prod_{c \in P(e)}(x - \mathcal{H}_\mathbb{F}(c))$ and $O_0'(x) = (x - \mathcal{H}_\mathbb{F}(p_k))$ (both of which are computed by the verifier)

We distinguish the following two cases and analyze them separately:

(a) $(x - \mathcal{H}_\mathbb{F}(p_k)) \nmid O_k(x)$ or $(x - \mathcal{H}_\mathbb{F}(p_k)) \nmid O_k'(x)$
(b) $(x - \mathcal{H}_\mathbb{F}(p_k)) \mid O_k(x)$ and $(x - \mathcal{H}_\mathbb{F}(p_k)) \mid O_k'(x)$

For case (a), without loss of generality we will focus on the first sub-case, i.e., $(x - \mathcal{H}_\mathbb{F}(p_k)) \nmid O_k(x)$. (The proof for the second sub-case proceeds identically.) First, let $\pi_1, \ldots, \pi_k$ denote the subset proofs from the membership proof π (i.e., $e(\sigma_i, g) = e(\sigma_{i-1}, \pi_{i-1}), \forall i \in [1, k]$). Second, observe that, by construction, $(x - \mathcal{H}_\mathbb{F}(p_k)) \mid O_0(x)$. Since, by assumption $(x - \mathcal{H}_\mathbb{F}(p_k)) \nmid O_k(x)$, there must exist some index $0 < \xi < k$ such that $(x - \mathcal{H}_\mathbb{F}(p_k)) \nmid O_\xi(x)$ and $(x - \mathcal{H}_\mathbb{F}(p_k)) \mid O_{\xi-1}(x)$, which can be efficiently deduced with access to all polynomials O_j. Therefore, by polynomial division there exist efficiently computable polynomials $Q_\xi(x), Q_{\xi-1}(x)$ and integer $\kappa \in \mathbb{Z}_p$ such that: $O_{\xi-1}(x) = (x - \mathcal{H}_\mathbb{F}(p_k)) \cdot Q_{\xi-1}(x)$ and $O_\xi(x) = (x - \mathcal{H}_\mathbb{F}(p_k)) \cdot Q_\xi(x) + \kappa$.

Based on the above, it must hold that:

$$\begin{aligned} e(\sigma_\xi, g) &= e(\sigma_{\xi-1}, \pi_{\xi-1}) \\ e(g^{O_\xi(s)}, g) &= e(g^{O_{\xi-1}(s)}, \pi_{\xi-1}) \\ e(g^{(s-\mathcal{H}_\mathbb{F}(p_k))\cdot Q_\xi(s)+\kappa}, g) &= e(g^{(s-\mathcal{H}_\mathbb{F}(p_k))\cdot Q_{\xi-1}(s)}, \pi_{\xi-1}) \\ e(g^{Q_\xi(s)+\frac{\kappa}{(s-\mathcal{H}_\mathbb{F}(p_k))}}, g) &= e(g^{Q_{\xi-1}(s)}, \pi_{\xi-1}) \\ e(g^{\frac{\kappa}{(s-\mathcal{H}_\mathbb{F}(p_k))}}, g) &= e(g^{Q_{\xi-1}(s)}, \pi_{\xi-1}) \cdot e(g^{-Q_\xi(s)}, g) \\ e(g^{\frac{1}{(s-\mathcal{H}_\mathbb{F}(p_k))}}, g) &= \left[e(g^{Q_{\xi-1}(s)}, \pi_{\xi-1}) \cdot e(g^{-Q_\xi(s)}, g)\right]^{\kappa^{-1}}. \end{aligned}$$

Hence, the pair $(\mathcal{H}_\mathbb{F}(p_k), \left[e(g^{Q_{\xi-1}(s)}, \pi_{\xi-1}) \cdot e(g^{-Q_\xi(s)}, g)\right]^{\kappa^{-1}})$ can be used to break the q-SBDH assumption.

In case (b), by assumption $(x - \mathcal{H}_\mathbb{F}(p_k)) \mid O_k(x)$ and $(x - \mathcal{H}_\mathbb{F}(p_k)) \mid O_k'(x)$. Therefore, by polynomial division there exist efficiently computable polynomials $Q_\xi(x), Q_{\xi-1}(x)$ such that: $O_{k-1}(x) = (x - \mathcal{H}_\mathbb{F}(p_k)) \cdot Q_{\xi-1}(x)$ and $O_k(x) = (x - \mathcal{H}_\mathbb{F}(p_k)) \cdot Q_\xi(x)$. Let $\delta_k = (\Delta_1, \Delta_2)$ be the proof of disjointness embedded in π' with respect to the roots accumulator σ_k of the BPT and a_k of the BFT. Since verification succeeds, it holds that:

$$\begin{aligned} e(\sigma_k, \Delta_1) \cdot e(a_k, \Delta_2) &= e(g, g) \\ e(g^{O_k(s)}, \Delta_1) \cdot e(g^{O_k'(s)}, \Delta_2) &= e(g, g) \\ e(g^{(s-\mathcal{H}_\mathbb{F}(p_k))\cdot Q_\xi(s)}, \Delta_1) \cdot e(g^{(s-\mathcal{H}_\mathbb{F}(p_k))\cdot Q_\xi'(s)}, \Delta_2) &= e(g, g) \\ e(g^{Q_\xi(s)}, \Delta_1) \cdot e(g^{Q_\xi'(s)}, \Delta_2) &= e(g, g)^{\frac{1}{(s-\mathcal{H}_\mathbb{F}(p_k))}}. \end{aligned}$$

Thus, the pair $(\mathcal{H}_\mathbb{F}(p_k), e(g^{Q_\xi(s)}, \Delta_1) \cdot e(g^{Q_\xi'(s)}, \Delta_2))$ can again be used to break the q-SBDH assumption.

Append-only security. Append-only security can be proven with the same techniques as the ones we used for membership security. Let p be the prefix of e that is used to prove non-membership with respect to $d_{i'}$. The membership proof for e with respect to d_i again involves a series of accumulators for which the corresponding polynomials can be extracted. By our previous analysis, $(x - \mathcal{H}_\mathbb{F}(p))$ must divide the polynomial extracted for the corresponding BPT root in d_i, otherwise the q-SBDH assumption can be broken. Continuing on this sequence of subset proofs, the append-only proof π_a "connects" this root accumulator to a root accumulator in $d_{i'}$.

By the same argument $(x - \mathcal{H}_{\mathbb{F}}(p))$ must also divide the polynomial extracted for this BPT root. Since non-membership also passed verification, the same holds with respect to the extracted polynomial for the root of the corresponding BFT in $d_{i'}$, else again q-SBDH can be broken. Finally, we apply the same argument as case (b) above, since $(x - \mathcal{H}_{\mathbb{F}}(p))$ divides both these polynomials and we have a disjointness proof for their accumulators, again breaking q-SBDH.

Fork-consistency. Fork-consistency follows directly from the collision-resistance of $\mathcal{H}$. Assume two parties receive digests $d_i \neq d_i'$ and subsequent digest d_j such that VerAppendOnly accepted when run on $\mathsf{VerAppendOnly}(VK, d_i, i, d_j, j, \pi_i)$ and $\mathsf{VerAppendOnly}(VK, d_i', i, d_j, j, \pi_i')$. In particular, suppose that d_i, d_i' disagree at BPT root r (there must exist at least one such root). Let a_r, a_r' be the accumulators of r in the two digests respectively and h_r, h_r' their respective hashes.

Since $\mathcal{H}$ is deterministic, it follows that $a_r \neq a_r'$. If $h_r = h_r'$ then this yields a collision in $\mathcal{H}$. Else, we argue as follows. Observe that the execution of MerkleHash (initiated at Line 21 of Algorithm 5) will perform a sequence of recursive hashes where: *(a)* the innermost input includes h_r when running VerAppendOnly for d_i and h_r' when running VerAppendOnly for d_i' , *(b)* the outermost hash is the same (the hash of the root of their common BPT, as included in d_j) in both cases, and *(c)* the label w is the same for both the nodes r and r', corresponding to their common position in the BPT (as designated by the old root's label). The last is ensured while checking that the Merkle proofs for all old roots are well-formed (as part of VerAppendOnly) and the fact that node labels are deterministically encoded based on append order. Thus, the assertion at Line 21 implies finding a collision in $\mathcal{H}$.

C. AAS Asymptotic Analysis

Suppose we have a *worst-case* AAS with $n = 2^i - 1$ elements. This AAS has i BPTs of size $n/2, n/4, \ldots, 1$. The BFTs corresponding to the BPTs will be of size $O(\lambda n/2), O(\lambda n/4), \ldots, O(1)$.

Space. The space is dominated by the BFTs, which take up $O(\lambda n/2) + O(\lambda n/4) + \cdots + O(1) = O(\lambda n)$ space. (The BPTs together only take up $O(n)$ space.)

Membership proof size. Suppose an element e is in the AAS in some BPT. To prove membership of e, we show a path from e's leaf in the BPT to the BPT's root accumulator consisting of constant-sized subset proofs at every node. Sine the largest BPT in the forest has height $\log(n/2)$, the membership proof is $O(\log n)$-sized.

Non-membership proof size. To prove non-membership of an element e, we show a frontier proof for a prefix of e in every BFT in the forest. The largest BFT has $O(\lambda n)$ nodes so frontier proofs are $O(\log(\lambda n))$-sized. Because there are $O(\log n)$ BFTs, all the frontier proofs are $O(\log n \log(\lambda n)) = O(\log^2 n)$-sized.

Append-only proof size. Our append-only proof is $O(\log n)$-sized. This is because our proof consists of paths from each old root in the old forest up to a single new root (excluding common roots). Because the old roots are roots of adjacent trees in the forest, there will be a single $O(\log n)$-sized Merkle path connecting the old roots to the new root. In other words, our append-only proofs are similar to the append-only proofs from history trees [9].

D. Append-only Authenticated Dictionary Definitions

Notation. Let $|S|$ denote the number of elements in a multiset S (e.g., $S = \{1, 2, 2\}$ and $|S| = 3$). Let $\mathcal{K}$ be the set of all possible keys and $\mathcal{V}$ be the set of all possible values. Formally, a *dictionary* is a function $D : K \to \mathcal{P}(\mathcal{V})$ that maps a key $k \in K$ to a multiset of values $V \in \mathcal{P}(\mathcal{V})$ (including the empty set), where $K \subset \mathcal{K}$ and $\mathcal{P}(\mathcal{V})$ denotes all possible multisets with elements from $\mathcal{V}$. Thus, $D(k)$ denotes the multiset of values associated with key k in dictionary D. Let $|D|$ denote the number of key-value pairs in the dictionary or its *version*. Appending $\langle k, v\rangle$ to a version i dictionary updates the multiset $V = D(k)$ of key to $V' = V \cup \{v\}$ and increments the version to $i + 1$.

We use $\langle k, v\rangle \in D$ to denote that v is one of the values of key k. We use $k \notin D$ to denote that key k has no value in D (i.e., $D(k) = \varnothing$). We use $k \in D$ to denote k has some value(s) in D that we don't know or care about. We use $D \subseteq D'$ to indicate D is a subset of D'. Subset here means that $\forall k \in D, D(k) \subseteq D'(k)$. We often use D_n to denote a dictionary of size n (i.e., $|D_n| = n$). We use $\mathcal{D}_n$ to denote the *authenticated* version of D_n which stores implementation-specific authentication information next to D_n. The $\in, \notin, \subseteq, \supseteq$ notation used with a dictionary D_n can be used with its authenticated version $\mathcal{D}_n$.

Server-side API. The untrusted server managing the dictionary implements:

$\mathsf{Setup}(1^\lambda, \beta) \to pp, VK$. Randomized algorithm that returns public parameters pp for the AAD scheme, which include a *verification key* VK used by clients. Here, λ is a security parameter and β is an upper-bound on the number of elements n in the dictionary (i.e., $n \leq \beta$). If the scheme has a trapdoor, anyone in possession of the trapdoor can break the security of the scheme (defined in Appendix D). In that case, $\mathsf{Setup}(\cdot)$ has to be run by a trusted entity who promises to "forget" the trapdoor.

$\mathsf{Append}(pp, \mathcal{D}_i, d_i, k, v) \to \mathcal{D}_{i+1}, d_{i+1}$. Deterministic algorithm that appends a new key-value pair $\langle k, v\rangle$ to the version i dictionary creating a new version $i+1$ dictionary. Succeeds only if the dictionary is not full (i.e., $i + 1 \leq \beta$). Returns the new authenticated dictionary $\mathcal{D}_{i+1}$ and its digest d_{i+1}. Importantly, clients can verify the version number $i + 1$ claimed by the server in d_{i+1}.

$\mathsf{ProveMemb}(pp, \mathcal{D}_i, k) \to V, \pi_{k,V}$. Deterministic algorithm that proves *complete* membership for all values of key k, if any. When $\mathcal{D}_i(k) = V$ and $V \neq \varnothing$, generates a membership proof $\pi_{k,V}$ that V is the complete multiset of values for key k. When $\mathcal{D}_i(k) = \varnothing$, generates a non-membership proof that key k has no values. Finally, the server cannot construct

a fake proof $\pi_{k,V'}$ for the wrong V', including for $V' = \varnothing$ (see Appendix D).

$\mathsf{ProveAppendOnly}(pp, \mathcal{D}_i, \mathcal{D}_j) \rightarrow \pi_{i,j}$. Deterministic algorithm that proves $\mathcal{D}_i \subseteq \mathcal{D}_j$ (see notation in Appendix D). Generates an *append-only proof* $\pi_{i,j}$ that all key-value pairs in $\mathcal{D}_i$ are also present and unchanged in $\mathcal{D}_j$. Importantly, a malicious server who removed or changed keys from $\mathcal{D}_j$ that were present in $\mathcal{D}_i$ cannot construct a valid append-only proof (see Appendix D).

Client-side API. Clients implement:

$\mathsf{VerMemb}(VK, d_i, k, V, \pi) \rightarrow \{T, F\}$. Deterministic algorithm that verifies proofs returned by $\mathsf{ProveMemb}(\cdot)$ against the digest d_i at version i of the dictionary. When $V \neq \varnothing$, verifies that V is the complete multiset of values for key k, ensuring no values have been left out and no extra values were added. When $V = \varnothing$, verifies that key k is not mapped to any value in the dictionary with digest d_i. (We formalize security in Appendix D.)

$\mathsf{VerAppendOnly}(VK, d_i, i, d_j, j, \pi_{i,j}) \rightarrow \{T, F\}$. Deterministic algorithm that ensures a dictionary remains append-only. Verifies that $\pi_{i,j}$ correctly proves that the dictionary with digest d_j is a superset of the dictionary with digest d_i (see Appendix D). Also, verifies that d_i and d_j are digests of dictionaries at version i and j, respectively.

AAD Correctness and Security Definitions Consider an ordered sequence of n key-value pairs $(k_i \in \mathcal{K}, v_i \in \mathcal{V})_{i\in[n]}$. Note that the same key (or key-value pair) can occur multiple times in the sequence. Let $\mathcal{D}', d' \leftarrow \mathsf{Append}^+(pp, \mathcal{D}, d, (k_i, v_i)_{i\in[n]})$ denote a sequence of $\mathsf{Append}(\cdot)$ calls arbitrarily interleaved with other $\mathsf{ProveMemb}(\cdot)$ and $\mathsf{ProveAppendOnly}(\cdot)$ calls such that $\mathcal{D}', d' \leftarrow \mathsf{Append}(pp, \mathcal{D}_{n-1}, d_{n-1}, k_n, v_n)$, $\mathcal{D}_{n-1}, d_{n-1} \leftarrow \mathsf{Append}(pp, \mathcal{D}_{n-2}, d_{n-2}, k_{n-1}, v_{n-1})$, $\ldots$, $\mathcal{D}_1, d_1 \leftarrow \mathsf{Append}(pp, \mathcal{D}, d, k_1, v_1)$. Let D_n denote the *corresponding dictionary* obtained after appending each $(k_i, v_i)_{i\in[n]}$ in order. Finally, let $\mathcal{D}_0$ denote an empty dictionary with empty digest d_0.

Definition A.3 (Append-only Authenticated Dictionary)**.** (Setup, Append, $\mathsf{ProveMemb}$, $\mathsf{ProveAppendOnly}$, $\mathsf{VerMemb}$, $\mathsf{VerAppendOnly}$) is a secure append-only authenticated dictionary (AAD) if, $\forall$ security parameters λ, $\forall$ upper-bounds $\beta = \mathsf{poly}(\lambda)$ and $\forall n \leq \beta$ it satisfies the following properties:

Complete membership correctness. $\forall$ sequences $(k_i \in \mathcal{K}, v_i \in \mathcal{V})_{i\in[n]}$ with corresponding dictionary D_n, $\forall$ keys $k \in \mathcal{K}$,

$$\Pr\left[\begin{array}{c}(pp, VK) \leftarrow \mathsf{Setup}(1^\lambda, \beta),\\ (\mathcal{D}, d) \leftarrow \mathsf{Append}^+(pp, \mathcal{D}_0, d_0, (k_i, v_i)_{i\in[n]}),\\ (V, \pi) \leftarrow \mathsf{ProveMemb}(pp, \mathcal{D}, k) :\\ V = D_n(k) \wedge \mathsf{VerMemb}(VK, d, k, V, \pi) = T\end{array}\right] = \eta(\lambda)$$

Observation: Note that this definition compares the returned multiset V with the "ground truth" in D_n and thus provides complete membership correctness. Also, it handles non-membership correctness since V can be the empty set. Finally, the definition handles all possible orders of inserting key-value pairs.

Complete membership security. $\forall$ adversaries A running in time $\mathsf{poly}(\lambda)$,

$$\Pr\left[\begin{array}{c}(pp, VK) \leftarrow \mathsf{Setup}(1^\lambda, \beta),\\ (d, k, V \neq V', \pi, \pi') \leftarrow \mathsf{A}(pp) :\\ \mathsf{VerMemb}(VK, d, k, V, \pi) = T \wedge\\ \mathsf{VerMemb}(VK, d, k, V', \pi') = T\end{array}\right] = \varepsilon(\lambda)$$

Observation: This definition captures the lack of any "ground truth" about what was inserted in the dictionary, since there is no trusted source in our model. Nonetheless, given a fixed digest d, our definition prevents *all* equivocation attacks about the complete multiset of values of a key, including the special case where the server equivocates about the key being present (i.e., $V \neq \varnothing$ and $V' = \varnothing$). Note that this definition also implies that different dictionaries cannot have the same digest.

Append-only correctness. $\forall$ sequences $(k_i \in \mathcal{K}, v_i \in \mathcal{V})_{i\in[n]}$ where $n \geq 2$

$$\Pr\left[\begin{array}{c}(pp, VK) \leftarrow \mathsf{Setup}(1^\lambda, \beta)\\ (\mathcal{D}_m, d_m) \leftarrow \mathsf{Append}^+(pp, \mathcal{D}_0, d_0, (k_i, v_i)_{i\in[m]}),\\ (\mathcal{D}_n, d_n) \leftarrow \mathsf{Append}^+(pp, \mathcal{D}_m, d_m, (k_j, v_j)_{j\in[m+1,n]}),\\ \pi \leftarrow \mathsf{ProveAppendOnly}(pp, \mathcal{D}_m, \mathcal{D}_n) :\\ \mathsf{VerAppendOnly}(VK, d_m, m, d_n, n, \pi) = T\end{array}\right] = \eta(\lambda)$$

Append-only security. $\forall$ adversaries A running in time $\mathsf{poly}(\lambda)$,

$$\Pr\left[\begin{array}{c}(pp, VK) \leftarrow \mathsf{Setup}(1^\lambda, \beta)\\ (d_i, d_j, i < j, \pi_a, k, V \neq V', \pi, \pi') \leftarrow \mathsf{A}(pp) :\\ \mathsf{VerAppendOnly}(VK, d_i, i, d_j, j, \pi_a) = T \wedge\\ \mathsf{VerMemb}(VK, d_i, k, V, \pi) = T \wedge\\ \mathsf{VerMemb}(VK, d_j, k, V', \pi') = T\end{array}\right] = \varepsilon(\lambda)$$

Observation: This definition ensures that values can only be added to a key and can never be removed or changed.

Fork consistency. This definition stays the same as in Section III-B.