Transparency Logs via Append-Only Authenticated Dictionaries

Alin Tomescu
Massachusetts Institute of Technology

Vivek Bhupatiraju
Lexington High School
MIT PRIMES

Dimitrios Papadopoulos
Hong Kong University of Science and Technology

Charalampos Papamanthou
University of Maryland

Nikos Triandopoulos
Stevens Institute of Technology

Srinivas Devadas
Massachusetts Institute of Technology

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 bandwidth-efficient when queried by users. Specifically, everyone should be able to efficiently *look up* log entries by their key *and* efficiently verify that the log remains *append-only*. Unfortunately, without additional trust assumptions, current transparency logs cannot provide both small-sized *lookup proofs* and small-sized *append-only proofs*. In fact, one of the proofs always requires bandwidth linear in the size of the log, making it expensive for everyone to query the log. 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 and helps reduce bandwidth consumption in transparency logs. This comes at the cost of increased append times and high memory usage, both of which remain to be improved to make practical deployment possible.

CCS CONCEPTS

• Security and privacy → Key management; **• Theory of computation → Cryptographic primitives**; **Data structures design and analysis**.

KEYWORDS

append-only; transparency logs; authenticated dictionaries; Merkle trees; bilinear accumulators; RSA accumulators; polynomials

ACM Reference Format:
Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. 2019. Transparency Logs via Append-Only Authenticated Dictionaries. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3319535.3345652

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '19, November 11–15, 2019, London, United Kingdom

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3345652

1 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 with the website. In general, many systems require a PKI or assume one exists [39, 40, 60, 84]. Yet, despite their necessity, PKIs have proven difficult to secure as evidenced by past CA compromises [1, 63, 77].

To address such attacks, *transparency logs* [29, 36, 59] have been proposed as a way of building *accountable* (and thus more secure) PKIs. A transparency log is a *dictionary* managed by an untrusted *log server*. The server periodically appends *key-value pairs* to the dictionary and is queried by mutually-distrusting *users*, who want to know certain keys' values. For example, in *key transparency* [8, 23, 55, 59, 65, 88, 95, 104], 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 log and CAs can be held 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) [59] project. Since its initial March 2013 deployment, CT has publicly logged over 2.1 billion certificates [44]. Furthermore, since April 2018, Google's Chrome browser requires all new certificates to be published in a CT log [93]. In the same spirit, there has been increased research effort into *software transparency* schemes [2, 38, 49, 78, 94, 96] for securing software updates. Furthermore, Google is prototyping *general transparency* logs [36, 45] via their Trillian project [45]. Therefore, it is not far-fetched to imagine generalized transparency improving our census system, our elections, and perhaps our government. But to realize their full potential, transparency logs must operate correctly or be easily caught otherwise. Specifically:

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 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. Ideally, a user with a "short" digest h_{old} should accept a new digest h_{new} only if it comes with a succinct *append-only proof* computed by the log. This proof should convince the user that the old log with digest h_{old} is a subset of the new log with digest h_{new}.

Logs should support lookups. When users have access to digests (instead of whole logs), the central question becomes: How can a user check *against their digest* which values are registered for a certain key k in the log? Ideally, a small *lookup proof* should convince the user 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, even though both users have the same digest and should thus see the same set of values for key k.

Logs should remain fork-consistent. An unavoidable issue is that a malicious log server can also equivocate about digests and *fork* users [29, 60]. For example, at time i, the server can append (k, v) to one user's log while appending (k, v') to another user's log. Since the two users' logs will differ at location i, their digests will also differ. Intuitively, *fork consistency* [60, 61] guarantees that if two users are given two different digests as above, they must forever be given different digests. Thus, users can *gossip* [28, 32, 94, 96] to check if they are seeing different digests and detect forks.

Challenges. Building transparency logs with succinct lookup and append-only proofs is a long-standing open problem. At first glance, a Merkle-based [68] solution seems possible. Unfortunately, it appears very difficult to organize a Merkle tree so as to support both succinct append-only proofs and succinct lookup proofs. On one hand, trees with chronologically-ordered leaves [29, 64, 97] support logarithmic-sized append-only proofs but at the cost of linear-sized lookup proofs. On the other hand, trees can be lexicographically-ordered by key [6, 23, 30, 79] to support succinct lookup proofs at the cost of linear append-only proofs (see Section 6.2).

It might seem natural to combine the two and obtain succinct lookup proofs via the lexicographic tree and succinct append-only proofs via the chronologic tree [88]. But this does not work either, since there must be a succinct proof that the two trees "correspond": they are correctly built over the same set of key-value pairs. While previous transparency logs [88, 104] work around this by having users "collectively" verify that the two trees correspond [26, 88, 104], this requires a sufficiently high number of honest users and can result in slow detection. An alternative, which we discuss in Section 7.1, is to use SNARKs [42, 48]. At second glance, *cryptographic accumulators* [13, 76] seem useful for building transparency logs (see Section 2.1). Unfortunately, accumulators are asymptotically-inefficient, requiring linear time to compute proofs or to update proofs after a change to the set. As a result, a computationally-efficient accumulator-based solution is not obvious.

Our contribution. We introduce a novel cryptographic primitive called an *append-only authenticated dictionary (AAD)*. An AAD maps a key to one or more values in an append-only fashion and is an abstraction for a transparency log. We are the first to give security definitions for AADs. We are also the first to instantiate asymptotically *efficient* AADs from *bilinear accumulators* [76] (see Section 5). Importantly, our design does not rely on collective verification by users or on trusted third parties and assumes only an untrusted log server. Our AAD offers logarithmic-sized append-only proofs, polylogarithmic-sized lookup proofs and polylogarithmic worst-case time appends (see Table 1).

We implement our AAD in C++ and evaluate it. Our code is available at https://github.com/alinush/libaad-ccs2019. Our lookup

Table 1: Asymptotic costs of our construction versus previous work. n is the number of key-value pairs in the dictionary and λ is the security parameter.

Time & bandwidth	Space	Append time	Lookup proof size	Append-only proof size
Lexicographic trees [65, 88]	$n \log n$	$\log n$	$\log n$	n
Chronologic trees [29, 59]	n	$\log n$	n	$\log n$
AAD (this work)	λn	$\lambda \log^3 n$	$\log^2 n$	$\log n$

and append-only proofs are in the order of tens of KiBs 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 KiB and verifies in 2.5 seconds. While our lookup proof sizes are larger than in previous work, our small-sized append-only proofs can help significantly reduce the overall bandwidth consumption in transparency logs, as we show in Section 6.2.1.

Limitations of our approach. Our construction has high append times (i.e., a few seconds per append) and high memory usage (i.e., hundreds of GiBs for an AAD of size 2^{20}). This means it is not yet practical and we discuss how future work might improve it in Sections 6.1.1 and 6.1.4. The security of our construction relies on the q-PKE "knowledge" assumption (commonly used in SNARKs [43, 47]). Hence, we need a large set of public parameters that must be generated via a *trusted setup* phase, which complicates deployment. We discuss how the trusted setup can be decentralized in Section 7.

Overview of techniques. We first build an efficient *append-only authenticated set* (AAS), instead of an AAD. An AAS is an append-only set of elements with proofs of (non)membership of any element. If we let elements be revoked certificates, then an AAS efficiently implements Revocation Transparency (RT) [58]. But to efficiently implement *any* transparency log, we must modify our AAS into an AAD, which is more "expressive." Specifically, an AAD can provably return all values of a key, while an AAS can only prove that an element is or is not in the set. One could attempt to build an AAD from an AAS in "black-box" fashion by representing an AAD key-value pair as an AAS element. Unfortunately, this is not sufficient if we want to convince AAD verifiers that *all* values of a key have been returned via a lookup proof. In Section 5, we describe a non-black-box modification of our AAS into an AAD.

Our first observation is that a *bilinear accumulator* (see Section 2.1) is already an AAS, albeit an expensive one. Specifically, updating the set and computing (non)membership proofs and append-only proofs takes time linear in the size of the set, which is prohibitive. Our work reduces these times to polylogarithmic, but at the cost of increasing proof sizes from constant to polylogarithmic in the size of the set. First, we introduce *bilinear trees*, a hierarchical way to precompute all membership proofs in a bilinear accumulator in quasilinear time (instead of quadratic). Second, instead of "accumulating" the elements directly, we build a "sparse" prefix tree (or trie) over all elements and accumulate the tree itself. Then, we precompute non-membership proofs for all prefixes at the *frontier* of this tree (see Figure 2) in quasilinear time. As a result, non-membership of an element is reduced to non-membership of one of its prefixes. (This frontier technique was originally proposed in [70].) Finally, we use classic amortization techniques [80, 81] to append in polylogarithmic time and to precompute append-only proofs between any version i and j of the set.

1.1 Related Work

The key difference between AADs and previous work [8, 23, 55, 59, 65, 88, 95, 104] is that we offer succinct proofs for everything while only relying on a single, untrusted log server. In contrast, previous work either has large proofs [59, 65], requires users to "collectively" verify the log [88, 104] (which assumes enough honest users and can make detection slow), or makes some kind of trust assumption about one or more actors [8, 55, 59, 95]. On the other hand, previous work only relies on collision-resistant hash functions, digital signatures and verifiable random functions (VRFs) [71]. This makes previous work much cheaper computationally, but since bandwidth is more expensive than computation, we believe this is not necessarily the right trade-off. In contrast, our bilinear construction requires trusted setup, large public parameters, and non-standard assumptions. Unlike previous work, our construction is not yet practical due to high append times and memory usage (see Sections 6.1.1 and 6.1.4). Finally, previous work [8, 55, 95, 104] addresses in more depth the subtleties of log-based PKIs, while our work is focused on improving the transparency log primitive itself by providing succinct proofs with no trust assumptions.

CT and ECT. Early work proposes the use of Merkle trees for public-key distribution but does not tackle the append-only problem, only offering succinct lookup proofs [23, 56, 75]. Accumulators are dismissed in [23] due to trusted setup requirements. Certificate Transparency (CT) [59] provides succinct append-only proofs via *history trees* (HTs). Unfortunately, CT does not offer succinct lookup proofs, relying on users to download each update to the log to discover fake PKs, which can be bandwidth-intensive (see Section 6.2.1). Alternatively, users can look up their PKs via one or more CT *monitors*, who download and index the entire log. But this introduces a trust assumption that a user can reach at least one honest CT monitor. Enhanced Certificate Transparency (ECT) addresses CT's shortcomings by combining a lexicographic tree with a chronologic tree, with collective verification by users (as discussed before). Alternatively, ECT can also rely on one or more "public auditors" to verify correspondence of the two trees, but this introduces a trust assumption.

A(RP)KI and PoliCert. Accountable Key Infrastructure (AKI) [55] introduces a checks-and-balances approach where log servers manage a lexicographic tree of certificates and so-called "validators" ensure log servers update their trees in an append-only fashion. Unfortunately, AKI must *"assume a set of entities that do not collude: CAs, public log servers, and validators"* [55]. At the same time, an advantage of AKI is that validators serve as nodes in a gossip protocol, which helps detect forks. ARPKI [8] and PoliCert [95] extend AKI by providing security against attackers controlling $n-1$ out of n actors. Unfortunately, this means ARPKI and PoliCert rely on an anytrust assumption to keep their logs append-only. On the other hand, AKI, ARPKI and PoliCert carefully consider many of the intricacies of PKIs in their design (e.g., certificate policies, browser policies, deployment incentives, interoperability). In addition, ARPKI formally verifies their design.

CONIKS and DTKI. CONIKS [65] uses a hash chain to periodically publish a digest of a lexicographic tree. However, users must collectively verify the tree remains append-only. Specifically, *in every published digest*, each user checks that their own public key has not been removed or maliciously changed. Unfortunately, this process can be bandwidth-intensive (see Section 6.2.1). DTKI [104] observes that relying on a multiplicity of logs (as in CT) creates overhead for domain owners who must check for impersonation in every log. DTKI introduces a *mapping log* that associates sets of domains to their own exclusive transparency log. Unfortunately, like ECT, DTKI also relies on users to collectively verify its many logs. To summarize, while previous work [8, 55, 95, 104] addresses many facets of the transparent PKI problem, it does not address the problem of building a transparency log with succinct proofs without trust assumptions and without collective verification.

Byzantine Fault Tolerance (BFT). If one is willing to move away from the single untrusted server model, then a transparency log could be implemented using BFT protocols [25, 57, 72]. In fact, BFT can trivially keep logs append-only and provide lookup proofs via sorted Merkle trees. With permissioned BFT [25], one must trust that 2/3 of BFT servers are honest. While we are not aware of permissioned implementations, they are worth exploring. For example, in the key transparency setting, it is conceivable that CAs might act as BFT servers. With permissionless BFT [72, 102], one needs a cryptocurrency secured by proof-of-work or proof-of-stake. Examples of this are Namecoin [73], Blockstack [4] and EthIKS [18].

Formalizations. Previous work formalizes Certificate Transparency (CT) [27, 34] and general transparency logs [27]. In contrast, our work formalizes append-only authenticated dictionaries (AAD) and sets (AAS), which can be used as transparency logs. Our AAD abstraction is more expressive than the *dynamic list commitment (DLC)* abstraction introduced in [27]. 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 be easily extended to support non-membership by insertion time. Finally, previous work carefully formalizes proofs of misbehavior for transparency logs [27, 34]. Although misbehavior in AADs is provable too, we do not formalize this in the paper. Neither our work nor previous work adequately models the network connectivity assumptions needed to detect forks in a gossip protocol. Lastly, previous work improves or extends transparency logging in various ways but does not tackle the append-only problem [31, 37, 83].

2 PRELIMINARIES

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 $\log x$ be shorthand for $\log_2 x$. Let $[n] = \{1, 2, \dots, n\}$ and $[i, j] = \{i, i+1, \dots, j-1, j\}$. Let $\mathcal{PP}_q(s) = \langle g^s, g^{s^2}, \dots, g^{s^q} \rangle$ denote q-SDH public parameters. and $\mathcal{PP}_q(s, \tau) = \langle g^s, g^{s^2}, \dots, g^{s^q}, g^{\tau s}, g^{\tau s^2}, \dots, g^{\tau s^q} \rangle$ denote q-PKE public parameters (see Appendix A). Let ε denote the empty string.

Cryptographic assumptions. Our work relies on the use of *pairings* or *bilinear maps* [51, 67]. 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 (although our implementation in Section 6 uses asymmetric pairings). Our assumptions can be re-stated in the setting of (the more efficient) asymmetric (Type II and III) pairings in a straightforward manner. Our AAS and AAD constructions from Sections 3 and 5 are provably secure under the q-SBDH [46] and q-PKE assumptions [47], which we define in Appendix A.

The q-PKE assumption is non-standard and often referred to as "non-falsifiable" in the literature. This terminology can be confusing, since previous, so-called "non-falsifiable" assumptions have been falsified [9]. Naor explored the nuance of these types of assumptions and proposed thinking of them as "not efficiently falsifiable" [74]. For example, to falsify q-PKE one must find an adversary and mathematically prove that *all* extractors fail for it.

2.1 Bilinear Accumulators

A *bilinear accumulator* [33, 76] is a cryptographic commitment to a set $T = \{e_1, e_2, \ldots, e_n\}$, referred to as the *accumulated set.*

Committing to a set. Let $C_T(x) = (x - e_1)(x - e_2) \cdots (x - e_n)$ denote the *characteristic polynomial* of T and s denote a trapdoor that nobody knows. The accumulator $\mathrm{acc}(T)$ of T is computed as $\mathrm{acc}(T) = g^{C_T(s)} = g^{(s-e_1)(s-e_2)\cdots(s-e_n)}$. 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 $\mathcal{PP}_q(s) = \langle g^s, g^{s^2}, \ldots, g^{s^q} \rangle$. This can be done via MPC protocols [20, 21, 54] as detailed in Section 7. Given coefficients $c_0, c_1, \ldots, c_n$ of $C_T(\cdot)$ where $n \leq q$, the accumulator is computed *without knowing* s as follows:

$$\mathrm{acc}(T) = g^{c_0}(g^s)^{c_1}(g^{s^2})^{c_2} \cdots (g^{s^n})^{c_n} = g^{c_0 + c_1 s + c_2 s^2 \cdots c_n s^n} = g^{C_T(s)}$$

In other words, the server computes a *polynomial commitment* [53, 76] to the characteristic polynomial of T. Since the accumulator only supports elements from $\mathbb{F}_p$, we assume a function $\mathcal{H}_{\mathbb{F}} : \mathcal{D} \to \mathbb{F}_p$ that maps elements to be accumulated from any domain $\mathcal{D}$ to values in $\mathbb{F}_p$. If $|\mathcal{D}| > |\mathbb{F}_p|$, then $\mathcal{H}_{\mathbb{F}}$ can be a CRHF.

Membership proofs. A *prover* who has T can convince a *verifier* who has $\mathrm{acc}(T)$ that an *element* e_i is in the set T. The prover simply convinces the verifier that $(x - e_i) \mid C_T(x)$ by presenting a commitment $\pi = g^{q(s)}$ to a quotient polynomial $q(\cdot)$ such that $C_T(x) = (x - e_i)q(x)$. Using a bilinear map, the verifier checks the property above holds at $x = s$, which is secure under q-SDH [53]:

$$e(g, \mathrm{acc}(T)) \stackrel{?}{=} e(\pi, g^s/g^{e_i}) \Leftrightarrow e(g,g)^{C_T(s)} \stackrel{?}{=} e(g,g)^{q(s)(s-e_i)}$$

Subset and disjointness proofs. To prove that $A \subseteq B$, the prover shows that $C_A(x) \mid C_B(x)$. Specifically, the prover presents a commitment $\pi = g^{q(s)}$ of a quotient polynomial $q(\cdot)$ such that $C_B(x) = q(x)C_A(x)$. The verifier checks that $e(g, \mathrm{acc}(B)) = e(\pi, \mathrm{acc}(A))$.

To prove that $A \cap B = \varnothing$, the prover uses the Extended Euclidean Algorithm (EEA) [98] to compute Bézout coefficients $y(\cdot)$ and $z(\cdot)$ such that $y(x)C_A(x) + z(x)C_B(x) = 1$. The proof consists of commitments to the Bézout coefficients $\gamma = g^{y(s)}$ and $\zeta = g^{z(s)}$. The verifier checks that $e(\gamma, \mathrm{acc}(A))e(\zeta, \mathrm{acc}(B)) = e(g,g)$. By setting $B = \{e\}$, we get another type of non-membership proof for $e \notin A$.

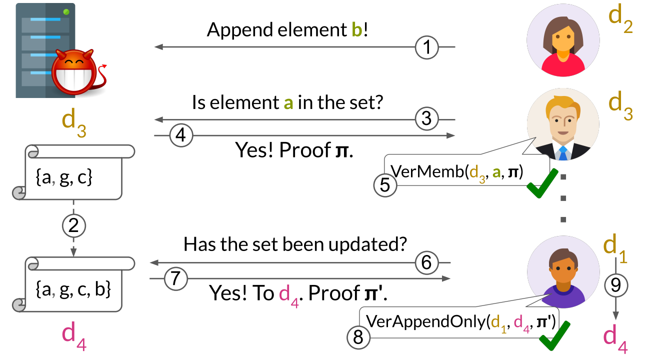


Figure 1: Our model: a single malicious *server* manages a *set* and many *clients* query the set. Clients will not necessarily have the digest of the latest set. The clients can (1) append a new element to the set, (2) query for an element and (3) ask for an updated digest of the set.

Fast Fourier Transform (FFT). We use FFT [99] to speed up polynomial multiplication and division. For polynomials of degree-bound n, we divide and multiply them in $O(n \log n)$ field operations [85]. We interpolate a polynomial from its n roots in $O(n \log^2 n)$ field operations [100]. 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 [98].

3 APPEND-ONLY AUTHENTICATED SETS

We begin by introducing a new primitive called an *append-only authenticated set* (AAS). An AAS can be used for Revocation Transparency (RT) as proposed by Google [58]. In Section 5, we modify our AAS into an *append-only authenticated dictionary* (AAD), which can be used for generalized transparency [36].

Overview. An AAS is a set of *elements* managed by an *untrusted server* and queried by *clients*. The server is the sole author of the AAS: it can append elements on its own and/or accept elements from clients. Clients can check membership of elements in the set (see Steps 3-5 in Figure 1). Clients, also known as *users*, are mutually-distrusting, potentially malicious, and do not have identities (i.e., no authorization or authentication). 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 nor change the element. After each append, the server signs and publishes a new, small-sized *digest* of the updated set (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). 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* $\pi_{i,j}$ 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. Clients will not necessarily have the latest digest of 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 [60], preventing them from seeing each other's appends. To deal with this, clients maintain

a *fork consistent* view [60, 61] of the set by checking append-only proofs. As a consequence, if the server ever withholds an append from one client, that client's digest will forever diverge from other clients' digests. To detect such *forks*, clients can *gossip* [28, 32, 94, 96] with one another about their digests. This is crucial for security in transparency logs.

This model is the same as in history trees (HTs) [29], assuming only a gossip channel and no trusted third parties. It also arises in encrypted web applications [39, 52, 84], Certificate Transparency (CT) [59] and software transparency schemes [38, 78]. Unlike the 2- and 3-party models [6, 82, 86], there is no *trusted source* that signs appends in this model. A trusted source trivially solves the AAS/AAD problem as it can simply vouch for the data structure's append-only property with a digital signature. Unfortunately, this kind of solution is useless for transparency logs [59, 65, 88], which lack trusted parties.

Server-side API. The untrusted server implements:

$\mathsf{Setup}(1^\lambda, \beta) \to pp, VK$. Randomized algorithm that returns public parameters pp used by the server 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., $n \le \beta$).

$\mathsf{Append}(pp, \mathcal{S}_i, d_i, k) \to \mathcal{S}_{i+1}, d_{i+1}$. 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., $i+1 \le \beta$). 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.

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. 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 3.1.)

$\mathsf{VerAppendOnly}(VK, d_i, i, d_j, j, \pi_{i,j}) \to \{T, F\}$. Deterministic algorithm that ensures a set remains append-only. Verifies that $\pi_{i,j}$ correctly proves that the set with digest d_i is a subset of the set with digest d_j. 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 or MPC protocol runs $\mathsf{Setup}(\cdot)$ and forgets the trapdoor (see Section 7). 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 to all clients.

Then, the server broadcasts the initial digest d_0 of the empty set $\mathcal{S}_0$ to its many clients. Clients can concurrently start appending elements using $\mathsf{Append}(\cdot)$ calls. If the server is honest, it serializes $\mathsf{Append}(\cdot)$ calls. Eventually, the server returns a new digest d_i to clients along with an append-only proof $\pi_{0,i}$ computed using $\mathsf{ProveAppendOnly}(\cdot)$. Some clients might be offline but eventually they will receive either d_i or a newer $d_j, j > i$. Importantly, whenever clients transition from version i to j, they 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 can prove it is a superset of any previous digest d_i using $\mathsf{ProveAppendOnly}(\cdot)$.

3.1 AAS Correctness and Security Definitions

We first introduce some helpful notation for our correctness definitions. Consider an ordered sequence of n appends $(k_i)_{i\in[n]}$. 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 $\mathcal{S}', d' \leftarrow \mathsf{Append}(pp, \mathcal{S}_{n-1}, d_{n-1}, k_n)$, $\mathcal{S}_{n-1}, d_{n-1} \leftarrow \mathsf{Append}(pp, \mathcal{S}_{n-2}, d_{n-2}, k_{n-1}), \dots, \mathcal{S}_1, d_1 \leftarrow \mathsf{Append}(pp, \mathcal{S}, d, k_1)$. Finally, let $\mathcal{S}_0$ denote an empty AAS with empty digest d_0.

Definition 3.1 (Append-only Authenticated Set)**.** (Setup, Append, $\mathsf{ProveMemb}$, $\mathsf{ProveAppendOnly}$, $\mathsf{VerMemb}$, $\mathsf{VerAppendOnly}$) is a secure append-only authenticated set (AAS) if $\exists$ a negligible function ε, $\forall$ security parameters λ, $\forall$ upper-bounds $\beta = \mathrm{poly}(\lambda)$ and $\forall n \le \beta$ it satisfies the following properties:

Membership correctness. $\forall$ ordered sequences of appends $(k_i)_{i\in[n]}$, for all elements k, where $b = 1$ if $k \in (k_i)_{i\in[n]}$ and $b = 0$ otherwise,

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

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 appending elements.

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

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

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 of appends $(k_i)_{i\in[n]}$ where $n \ge 2$,

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

Append-only security. $\forall$ adversaries A running in time poly(λ),

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

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

Fork consistency. $\forall$ adversaries A running in time poly(λ),

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

Observation: This is our own version of fork consistency that captures what is known in the literature about fork consistency [29, 61]. Specifically, it allows a server to fork the set at version i by presenting two different digests d_i and d_i' and prevents the server from forging append-only proofs that "join" the two forks into some common digest d_j at a later version j.

4 AAS FROM ACCUMULATORS

This section presents our accumulator-based AAS construction. We focus on bilinear accumulators here and discuss how our construction would benefit from RSA accumulators in Section 7. We give a more formal algorithmic description in Appendix B.

As mentioned in Section 1, a bilinear accumulator over n elements is already an AAS, albeit an inefficient one. Specifically, proving (non)membership in a bilinear accumulator requires an $O(n)$ time polynomial division. As a consequence, precomputing all n membership proofs (naively) takes $O(n^2)$ time, which is prohibitive for most use cases. Even worse, for non-membership, one must precompute proofs for all possible missing elements, of which there are exponentially many (in the security parameter λ). 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 2b). 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 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 $S = \{e_1, \ldots, e_n\}$ (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 $T(n) = 2T(n/2) + O(n \log n) = O(n \log^2 n)$ where $O(n \log n)$ is the time to multiply the characteristic polynomials of two sets of size n in the tree. 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 2.1). Specifically, the proof contains the accumulators along the path from e_i's leaf to the root,

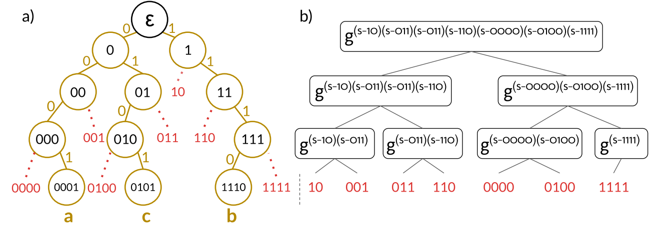


Figure 2: On the left side, we depict a trie over set $S = \{a, b, c\}$. Each element is mapped to a unique path of length 4 in the trie. Nodes that are not in the trie but are at its *frontier* are depicted in red. On the right side, we depict a *bilinear frontier tree* (BFT) corresponding to the set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

as well as the accumulators of all sibling nodes along this path (see Figure 2b). The client 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 2λ bits (or is made so using a CRHF) and can be mapped to an element in $\mathbb{F}_p$ (which is also of size $\approx 2\lambda$ bits) and thus can be accumulated. 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 $2\lambda+1$ 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

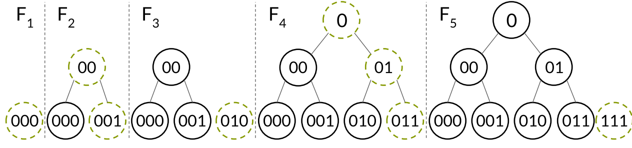


Figure 3: A forest starting empty and going through a sequence of five appends. A forest only has trees of exact size 2^j for distinct j's. A forest of n leaves has *at most* $\log n$ trees.

before, the client 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 might exist other elements e'' who share ρ as a prefix. As a result, the non-membership proof for e' could be "reused" as a non-membership proof for e''. This is best illustrated in Figure 2a using our previous example where $S = \{a, b, c\}$. Consider elements $d = \underline{011}1$ and $f = \underline{011}0$ that are not in S. To prove non-membership for either element, it suffices to prove the same statement: $\underline{011} \notin P(S)$. Thus, if we can identify all such *shared prefixes*, we can use them to prove the non-membership of (exponentially) many elements. (This technique is also used in Micali et al's zero-knowledge sets [70].)

To do this, we insert all elements from S in a trie as depicted in Figure 2a. Next, we keep track of the prefixes at the "frontier" of the trie depicted in red in Figure 2a. We immediately notice that to prove non-membership of any element, 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 as a prefix. Thus, we formally define the *frontier* of S as:

$$F(S) = \{\rho \in \{0,1\}^{\le 2\lambda} : \rho \notin P(S) \wedge \mathsf{parent}(\rho) \in P(S)\},$$

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

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 $\rho \in F(S)$. 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 2.1 (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.

Static AAS construction. Combining all the above techniques, we obtain a *static* AAS that does not support updates efficiently (nor append-only proofs). This 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 is just the root accumulator of the BPT.

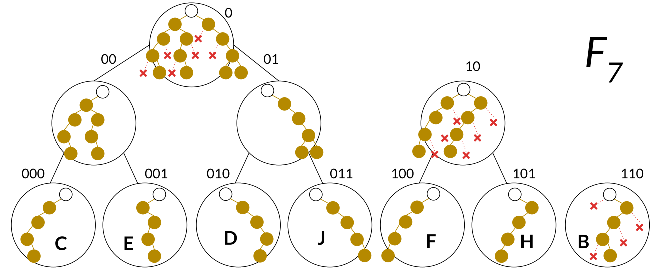


Figure 4: A dynamic AAS with $\lambda = 2$ for set $\{B, C, D, E, F, H, J\}$. Our AAS is a forest of BPTs with corresponding BFTs. Each node stores a BPT accumulator (and subset proof), depicted as a trie, in yellow. Root nodes store a BFT, depicted as the missing red nodes.

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 [80] also used in [87].

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 no elements in the AAS. When the first element e_1 is appended, we build its *tree-pair*: a BPT over the set $\{e_1\}$, its BFT and a disjointness proof. When the second element e_2 is appended, we "merge": we build a size-2 tree-pair over $\{e_1, e_2\}$. The rule is we always merge equal-sized tree-pairs. When e_3 is appended, we cannot merge it because there's no other tree-pair of size 1. Instead, we create a tree-pair over $\{e_1\}$. In general, after $2^\ell - 1$ appends, we end up with ℓ separate tree-pairs corresponding to sets of elements $S_1, \ldots, S_\ell$. The final set is $S = \bigcup_{j=1}^{\ell} S_j$ where $|S_j| = 2^j$. The evolution of such a forest is depicted in Figure 3 and the final data structure can be seen in Figure 4.

Let us analyze the time to merge two size-n tree-pairs for S_1 and S_2 into a size-$2n$ tree-pair for $S = S_1 \cup S_2$. To compute S's BPT, we need to (i) compute its root accumulator, (ii) set its children to the "old" root accumulators 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 $O(\lambda n \log^2 n)$ time. Finally, we can compute S's BFT from scratch in $O(\lambda n \log^2 n)$ time.

To analyze the append time, consider the time $T(n)$ to create an AAS over a set S with $n = 2^\ell$ 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 one append is $O(\lambda \log^3 n)$ and can be de-amortized into *worst-case* time using generic techniques [80, 81].

The downside of our amortized approach is that proving non-membership becomes slightly more expensive than in the static AAS data structure from above. Specifically, 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 set to the root accumulators of all BPTs and has size $O(\log n)$. We analyze the complexity of our AAS in Appendix D.

Efficient append-only proofs. Our append-only proofs are similar to the ones in history trees [29]. An append-only proof must relate the root BPT accumulator(s) in the old AAS to the root BPT accumulator(s) in the new AAS. We'll refer to these as "old roots" and "new roots" respectively. Specifically, it must show that every old root either (i) became a new root or (ii) has a path to a new root with valid subset proofs at every level. Such a 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 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. To summarize, the append-only proof guarantees that each old root (1) has a valid subset path to a new root or (2) became a new root.

Ensuring fork-consistency. For gossip protocols to work [28, 32], our AAS must be 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 been: the two users had diverging views that were subsequently merged.

To avoid this, we will "Merkle-ize" each BPT using a CRHF $\mathcal{H}$ in the standard manner (i.e., a node hashes its accumulator and its two children's hashes). Our AAS digest is now set to the Merkle roots of all BPTs, which implicitly commit to all root accumulators in the BPTs. As a result, after merging BPTs for elements e_1 and e_2, the Merkle root of the merged BPT will differ based on how appends were ordered: (e_1, e_2), or (e_2, e_1). Thus, violating fork-consistency becomes as hard as finding a collision in $\mathcal{H}$ (see Appendix C).

5 FROM SETS TO DICTIONARIES

In this section, we turn our attention to constructing an append-only authenticated dictionary (AAD). Recall that an AAS stores *elements* and supports (non)membership queries of the form "Is $e \in S$?" In contrast, an AAD stores *key-value pairs* and supports *lookups* of the form "Is V the complete set of values for key k?" In other words, an AAD maps a *key* k to a multiset of *values* V in an append-only fashion. Specifically, once a value has been added to a key, it cannot be removed nor changed. For example, if a key is a domain name and its values are certificates for that domain, then an AAD can be used as a Certificate Transparency (CT) log. In general, keys and values can have any application-specific type, as long as they can be hashed to a bit string.

Our construction has great similarities with the AAS of Section 4. However, the different functionality calls for modifications. Indeed, even the security notion for AADs is different (see Appendix E). Specifically, 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. Next, we describe how we modify our AAS from Section 4 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 2λ bits to 4λ 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 Appendix B. Thus AAD clients now have twice the number of public parameters: $g^{\tau}, (g^{s^i})_{i=0}^{4\lambda+1}$.

Proving lookups. For simplicity, let us restrict ourselves to an AAD of size 2^i (i.e., with just *one* tree-pair). For a key k with no values, a lookup proof is simply a frontier proof for a prefix of k in the BFT, much like a non-membership proof in the AAS (see Section 4). What if k has one or more values? First, the lookup proof contains paths to BPT leaves with k's values (i.e., with elements of the form $k|v$), much like a membership proof in an AAS. But what is to guarantee *completeness* of the response? What if a malicious server leaves out one of the values of key k? (This is important in transparency logs where users look up their own PKs and must receive all of them to detect impersonation attacks.) We use the same frontier technique as in the AAS to convince clients values are not being left out. Specifically, the server 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.

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 4). 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 generalizes to AADs of arbitrary size: the server (i) proves non-membership of k in BPTs with no values for k (via the BFT) and (ii) proves V_i is the complete set of values of k in each remaining BPT i (via the BFT lower frontier technique). In that case, a lookup proof for a key k with a single value v consists of (1) an $O(\log n)$-sized path in some BPT with an $O(\lambda \log n)$-sized frontier proof in its corresponding BFT (to guarantee completeness) and (2) an $O(\log n)$-sized frontier proof for k in all other $O(\log n)$ BFTs, to prove k has no values there.

Smaller lookup proofs. When k has one value, it follows from above that a lookup 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 BFT leaf which contains all lower frontier

prefixes of k. 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 $4\lambda+1$. We remind the reader that clients have enough public parameters $(g^{s^i})_{i=0}^{4\lambda+1}$ to reconstruct the accumulators in these BFT leaves and verify the frontier proof.

Supporting large domains and multisets. To handle keys and values longer than 2λ bits, we store $\mathcal{H}(k)|\mathcal{H}(v)$ in the AAD (rather than $k|v$), where $\mathcal{H}$ is a CRHF 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).

Supporting inclusion proofs. Another useful proof for a transparency log is an *inclusion proof* which only returns *one of the values* of key k (while lookup proofs return *all* values of a key k). For example, in Certificate Transparency (CT), browsers are supposed to verify an inclusion proof of a website's certificate before using it. Our AAD supports inclusion proofs too. They consist of a path to a BPT leaf with the desired key-value pair. Since they do not require frontier proofs, inclusion proofs are only $O(\log n)$-sized.

6 EVALUATION

In this section, we evaluate our AAD (not AAS) construction's proof sizes, append times and memory usage. We find that append times and memory usage are too high for a practical deployment and discuss how they might be improved in future work (see Sections 6.1.1 and 6.1.4). If they are improved, we find AADs can save bandwidth relative to CT and CONIKS and we describe exactly when and how much in Section 6.2.1.

Codebase and testbed. We implemented our *amortized* AAD construction from Section 5 in 5700 lines of C++. Its *worst-case* append time is $O(\lambda n \log^2 n)$ while its amortized append time is $O(\lambda \log^3 n)$. We used Zcash's `libff` [90] as our elliptic curve library with support for a 254-bit Barretto-Naehrig curve with a Type III pairing [7]. We used `libfqfft` [91] to multiply polynomials and `libntl` [92] to divide polynomials and compute GCDs. Our code is available at:

https://github.com/alinush/libaad-ccs2019.

We ran our evaluation in the cloud on Amazon Web Services (AWS) on a r4.16xlarge instance type with 488 GiB of RAM and 64 VCPUs, running Ubuntu 16.04.4 (64-bit). 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."

6.1 Microbenchmarks

6.1.1 *Append times.*

Starting with an empty AAD, we append key-value pairs to it and keep track of the *cumulative average append-time.* Recall that appends are amortized in our construction (but can be de-amortized using known techniques [80, 81]). 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. Batching $k = 2^i$ appends together means we only compute one BFT for the full tree of size k created after inserting the batch. In contrast, without batching, we would compute k BFTs, one for each new forest root created after an append. Figure 5c shows that the average append time is 5.75 seconds with no batching and 0.76 seconds with batch size 8192. (For batch sizes $32, 64, \ldots, 4096$, the average times per append in milliseconds are 3422, 3064, 2644, 2361, 1848, 1548, 1353 and 976 respectively.) These times should increase by around 3.5 seconds if we benchmarked 2^{20} appends.

Speeding up appends. The bottleneck for appends is computing the BFTs. Although we used `libff`'s multi-threaded multi-exponentiation to compute accumulators faster, there are other ways to speed up appends that we have not explored. First, we can parallelize computing (1) the polynomials on the same level in a BFT, (2) the smaller accumulators at lower levels of the BFT, where multi-threaded multi-exponentiation does not help as much and (3) the subset proofs in the forest. Second, we can reuse some of the previously computed accumulators when computing a new BFT. Third, our BPT and BFT constructions require "extractable" counterparts of the accumulators, which almost triple the time to commit to a polynomial. We hope to remove this expensive requirement by proving our construction secure in the generic group model, similar to new SNARK constructions [48]. Finally, techniques for distributed FFT could speed up polynomial operations [103].

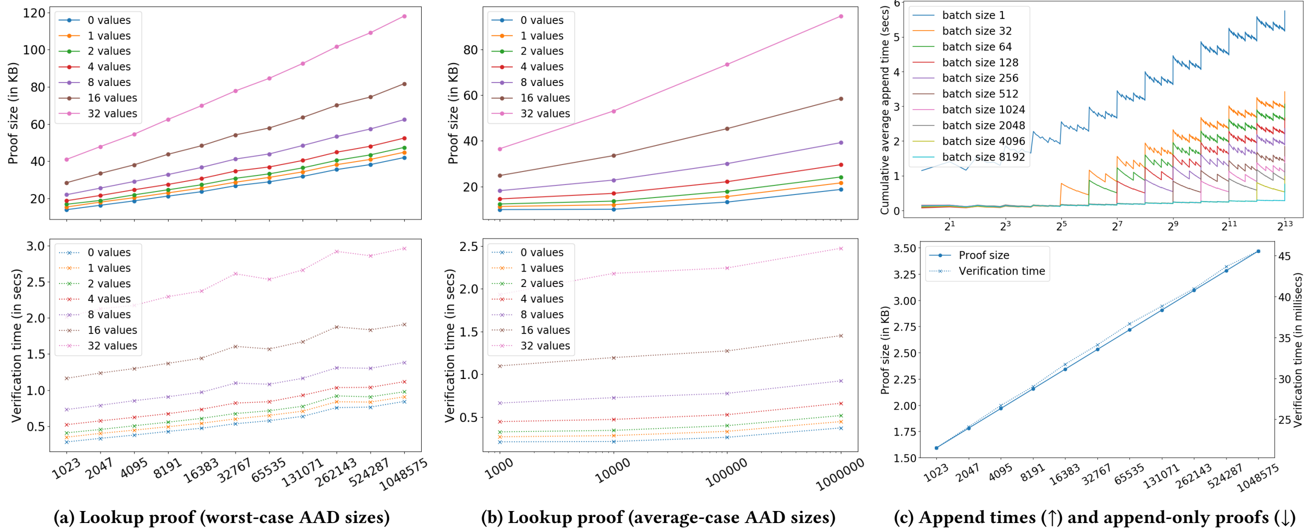
6.1.2 *Lookup proofs.*

We investigate three factors that affect lookup proof size and verification time: (1) the dictionary size, (2) the number of trees in the forest and (3) the number of values of a key. Our benchmark creates AADs of ever-increasing size n. For speed, 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 $\ell \in \{0, 1, 2, 4, 8, 16, 32\}$. (Recall that a key with ℓ values requires ℓ frontier proofs.) To get an average, for every ℓ, we set up 10 different *target* keys so each key has ℓ values. 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 to avoid having all 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 lookup proof, and measure the size and verification time. Then, for each ℓ, we take an average over its 10 target keys. We repeat the experiment for increasing dictionary sizes n and summarize the numbers in Figures 5a and 5b. Proof verification is single-threaded.

Worst-case versus best-case dictionary sizes. Recall that some dictionary sizes are "better" than others because they have fewer trees in the forest. For example, a dictionary of (worst-case) size $2^i - 1$ will have i trees in the forest and thus i BFTs. Thus, a lookup proof must include frontier proofs in all i BFTs. In contrast, a dictionary of size 2^i only has a single tree in the forest, so a lookup proof needs only one frontier proof. Indeed, our evaluation shows that lookup proofs are smaller in AADs of size 10^i (see Figure 5b) compared to $2^i - 1$ (see Figure 5a). For example, for a key with 32 values, the proof averages 95 KiB for size 10^6 and 118 KiB for size $2^{20} - 1$.

6.1.3 *Append-only proofs.*

This benchmark appends random key-value pairs until it reaches a target size $n = 2^{i+1} - 1$. Then, it



(a) Lookup proof (worst-case AAD sizes)

(b) Lookup proof (average-case AAD sizes)

(c) Append times (↑) and append-only proofs (↓)

Figure 5: The x-axes always indicate AAD sizes. Sections 6.1.1 to 6.1.3 explain the experiments. In Figure 5c (↑), "spikes" occur when two trees of size B are merged, which triggers a new BFT computation, where B is the batch size.

measures the append-only proof size (and verification time) between AADs of size n and $m = 2^i - 1$. We benchmarked on $2^i - 1$ AAD sizes to illustrate worst-case $\Theta(i)$ append-only proof sizes. To speed up the benchmark, we randomly pick accumulators in the forest. Append-only proof verification is single-threaded. Our results show append-only proofs are reasonably small and fast to verify (see Figure 5c). For example, the append-only proof between sizes $2^{19} - 1$ and $2^{20} - 1$ is 3.5 KiB and verifies in 45 milliseconds.

6.1.4 Memory usage. Our lookup proof benchmark was the most memory-hungry: it consumed 263 GiB of RAM for AAD size $n = 2^{20} - 1$. In contrast, the append-only proof benchmark consumed only 12.5 GiBs of memory, since it did not need BFTs. As an example, when $n = 2^{20} - 1$, all BFTs combined have no more than $390n$ nodes. Since we are using Type III pairings, each node stores three accumulators (two in $\mathbb{G}_1$ and one in $\mathbb{G}_2$) in 384 bytes (due to `libff`'s 3x overhead). Thus, the BFT accumulators require no more than 147 GiB of RAM. The rest of the overhead comes from our pointer-based BFT implementation and other bookkeeping (e.g., polynomials). The q-PKE public parameters could have added 64 GiBs of RAM, but these two benchmarks did not need them.

Improving memory. A new security proof could eliminate the additional $\mathbb{G}_1$ and $\mathbb{G}_2$ accumulators and reduce BFT memory by 2.66x and the size of the public parameters by 1.33x (see Section 6.1.1). A more efficient representation of group elements than `libff`'s could also reduce BFT memory by 3x. An efficient array-based implementation of BPTs and BFTs could further reduce memory by tens of gigabytes. Finally, the 390x overhead of BFTs can be drastically reduced by carefully grouping upper frontier prefixes together in a BFT leaf, similar to the grouping of lower frontier prefixes from Section 5. However, doing this without increasing the lookup proof size too much remains to be investigated.

6.2 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 and remain to be improved in future work (see Section 6.1.1).

Lookup 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 trees in the forest). On the other hand, append-only proofs in AADs are $O(\log n)$, much smaller than the $O(n)$ in prefix trees. For example, our Golang implementation of prefix trees, shows that the append-only proof between trees of size 2^{19} and 2^{20} is 32 MiB (as opposed to 3.5 KiB in AADs). The proof gets a bit smaller when the size gap between the dictionaries is larger but not by much: 14.6 MiB between 10^5 and 10^6.

Lookup proofs in history trees (HTs) are $O(n)$-sized, compared to $O(\log^2 n)$ in AADs. This is because, to guarantee completeness, the HT proof must consist of all key-value pairs. On the other hand, append-only proofs in AADs are slightly larger than in HTs. While our proofs contain approximately the same number of nodes as in HT proofs, our nodes store two BPT accumulators in $\mathbb{G}_1$ and a subset proof in $\mathbb{G}_2$ (in addition to a Merkle hash). This increases the per-node proof size from 32 bytes to 32 + 64 + 64 = 160 bytes.

6.2.1 When do AADs reduce bandwidth? Asymptotically, AAD proof sizes outperform previous work. But in practice, our evaluation shows AAD proof sizes are still larger than ideal, especially lookup proofs. This begs the question: *In what settings do AADs reduce bandwidth in transparency logs?* We answer this question below while acknowledging that AAD append times and memory

usage are not yet sufficiently fast for a practical deployment (see Section 6.1.1).

Consider a key transparency log with approximately one billion entries (i.e., an entry is a user ID and its PK). If this were a CONIKS log, then each user must check their PK in every digest published by the log server. Let D denote the number of digests published per day by the log server. This means the CONIKS log server will, on average, send 960 · D bytes per day per user (without accounting for the overhead of VRFs [71] in CONIKS). If this were an AAD log, then each user (1) gets the most recent digest via an append-only proof and (2) checks their PK only in this digest via a lookup proof. Let C denote the number of times per day a user checks his PK (and note that, in CONIKS, $C = D$). Since the lookup proof is for the user's PKs not having changed, it only needs to contain frontier proofs. Extrapolating from Figure 5b, such an average-case lookup proof is 40 KiB (in an AAD of size one billion). Similarly, an append-only proof would be 7 KiB. This means the AAD log server will, on average, send $47 \cdot 1024 \cdot C$ bytes per day per user. Thus, AADs are more efficient when $.0199 \cdot D/C > 1$. In other words, AADs will be more bandwidth-efficient in settings where log digests must be published frequently (i.e., D is high) but users check their PK sporadically (i.e., C is low). For example, if $D = 250$ (i.e., a new digest every 6 minutes) and $C = 0.5$ (i.e., users check once every two days), then AADs result in 10x less bandwidth.

What about CT? Recall that CT lacks succinct lookup proofs. As a result, domains usually trust one or more *monitors* to download the log, index it and correctly answer lookup queries. Alternatively, a domain can act as a monitor itself and keep up with every update to the log. We call such domains *monitoring domains*. Currently, CT receives 12.37 certificates per second on average [44], with a mean size of 1.4 KiB each [35]. Thus, a CT log server will, on average, send $12.37 \cdot 1.4 \cdot 1024 = 17,733.63$ bytes per second per monitoring domain. In contrast, AADs require $47 \cdot 1024 \cdot C/86,400 = .557 \cdot C$ bytes per second per monitoring domain. As before, C denotes how many times per day a monitoring domain will check its PK in the log. Thus, AADs are more efficient when $31,837/C > 1$. So even if domains monitor very frequently (e.g., $C = 100$), AADs are more bandwidth efficient. However, we stress that our append times and memory usage must be reduced for a practical deployment to achieve these bandwidth savings (see Sections 6.1.1 and 6.1.4).

7 DISCUSSION

Privacy via VRFs. When a user's identity (e.g., email address) is hashed to determine a path in the tree, the existence of the path leaks that the user is registered. To avoid this, CONIKS proposed using a *verifiable random function (VRF)* [65, 71] to map users to paths in the tree in a private but verifiable manner. We note that our construction is compatible with VRFs as well and can provide the same guarantees. For fairness, our comparison to CONIKS from Section 6.2.1 assumes CONIKS does not use VRFs.

Constant-sized digests. Digests in our constructions are $O(\log n)$-sized where n is the size of the set (or dictionary). We can make the digest constant-sized by concatenating and hashing all Merkle roots. Then, we can include the Merkle roots as part of our append-only and lookup proofs, without increasing our asymptotic costs.

Large, bounded public parameters. Our bilinear-based constructions from Sections 4 and 5 are bounded: they support at most N appends (given $q \approx 4\lambda N$ public parameters). One way to get an unbounded construction might be to use RSA accumulators as explained later in this section. Another way is to simply start a new data structure, when the old one gets "full," similar to existing CT practices [62]. The old digest could be committed in the new data structure to preserve the append-only property and fork consistency. (This will slightly increase our proof sizes for users who are not caught up with the latest digest.)

Trusted setup ceremony. Previous work shows how to securely generate public parameters for QAP-based SNARKs [47, 48] via MPC protocols [20, 21]. For our AAD, we can leverage simplified versions of these protocols, since our public parameters are a "subset" of SNARK parameters. In particular, the protocol from [21] makes participation very easy: it allows any number of players to join, participate and optionally drop out of the protocol. In contrast, the first protocol [20] required a fixed, known-in-advance set of players. For our scheme, we believe tech companies with a demonstrated interest in transparency logs such as Google, Facebook and Apple can be players in the protocol. Furthermore, any other interested parties can be players too, thanks to protocols like [21]. Finally, the practicality of these MPC protocols has already been demonstrated. In 2016, six participants used [20] to generate public parameters for the Sprout version of the Zcash cryptocurrency [50]. Two years later, nearly 200 participants used [21] to generate new public parameters for the newer Sapling version of Zcash.

RSA-based construction. In principle, the bilinear accumulator in our constructions from Sections 4 and 5 could be replaced with other accumulators that support subset proofs and disjointness proofs. Very recent work by Boneh et al [17] introduces new techniques for aggregating non-membership proofs in RSA accumulators. We believe their techniques can be used to create constant-sized disjointness proofs for RSA accumulators. This, in turn, can be used to build an alternative AAD as follows.

Let us assume we have an RSA accumulator over m elements. First, RSA accumulators allow precomputing all *constant-sized* membership proofs in $O(m \log m)$ time [89]. In contrast, our bilinear tree precomputes all logarithmic-sized proofs in $O(m \log^2 m)$ time. As a result, frontier proofs would be constant-sized rather than logarithmic-sized (i.e., the frontier tree corresponding to an RSA accumulator would be "flat"). This decreases our AAD lookup proof size from $O(\log^2 n)$ to $O(\log n)$. This asymptotic improvement should also translate to a concrete improvement in proof sizes. Our memory consumption should also decrease, since BFTs are no longer required. Second, RSA accumulators have constant-sized parameters rather than linear in dictionary size. This requires a simpler trusted setup ceremony [41] and further saves memory on the server. However, unless RSA accumulators can be sped up, it would result in even slower appends, due to more expensive exponentiations. We leave it to future work to instantiate this RSA construction and prove it secure.

7.1 Constructions from argument systems

A promising direction for future work is to build AADs from generic argument systems [5, 10, 11, 24, 42, 47, 48, 69, 101]. Such AAD

constructions would also require non-standard assumptions [43], possibly different than q-PKE (e.g., random oracle model, generic group model, etc.). Depending on the argument system, they might or might not require trusted setup and large public parameters.

A static AAD can be built from an argument system (e.g., a SNARK [42, 48]) as follows. The AAD maintains one unsorted tree U and one sorted tree S whose leaves are sorted by key. The digest of the AAD consists of (1) the Merkle roots $(d(S), d(U))$ of S and U and (2) a SNARK *proof of "correspondence"* $\pi(S, U)$ between S and U. This proof shows that S's leaves are the same as U's *but in a different, sorted by key, order*. The SNARK circuit takes as input U's leaves and S's leaves, hashes them to obtain $d(U)$ and $d(S)$ and checks that S's leaves are sorted by key.

Now, given a digest $(d(S), d(U), \pi(S, U))$, lookups can be efficiently proven using Merkle proofs in the sorted tree S. The append-only property of two digests $(d(S), d(U), \pi(S, U))$ and $(d(S'), d(U'), \pi(S', U'))$ can be efficiently proven using a history tree append-only proof between $d(U)$ and $d(U')$. This proves U is a subset of U' and, crucially, it also proves that S is a subset of S', since the SNARK $\pi(S, U)$ proves that S "corresponds" to U and S' to U'. Unfortunately, updates would invalidate the SNARK proof and take time at least linear in the dictionary size to recompute it. However, we can apply the same Overmars technique [80, 81] to make updates polylogarithmic time. (This would now require a family of circuits, one for each size 2^i of the trees.)

This approach would result in much shorter lookup proofs while maintaining the same efficiency for append-only proofs, since state-of-the-art SNARKs have proofs consisting of just 3 group elements [48]. On the other hand, this approach might need more public parameters and could have slower appends. This is because, even with SNARK-friendly hashes (e.g., Ajtai-based [12], MiMC [3] or Jubjub [105]), we estimate the number of multiplication gates for hashing trees of size 2^{20} to be in the billions. (And we are not accounting for the gates that verify tree S is sorted.) In contrast, the degrees of the polynomials in our bilinear-based constructions are only in the hundreds of millions for dictionaries of size 2^{20}.

Nonetheless, optimizing such a solution would be interesting future work. For example, replacing SNARKs with STARKs [10] would eliminate the large public parameters and the trusted setup, at the cost of larger append-only proofs. This may well be worth it if the proof size and prover time are not too large. Other argument systems such as Hyrax [101], Ligero [5] and Aurora [11] could achieve the same result. Unfortunately, Aurora and Ligero would increase the append-only proof verification time to linear, which could be prohibitive. Bulletproofs [24] would further increase this verification time to quasilinear. Hyrax can make this time sublinear if the circuit is sufficiently parallel or has "a wiring pattern [that] satisfies a technical regularity condition" [101].

Recursively-composable arguments. Another interesting approach is to obtain AADs from recursively-composable SNARKs [12, 14]. Such SNARKs could structure the verification of the append-only property recursively so that circuits need not operate on the entire dictionary, thus lowering overheads. We are aware of concurrent work that explores this approach, but unfortunately it is not peer-reviewed nor published in an online archive. While such an approach could be very promising, currently implemented systems operate at the 80-bit security level. This is because increasing the security of the elliptic curves used in recursive SNARK constructions is costly, since they have low embedding degree [12]. In contrast, our implementation is 100-bit-secure after accounting for recent progress on computing discrete logs [66] and our q-SDH assumption with $q = 2^{20}$ [16]. We can increase this to 118 bits, with no loss in performance, by adopting 128-bit-secure BLS12-381 curves [19].

8 CONCLUSION

In this work, we introduced the first append-only authenticated dictionary (AAD) that achieves polylogarithmic proof sizes and append times. Unlike previous work, our construction only assumes a single fully-malicious server and does not rely on users to "collectively" verify the dictionary. Our analysis shows that AADs can reduce the bandwidth in current CT logs and in CONIKS logs that publish digests much more frequently than users check their PK in the log. However, as our evaluation shows, AADs are not yet practical enough for deployment, particularly because they have high append times and memory usage. We hope future work can overcome this by optimizing the construction, the implementation or both. Finally, we also introduced the first efficient append-only authenticated set (AAS), which can be used to implement Google's Revocation Transparency (RT) [58].

Open problems. We identify two interesting directions for future work. First, can we build efficient AADs with polylogarithmic proof sizes from standard assumptions, such as the existence of CRHFs? If not, what are the lower bounds? Second, can we obtain "zero-knowledge" AADs which leak nothing during queries?

ACKNOWLEDGMENTS

We would like to thank Marten van Dijk for suggesting the "sparse" prefix tree approach and Madars Virza for productive conversations that helped steer this work. We also thank the anonymous reviewers for their useful feedback. This research was supported in part from USA NSF under CNS grants 1413920, 1718782, 1514261, 1652259, by DARPA & SPAWAR under grant N66001-15-C-4066, by HK RGC under grant ECS-26208318, and by a NIST grant.

REFERENCES

[1] Heather Adkins. 2011. 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.

[2] Mustafa Al-Bassam and Sarah Meiklejohn. 2018. Contour: A Practical System for Binary Transparency. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*.

[3] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. 2016. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In *ASIACRYPT'16*.

[4] Muneeb Ali, Jude Nelson, Ryan Shea, and Michael J. Freedman. 2016. Blockstack: A Global Naming and Storage System Secured by Blockchains. In *USENIX ATC'16*.

[5] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. In *ACM CCS'17*.

[6] Aris Anagnostopoulos, Michael T. Goodrich, and Roberto Tamassia. 2001. Persistent Authenticated Dictionaries and Their Applications. In *Information Security*.

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

[8] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. 2014. ARPKI: Attack Resilient Public-Key Infrastructure. In *ACM CCS'14*.

[9] Mihir Bellare and Adriana Palacio. 2004. The Knowledge-of-Exponent Assumptions and 3-Round Zero-Knowledge Protocols. In *CRYPTO'04*.

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

[11] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. 2019. Aurora: Transparent Succinct Arguments for R1CS. In *EUROCRYPT'19*.

[12] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2017. Scalable Zero Knowledge Via Cycles of Elliptic Curves. *Algorithmica* 79, 4 (01 Dec 2017).

[13] Josh Benaloh and Michael de Mare. 1994. One-Way Accumulators: A Decentralized Alternative to Digital Signatures. In *EUROCRYPT'93*.

[14] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2013. Recursive Composition and Bootstrapping for SNARKS and Proof-carrying Data. In *STOC'13*.

[15] Nir Bitansky, Ran Canetti, Omer Paneth, and Alon Rosen. 2014. On the Existence of Extractable One-way Functions. In *STOC'14*.

[16] Dan Boneh and Xavier Boyen. 2008. Short signatures without random oracles and the SDH assumption in bilinear groups. *Journal of Cryptology* 21, 2 (2008).

[17] Dan Boneh, Benedikt Bünz, and Ben Fisch. 2018. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. Cryptology ePrint Archive, Report 2018/1188. https://eprint.iacr.org/2018/1188.

[18] Joseph Bonneau. 2016. EthIKS: Using Ethereum to audit a CONIKS key transparency log. BITCOIN'16.

[19] Sean Bowe. 2017. Switch from BN254 to BLS12-381. https://github.com/zcash/zcash/issues/2502. Accessed: 2019-02-03.

[20] Sean Bowe, Ariel Gabizon, and Matthew D. Green. 2019. A Multi-party Protocol for Constructing the Public Parameters of the Pinocchio zk-SNARK. In *Financial Cryptography '19*.

[21] Sean Bowe, Ariel Gabizon, and Ian Miers. 2017. Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model. Cryptology ePrint Archive, Report 2017/1050. https://eprint.iacr.org/2017/1050.

[22] Elette Boyle and Rafael Pass. 2015. Limits of Extractability Assumptions with Distributional Auxiliary Input. In *ASIACRYPT'15*.

[23] Ahto Buldas, Peeter Laud, and Helger Lipmaa. 2000. Accountable Certificate Management using Undeniable Attestations. In *ACM CCS'00*.

[24] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. 2018. Bulletproofs: Short Proofs for Confidential Transactions and More. In *IEEE S&P'18*.

[25] M. Castro and B. Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *TOCS* 20, 4 (2002).

[26] Melissa Chase, Apoorvaa Deshpande, and Esha Ghosh. 2018. Privacy Preserving Verifiable Key Directories. Cryptology ePrint Archive, Report 2018/607. https://eprint.iacr.org/2018/607.

[27] Melissa Chase and Sarah Meiklejohn. 2016. Transparency Overlays and Applications. In *ACM CCS'16*.

[28] Laurent Chuat, Pawel Szalachowski, Adrian Perrig, Ben Laurie, and Eran Messeri. 2015. Efficient gossip protocols for verifying the consistency of Certificate logs. In *IEEE CNS'15*.

[29] Scott A. Crosby and Dan S. Wallach. 2009. Efficient Data Structures for Tamper-evident Logging. In *USENIX Security '09*.

[30] Scott A. Crosby and Dan S. Wallach. 2011. Authenticated Dictionaries: Real-World Costs and Trade-Offs. *ACM Transactions on Information and System Security* 14, 2, Article 17 (Sept. 2011).

[31] Rasmus Dahlberg and Tobias Pulls. 2018. Verifiable Light-Weight Monitoring for Certificate Transparency Logs. In *NordSec 2018: Secure IT Systems*.

[32] Rasmus Dahlberg, Tobias Pulls, Jonathan Vestin, Toke Høiland-Jørgensen, and Andreas Kassler. 2018. Aggregation-Based Gossip for Certificate Transparency. *CoRR* abs/1806.08817 (2018). arXiv:1806.08817 http://arxiv.org/abs/1806.08817

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

[34] Benjamin Dowling, Felix Günther, Udyani Herath, and Douglas Stebila. 2016. Secure Logging Schemes and Certificate Transparency. In *ESORICS'16*.

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

[36] Adam Eijdenberg, Ben Laurie, and Al Cutter. 2016. Verifiable Data Structures. https://github.com/google/trillian/blob/master/docs/papers/VerifiableDataStructures.pdf. Accessed: 2018-04-12.

[37] Saba Eskandarian, Eran Messeri, Joseph Bonneau, and Dan Boneh. 2017. Certificate Transparency with Privacy. *PoPETs* 2017, 4 (2017).

[38] Sascha Fahl, Sergej Dechand, Henning Perl, Felix Fischer, Jaromir Smrcek, and Matthew Smith. 2014. Hey, NSA: Stay Away from My Market! Future Proofing App Markets Against Powerful Attackers. In *ACM CCS'14*.

[39] Ariel J. Feldman, Aaron Blankstein, Michael J. Freedman, and Edward W. Felten. 2012. Social Networking with Frientegrity: Privacy and Integrity with an Untrusted Provider. In *USENIX Security '12*.

[40] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. 2010. SPORC: Group Collaboration Using Untrusted Cloud Resources. In *OSDI'10*.

[41] Tore Kasper Frederiksen, Yehuda Lindell, Valery Osheter, and Benny Pinkas. 2018. Fast Distributed RSA Key Generation for Semi-honest and Malicious Adversaries. In *CRYPTO'18*.

[42] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. 2013. Quadratic Span Programs and Succinct NIZKs without PCPs. In *EUROCRYPT'13*.

[43] Craig Gentry and Daniel Wichs. 2011. Separating Succinct Non-interactive Arguments from All Falsifiable Assumptions. In *STOC'11*.

[44] Google. 2016. HTTPS encryption on the web: Certificate transparency. https://transparencyreport.google.com/https/certificates. Accessed: 2018-04-12.

[45] Google. 2016. Trillian: General Transparency. https://github.com/google/trillian. Accessed: 2018-04-12.

[46] Vipul Goyal. 2007. Reducing Trust in the PKG in Identity Based Cryptosystems. In *CRYPTO'07*.

[47] Jens Groth. 2010. Short Pairing-Based Non-interactive Zero-Knowledge Arguments. In *ASIACRYPT'10*.

[48] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *EUROCRYPT'16*.

[49] Benjamin Hof and Georg Carle. 2017. Software Distribution Transparency and Auditability. *CoRR* abs/1711.07278 (2017). arXiv:1711.07278 http://arxiv.org/abs/1711.07278

[50] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. 2015. Zcash Protocol Specification. https://github.com/zcash/zips/blob/master/protocol/protocol.pdf. Accessed: 2017-11-17.

[51] Antoine Joux. 2000. A One Round Protocol for Tripartite Diffie–Hellman. In *Algorithmic Number Theory*.

[52] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. 2016. Verena: End-to-End Integrity Protection for Web Applications. In *IEEE S&P'16*.

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

[54] Aggelos Kiayias, Ozgur Oksuz, and Qiang Tang. 2015. Distributed Parameter Generation for Bilinear Diffie Hellman Exponentiation and Applications. In *Information Security*.

[55] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perring, Collin Jackson, and Virgil Gligor. 2013. Accountable Key Infrastructure (AKI): A Proposal for a Public-key Validation Infrastructure. In *WWW'13*.

[56] Paul C. Kocher. 1998. On certificate revocation and validation. In *Financial Cryptography '98*.

[57] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 20.

[58] Ben Laurie. 2015. Revocation Transparency. https://www.links.org/files/RevocationTransparency.pdf. Accessed: 2018-07-31.

[59] Ben Laurie, Adam Langley, and Emilia Kasper. 2013. RFC: Certificate Transparency. http://tools.ietf.org/html/rfc6962. Accessed: 2015-5-13.

[60] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. 2004. Secure Untrusted Data Repository (SUNDR). In *OSDI'04*.

[61] Jinyuan Li and David Maziéres. 2007. Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems. In *NSDI'07*.

[62] Vincent Lynch. 2018. Scaling CT Logs: Temporal Sharding. https://www.digicert.com/blog/scaling-certificate-transparency-logs-temporal-sharding/. Accessed: 2019-02-03.

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

[64] Petros Maniatis and Mary Baker. 2003. Authenticated Append-only Skip Lists. *CoRR* cs.CR/0302010 (2003). http://arxiv.org/abs/cs.CR/0302010

[65] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. 2015. Bringing Deployable Key Transparency to End Users. In *USENIX Security '15*.

[66] Alfred Menezes, Palash Sarkar, and Shashank Singh. 2017. Challenges with Assessing the Impact of NFS Advances on the Security of Pairing-Based Cryptography. In *Mycrypt'16*.

[67] Alfred Menezes, Scott Vanstone, and Tatsuaki Okamoto. 1991. Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field. In *STOC'91*.

[68] Ralph C. Merkle. 1982. Method of providing digital signatures.

[69] Silvio Micali. 2000. Computationally Sound Proofs. *SIAM J. Comput.* 30, 4 (2000).

[70] Silvio Micali, Michael Rabin, and Joe Kilian. 2003. Zero-Knowledge Sets. In *FOCS'03*.

[71] Silvio Micali, Salil Vadhan, and Michael Rabin. 1999. Verifiable Random Functions. In *FOCS'99*.

[72] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf. Accessed: 2017-03-08.

[73] Namecoin. 2011. Namecoin. https://namecoin.info/. Accessed: 2015-08-23.

[74] Moni Naor. 2003. On Cryptographic Assumptions and Challenges. In *CRYPTO'03*.
[75] Moni Naor and Kobbi Nissim. 1998. Certificate Revocation and Certificate Update. In *USENIX Security '98*.
[76] Lan Nguyen. 2005. Accumulators from Bilinear Pairings and Applications. In *CT-RSA'05*.
[77] André Niemann and Jacqueline Brendel. 2014. A Survey on CA Compromises.
[78] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. 2017. CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds. In *USENIX Security '17*.
[79] Alina Oprea and Kevin D. Bowers. 2009. Authentic Time-Stamps for Archival Storage. In *ESORICS'09*.
[80] Mark H. Overmars. 1987. *Design of Dynamic Data Structures*.
[81] Mark H. Overmars and Jan van Leeuwen. 1981. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Inform. Process. Lett.* 12, 4 (1981).
[82] Charalampos Papamanthou and Roberto Tamassia. 2007. Time and Space Efficient Algorithms for Two-Party Authenticated Data Structures. In *Information and Communications Security.*
[83] Roel Peeters and Tobias Pulls. 2016. Insynd: Improved Privacy-Preserving Transparency Logging. In *ESORICS'16*.
[84] Raluca Ada Popa, Emily Stark, Jonas Helfer, Steven Valdez, Nickolai Zeldovich, M. Frans Kaashoek, and Hari Balakrishnan. 2014. Building Web Applications on Top of Encrypted Data Using Mylar. In *NSDI'14*.
[85] Franco P. Preparata and Dilip V. Sarwate. 1977. Computational Fourier Transforms Complexity of Over Finite Fields. *Math. Comp.* 31, 139 (1977).
[86] Tobias Pulls and Roel Peeters. 2015. Balloon: A Forward-Secure Append-Only Persistent Authenticated Data Structure. In *ESORICS'15*.
[87] Leonid Reyzin and Sophia Yakoubov. 2016. Efficient Asynchronous Accumulators for Distributed PKI. In *Security and Cryptography for Networks*.
[88] Mark D. Ryan. 2014. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS'14*.
[89] Tomas Sander, Amnon Ta-Shma, and Moti Yung. 2001. Blind, Auditable Membership Proofs. In *Financial Cryptography '01*.
[90] SCIPR Lab. 2016. libff. https://github.com/scipr-lab/libff. Accessed: 2018-07-28.
[91] SCIPR Lab. 2016. libfqfft. https://github.com/scipr-lab/libfqfft. Accessed: 2018-07-28.
[92] Victor Shoup. 2016. libntl. https://www.shoup.net/ntl/. Accessed: 2018-07-28.
[93] Ryan Sleevi. 2017. 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.
[94] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford. 2016. Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning. In *IEEE S&P'16*.
[95] Pawel Szalachowski, Stephanos Matsumoto, and Adrian Perrig. 2014. PoliCert: Secure and Flexible TLS Certificate Management. In *ACM CCS'14*.
[96] Alin Tomescu and Srinivas Devadas. 2017. Catena: Efficient Non-equivocation via Bitcoin. In *IEEE S&P'17*.
[97] Jelle van den Hooff, M Frans Kaashoek, and Nickolai Zeldovich. 2014. VerSum: Verifiable Computations over Large Public Logs. In *ACM CCS'14*.
[98] Joachim von zur Gathen and Jurgen Gerhard. 2013. Fast Euclidean Algorithm. In *Modern Computer Algebra* (3rd ed.). Cambridge University Press, New York, NY, USA, Chapter 11, 313–333.
[99] Joachim von zur Gathen and Jurgen Gerhard. 2013. Fast Multiplication. In *Modern Computer Algebra* (3rd ed.). Cambridge University Press, New York, NY, USA, Chapter 8, 221–254.
[100] Joachim von zur Gathen and Jurgen Gerhard. 2013. Fast polynomial evaluation and interpolation. In *Modern Computer Algebra* (3rd ed.). Cambridge University Press, New York, NY, USA, Chapter 10, 295–310.
[101] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. 2018. Doubly-Efficient zkSNARKs Without Trusted Setup. In *IEEE S&P'18*.
[102] Gavin Wood. 2015. Ethereum: A Secure Decentralised Generalised Transaction Ledger. http://gavwood.com/paper.pdf. Accessed: 2016-05-15.
[103] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. 2018. DIZK: A Distributed Zero Knowledge Proof System. In *USENIX Security '18*.
[104] Jiangshan Yu, Vincent Cheval, and Mark Ryan. 2016. DTKI: A New Formalized PKI with Verifiable Trusted Parties. *Comput. J.* 59, 11 (2016).
[105] Zcash. 2017. What is Jubjub. https://z.cash/technology/jubjub/. Accessed: 2019-02-03.

A CRYPTOGRAPHIC ASSUMPTIONS

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

Definition A.2 (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 $\langle g, g^s, g^{s^2}, \dots, g^{s^q}\rangle$ 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.3 (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 $z \in \{0, 1\}^{\mathsf{poly}(\lambda)}$

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

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 [15, 22].

B AAS ALGORITHMS

Here, we give detailed algorithms that implement our AAS from Section 3. Recall 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 operate on this forest, adding new leaves, 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 denoted by $\mathsf{sibling}(w) = v|\overline{b}$, where $\overline{b} = 1 - b$. A *path* from one node v to its ancestor node w is denoted by $\mathsf{path}[v, w] = \{u_1 = v, u_2 = \mathsf{parent}(u_1), \dots, u_\ell = \mathsf{parent}(u_{\ell-1}) = w\}$. The parent node of $v = w|b$ is denoted by $\mathsf{parent}(v) = \mathsf{parent}(w|b) = w$. 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 trees where each tree's size is a *unique* power of two (e.g., see F_5 in Figure 3). The unique tree sizes are maintained by constantly merging trees of the same size. Let $\mathsf{bin}^\beta(x)$ denote the $\lceil \log \beta\rceil$-bit binary expansion of a number x (e.g., $\mathsf{bin}^{14}(6) = 0110$). (Note that $\mathsf{bin}^1(x) = \varepsilon, \forall x$ because $\lceil \log 1\rceil = 0$.) In our AAS, $\mathsf{bin}^\beta(i)$ denotes the ith inserted leaf, where i starts at 0 (e.g., see leaves 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., $\text{roots}(F_5) = \{0, 111\}$ in Figure 3). Let $\text{leaves}(F_i)$ denote all the leaves in the forest (e.g., $\text{leaves}(F_3) = \{000, 001, 010\}$ in Figure 3).

AAS notation. Note that **assert**$(\cdot)$ ensures a condition is true or fails the calling function otherwise. Let $\text{Dom}(f)$ be the domain of a function f. We use $f(x) = \bot$ to indicate $x \notin \text{Dom}(f)$. 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 $\mathbf{a}_{\text{parent}(w)}$. The digest d_i of $\mathcal{S}_i$ maps each root r to its Merkle hash $\mathbf{h}_r$. Every root r stores a disjointness proof $\boldsymbol{\psi}_r$ between its BPT and BFT. For simplicity, we assume server algorithms implicitly parse out the **bolded blue variables** from $\mathcal{S}_i$.

Server algorithms. $\text{Setup}(\cdot)$ generates large enough q-PKE public parameters $\mathcal{PP}_q(s, \tau)$ (see Definition A.3), given an upper bound β on the number of elements. Importantly, the server forgets the trapdoors s and τ used to generate the public parameters. In other words, this is a *trusted setup* phase (see Section 7).

Algorithm 1 Computes public parameters (trusted setup)

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

$\text{Append}(\cdot)$ 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 4 (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. Recall from Section 2.1 that $\mathcal{H}_{\mathbb{F}}$ maps elements to be accumulated to field elements in $\mathbb{F}_p$.

Algorithm 2 Appends a new ith element to the AAS, $i \in [0, \beta - 1]$

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

If k is in the set, $\text{ProveMemb}(\cdot)$ sends a Merkle path to k's leaf in some tree with root r (Lines 3 to 5) via $\text{ProvePath}(\cdot)$ (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 $\text{ProveMemb}(\cdot)$ sends frontier proofs in each BFT in the forest (Lines 6 to 8) via $\text{ProveFrontier}(\cdot)$ (see Algorithm 6).

Algorithm 3 Constructs a (non)membership proof

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

For each root r in F_i, $\text{ProveAppendOnly}(\cdot)$ sends a Merkle path to an ancestor root in F_j, if any. The Merkle path contains subset proofs between all BPT accumulators along the path. It also contains the root BPT accumulators from F_i, which the client will verify against his digest d_i.

Client algorithms. $\text{VerAppendOnly}(\cdot)$ first ensures that d_i and d_j are digests at version i and j respectively (Lines 7 to 8). Before checking subset proofs, $\text{VerAppendOnly}(\cdot)$ validates the old root BPT accumulators in $\pi_{i,j}$ against the Merkle roots in d_i (Lines 11 to 13). Then, checks that each root r from F_i is a subset of some root in F_j by checking subset proofs (Line 16) via $\text{VerPath}(\cdot)$ (see Algorithm 5). $\text{VerAppendOnly}(\cdot)$ enforces fork-consistency implicitly when verifying Merkle hashes.

Algorithm 4 Creates and verifies append-only proofs

1: **function** $\text{ProveAppendOnly}(pp, \mathcal{S}_i, \mathcal{S}_j) \to \pi$
2: $\quad$ **if** $\text{roots}(F_i) \subset \text{roots}(F_j)$ **then return** $\bot$
3: $\quad$ Let $R = \{\text{roots} \in F_i \text{ but } \notin F_j\}$ and $r' \in \text{roots}(F_j)$ be their ancestor root
4: $\quad \pi \leftarrow \text{ProvePath}(\mathcal{S}_j, r, r', \pi), \forall r \in R$ $\quad \pi \leftarrow \text{ProveRootAccs}(\mathcal{S}_i, \pi)$
5: $\quad$ **return** π
6: **function** $\text{VerAppendOnly}(VK, d_i, i, d_j, j, \pi_{i,j}) \to \{T, F\}$
7: $\quad$ **assert** $d_i(r) \neq \bot \Leftrightarrow r \in \text{roots}(F_i)$ ▹ Is valid version i digest?
8: $\quad$ **assert** $d_j(r) \neq \bot \Leftrightarrow r \in \text{roots}(F_j)$ ▹ Is valid version j digest?
9: $\quad$ **assert** $\forall r \in \text{roots}(F_i) \cap \text{roots}(F_j), d_i(r) = d_j(r)$
10: $\quad$ Let $R = \{\text{roots} \in F_i \text{ but } \notin F_j\}$ ▹ i.e., old roots with paths to new root
11: $\quad$ **for all** $r \in \text{roots}(F_i)$ **do** ▹ Check proof gives correct old root accumulators
12: $\quad\quad (\cdot, a_r, \cdot, \cdot) \leftarrow \pi(r)$ $\quad (h_{r|b}, \cdot, \cdot, \cdot) \leftarrow \pi(r|b), \forall b \in \{0, 1\}$
13: $\quad\quad$ **assert** $d_i(r) = \mathcal{H}(r|h_{r|0}|a_r|h_{r|1})$
14: $\quad \forall r \in R$, fetch h_r from $d_i(r)$ and update $\pi_{i,j}(r)$ with it
15: $\quad$ **assert** $\pi_{i,j}$ is well-formed Merkle proof for all roots in R
16: $\quad$ **assert** $\forall r \in R, \text{VerPath}(d_j, r, \pi_{i,j})$

If k is stored at leaf ℓ in the AAS, $\text{VerMemb}(\cdot)$ 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 via $\text{VerPath}(\cdot)$ (see Algorithm 5). If k is not in the AAS, $\text{VerMemb}(\cdot)$ verifies frontier proofs for k in each BFT in the forest via $\text{VerFrontier}(\cdot)$ (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 $\ell, \pi, (\chi_r)_{r \in \text{roots}(F_i)}, (y_r, z_r)_{r \in \text{roots}(F_i)}$
3: **if** $b = 1$ **then** ▷ This is a membership proof being verified
4: $(\cdot, a_\ell, \hat{a}_\ell) \leftarrow \text{Accum}(P(\{k\}))$ $\quad h_\ell \leftarrow \mathcal{H}(\ell|\bot|a_\ell|\bot)$
5: Update $\pi(\ell)$ with h_ℓ and accumulators a_ℓ and $\hat{a}_\ell$
6: **assert** π is well-formed Merkle proof for leaf $\ell \wedge \text{VerPath}(d_i, \ell, \pi)$
7: **else** ▷ This is a non-membership proof being verified
8: **for all** $r \in \text{roots}(F_i)$ **do** ▷ Check BFTs
9: $(\cdot, a_r, \cdot, \cdot) \leftarrow \pi(r) \quad (o_r, \cdot) \leftarrow \chi_r(\varepsilon)$
10: $(h_{r|b}, \cdot, \cdot, \cdot) \leftarrow \pi(r|b), \forall b \in \{0, 1\}$
11: **assert** $d_i(r) = \mathcal{H}(r|h_{r|0}|a_r|h_{r|1})$
12: **assert** $e(a_r, y_r)e(o_r, z_r) = e(g, g) \wedge \text{VerFrontier}(k, \chi_r)$
13: **function** VerPath$(d_k, w, \pi) \to \{T, F\}$
14: Let $r \in \text{roots}(F_k)$ denote the ancestor root of w
15: ▷ Walk path invariant: u is *not* a root node (but parent(u) might be)
16: **for** $u \leftarrow w; u \neq r; u \leftarrow \text{parent}(u)$ **do**
17: $p \leftarrow \text{parent}(u)$ ▷ Check subset proof and extractability (below)
18: $(\cdot, a_u, \hat{a}_u, \pi_u) \leftarrow \pi(u) \quad (\cdot, a_p, \hat{a}_p, \cdot) \leftarrow \pi(p)$
19: **assert** $e(a_u, \pi_u) = e(a_p, g) \wedge e(a_u, g^\tau) = e(\hat{a}_u, g)$
20: **assert** $d_k(r) = \text{MerkleHash}(\pi, r)$ ▷ Invariant: u equals r now
21: **assert** $e(a_r, g^\tau) = e(\hat{a}_r, g)$ ▷ Is root accumulator extractable?
22: **function** MerkleHash$(\pi, w) \to h_w$ ▷ Precondition: π is well-formed proof
23: $(h_w, a_w, \cdot, \cdot) \leftarrow \pi(w)$
24: **if** $h_w = \bot$ **then**
25: **return** $\mathcal{H}(w|\text{MerkleHash}(\pi, w|0)|a_w|\text{MerkleHash}(\pi, w|1))$
26: **else**
27: **return** h_w

Frontier algorithms. CreateFrontier$(\cdot)$ creates a BFT level by level, starting from the leaves, given a set of frontier prefixes F. 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. VerFrontier$(\cdot)$ verifies a frontier proof for one of k's prefixes against a specific root BFT accumulator.

Algorithm 6 Manages BFT of a set

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

Theorem B.1. Under the q-SBDH and q-PKE assumptions, and assuming that $\mathcal{H}$ is a secure CRHF, our construction is a secure AAS as per Definition 3.1.

We prove Theorem B.1 in Appendix C.

C AAS SECURITY PROOFS

Membership and append-only correctness follow from close inspection of the algorithms. Here, we prove our AAS construction offers membership and append-only security, as well as fork-consistency.

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

First, let us focus on the membership proof π, which consists of a path to k's leaf in some BPT of size 2^ℓ leaves. Let $a_0, a_1, \ldots, a_\ell$ be the accumulators along this path (part of π), where a_0 is the leaf accumulator for element k with characteristic polynomial $A_0(x) = \prod_{c \in P(k)}(x - \mathcal{H}_\mathbb{F}(c))$. Let $\pi_0, \ldots, \pi_{\ell-1}$ denote the corresponding subset proofs, such that $e(a_j, g) = e(a_{j-1}, \pi_{j-1}), \forall j \in [\ell]$.

Second, let us consider the other (contradictory) non-membership proof π', which consists of a path to a BFT leaf storing a prefix ρ of k. Let $o_0, o_1, \ldots, o_{\ell'}$ be the frontier accumulators along this path, where o_0 is the leaf accumulator for ρ with characteristic polynomial $O_0(x) = x - \mathcal{H}_\mathbb{F}(\rho)$. Note that this BFT is of size $2^{\ell'}$ and might differ from 2^ℓ, the size of π's BPT. Let a^*_ℓ be the root accumulator for this BFT's corresponding BPT, as contained in π' (see Algorithm 3).

When verifying π and π', both a_ℓ and a^*_ℓ are hashed (together with the two claimed hash values of their children) and the result is checked against the hash from digest d. Since verification of π and π' succeeds, if $a_\ell \neq a^*_\ell$ this would produce a collision in $\mathcal{H}$.

Else, we argue as follows. Each accumulator $a_1, \ldots, a_\ell$ is accompanied by an extractability term $\hat{a}_1, \ldots, \hat{a}_\ell$, which the client checks as $e(a_j, g^\tau) = e(\hat{a}_j, g)$ for $j \in [\ell]$ (see Line 19 in Algorithm 5). Hence, from the q-PKE assumption, it follows that there exists a polynomial time algorithm that, upon receiving the same input as A, outputs polynomials $(A_j(x))_{j \in [\ell]}$ (in coefficient form) such that $g^{A_j(s)} = a_j$ with all but negligible probability. The same holds for all frontier accumulators $o_1, \ldots, o_{\ell'}$ and terms $\hat{o}_1, \ldots, \hat{o}_{\ell'}$ included in π', and let $(O_j(x))_{j \in [\ell']}$ denote their polynomials.

We distinguish two cases and analyze them separately:

(a) $(x - \mathcal{H}_\mathbb{F}(\rho)) \nmid A_\ell(x)$ or $(x - \mathcal{H}_\mathbb{F}(\rho)) \nmid O_{\ell'}(x)$
(b) $(x - \mathcal{H}_\mathbb{F}(\rho)) \mid A_\ell(x)$ and $(x - \mathcal{H}_\mathbb{F}(\rho)) \mid O_{\ell'}(x)$

For case (a), without loss of generality we will focus on the $(x - \mathcal{H}_\mathbb{F}(\rho)) \nmid A_\ell(x)$ sub-case. (The proof for the second sub-case proceeds identically.) Observe that, by construction, $(x - \mathcal{H}_\mathbb{F}(\rho)) \mid A_0(x)$ and, by assumption, $(x - \mathcal{H}_\mathbb{F}(\rho)) \nmid A_\ell(x)$. Thus, there must exist some index $0 < i \leq \ell$ such that $(x - \mathcal{H}_\mathbb{F}(\rho)) \mid A_{i-1}(x)$ and $(x - \mathcal{H}_\mathbb{F}(\rho)) \nmid A_i(x)$. Note that i can be easily deduced given all $(A_j(x))_{j \in [\ell]}$. Therefore, by polynomial division there exist efficiently computable polynomials $q_i(x), q_{i-1}(x)$ and $\kappa \in \mathbb{F}_p$ such that: $A_{i-1}(x) = (x - \mathcal{H}_\mathbb{F}(\rho)) \cdot q_{i-1}(x)$ and $A_i(x) = (x - \mathcal{H}_\mathbb{F}(\rho)) \cdot q_i(x) + \kappa$.

Now, during the verification of the ith subset proof, it holds that:

$$e(a_i, g) = e(a_{i-1}, \pi_{i-1}) \Leftrightarrow$$
$$e(g^{A_i(s)}, g) = e(g^{A_{i-1}(s)}, \pi_{i-1}) \Leftrightarrow$$
$$e(g^{(s-\mathcal{H}_{\mathbb{F}}(\rho))\cdot q_i(s)+\kappa}, g) = e(g^{(s-\mathcal{H}_{\mathbb{F}}(\rho))\cdot q_{i-1}(s)}, \pi_{i-1}) \Leftrightarrow$$
$$e(g^{q_i(s)+\frac{\kappa}{(s-\mathcal{H}_{\mathbb{F}}(\rho))}}, g) = e(g^{q_{i-1}(s)}, \pi_{i-1}) \Leftrightarrow$$
$$e(g^{\frac{\kappa}{(s-\mathcal{H}_{\mathbb{F}}(\rho))}}, g) = e(g^{q_{i-1}(s)}, \pi_{i-1}) \cdot e(g^{-q_i(s)}, g) \Leftrightarrow$$
$$e(g^{\frac{1}{(s-\mathcal{H}_{\mathbb{F}}(\rho))}}, g) = \left[e(g^{q_{i-1}(s)}, \pi_{i-1}) \cdot e(g^{-q_i(s)}, g)\right]^{\kappa^{-1}}.$$

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

In case (b), by assumption, $(x-\mathcal{H}_{\mathbb{F}}(\rho)) \mid A_\ell(x)$ and $(x-\mathcal{H}_{\mathbb{F}}(\rho)) \mid O_{\ell'}(x)$. Therefore, by polynomial division there exist efficiently computable polynomials $q_A(x), q_o(x)$ such that: $A_\ell(x) = (x - \mathcal{H}_{\mathbb{F}}(\rho)) \cdot q_A(x)$ and $O_{\ell'}(x) = (x - \mathcal{H}_{\mathbb{F}}(\rho)) \cdot q_o(x)$. Let $\psi = (y, z)$ be the proof of disjointness from π'. Since ψ verifies against accumulators a_ℓ and $o_{\ell'}$, it holds that:

$$e(a_\ell, y) \cdot e(o_{\ell'}, z) = e(g, g) \Leftrightarrow$$
$$e(g^{A_\ell(s)}, y) \cdot e(g^{O_{\ell'}(s)}, z) = e(g, g) \Leftrightarrow$$
$$e(g^{(s-\mathcal{H}_{\mathbb{F}}(\rho))\cdot q_A(s)}, y) \cdot e(g^{(s-\mathcal{H}_{\mathbb{F}}(\rho))\cdot q_o(s)}, z) = e(g, g) \Leftrightarrow$$
$$e(g^{q_A(s)}, y) \cdot e(g^{q_o(s)}, z) = e(g, g)^{\frac{1}{(s-\mathcal{H}_{\mathbb{F}}(\rho))}}.$$

Thus, the pair $(\mathcal{H}_{\mathbb{F}}(\rho), e(g^{q_A(s)}, y) \cdot e(g^{q_o(s)}, z))$ can again be used to break the q-SBDH assumption.

Append-only security. We can prove append-only security with the same techniques used above. Let ρ be the prefix of k used to prove non-membership w.r.t. the new digest d_j. The membership proof for k w.r.t. the old digest d_i again involves a series of BPT accumulators whose corresponding polynomials can be extracted. By our previous analysis, $(x - \mathcal{H}_{\mathbb{F}}(\rho))$ must divide the polynomial extracted for the BPT root accumulator in d_i, otherwise the q-SBDH assumption can be broken. Continuing on this sequence of subset proofs, the append-only proof "connects" this BPT root accumulator to a BPT root accumulator in d_j. By the same argument, $(x-\mathcal{H}_{\mathbb{F}}(\rho))$ must also divide the polynomial extracted for this BPT root. Since non-membership also verifies, $(x-\mathcal{H}_{\mathbb{F}}(\rho))$ must divide the extracted polynomial for the root BFT accumulator in d_j, or else q-SBDH can be broken. Finally, we apply the same argument as case (b) above, since $(x - \mathcal{H}_{\mathbb{F}}(\rho))$ divides both these polynomials and we have a disjointness proof for their accumulators, again breaking q-SBDH.

Fork consistency. Assume there exists a polynomial-time adversary A that breaks fork consistency, producing digests $d_i \neq d_i'$ with append-only proofs π_i, π_i' to a new digest d_j. Since $d_i \neq d_i'$, there exists a root r such that its hash h_r in d_i differs from its hash h_r' in d_i'. Since d_i and d_i' get "joined" into d_j, let $r^* \neq r$ denote the ancestor root of r in d_j. (Note that $r^* \neq r$, since VerAppendOnly always makes sure that common roots between an old digest and a new digest have the same hash.) Now, note that both proofs π_i, π_i' are Merkle proofs from node r to r^*. Importantly, because every node w is hashed together with its label w (as $h_w = \mathcal{H}(w, h_{w|0}, a_w, h_{w|1})$), the two Merkle proofs take the same path (i.e., path$[r, r^*]$)! In other words, the adversary produced two Merkle proofs that (1) verify against the same digest d_j, (2) take the same path to the same leaf r, but (3) attest for different leaf hashes h_r and h_r'. This breaks Merkle hash tree security and can be used to produce a collision in $\mathcal{H}$.

D AAS ASYMPTOTIC ANALYSIS

Suppose we have a *worst-case* AAS with $n = 2^i - 1$ elements.

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. (BPTs only take up $O(n)$ space.)

Membership proof size. Suppose an element e is in some BPT of the AAS . 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. Since 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, once we exclude common roots between the old and new digest, our proof consists of paths from each old root in the old forest up to a single new root in the new forest. Because the old roots are roots of adjacent trees in the old 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 [29].

E AAD 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. ($\mathcal{K}$ and $\mathcal{V}$ are application-specific; e.g., in software transparency, a key is the software package name and a value is the hash of a specific version of this software package.) 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 (k, v) to a version i dictionary updates the multiset $V = D(k)$ of key k to $V' = V \cup \{v\}$ and increments the dictionary version to $i + 1$.

Server-side API. The untrusted server implements:

Setup$(1^\lambda, \beta) \to pp, VK$. Randomized algorithm that returns public parameters pp used by the server 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 dictionary (i.e., $n \leq \beta$).

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 (k, v) 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}.

ProveLookup$(pp, \mathcal{D}_i, k) \to V, \pi_{k,V}$. Deterministic algorithm that generates a proof $\pi_{k,V}$ that V is the *complete* multiset of values for key k. In particular, when $\mathcal{D}_i(k) = \varnothing$, this is a proof that key

k has no values. Finally, the server cannot construct a fake proof $\pi_{k,V'}$ for the wrong V', including for $V' = \varnothing$.

$\mathsf{ProveAppendOnly}(pp, \mathcal{D}_i, \mathcal{D}_j) \to \pi_{i,j}$. Deterministic algorithm that proves $\mathcal{D}_i$ is a subset of $\mathcal{D}_j$. 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.

Client-side API. Clients implement:

$\mathsf{VerLookup}(VK, d_i, k, V, \pi) \to \{T, F\}$. Deterministic algorithm that verifies proofs returned by $\mathsf{ProveLookup}(\cdot)$ 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.

$\mathsf{VerAppendOnly}(VK, d_i, i, d_j, j, \pi_{i,j}) \to \{T, F\}$. Deterministic algorithm that ensures a dictionary remains append-only. Verifies that $\pi_{i,j}$ correctly proves that the dictionary with digest d_i is a subset of the dictionary with digest d_j. 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 $\mathsf{ProveLookup}(\cdot)$ 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 authenticated dictionary with (empty) digest d_0.

Definition E.1 (Append-only Authenticated Dictionary)**.** (Setup, Append, ProveLookup, ProveAppendOnly, VerLookup, VerAppendOnly) is a secure append-only authenticated dictionary (AAD) if $\exists$ negligible function ε, $\forall$ security parameters λ, $\forall$ upper-bounds $\beta = \mathrm{poly}(\lambda)$ and $\forall n \leq \beta$ it satisfies the following properties:

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

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

Observation: Note that this definition compares the returned multiset V with the "ground truth" in D_n and thus provides lookup 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.

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

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

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 $V' = \varnothing$).

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

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

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

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

Observation: This definition ensures that values can only be added to a key and can never be removed nor changed.

Fork consistency. This definition stays the same as in Section 3.1.