Forget-me-not Trees: Mass-scale Auditable Key Transparency from Hash Functions

Gabriel Kaptchuk[1]

[1]University of Maryland, kaptchuk@umd.edu

Abstract

Modern, deployed key transparency systems rely on auditors to ensure that updates to the set of keys are well-structured, allowing clients to efficiently monitor their own keys. In practice, the server's consistency proofs are very large, requiring computationally powerful auditors; as a result, real-world deployments have very few auditors.

We propose a new key transparency system based on a new data structure called Forget-me-not trees, which is a careful composition of Merkle trees and Bloom filters. The resulting system reduces the size of audit proofs by $\approx 500\times$, from 15MB–30MB down to only 30KB–60KB. Our construction is the first mass-scale auditable key transparency system that relies only on hash functions.

1 Introduction

Modern end-to-end encrypted (E2EE) messaging platforms provide users with strong confidentiality guarantees: the service provider cannot read or manipulate any exchanged messages. These guarantees crucially rely on a robust, verifiable mapping between users' identities (e.g., usernames or phone numbers) and the cryptographic key material used to encrypt messages; if this mapping is not accurate, a user may encrypt messages to an attacker rather than the intended recipient, undermining any cryptographic guarantees.

The main technical tool used to ensure that E2EE providers distribute the correct key material to clients is *key transparency*. First introduced with CONIKS [MBB+15], key transparency systems ensure that users have a consistent view of this mapping. This allows clients to *monitor* their own keys and identify if their authentic key material has been replaced by a malicious key. Because key transparency ensures consistency, a client who monitors their own key material can be assured that all other clients will receive the correct key material.

Scaling key transparency. Significant recent research has focused on constructing deployable key transparency systems that scale to hundreds of millions or billions of daily active users [MBB+15, Bon16, CDGM19, TFZ+22, MKKS+23, LCG+23, LCG+24, HSBB26], resulting in multiple real-world deployments [RWC, LL23, EA23].[1] At a high level, key transparency systems publish a succinct commitment to the mapping from identifiers to cryptographic key material in each epoch and then provide partial openings to this commitment in response to client queries. In this approach,

[1]While we focus primarily on key transparency, these techniques have also found other applications in certificate management, binary transparency, and JavaScript transparency [ABB+26, Gooa, Gooc].

there are two dominant computational costs: (1) auditing the consistency of commitments between epochs, and (2) the monitoring work that each client must do to ensure their key material is correct. In deployed systems (e.g., [ACG+26]), the computational costs associated with auditing epoch consistency are too expensive for clients to perform directly; in 2024, WhatsApp reported that their system processes tens of thousands of key updates per minute with ≈ 200MB consistency proofs that need to be verified each epoch [RWC]. As of early 2026, these consistency proofs are issued every 30 seconds and are between 15MB–30MB each.[2] To bridge this gap, modern systems make use of a small number of computationally powerful *auditors* that verify these proofs and digitally sign the announced state, allowing clients to just monitor their own key material—a process which grows only logarithmically in the number of epochs [CDGM19, MKKS+23].

The main open problem in modern key transparency research is designing key transparency systems that do not require the relatively centralized trust assumptions that come with computationally powerful third-party auditors. Research building on the techniques in deployed systems, which rely exclusively on cryptographic hash functions [MBB+15, MKKS+23, LCG+23, LCG+24, RWC, EA23, Gooa, ABB+26, Gooc], appears to have stalled, falling several orders of magnitude short of allowing lightweight clients to participate in checking consistency between epochs. An alternative, recently studied approach is leveraging more advanced cryptographic machinery, e.g., public key cryptography [TFZ+22] and structured reference strings [HSBB26], in order to make proof of consistency succinct. While these constructions get significantly closer to meeting our goal, they come with significant deployability challenges of their own, including costly lookup proofs [TFZ+22], the use of structured reference strings and long proof times [HSBB26], and the use of assumptions that are not post-quantum secure, resulting in constructions that are unlikely to be deployment candidates.

1.1 Our Contributions

In this work we design the first key transparency system from hash functions that enables mass-scale client auditing of epoch consistency. At the heart of our construction is a new data structure we call Forget-me-not Trees, which marries Bloom filters with Merkle trees. Forget-me-not Trees embed a Bloom filter into each node of a Merkle tree, and insert all the descendant leaves into those filters. The Bloom filters and Merkle trees provide an elegant balance: Bloom filters are excellent at proving *non-membership* and Merkle trees excel at proving *membership*. When combined carefully, this results in (roughly) $\mathcal{O}(\log^2(n))$ size proofs that a particular value is the *latest* update applied to a particular identifier, where n is the total number of updates that have been applied to all identifiers.

Specifically, we make the following contributions:

– *New data structure: forget-me-not tree.* First, we design *forget-me-not trees*, a new data structure that embeds Bloom filters into each node of a Merkle tree. Membership proofs have size $\mathcal{O}(\log(n)\log(m))$, where n is the number of leaves in the tree and m is the size of the Bloom filters. The size of non-membership proofs is probabilistic, with expected size $\mathcal{O}(\log(m) \times \mathsf{max}(\frac{n}{m}, 1))$, which is roughly $\log(m)$ when $n \approx m$.

– *Building key transparency from Forget-me-nots.* We show how to implement key transparency using Forget-me-not trees as an append-only log, combining well-known techniques (e.g., using

[2] Data from October 2025 and February 2026. In March 2026, WhatsApp switched to a new version of their key transparency system, resulting in short-term, non-characteristic behavior.

forests) with several Forget-me-not specific approaches (e.g., including timestamps in node hashes). To look up a key, the server combines a membership proof with multiple sub-tree non-membership proofs to demonstrate that the returned value is the *latest* version in the log. Key monitoring is implemented via non-membership proofs, which are dependent only on the number of updates made to the tree and not the number of epochs.

– *Prototype Implementation and evaluation.* We implement our data structure in Rust[3] and profile its performance. Assuming a key transparency system like WhatsApp's, with billions of active users and tens of thousands of key updates per minute, our evaluation indicates that Forget-me-not Key Transparency significantly outperforms state-of-the-art deployed systems. Our consistency proofs require ≈22–30 bits of communication per updated key value, ≈ 500× less communication than existing deployments, and take ≈ 3.5–8ms to verify on commodity hardware.[4] In exchange for significantly smaller epoch consistency proofs, key lookups grow with the number of leaves added to the tree after the key in question. Concretely, lookups require < 30 bytes per million unique leaves; we note, however, that key lookups are much rarer events. Key monitoring proofs also remain very small; a client can monitor a month's worth of updates to the tree with < 30KB of communication and monitoring once a day requires < 8KB of communication.

Limitations. We note that our approach does have a notable limitation compared to prior work: in exchange for the stronger security guarantees that come with mass-scale auditability, our construction provides a weaker privacy property. Existing work [CDGM19, MKKS+23, LCG+24, TFZ+22, HSBB26] hides which keys are updated from the auditors. In our construction, auditors can (with high confidence) link multiple updates to the same key together, although we can hide the identity of the owner of that key and the new public key. We believe that this tradeoff is a reasonable one in order to radically reconfigure auditing.

2 Background and Related Work

We briefly summarize the most closely related work, focusing on hash-based constructions. Additional relevant citations can be found in [BFM24, MFZE+26].

Key transparency from hash functions. Key transparency was first proposed by Melara et al. in CONIKS [MBB+15]. CONIKS used a sparse Merkle tree to store the current values associated with each identity. In each epoch, the server would publish the root of this Merkle tree and each client was responsible for monitoring their own key material in each epoch. This monitoring burden was prohibitive for clients that go offline, as new epochs occur frequently.

Chase et al. designed SEEMless [CDGM19], a clever approach that reduced the monitoring burden on clients by keeping two trees, one that keeps *all* versions of each identifier and one that keeps all *old* versions of each identifier. Each version of the identifier is stored at a deterministic,

[3] All code was generated using Claude Opus 4.6, with careful testing performed by the authors. While we do not believe that the code is sufficiently well tested for immediate deployment, we believe it is of equivalent quality to most academic code bases that are regularly published by our community.

[4] Verification of epoch consistency proofs is trivially parallelizable, and thus performance will be driven by the verifier's available parallelism. We note that these numbers make some (we believe) reasonable guesses about good parameters for real world deployments, but other parameter choices are viable which may slightly impact performance and modulate tradeoffs.

pseudorandom location in the tree, which provides both predictability and privacy. When doing a lookup, the client can check that the retrieved version of their identifier is the latest by querying both trees, as appropriate. The logarithmic overhead on monitoring keys is from showing that the returned version is the *latest* version in the tree; the construction leverages special "marker" versions of keys when the version is a power of two, non-presence of which convinces the monitor that no version of the identifier between 2^i and 2^{i+1} could be read from the tree. This approach only provides the requisite level of security if the two trees are *append only*, i.e., data cannot be deleted from the tree between epochs. To ensure that the tree is append only, SEEMless introduces *auditors*.

Parakeet [MKKS+23] added a simple way to reduce the server-side costs of SEEMless' basic approach by including the epoch number when a value was last updated. When the server wants to remove old versions of an identifier from the tree, the server marks the version with a "tombstone" and the client checks that only old versions are marked as such. After a suitable amount of time, the server can delete old versions from the tree and auditors ensure that only appropriately marked versions are deleted between epochs.

Most recently, OPTIKS [LCG+24] simplified the constructions of SEEMless and Parakeet by modifying the way that lookups are performed: rather than just reading the latest version of an identifier, the client additionally reads *all* previous versions of the identifier from the tree. To demonstrate that the returned version is the *latest* one, the server also provides a non-membership proof that the next version is not in the tree. Because a malicious server cannot skip version numbers without making later versions unreadable, this streamlines the proof that a version is the latest.

All these techniques suffer from a significant drawback: CONIKS's [MBB+15] monitoring costs are *linear* in the number of epochs and SEEMless [CDGM19], Parakeet [MKKS+23] and OPTIKS [LCG+24] rely on computationally powerful auditors in order to enforce the append-only nature of the Merkle trees they construct. In practice, the basic paradigm proposed by SEEMless/Parakeet/OPTIKS has seen successful deployments [RWC, EA23, Gooa]. While auditing could, in principle, be conducted by anyone, currently the set of auditors is extremely small (e.g., just Cloudflare) due to the high costs associated with auditing.

Merkle^2. A slightly different approach was proposed by Hu et al. in Merkle^2 [HHK+21], which heavily inspired this work. Merkle^2 proposes a modified Merkle tree structure, the leaves of which contain identifier-value pairs in chronological order. In typical chronologically-ordered Merkle trees, there is no way to prevent a malicious server from adding two conflicting leaves to the tree, e.g., $(\mathsf{label}, \mathsf{val}_1)$ and $(\mathsf{label}, \mathsf{val}_2)$, and serving inconsistent views to different clients. To solve this issue, Merkle^2 embeds an index-ordered Merkle tree within each node of the tree that contains all of the tuples contained in descendant leaves; because these trees are index-ordered, only a single value can be embedded for each label. When doing a lookup in Merkle^2, the server proves that the claimed $(\mathsf{label}, \mathsf{val})$ exists in the leaf node and also is in the index-ordered Merkle tree within each node along the paths from the leaf to the root. As a result, only one value can be read for any given label.

While conceptually elegant, Merkle^2 fails to realize the appropriate functionality for key transparency, as there is no way to efficiently monitor keys over many epochs. To rectify this, Merkle^2 [HHK+21] integrated a *signature chain* that ensures only an authorized party can update a particular label. While this effectively prevents malicious writes, it also means that key material can only be updated while the corresponding client still has their old key material. Unfortunately, this assumption is

inconsistent with key transparency.

Looking ahead, our approach takes inspiration from Merkle^2, but replaces the index-ordered trees with Bloom filters, which allow auditors to more efficiently check that two forests have been properly merged and prove non-membership.

Key transparency from public key cryptography. One promising potential direction to avoid the high computational costs associated with auditing key transparency systems is to use public key assumptions to make verification succinct. VeRSA [TFZ+22], by Tyagi et al. was the first work to explore this direction, leveraging RSA accumulators to store the identifier-value pairs. While this provided efficient client verification of look-ups and auditing proofs, the server had to perform (asymptotically) linear work in the number of items in the dictionary and the concrete performance of the design was insufficient for deployment. Recently, Hafezi et al. [HSBB26] proposed IronDict, which leverages a polynomial commitment scheme based on a structured setup assumption to significantly improve the concrete performance of this approach. Their construction has remarkably efficient lookup proofs and auditing, each below 10KB with verification times < 50ms. Moreover, they leverage folding to allow clients to "fast-forward" their monitoring over multiple epochs. The drawback of their proposal is that producing the auditing proofs is extremely computationally expensive (a process that depends on the maximum number of labels in the dictionary) and that it relies on a structured setup assumption. Finally, both these approaches rely on assumptions that are not post-quantum secure, which will make it challenging to convince deployed systems to transition to these approaches.

3 Technical Overview

At the core of our approach is a new variant of a Merkle tree, which we call a Forget-me-not tree. Our approach leverages the observation that Merkle trees and Bloom filters complement one another:

– *Merkle trees are good at proving membership, but bad at proving non-membership*: Merkle trees facilitate logarithmic membership proofs, but proving non-membership requires significant additional work; when done naïvely, this would require a linear scan of the tree. While better techniques exist (e.g., sorting the tree such that particular values can only appear in specific locations or computing a complementary tree that encodes the ranges of values that are not in the tree), they all fundamentally break the ability to merge forests together efficiently.

– *Bloom filters are good at proving non-membership, but bad at proving membership*: because Bloom filters guarantee no false negatives, they are optimized for proving non-membership; if checking a Bloom filter for some element data returns 0, then the caller can be certain that data was never inserted. However, if it returns 1, it could be a *false positive* and there is no guarantee that data was ever added.

Recall that the digest of each internal node of a Merkle tree is computed as

$$\mathsf{digest} = H(\mathsf{digest}_{\text{left}} \| \mathsf{digest}_{\text{right}}),$$

where $\mathsf{digest}_{\text{left}}$ and $\mathsf{digest}_{\text{right}}$ are the digests of its left and right children respectively. Intuitively, Forget-me-not trees augment the ways digests are computed by embedding a hash of a Bloom filter

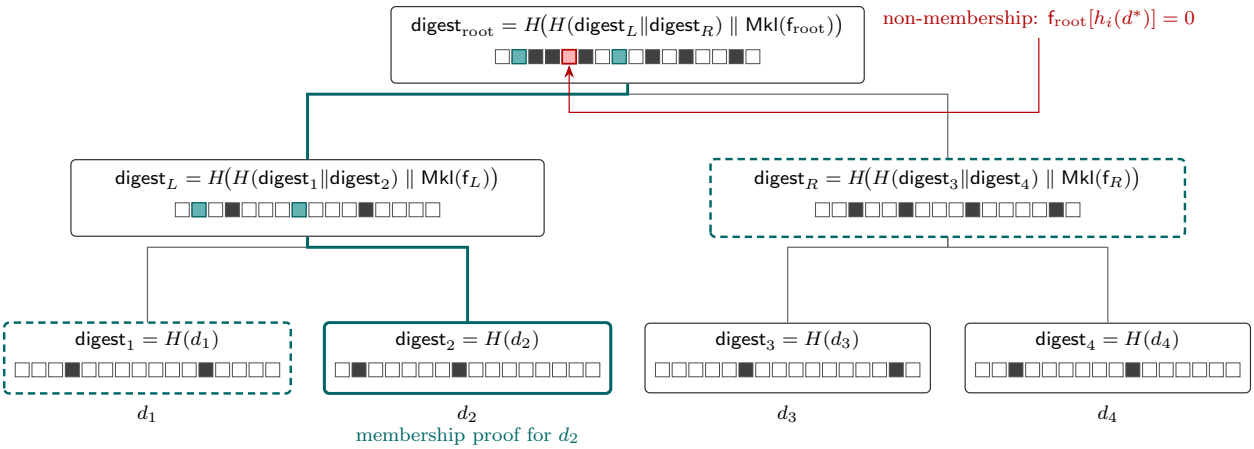


Figure 1: Illustrative Forget-me-not tree with $m = 16$, $k = 2$, and proof annotations. Each node stores a digest and a Merklized Bloom filter (■ / ■ = 1, □ / ■ = 0); parent filters are the bitwise OR of children. Teal highlights a membership proof for d_2: the prover sends sibling digests (dashed) and Merkle proofs that bits $h_1(d_2), h_2(d_2)$ are 1 at each node along the path. Red highlights a potential non-membership for some data d^* for which $\mathsf{f}_{\text{root}}[h_i(d^*)] = 0$ for some $i \in [k]$; in this case, the proof need only demonstrate that the bit highlighted in red is 0. When $h_i(d^*) = 1\ \forall i \in [k]$, the non-membership prover descends the tree recursively until it finds nodes for which $h_i(d^*) = 0$.

into the digest of each node. That is, the digest is computed as

$$\mathsf{digest} = H(H(\mathsf{digest}_{\text{left}} \| \mathsf{digest}_{\text{right}}) \| H(\mathsf{f})),$$

where f is an m bit Bloom filter after adding the data stored in each leaf of the subtree for which the node with label digest is the root. Computing non-membership proofs for some value data becomes direct: simply send f to the verifier and they can locally check that data was never added and that digest is computed honestly. If this check results in a false positive, the prover can simply recurse down the tree until it finds a set of nodes that do not return a false positive and also "cover" all of the leaf nodes in the subtree.

While natural, the naïve version of this approach is flawed, both from a security perspective and an efficiency perspective. Notice that there is nothing ensuring that f was computed correctly. If a malicious party computes the tree, they can simply set f to be 0^m and the Bloom filter check will return 0 for all values of data. Moreover, the size of f needs to be linear in the number of leaves in order to ensure that the Bloom filter has a low false-positive rate, meaning sending f is not logarithmic in the number of leaves.

We fix these two problems as follows:

(1) *Check Bloom filters during membership proofs:* Rather than trying to prevent a malicious party from mis-computing the Bloom filter, we simply ensure that it is impossible to provide a membership proof for any data that has been omitted from the Bloom filter—making omitting bits from the Bloom filter a pointless strategy. We do this by requiring that membership proofs for data demonstrate that data is in the Bloom filter within each node along the path from the leaf to the root. Setting too many bits to 1 in the Bloom filter is similarly pointless, as the core Merkle tree structure will prevent lying about membership.

(2) *Merklize Bloom filters:* Rather than storing the hash of the Bloom filter directly, we Merklize the Bloom filter and store the root of this tree. That is, we compute the node label as

$$\mathsf{digest} = H(H(\mathsf{digest}_{\text{left}} \| \mathsf{digest}_{\text{right}}) \| \mathsf{Merklize}(\mathsf{f}))$$

Recall that checking whether a filter f contains a value data requires reading only k bits of the Bloom filter, where k is the number of hash functions. Similarly, if f does not contain data, reading only a single bit of f is sufficient. Thus, the prover can simulate oracle access to f by providing Merkle proofs that the appropriate bits are set to 0 (non-membership) or 1 (membership).

Putting these techniques together results in membership proofs with size $\mathcal{O}(k \log(n) \log(m))$, as the prover must provide a sibling hash and a Bloom membership proof for each node along the path from the leaf to the root. We visualize this approach with a small tree ($n = 4, m = 16, k = 2$) in Figure 1. The size of non-membership proofs depends on chosen parameters of the Bloom filter. If the number of leaves n is known in advance, then it is possible to set $m > n$ and select a k such that the false positive rate from randomly chosen inputs is arbitrarily small. In such a setting, non-membership proofs will almost never need to recurse, resulting in non-membership proofs of size $\mathcal{O}(\log m)$. If m and k cannot be selected in such a way, however, it may be that recursion is required, which will result in non-membership proofs of expected size $\mathcal{O}(n \log m)$. As we will see below, despite this linear term, this is highly effective in practice.

Merging trees. Just as with Merkle trees, we note that each subtree within a Forget-me-not tree is, itself, a Forget-me-not tree. As such, merging two trees can be done by only considering the data stored in the roots of the two trees without compromising security. That is, given the roots of two Forget-me-not trees, with digest-filter pairs $(\mathsf{digest}_1, \mathsf{f}_1)$ and $(\mathsf{digest}_2, \mathsf{f}_2)$ respectively, a verifier can simply OR f_1 and f_2 together and then compute the digest of a new root node with digest_1 as its left child and digest_2 as its right child.

Approached naïvely, this approach does not seem like it would have good performance, as f needs to be linear in the maximum number of leaf nodes in order to maintain logarithmic-sized non-membership proofs. We observe, however, that in many applications (and in key transparency in particular) the root's filter is usually *very* sparse, allowing for highly efficient compression.

Constructing key transparency. There are three main operations in key transparency systems, listed here from most frequent to least frequent: (1) auditing the bulk addition of new and updated key material, (2) monitoring the log for changes to clients' key material, and (3) performing lookups on peers' key material.

We implement these operations by maintaining a forest of complete Forget-me-not trees, the natural correlate of Merkle forests. Bulk addition of key updates requires the server to organize the updates into a new Forget-me-not forest and then merge the two forests together. Specifically, the leaves embed $(\mathsf{label}, \mathsf{val})$ pairs, where label is used to store the phone number or identifier associated with the user and val holds the public key; the Bloom filter in the leaf is created using only label, but the full tuple is hashed into the digest.

To verify that a merge was performed correctly, auditors need access to the Bloom filters embedded into the roots of the old forest and the (compressed) filters in the roots of the forest representing the updates. Assuming the auditors are stateful, they will already have the filters for the old forest and only need to download the filters for the updates. When compressed, this requires ≈22–30 bits for each update, resulting in $\approx 500\times$ less communication than the state-of-the-art.

To support efficient monitoring and lookups, we timestamp each node in the forest with an epoch by hashing it into the digest; the consistency of these epochs is audited along with the Bloom filters. Monitoring proofs are then a non-membership proof for the key in question for all nodes with an epoch *after* the last time the client performed a monitoring proof. Using epochs means that this proof only depends on the number of keys added since the last time the client monitored their key, not the number of keys since their last update. A query is implemented as a proof that some leaf is the *latest* leaf in the tree with the label in question, i.e., a single membership proof for the leaf and a non-membership proof that covers all nodes with an epoch *after* the epoch embedded into the leaf.

4 Preliminaries

Notation. We use $[\ell]$ to indicate $\{1, 2, \ldots, \ell\}$. Let H be a cryptographic hash function and $\{h_i\}_{i\in[k]}$ the (possibly non-cryptographic) hash functions for a Bloom filter.

4.1 Bloom Filters

A Bloom filter BF is a tuple of algorithms $(\mathsf{Setup}, \mathsf{Add}, \mathsf{Contains}, \mathsf{Merge})$ and is parameterized by a size m and a number of hash functions k. The algorithms have the following interfaces:

- $\mathsf{Setup}() \to \mathsf{f}$ outputs some initial, m bit state f.[5]
- $\mathsf{Add}(\mathsf{f}, \mathsf{data}) \to \mathsf{f}'$ takes in a state f and some arbitrary data data and outputs an updated state f'.
- $\mathsf{Contains}(\mathsf{f}, \mathsf{data}) \to \{0, 1\}$ takes in a state f and some arbitrary data data and outputs either 0 or 1.
- $\mathsf{Merge}(\mathsf{f}_0, \mathsf{f}_1) \to \mathsf{f}$ takes in two states $\mathsf{f}_0, \mathsf{f}_1$ and outputs some new state f.

Bloom filters work as follows: $\mathsf{Setup}()$ outputs the bitstring $\mathsf{f} = 0^m$. $\mathsf{Add}(\mathsf{f}, \mathsf{data})$ sets $\mathsf{f}[h_i(\mathsf{data})] = 1$ for $i \in [k]$. $\mathsf{Contains}(\mathsf{f}, \mathsf{data}) = \bigwedge_{i=1}^{k} \mathsf{f}[h_i(\mathsf{data})]$ and $\mathsf{Merge}(\mathsf{f}_0, \mathsf{f}_1) = \mathsf{f}_0 \vee \mathsf{f}_1$.

No false negatives. An important property of the $\mathsf{Contains}$ interface of Bloom filters is that it has no false negatives. More formally, for all n and all sequences of data inputs $\mathsf{data}_1, \ldots, \mathsf{data}_n$, then for any $\mathsf{data}^* \in \{\mathsf{data}_i\}_{i\in[n]}$

$$\Pr\left[\mathsf{Contains}(\mathsf{f}_n, \mathsf{data}^*) = 1 \;\middle|\; \begin{array}{c} \mathsf{f}_0 \leftarrow \mathsf{Setup}() \\ \mathsf{f}_i \leftarrow \mathsf{Add}(\mathsf{f}_{i-1}, \mathsf{data}_i)\ \forall\ i \in [n] \end{array}\right] = 1$$

Moreover, we have the property that Merge preserves this property; formally, for any pairs of states $\mathsf{f}_0, \mathsf{f}_1$, if $\mathsf{f} \leftarrow \mathsf{Merge}(\mathsf{f}_0, \mathsf{f}_1)$, then for all data, the following holds:

$$\Pr\left[\mathsf{Contains}(\mathsf{f}, \mathsf{data}) = 1 \mid \mathsf{Contains}(\mathsf{f}_0, \mathsf{data}) = 1 \text{ or } \mathsf{Contains}(\mathsf{f}_1, \mathsf{data}) = 1\right] = 1$$

False positive rate. When the hash functions are modeled as independent random functions, the false positive rate, i.e., the probability that $\mathsf{Contains}(\mathsf{f}, \mathsf{data}^*) = 1$ for some random $\mathsf{data}^* \notin$

[5] The k hash functions are often sampled fresh each time Setup is called; in our setting, we assume that these hash functions are sampled once for the scheme and thus are implicit in the above interfaces.

$\{\mathsf{data}_i\}_{i\in[n]}$ after all n of these elements have been added to a filter of size m with k hash functions is

$$\Pr\left[\mathsf{Contains}(\mathsf{f}_n, \mathsf{data}^*) = 1 \;\middle|\; \begin{array}{c} \mathsf{f}_0 \leftarrow \mathsf{Setup}() \\ \mathsf{f}_i \leftarrow \mathsf{Add}(\mathsf{f}_{i-1}, \mathsf{data}_i)\ \forall\ i \in [n] \end{array}\right] = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k.$$

Later in this work, we will primarily consider Bloom filters with $k = 2$; although this is non-optimal for minimizing the false positive rate in general, we show that it outperforms other choices of k in our setting.

Efficiency. Finally, from an efficiency perspective, it is important to note that $\mathsf{Contains}$ can be checked by reading at most k bits from the filter, no matter the choice of m.

4.2 Merkle Trees

Merkle trees are a standard cryptographic data structure in which a digest of a list of inputs is computed by organizing the inputs as the leaves of a binary tree and computing the digest digest of each internal node as $H(\mathsf{digest}_{\text{left}} \| \mathsf{digest}_{\text{right}})$, where $\mathsf{digest}_{\text{left}}$ and $\mathsf{digest}_{\text{right}}$ are the digests of the left and right children respectively; the digests of the leaves are computed directly as the hash of the inputs. Given this structure, a prover can demonstrate that some data is a member of the input (i.e., is stored in a leaf) by providing the digests of the siblings of the nodes that fall along the path from the leaf in which that data is stored to the root. Importantly, the size of this proof is logarithmic in the number of leaves.

In our construction, we interface with Merkle trees using the following interfaces:

- $\mathsf{Merklize}(\mathsf{data}) \to \mathsf{digest}$
- $\mathsf{Prove}(\mathsf{data}, \mathsf{index}, \mathsf{val}) \to \mathsf{digest}, \pi$
- $\mathsf{Verify}(\mathsf{digest}, \pi, \mathsf{index}, \mathsf{val}) \to \{0, 1\}$

We say that a Merkle tree is *binding* in that it should be computationally infeasible for an adversary to produce proofs for $(\mathsf{index}, \mathsf{val}_1)$ and $(\mathsf{index}, \mathsf{val}_2)$ for $\mathsf{val}_1 \neq \mathsf{val}_2$ that both verify with respect to some fixed digest.

Merkle forests or Merkle mountain ranges. When the input for a Merkle tree is non-binary, a standard technique is to compute a set of trees (usually called a forest or a mountain range), rather than a single tree; this technique has been re-discovered multiple times in the cryptographic literature [CW09, Tod13, RY16, GHMR18, Pro]. When storing data in this format, it is also possible to *merge* two forests together very efficiently, requiring combining at most a logarithmic number of roots together. Recently, Christ et al. [BCCK25] proved that this approach is optimal.

4.3 Transparent Dictionaries

There are a number of alternative definitions one could use to define the primitive underlying key transparency, including a Verifiable Key Directory [CDGM19] and a Transparent Dictionary [HSBB26]. These interfaces are functionally similar, but have slight differences to accommodate differences in their underlying definitions. We take the interfaces from Hafezi et al. [HSBB26] as our starting point, as the semantics of Verifiable Key Directories surface a key history as a first-order

concept and our construction doesn't require this concept. Additionally, because we rely only on hash functions, we don't require the Setup interface.

A Transparent Dictionary (TD) [HSBB26] involves three types of parties: a server, auditors, and clients. A TD is the following tuple of algorithms:

- $\mathsf{Server.Init}((\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n)) \to (\mathsf{digest}_0, \mathsf{state}_0)$ initializes the dictionary state and computes an initial digest digest_0.
- $\mathsf{Server.Update}(\mathsf{state}_{i-1}, \Delta) \to (\mathsf{digest}_i, \mathsf{state}_i, \pi_i)$ applies a set of changes Δ, consisting of $(\mathsf{label}, \mathsf{val})$ pairs to obtain a new state state_i, and creates a new digest digest_i along with an invariance proof π_i.
- $\mathsf{Server.Lookup}(\mathsf{label}, \mathsf{state}_i) \to (\mathsf{val}, \pi_{\mathsf{Lookup}})$ returns the value associated with label label at epoch i together with a lookup proof, or $\perp$ if label is not present.
- $\mathsf{Client.VerifyLookup}(\mathsf{digest}, \mathsf{label}, \mathsf{val}, \pi_{\mathsf{Lookup}}) \to \{0, 1\}$ verifies a lookup proof against the epoch digest.
- $\mathsf{Auditor.VerifyInvariance}(\mathsf{digest}_i, \mathsf{digest}_{i+1}, \pi_{i+1}) \to \{0, 1\}$ verifies that the transition from epoch i to $i + 1$ preserves the required invariants.
- $\mathsf{Server.Consistency}(\mathsf{label}, \mathsf{state}_i, \mathsf{state}_j) \to \pi_{\mathsf{consis}}$ returns a proof that the value associated with label remained unchanged throughout epochs i to j (where $i < j$), or $\perp$ if the value changed.
- $\mathsf{Client.VerifyConsistency}(\mathsf{digest}_i, \mathsf{digest}_j, \mathsf{label}, \pi_{\mathsf{consis}}) \to \{0, 1\}$ verifies a consistency proof for label label between epochs i and j.

Completeness [HSBB26]**.** A transparent dictionary construction is said to be *complete* if the following holds with probability 1 over the randomness of the honest server and honest setup:

- *Consistency correctness.* Let label be a label and v be its corresponding value at epoch i. If no tuples with the label label have been included in any of the Server.Update calls that produce the digests $\mathsf{digest}_{i+1}, \ldots, \mathsf{digest}_j$, then:

$$\Pr\big[\mathsf{Client.VerifyConsistency}(\mathsf{digest}_i, \mathsf{digest}_j, \mathsf{label}, \pi_{\mathsf{consis}}) = 1 \;\big|\; \mathsf{Server.Consistency}(\mathsf{label}, \mathsf{state}_i, \mathsf{state}_j) \to \pi_{\mathsf{consis}}\big] = 1$$

  Additionally, for every $i \le t \le j$:

$$\Pr\left[\mathsf{Client.VerifyLookup}(\mathsf{digest}_t, \mathsf{label}, \mathsf{val}, \pi_{\mathsf{Lookup}}) = 1 \wedge \mathsf{val} = v \;\middle|\; \mathsf{Server.Lookup}(\mathsf{label}, \mathsf{state}_t) \to (\mathsf{val}, \pi_{\mathsf{Lookup}})\right] = 1$$

- *Audit correctness.* For all epochs $i \in \{0, 1, 2, \ldots\}$, given some initial state state_i and honestly computed associated digest digest_i, the following holds for all Δ:

$$\Pr\big[\mathsf{Auditor.VerifyInvariance}(\mathsf{digest}_i, \mathsf{digest}_{i+1}, \pi_{i+1}) = 1 \;\big|\; (\mathsf{digest}_{i+1}, \mathsf{state}_{i+1}, \pi_{i+1}) \leftarrow \mathsf{Server.Update}(\mathsf{state}_i, \Delta)\big] = 1.$$

Soundness [HSBB26]**.** Soundness guarantees that, if there is at least one honest auditor in each round and the auditor accepts the new epoch along with its accompanying proof, then the following properties hold, except with negligible probability:

– *Lookup soundness (binding).* Given a commitment digest_i and a single label label, the following probability is negligible:

$$\Pr\left[\begin{array}{c|c} \mathsf{Client.VerifyLookup}(\mathsf{digest}_i, \mathsf{label}, v_0, \pi_0) = 1 \wedge & \\ \mathsf{Client.VerifyLookup}(\mathsf{digest}_i, \mathsf{label}, v_1, \pi_1) = 1 \wedge & (v_0, \pi_0), (v_1, \pi_1) \leftarrow \mathsf{Server} \\ v_0 \neq v_1 & \end{array}\right] < \mathsf{negl}(\lambda)$$

– *Consistency soundness.* For any two epochs i and j, with associated digests digest_i and digest_j the following probability is negligible:

$$\Pr\left[\begin{array}{c|c} \mathsf{Client.VerifyLookup}(\mathsf{digest}_i, \mathsf{label}, v_0, \pi_0) = 1 \wedge & \\ \mathsf{Client.VerifyLookup}(\mathsf{digest}_t, \mathsf{label}, v_1, \pi_1) = 1 & \\ i < t \leq j \wedge v_0 \neq v_1 & \\ & (\mathsf{label}, v_0, \pi_0, t, v_1, \pi_1, \pi_{\mathsf{consis}}) \leftarrow \mathsf{Server} \\ & \mathsf{Client.VerifyConsistency}(\mathsf{digest}_i, \mathsf{digest}_j, \mathsf{label}, \pi_{\mathsf{consis}}) = 1 \end{array}\right] < \mathsf{negl}(\lambda)$$

5 Forget-me-not Trees

In this section we describe the basic Forget-me-not tree data structure, as it may be of independent interest beyond key transparency systems. In Section 5.1 we give formal interfaces and definitions. In Section 5.2 we formally describe our construction and in Section A we prove that this construction is secure. In Section 5.3 we analyze the asymptotic and concrete efficiency of our construction.

5.1 Definitions

Formally, a Forget-me-not tree is an instantiation of a cryptographic accumulator that supports membership and non-membership proofs. For notational convenience, we consider only full binary trees, but it is trivial to modify the definitions to consider arbitrary numbers of leaves. Forget-me-not trees consist of the following algorithms:

– $\mathsf{Merklize}(\mathsf{data}_1, \ldots, \mathsf{data}_n) \to (\mathsf{digest}, \mathsf{st})$ takes in a sequence of data inputs $\mathsf{data}_1, \ldots, \mathsf{data}_n$ and outputs a digest digest and some state st.

– $\mathsf{ProveMembership}(\mathsf{digest}, \mathsf{st}, \mathsf{data}) \to \{\pi, \perp\}$ takes in a digest digest, some state st, and some arbitrary data data and outputs either a proof π or $\perp$ if data is not a member.

– $\mathsf{ProveNonMembership}(\mathsf{digest}, \mathsf{st}, \mathsf{data}) \to \{\pi, \perp\}$ takes in a digest digest, some state st, and some arbitrary data data and outputs either a proof π or $\perp$ if data is a member.

– $\mathsf{VerifyMembership}(\mathsf{digest}, \mathsf{data}, \pi) \to \{0, 1\}$ takes in a digest digest, some arbitrary data data, and a proof π and outputs either 0 or 1.

– $\mathsf{VerifyNonMembership}(\mathsf{digest}, \mathsf{data}, \pi) \to \{0,1\}$ takes in a digest digest, some arbitrary data data, and a proof π and outputs either 0 or 1.

– $\mathsf{MergeAndProve}(\mathsf{digest}_1, \mathsf{st}_1, \mathsf{digest}_2, \mathsf{st}_2) \to (\mathsf{digest}, \mathsf{st}, \pi)$ takes in two digests $\mathsf{digest}_1, \mathsf{digest}_2$ and their corresponding states $\mathsf{st}_1, \mathsf{st}_2$ and outputs a new digest digest, a new proving state st, and a proof π that the merge was performed correctly.

– $\mathsf{VerifyMerge}(\mathsf{digest}_1, \mathsf{digest}_2, \mathsf{digest}, \pi) \to \{0,1\}$ takes in two digests $\mathsf{digest}_1, \mathsf{digest}_2$, a merged digest digest, and a proof π and outputs either 0 or 1.

These algorithms provide the following guarantees:

Correctness. We require the three natural correctness properties, *correct membership*, *correct non-membership*, and *correct merge*, i.e., if the respective proof is generated honestly, then it will verify with probability 1. Formally,

(1) *Membership:* If data was included in the input to $\mathsf{Merklize}$, then the resulting proof is accepted by the verifier. Formally, for all sets on input data $\mathsf{data}_1, \ldots, \mathsf{data}_n$ and all $i \in [n]$,

$$\Pr\left[\mathsf{VerifyMembership}(\mathsf{digest}, \mathsf{data}_i, \pi) = 1 \;\middle|\; \begin{matrix}(\mathsf{digest}, \mathsf{st}) \leftarrow \mathsf{Merklize}(\mathsf{data}_1, \ldots, \mathsf{data}_n) \\ \pi \leftarrow \mathsf{ProveMembership}(\mathsf{digest}, \mathsf{st}, \mathsf{data}_i)\end{matrix}\right] = 1$$

(2) *Non-Membership:* If data was not included in the input to $\mathsf{Merklize}$, then the resulting proof is accepted by the verifier with overwhelming probability. Formally, for all $\mathsf{data}_1, \ldots, \mathsf{data}_n$ and all $\mathsf{data}^* \notin \{\mathsf{data}_1, \ldots, \mathsf{data}_n\}$,

$$\Pr\left[\mathsf{VerifyNonMembership}(\mathsf{digest}, \mathsf{data}^*, \pi) = 1 \;\middle|\; \begin{matrix}(\mathsf{digest}, \mathsf{st}) \leftarrow \mathsf{Merklize}(\mathsf{data}_1, \ldots, \mathsf{data}_n) \\ \pi \leftarrow \mathsf{ProveNonMembership}(\mathsf{digest}, \mathsf{st}, \mathsf{data}^*)\end{matrix}\right] = 1$$

(3) *Merge:* If two trees are honestly constructed and merged, then the resulting merge proof is accepted by the verifier. Formally, for all $\mathsf{data}_1, \ldots, \mathsf{data}_n$ and $\mathsf{data}'_1, \ldots, \mathsf{data}'_\ell$,

$$\Pr\left[\mathsf{VerifyMerge}(\mathsf{digest}_1, \mathsf{digest}_2, \mathsf{digest}, \pi) = 1 \;\middle|\; \begin{matrix}(\mathsf{digest}_1, \mathsf{st}_1) \leftarrow \mathsf{Merklize}(\mathsf{data}_1, \ldots, \mathsf{data}_n) \\ (\mathsf{digest}_2, \mathsf{st}_2) \leftarrow \mathsf{Merklize}(\mathsf{data}'_1, \ldots, \mathsf{data}'_\ell) \\ (\mathsf{digest}, \mathsf{st}, \pi) \leftarrow \mathsf{MergeAndProve}(\mathsf{digest}_1, \mathsf{st}_1, \mathsf{digest}_2, \mathsf{st}_2)\end{matrix}\right] = 1$$

Binding. We require that a malicious party that computes digest cannot produce both a membership and non-membership proof for any leaf data data. Formally, for all p.p.t. adversaries $\mathcal{A}$, we require that

$$\Pr\left[\begin{matrix}\mathsf{VerifyMembership}(\mathsf{digest}, \mathsf{data}, \pi_1) = 1 \text{ and} \\ \mathsf{VerifyNonMembership}(\mathsf{digest}, \mathsf{data}, \pi_2) = 1\end{matrix} \;\middle|\; (\mathsf{digest}, \mathsf{data}, \pi_1, \pi_2) \leftarrow \mathcal{A}(1^\lambda)\right] \leq \mathsf{negl}(\lambda)$$

Merge Consistency. When we merge two trees, the new resulting digest must respect the membership and non-membership properties of the subtrees. Formally, for all p.p.t. adversaries $\mathcal{A}$,

we require that

$$\Pr\left[\begin{array}{c}\mathsf{VerifyMerge}(\mathsf{digest}_1,\mathsf{digest}_2,\mathsf{digest},\pi_{\text{merge}})=1 \text{ and}\\ \mathsf{VerifyMembership}(\mathsf{digest},\mathsf{data},\pi_{\text{mem}})=1 \text{ and}\\ \left(\begin{array}{c}\mathsf{VerifyNonMembership}(\mathsf{digest}_1,\mathsf{data},\pi_{\text{non-mem},1})=1 \text{ and}\\ \mathsf{VerifyNonMembership}(\mathsf{digest}_2,\mathsf{data},\pi_{\text{non-mem},2})=1\end{array}\right)\end{array}\middle|\begin{array}{c}\left(\begin{array}{c}\mathsf{digest}_1,\mathsf{digest}_2,\\ \mathsf{digest},\mathsf{data},\\ \pi_{\text{mem}},\pi_{\text{merge}},\\ \pi_{\text{non-mem},1},\\ \pi_{\text{non-mem},2}\end{array}\right)\leftarrow\mathcal{A}(1^\lambda)\end{array}\right]\leq\mathsf{negl}(\lambda)$$

and symmetrically,

$$\Pr\left[\begin{array}{c}\mathsf{VerifyMerge}(\mathsf{digest}_1,\mathsf{digest}_2,\mathsf{digest},\pi_{\text{merge}})=1 \text{ and}\\ \mathsf{VerifyNonMembership}(\mathsf{digest},\mathsf{data},\pi_{\text{non-mem}})=1 \text{ and}\\ \left(\begin{array}{c}\mathsf{VerifyMembership}(\mathsf{digest}_1,\mathsf{data},\pi_{\text{mem}})=1 \text{ or}\\ \mathsf{VerifyMembership}(\mathsf{digest}_2,\mathsf{data},\pi_{\text{mem}})=1\end{array}\right)\end{array}\middle|\begin{array}{c}\left(\begin{array}{c}\mathsf{digest}_1,\mathsf{digest}_2,\\ \mathsf{digest},\mathsf{data},\\ \pi_{\text{mem}},\pi_{\text{merge}},\\ \pi_{\text{non-mem}}\end{array}\right)\leftarrow\mathcal{A}(1^\lambda)\end{array}\right]\leq\mathsf{negl}(\lambda)$$

5.2 Construction

We give formal descriptions of our construction of a Forget-me-not tree in Algorithm 1, Algorithm 2, Algorithm 3, and Algorithm 4. In Algorithm 1 we provide a recursive description of the process of building a tree. Each leaf node is hashed and a Bloom filter is initialized with the data contained in the leaf. Then, nodes are merged together with Merge. Finally Merklize returns the `digest` of the root node and the full tree structure. In Algorithm 2, the prover augments a "standard" Merkle proof by additionally proving that $\mathsf{BF.Contains}(\mathsf{f},\mathsf{data})=1$, for each Bloom filter f embedded in the nodes along the path from the leaf to the root. In Algorithm 3 we use a depth-first search to generate a non-membership proof; when a node contains a Bloom filter f such that $\mathsf{BF.Contains}(\mathsf{f},\mathsf{data})=0$, then the prover generates a Merkle proof that one of the relevant indexes in the Bloom filter is 0. Otherwise, the prover recurses to the node's children. In the extreme case, this process descends to the leaf node and the value itself is opened. Finally, in Algorithm 4, the prover returns the filters of the two roots, along with their hashes. The verifier directly merges the filters together and verifies that each of the digests was computed correctly.

In Section A we prove that this construction provides the properties described in Section 5.1.

5.3 Efficiency

We analyze the efficiency of Forget-me-not trees asymptotically and concretely.

Asymptotic efficiency. We are primarily concerned with the efficiency of the three main proof interfaces of Forget-me-not trees: (1) membership proofs, (2) non-membership proofs, and (3) merge proofs. Recall that n is the total number of leaves in the tree, m is the size of the Bloom filters, and k is the number of hash functions used for the Bloom filters.

(1) *Membership proofs:* Membership proof require producing the siblings of nodes along the path from the root to the relevant leaf and producing k proofs that the appropriate bits in each Bloom filter are set. The path from the root to the leaf is of length $\mathcal{O}(\log(n))$ and each proof in each Bloom filter is of size $\mathcal{O}(\log(m))$. Thus, the total proof size is $\mathcal{O}(k\log(n)\log(m))$.

Algorithm 1 Building a Forget-me-not tree

procedure Merge($t_{\text{left}}, t_{\text{right}}$)
  $\texttt{self.f} = \mathsf{BF.Merge}(t_{\text{left}}.\texttt{f}, t_{\text{right}}.\texttt{f})$
  $\texttt{self.digest} = H(H(t_{\text{left}}.\texttt{digest} \| t_{\text{right}}.\texttt{digest}) \| \mathsf{MT.Merklize}(\texttt{self.f}))$
  $\texttt{self.left} \leftarrow t_{\text{left}}$,
  $\texttt{self.right} \leftarrow t_{\text{right}}$
  return self
end procedure

procedure TreeBuilder($\mathsf{data}_1, \ldots, \mathsf{data}_n$)
  if $n = 1$ **then**
    $\mathsf{f} \leftarrow \mathsf{BF.Setup}(), \texttt{self.f} \leftarrow \mathsf{BF.Add}(\mathsf{f}, \mathsf{data}_1)$
    $\texttt{self.digest} \leftarrow H(H(\mathsf{data}_1) \| \mathsf{MT.Merklize}(\mathsf{f}))$
    $\texttt{self.left} \leftarrow \perp$, $\texttt{self.right} \leftarrow \perp$
  else
    $t_{\text{left}} \leftarrow \mathsf{TreeBuilder}(\mathsf{data}_1, \ldots, \mathsf{data}_{\frac{n}{2}})$
    $t_{\text{right}} \leftarrow \mathsf{TreeBuilder}(\mathsf{data}_{\frac{n}{2}+1}, \ldots, \mathsf{data}_n)$
    $\texttt{self} \leftarrow \mathsf{Merge}(t_{\text{left}}, t_{\text{right}})$
  end if
  return self
end procedure

procedure Merklize($\mathsf{data}_1, \ldots, \mathsf{data}_n$)
  $t = \mathsf{TreeBuilder}(\mathsf{data}_1, \ldots, \mathsf{data}_n)$
  return $t.\texttt{digest}, t$
end procedure

Algorithm 2 Membership Proofs

procedure ProveMembership(digest, st, data)
  $\texttt{f} \leftarrow \mathsf{BF.Setup}(), \texttt{f} \leftarrow \mathsf{BF.Add}(\texttt{f}, \mathsf{data})$
  $d \leftarrow H(H(\mathsf{data}) \| \mathsf{MT.Merklize}(\texttt{f}))$
  Find leaf node in st where $\texttt{node.digest} = d$; **return** $\perp$ if none exists. Denote the index of this leaf as index
  $\pi \leftarrow \texttt{index}$
  while node is not the root of st **do**
    Let par be node's parent and sib be its sibling
    $\forall i \in [k]$:
      $(d_{\mathsf{BF}}, \pi_{\mathsf{BF},i}) \leftarrow \mathsf{MT.Prove}(\texttt{par.f}, h_i(\mathsf{data}), 1)$
    $\pi \leftarrow \pi \| (\texttt{sib.digest}, d_{\mathsf{BF}}, \pi_{\mathsf{BF},1}, \ldots, \pi_{\mathsf{BF},k})$
    $\texttt{node} \leftarrow \texttt{par}$
  end while
  return π
end procedure

procedure VerifyMembership(digest, data, π)
  $\texttt{f} \leftarrow \mathsf{BF.Setup}(), \texttt{f} \leftarrow \mathsf{BF.Add}(\texttt{f}, \mathsf{data})$
  $d \leftarrow H(H(\mathsf{data}) \| \mathsf{MT.Merklize}(\texttt{f}))$
  for $(d_{\texttt{sib}}, d_{\mathsf{BF}}, \pi_{\mathsf{BF},1}, \ldots \pi_{\mathsf{BF},k}) \in \pi$ **do**
    $\forall i \in [k]$:
      assert$(\mathsf{MT.Verify}(d_{\mathsf{BF}}, \pi_{\mathsf{BF},i}, h_i(\mathsf{data}), 1) = 1)$
    if $\texttt{dir} = \texttt{left}$ **then** ▷ Compute dir based on index
      $d \leftarrow H(H(d \| d_{\texttt{sib}}) \| d_{\mathsf{BF}})$
    else
      $d \leftarrow H(H(d_{\texttt{sib}} \| d) \| d_{\mathsf{BF}})$
    end if
  end for
  return $d \stackrel{?}{=} \mathsf{digest}$
end procedure

(2) *Non-membership proofs:* The size of non-membership proofs is data-dependent. In the worst case, each of $\{h_i(\mathsf{data}_1)\}_{i\in[k]} = \ldots = \{h_i(\mathsf{data}_n)\}_{i\in[k]} = \{h_i(\mathsf{data})\}_{i\in[k]}$ where $\mathsf{data}_1, \ldots, \mathsf{data}_n$ are the data in the leaves of the tree and data is the element for which we are proving non-membership. In this case, proving non-membership is obviously equivalent to opening the whole tree. However, recall that the false positive rate is an inverse exponential function (see Section 4). Thus, in the case that either the hash functions are modeled as random oracles or the hash functions are selected after the data is chosen, we instead can estimate the expected number of nodes the recursive algorithm must visit. Notice that the driving factor in the false-positive rate is the ratio n/m; as n/m grows, the false-positive rate approaches 1 and when $n > m$, the size of the proofs grows exponentially, as the number of nodes visited grows exponentially.[6] For each visited node, the prover produces a Merkle proof for the Bloom filter of size $\mathcal{O}(\log(m))$. Thus, the total expected size of the proofs is $\mathcal{O}(\log(m) \times \mathsf{max}(\frac{n}{m}, 1))$.

(3) *Merge proofs:* Merge proofs are linear in m, as the two Bloom filters must be sent in the clear. Notice, however, that Bloom filters are quite compressible when they are sparse, resulting in roughly linear proof sizes in n.

[6]When accounting for this carefully, one will observe that the probabilities of false-positives among parent and children nodes are not independent, as parent Bloom filters are computed as the OR of the children's. However, it is clear that this approximation is correct.

Algorithm 3 Non-Membership Proofs

procedure ProveNonMembership(digest, st, data)
  $\pi \leftarrow \varepsilon$, $S \leftarrow \{\texttt{st.root}\}$
  while $S \neq \emptyset$ **do**
    node $\leftarrow S$.POP()
    if node is a leaf **then**
      $\pi \leftarrow \pi \| (\texttt{type=leaf}, \texttt{node.digest})$
    else if BF.Contains(node.f, data) $= 0$ **then**
      Pick $j \in [k]$ such that node.f$[j] = 0$
      $\pi_{\mathsf{BF}} \leftarrow$ MT.Prove(node.f, h_j(data), 0)
      $\pi \leftarrow \pi \| (\texttt{type=bloom}, H(d_{\text{left}} \| d_{\text{right}}), j, \pi_{\mathsf{BF}})$
    else ▷ False positive: recurse into children
      $\pi \leftarrow \pi \| (\texttt{type=recurse}, \mathsf{MT.Merklize}(\texttt{node.f}))$
      S.PUSH(node.right), S.PUSH(node.left)
    end if
  end while
  return π
end procedure

procedure VerifyNonMembership(digest, data, π)
  $(d, i) \leftarrow$ VerifyNode(π, 1, data)
  return 1 if $d =$ digest and $i = n{+}1$; otherwise **return** 0
end procedure

procedure VerifyNode(π, i, data)
  Parse π as a sequence of entries $e_1, \ldots, e_n$
  if $e_i = (\texttt{type=leaf}, d_\ell)$ **then**
    f $\leftarrow$ BF.Setup(), f $\leftarrow$ BF.Add(f, data)
    assert $(d_\ell \neq H(H(\text{data}) \| \mathsf{MT.Merklize}(\texttt{f})))$
    return $(d_\ell, i+1)$
  else if $e_i = (\texttt{type=bloom}, d_{\text{inner}}, j, \pi_{\mathsf{BF}})$ **then**
    assert $(j \in \{h_{j'}(\text{data})\}_{j' \in [k]})$
    assert (MT.Verify(π_{BF}.digest, $\pi_{\mathsf{BF}}.\pi, j, 0) = 1$)
    return $(H(d_{\text{inner}} \| \pi_{\mathsf{BF}}.\texttt{digest}), i+1)$
  else ▷ $e_i = (\texttt{type=recurse}, d_{\mathsf{BF}})$
    $(d_{\text{left}}, i') \leftarrow$ VerifyNode($\pi, i+1$, data)
    $(d_{\text{right}}, i'') \leftarrow$ VerifyNode(π, i', data)
    return $(H(H(d_{\text{left}} \| d_{\text{right}}) \| d_{\mathsf{BF}}), i'')$
  end if
end procedure

Algorithm 4 Merge Proofs

procedure MergeAndProve(digest$_1$, st$_1$, digest$_2$, st$_2$)
  st $\leftarrow$ Merge(st$_1$, st$_2$)
  $\pi_1 \leftarrow$ (st$_1$.f, H(st$_1$.left.digest$\|$st$_1$.right.digest))
  $\pi_2 \leftarrow$ (st$_2$.f, H(st$_2$.left.digest$\|$st$_2$.right.digest))
  return (st.digest, st, $\pi_1 \| \pi_2$)
end procedure

procedure VerifyMerge(digest$_1$, digest$_2$, digest, π)
  Parse π as $(\mathsf{f}_1, d_{\text{inner},1}, \mathsf{f}_2, d_{\text{inner},2})$
  $d_{\mathsf{BF},1} \leftarrow \mathsf{MT.Merklize}(\mathsf{f}_1)$
  $d_{\mathsf{BF},2} \leftarrow \mathsf{MT.Merklize}(\mathsf{f}_2)$
  f $\leftarrow$ BF.Merge(f$_1$, f$_2$)
  $d_{\mathsf{BF}} \leftarrow \mathsf{MT.Merklize}(\mathsf{f})$
  assert(digest$_1 = H(d_{\text{inner},1} \| d_{\mathsf{BF},1})$)
  assert(digest$_2 = H(d_{\text{inner},2} \| d_{\mathsf{BF},2})$)
  return digest $\stackrel{?}{=} H(H(\text{digest}_1 \| \text{digest}_2) \| d_{\mathsf{BF}})$
end procedure

Concrete efficiency. To get a better sense of the concrete performance of Forget-me-not trees, we implement and benchmark the performance of membership proofs, non-membership proofs, and merge proofs.[7] We implement four small optimizations in our implementation to improve performance: (1) when Merklizing the Bloom filters, we chunk up the filters into λ-bit chunks, rather than splitting each bit separately, (2) when beneficial, we re-use sibling hashes when proving that all k bits are set in the Bloom filter for membership proofs, (3) we use sparse Merkle trees whenever possible, and (4) we compress the representation of the filters when possible, using Elias-Fano encoding [Fan71, Eli74] when they are sparse (either in terms of the number of 0's or 1's) or simply replacing it with some special value when the filter is fully saturated. Rather than build full trees for this benchmarking process, we generate proofs in a "bottom-up" way so that the prover

[7] Our implementation is derived from the implementation used in the key transparency system described below. As such, we also include epoch information in each digest, which is not described in this section. This very slightly increases proof sizes. We note that all trees in these benchmarks contain only nodes from the same epoch, so there is no epoch-dependent performance impact to non-membership.

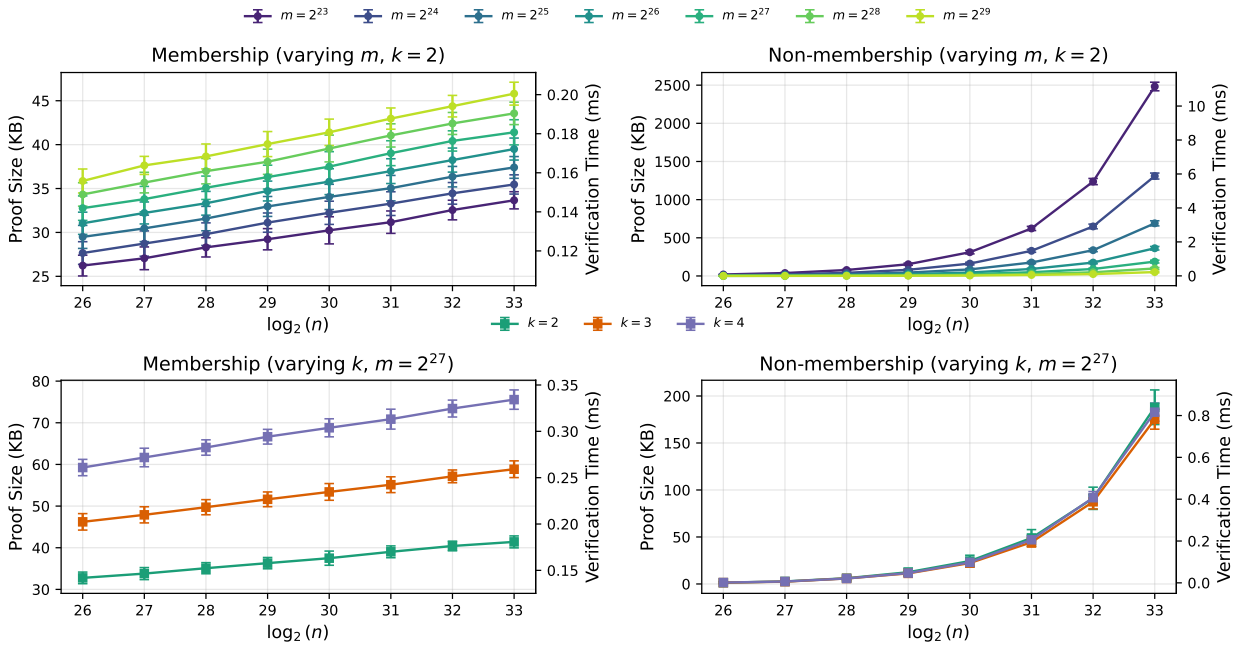


Figure 2: Membership and non-membership proofs for a Forget-me-not tree with n leaves, averaged over 100 trials with standard error bars. Each plot shows exact proof sizes (left y-axis labels) and approximate verification times (right y-axis labels). Top pair: varying Bloom filter size m while keeping $k = 2$. Bottom pair: varying k while keeping $m = 2^{27}$.

can generate the contents of the Bloom filters as needed from random leaf data. When there are nodes that the prover does not need to visit, their labels are replaced by a dummy hash and the root label is generated at the end of the process. We check that these proofs are well-formed by running them through the full proof verification algorithm. This approach lets us sweep through the parameter set more efficiently.[8]

We show the size of membership and non-membership proofs in Figure 2. Additionally, because verification time is linear in the proof size, throughout this work we fit a second y-axis on each plot showing approximate verification times; the fit for most of these axes is extremely accurate, but there is small error due to jitter. The left two plots illustrate the impact of varying m, the size of the Bloom filters, on proof sizes; both plots hold $k = 2$ for all experiments. We see that the size of membership proofs grows roughly logarithmically in n (note that the x-axis is log-scale) and that the concrete proof sizes are quite good, with proofs ranging between 25KB and 50KB even for very large trees. As m increases, the concrete proof sizes also increase logarithmically (note that m doubles between each line), thus tightly matching the $\mathcal{O}(\log(m)\log(n))$ asymptotics.

Non-membership proofs, illustrated in the second plot in Figure 2, show the "piecewise" proof sizes described above. Specifically, when $\frac{n}{m} < 1$, the proof sizes are *constant* in n. But, once $\frac{n}{m} \geq 1$, the proof sizes grow linearly in n (again, recall that the x-axis is log-scale). Concretely, the size of the proof is quite good as long as $\frac{n}{m}$ does not exceed 1 too dramatically, but eventually the size of the proof becomes too large.

[8] All benchmarks are generated pseudorandomly, so all results can be replicated from https://github.com/gkaptch1/ForgetMeNotTrees.

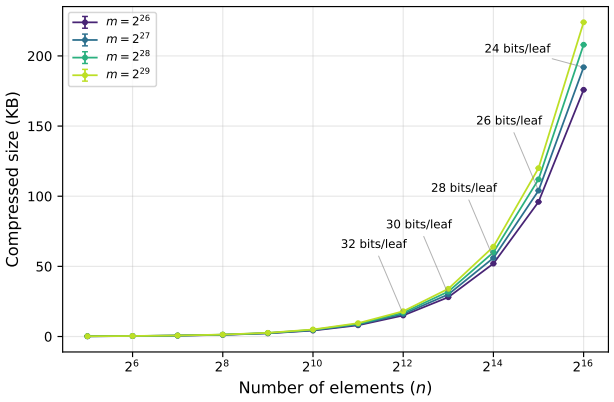


Figure 3: Size of compressed Bloom filters, where each line captures a different size of the Bloom filter (m bits) and n inserted elements. $k = 2$ for all cases. Each datapoint is the average size of compressing 100 Bloom filters with random data inserted.

We note that there is a clear trade-off between membership proof size and non-membership proof size when choosing m. Larger m result in larger membership proofs, growing by a several KB each time m is doubled. On the other hand, if m is set to be too small, then non-membership proofs quickly reach their rapid-growth phase and become unusable. Thus, selecting the right parameters will depend heavily on the anticipated use case.

The right two subplots in Figure 2 illustrate the impact of changing $k \in [2, 3, 4]$.[9] As k increases, membership proofs need to prove that more bits are set in each Bloom filter, resulting in concretely larger proofs. The final subplot demonstrates, however, that increasing k does not help produce smaller non-membership proofs. While increasing k does reduce the false positive rate of the Bloom filters, it also means that the Bloom filters reach saturation more quickly. Thus, any gains from improving the false positive rate are immediately offset by the need to prove non-membership in more nodes. As such, for the rest of this work we only consider $k = 2$.

Finally, in Figure 3 we show the performance of the compression used to prove that merge operations were correct. Notably, for sparse Bloom filters, the size of the compressed filter is largely determined by n, the number of inserted elements, and not m, the size of the Bloom filters. Doubling m results in only slightly larger compressed filters.

6 Building Key Transparency from Forget-me-not Trees

Building an efficient key transparency system on top of Forget-me-not trees requires small changes to the Forget-me-not tree construction itself and also specifying the way in which the tree is used. We need to construct a Transparent Dictionary, a definition adapted from [HSBB26] with slight modifications, described in Section 4.3. Notably, the interfaces for this are not simply membership and non-membership, but rather need to account for the fact that the data structure is over $(\mathsf{label}, \mathsf{val})$ pairs, the data changes over time, and must expose interfaces for querying the latest version of a key, monitoring keys, and auditing epoch consistency. We begin by discussing the minor changes we need to make to the Forget-me-not trees before proceeding to show how to

[9] Setting $k = 1$ has too poor of a false positive rate to be worth seriously considering for deployment.

leverage these into highly efficient key transparency systems.

6.1 Optimizing Forget-me-not Trees

We make three simple changes to the underlying Forget-me-not data structure in order to optimize it for our key transparency use case: using label-value semantics, using a forest of trees to support numbers of leaves that are not a power of two, and embedding epochs into the digests of nodes.

Making this construction label-value. We adapt the standard approach for transforming a standard Merkle tree into a label-value store. Specifically, we logically use only the label when inserting the tuple into the tree, but then additionally hash the value into the leaf node. In prefix-sorted trees [CDGM19, MKKS+23], for example, only the label determines where in the tree the tuple is stored. In our case, we compute the label of leaf nodes as $H(\mathsf{label}\|\mathsf{val})$, but then only add label to the initial filter. With respect to Algorithm 1, we have each $\mathsf{data} = (\mathsf{label}, \mathsf{val})$ and replace line 10 with

$$\texttt{self.f} \leftarrow \mathsf{BF.Add}(\mathsf{f}, \mathsf{label}).$$

Using the Merkle forest representation. In Section 5, we assumed that n was a perfect power of two. In real systems, however, this assumption does not hold. To accommodate an arbitrary number of leaves, we hold the data using a *Merkle forest* approach, as in many prior works (e.g., [Goob, Gooa]). This stores the n leaves in a set of at most $\log(n)$ perfect binary Forget-me-not trees. This representation has the property that merging two forests into a new forest requires at most $\mathcal{O}(\log(n))$ merge operations and has been shown to be optimal [BCCK25]. For clarity, we call this use Forget-me-not forests.

Epochs. In order to determine which version of a label is the most recent, we need to embed a notion of time into the tree itself. We do this through the notion of epochs, a counter which increments by one each time the server publishes a new commitment to the state of the key transparency dictionary. When computing node digests, we add the epoch in which the leaf or node was created: for leaves, we compute

$$\texttt{digest} = H(\mathsf{label}\|\mathsf{val}\|\texttt{epoch})$$

and for internal nodes, we compute the hash as

$$\texttt{digest} = H(H(t_{\text{left}}.\texttt{digest}\|t_{\text{right}}.\texttt{digest})\|\mathsf{MT.Merklize}(\texttt{self.f})\|\texttt{epoch})$$

With epochs embedded into each node, we change non-membership and membership proofs to be "epoch-aware." Specifically, epoch-aware non-membership proofs shortcut their descent at nodes with epochs earlier than some target epoch. Non-membership proofs now contain a set of nodes that are the ancestors of every leaf in the forest, and for each of those nodes, either (1) $\mathsf{BF.Contains}(\mathsf{f}, \mathsf{label}) = 0$ or (2) the epoch of the node is less than the target epoch. Similarly, epoch-aware membership proofs also check the invariant between children and parent nodes along the path from the root to the leaf that all children's embedded epochs are no later than their parents'. Finally, during merging, verifiers check that the correct epoch is embedded into the new root node. When these checks are combined, a malicious server embedding a bad epoch into a leaf only succeeds in making leaves unreadable.

Modified interfaces. The above changes require us to modify the interfaces for the Forget-me-not trees to expose the notion of an epoch and change the semantics to be label-value. The new interfaces are:

- $\mathsf{FMNForest.Merklize}((\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n), \mathsf{epoch}) \to (\mathsf{digest}, \mathsf{st})$ takes in a sequence of label-value pairs $(\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n)$ and an epoch epoch, and outputs a forest digest digest and state st.
- $\mathsf{FMNForest.ProveMembership}(\mathsf{digest}, \mathsf{st}, \mathsf{label}, \mathsf{val}, \mathsf{epoch}) \to \{\pi, \bot\}$ takes in a forest digest digest, state st, a label label, a value val, and an epoch epoch and outputs either a proof π or $\bot$ if $(\mathsf{label}, \mathsf{val})$ is not in the forest with epoch epoch.
- $\mathsf{FMNForest.VerifyMembership}(\mathsf{digest}, \mathsf{label}, \mathsf{val}, \mathsf{epoch}, \pi) \to \{0, 1\}$ takes in a forest digest digest, a label label, a value val, an epoch epoch, and a proof π and outputs either 0 or 1.
- $\mathsf{FMNForest.ProveNonMembership}(\mathsf{digest}, \mathsf{st}, \mathsf{label}, \mathsf{epoch}) \to \{\pi, \bot\}$ takes in a forest digest digest, state st, a label label, and an epoch epoch, and outputs a proof π.
- $\mathsf{FMNForest.VerifyNonMembership}(\mathsf{digest}, \mathsf{label}, \mathsf{epoch}, \pi) \to \{0, 1\}$ takes in a label label, a digest digest, an epoch epoch, and a proof π and outputs either 0 or 1.
- $\mathsf{FMNForest.MergeAndProve}(\mathsf{digest}_1, \mathsf{st}_1, \mathsf{digest}_2, \mathsf{st}_2, \mathsf{epoch}) \to (\mathsf{digest}, \mathsf{st}, \pi)$ takes in two digests $\mathsf{digest}_1, \mathsf{digest}_2$, their corresponding states $\mathsf{st}_1, \mathsf{st}_2$, and an epoch epoch and outputs a new digest digest, a new state st, and a proof π that the merge was performed correctly.
- $\mathsf{FMNForest.VerifyMerge}(\mathsf{digest}_1, \mathsf{digest}_2, \mathsf{digest}, \mathsf{epoch}, \pi) \to \{0, 1\}$ takes in two forest digests $\mathsf{digest}_1, \mathsf{digest}_2$, a merged forest digest, an epoch epoch and a proof π and outputs either 0 or 1.

We include formal descriptions of each of these interfaces and prove the appropriate updated properties about them in Section B. All of these algorithms and properties are natural generalizations of the base construction in Section 5.

6.2 Constructing Key Transparency

With our modified interfaces for Forget-me-not forests in hand, we now discuss how to construct key transparency. Recall that the four main operations in key transparency are (1) updating the dictionary, (2) querying the latest version of a key, (3) monitoring a key to see if there are any malicious updates, and (4) auditing epoch transitions.

Update operations. Update operations build a new tree using the $\mathsf{Merklize}$ interface of the Forget-me-not forest and then use $\mathsf{MergeAndProve}$ to merge this new forest into the existing forest (see $\mathsf{Server.Update}$ in Algorithm 5). The verifier simply checks the resulting merge proof.

One subtle observation about this process is that the leaves need not stay sorted in epoch order due to the batched nature of the merge process. Imagine, for example, the old tree holds an odd number of leaves, meaning there is a tree of size 1 with some epoch epoch_1. If Update adds an even number of leaves in $\mathsf{epoch}_2 > \mathsf{epoch}_1$, then the tree of size 1 will be unaffected.

Query operations. When multiple versions of a label are present in the forest, a client needs to be convinced that the server has delivered the *latest* version of that label. The server proves this by giving a membership proof for the relevant leaf and then a monitoring proof for all subsequent epochs. Concretely, this constitutes an epoch-aware membership proof for the leaf that they claim is the most recent version of label, then epoch-aware non-membership proofs for label for each

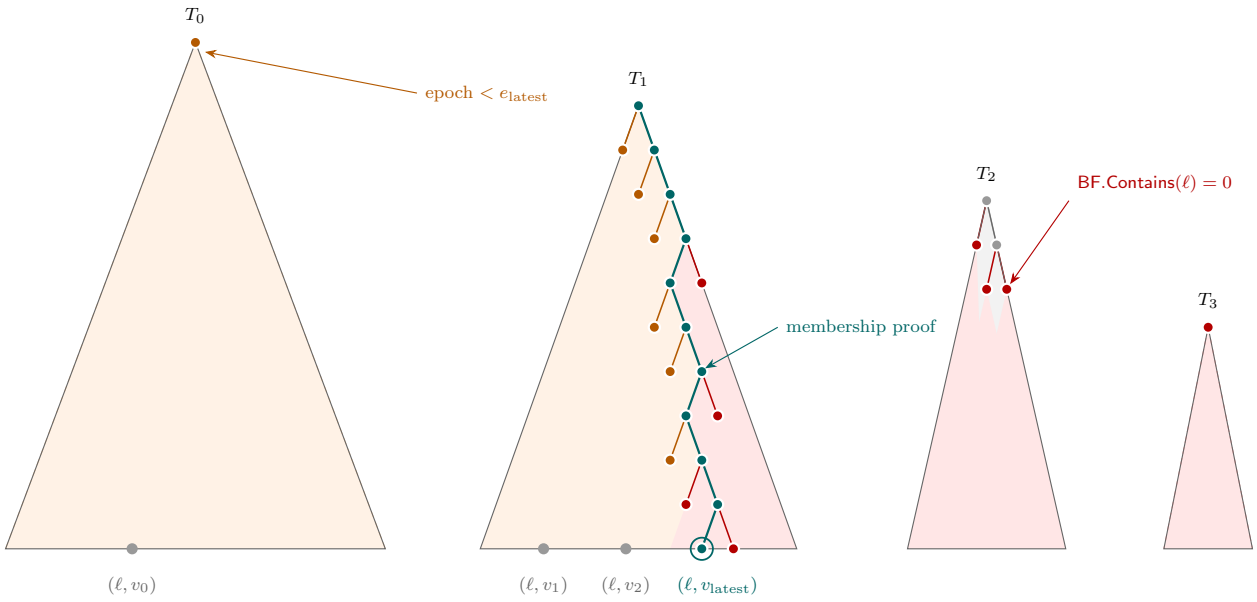


Figure 4: A simplified visualization of the proof produced when reading the latest version of a label ℓ in a Forget-me-not forest. In this example, we assume that all leaf nodes are sorted in exactly the order in which they are added to the tree; in practice, this assumption will not hold due to the ways in which large batches of keys are added. Teal traces the membership proof path for the latest version of label ℓ in T_1. Everything to the left of the path (orange) is covered by epoch checks: these subtrees predate e_{latest}. Near the leaf we see that a whole subtree shares the same epoch e_{latest}, so a non-membership proof must be used instead of an epoch check. The entire older tree T_0 is similarly epoch-covered. To the right of the path and at the bottom, each sibling requires a Bloom filter non-membership proof (red, BF.Contains(ℓ) = 0); the dashed line marks the frontier below which these checks resolve. The same frontier applies to T_2 and T_3. For tree T_2, the root Bloom filter contains a false-positive, so the prover recursively visits the children. In tree T_3 the root alone is sufficient for proving non-membership.

node along the membership proof's co-path. We give a description of this in Server.Lookup in Algorithm 5, and the client's verification procedure in Client.VerifyLookup in Algorithm 6.

We illustrate an example of our query proofs in Figure 4, with nodes along the membership path highlighted in teal, and nodes along the co-path either highlighted in orange, when the non-membership prover finds a node with an earlier epoch, or red, when the non-membership prover finds a node with a Bloom filter miss. For easy illustration, we make the simplifying assumption that all leaves are epoch ordered, which is not true in practice as leaves are added in batches.

Monitoring operations. Clients can monitor updates to their label using the modified, epoch-aware non-membership proof outlined above. To prove that label has not been modified since epoch e, the server simply generates a non-membership proof for each tree in the Forget-me-not forest, where each node making up the non-membership set either has BF.Contains(f, label) $= 0$ or has some epoch $e' \leq e$.

Auditing operations. Auditors need to verify that two forests—one constituting the previous state of the directory holding n elements and one holding all n' new updates—have been properly

Algorithm 5 Server operations

procedure Server.Init($(\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n), \mathsf{epoch} = 0$)
  return FMNForest.Merklize($(\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n), \mathsf{epoch} = 0$)
end procedure

procedure Server.Update($\mathsf{state}_{i-1}, \Delta$)
  Parse $\Delta = (\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_{n'}, \mathsf{val}_{n'})$
  $\mathsf{digest}_\Delta, t' = \mathsf{FMNForest.Merklize}((\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_{n'}, \mathsf{val}_{n'}), i)$
  $\mathsf{digest}_i, t_i, \pi_{\text{merge}} \leftarrow \mathsf{FMNForest.MergeAndProve}(\mathsf{digest}_{i-1}, t_{i-1}, \mathsf{digest}_\Delta, t', i)$
  return $(\mathsf{digest}_i, t_i, (\mathsf{digest}_\Delta, \pi_{\text{merge}}))$
end procedure

procedure Server.Lookup($\mathsf{label}, \mathsf{state}_i$)
  Find latest leaf in state_i associated with label label. Denote its value as val, its epoch as epoch, and its index in the forest as index.
  $\pi \leftarrow \mathsf{index} \| \mathsf{FMNForest.ProveMembership}(\mathsf{digest}_i, \mathsf{state}_i, \mathsf{label}, \mathsf{val}, \mathsf{epoch})$
  for each subtree root sub that is a sibling along the membership path or a sibling tree **do**
    $\pi = \pi \| (\mathsf{sub.digest}, \mathsf{FMNForest.ProveNonMembership}(\mathsf{sub.digest}, \mathsf{sub.st}, \mathsf{label}, \mathsf{epoch}))$
  end for
  return π
end procedure

procedure Server.Consistency($\mathsf{label}, \mathsf{state}_i, \mathsf{state}_j$)
  $\pi_{\text{consis}} = \mathsf{FMNForest.ProveNonMembership}(\mathsf{digest}_j, \mathsf{state}_j, \mathsf{label}, \mathsf{state}_i.\mathsf{epoch} + 1)$
  return π_{consis}
end procedure

merged. This requires producing $\mathcal{O}(\log(n))$ proofs using MergeAndProve (as there are at most $\log(n)$ trees in the forest) which auditors can verify with VerifyMerge. These proofs are largely just the (compressed) Bloom filters associated with each tree root in the forest. Notably, merging a forest of size n with a forest of size n' will usually require only $\mathcal{O}(\log(n'))$ proofs.

One additional opportunity for optimization comes from recognizing that auditors are likely to verify many sequential epoch transitions. As a result, there are two types of verifiers:

– *Stateless verifiers:* When the verifier has no state from the previous epoch, then the merge proof π must contain the filters for both the old forest and the new forest.

– *Stateful verifiers:* If the verifier already has the filters for the old forest, then the server only sends filters for the roots in the new forest. This is the common condition, because once a stateless verifier verifies a single epoch transition, they can now become a stateful verifier for all future epoch transitions.

Thus, we assume that *most* auditing verification will be *stateful*, allowing us to focus on the complexity of stateful verification in our future benchmarks.

Privacy. Finally, we reason about the leakage of our approach. In each epoch, the server reveals a Bloom filter filled with the labels from each new update. While Bloom filters have less leakage than sending the updates in plaintext, this leakage may be sufficient to check if any given label is included in the updates. Given the deterministic nature of Bloom filters, this leakage appears to be inherent to our approach; if we randomize label, then different non-membership proofs will have

Algorithm 6 Client and Auditor operations

procedure Client.VerifyLookup(digest, label, val, π_{Lookup})
  Parse $\pi_{\mathsf{Lookup}} = \mathsf{index} \| \pi_{\text{mem}} \| (\mathsf{digest}_1, \pi_{\text{non-mem},1}) \| \ldots \| (\mathsf{digest}_\ell, \pi_{\text{non-mem},\ell})$
  Let epoch be the leaf epoch parsed from π_{mem}
  For $i \in [\ell]$
    assert(FMNForest.VerifyNonMembership(digest_i, label, epoch, $\pi_{\text{non-mem},i}$) $= 1$)
  assert(FMNForest.VerifyMembership(digest, label, val, epoch, π_{mem}) $= 1$)
  Reconstruct root digests for each tree $(\mathsf{digest}'_1, \ldots, \mathsf{digest}'_j)$ using the provided proofs
  For $i \in [j]$
    assert($\mathsf{digest}[i] = \mathsf{digest}'_i$)
  return 1
end procedure

procedure Auditor.VerifyInvariance($\mathsf{digest}_i, \mathsf{digest}_{i+1}, \pi$)
  Parse $\pi = (\mathsf{digest}_\Delta, \pi_{\text{merge}})$
  assert(digest_{i+1}.epoch is the appropriate epoch for the $(i+1)^{\text{th}}$ epoch)
  return FMNForest.VerifyMerge($\mathsf{digest}_i, \mathsf{digest}_\Delta, \mathsf{digest}_{i+1}, \mathsf{digest}_{i+1}.\mathsf{epoch}, \pi_{\text{merge}}$)
end procedure

procedure Client.VerifyConsistency($\mathsf{digest}_i, \mathsf{digest}_j$, label, π_{consis})
  return FMNForest.VerifyNonMembership(digest_j, label, $\mathsf{digest}_i.\mathsf{epoch} + 1, \pi_{\text{consis}}$)
end procedure

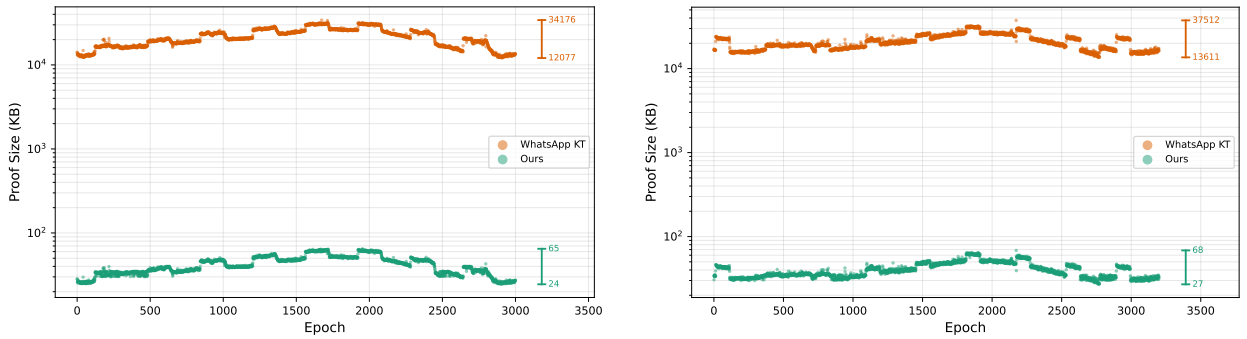
to be produced for each possible randomized value of label. It is possible that such an approach could be carefully tuned to produce a performant system, but it is beyond the scope of this work.

To limit the risks associated with this leakage, we can leverage the same approach outlined by [CDGM19]: rather than inserting a label that corresponds directly to a phone number (e.g., $\mathsf{label} = 555 - 555 - 5555$), we can instead randomize the label using a VRF keyed by the server (e.g., $\mathsf{label} = \mathsf{VRF}(\mathsf{sk}_{\mathsf{server}}, 555 - 555 - 5555)$). Using this approach, an auditor can only observe that the same label appears in multiple epoch updates, but will not know to what phone number that label corresponds. We leave this approach as a straight-forward optimization and omit it in our implementation.

7 Implementation and Evaluation

We implement Forget-me-not trees, along with the key-transparency specific optimizations described in Section 6.1, in Rust.[10] All code was written using Claude Opus 4.6, with testing conducted by the research team (see Section D for a discussion regarding the use of AI). While the produced code is likely not sufficiently well audited to be deployed on live systems, we have high confidence that the produced benchmarks are accurate. To evaluate our approach, we carefully benchmark its performance on the following three operations: (1) auditing epoch consistency proofs, (2) monitoring a key to see if it has been updated, and (3) querying the latest version of a key. For all benchmarks, we set $k = 2$ and $m = 2^{27}$. The choice of $m = 2^{27}$ is somewhat arbitrary, and larger values of m may be appropriate for real deployments; larger m only slightly increases the size of epoch proofs and monitoring proofs but can significantly decrease the size of key query

[10] All code and instructions for replicating results are available at https://github.com/gkaptch1/ForgetMeNotTrees.



(a) October 1, 2025

(b) February 1, 2026

Figure 5: Proof sizes per epoch: WhatsApp Key Transparency vs. our approach, measured over ≈ 3000 consecutive epochs (about 1 day). (Orange) data is loaded directly from WhatsApp logs, while Teal data is computed using our approach from the equivalent number of random key operations. The proofs benchmarked here assume that the verifier already has the filters at the roots of the trees in the old forest. Each epoch contains ≈ 9000–10000 key operations. Our proofs are generally 30–60KB, around 500× smaller than WhatsApp's 15MB to 30MB proofs. Our proofs can be verified in ≈ 3.5–8ms.

proofs. Larger m, however, does increase the maximum on-disk resources required to be a stateful verifier for epochs.

As discussed in Section 5, larger proofs are generated in a bottom-up way, where the digests of irrelevant sub-trees are replaced with a random value but all Bloom filters are computed honestly. Proofs are always verified with the full, honest verification algorithm. Verification times are all single-threaded on either `AMD EPYC 7502 @ 2.50GHz` or `Intel Xeon CPU E5-2660 v3 @ 2.60GHz`. Each of these processors has a significant number of cores and a large amount of RAM, but the processors themselves are slow compared to consumer-grade hardware, making our verification times an upper bound on real-world performance.

Epoch auditing. We compare the epoch consistency proofs in our construction to those created by WhatsApp's key transparency system [ACG+26], which is currently used in production. We download the epoch consistency proofs posted by WhatsApp during two recent days, October 1, 2025 and February 1, 2026 using a modified version of the `akd` library designed by Facebook [ACG+26]. WhatsApp generates two epoch proofs a minute, resulting in ≈ 3000 epochs each day. These two dates were chosen arbitrarily subject to the constraint that they were both recent and one fell on a weekend and one fell during the week. For each epoch proof generated by WhatsApp's system, we extracted the number of new keys and updated keys, then generated a *stateful* epoch proof for the same number of leaves, with the content of each leaf set at random. Notably, this assumes that the verifier already has the state for the previous epoch loaded; stateless proofs would be (roughly) twice as large.

In Figure 5 we compare the size of the proofs our system produces (Teal) to those generated by WhatsApp's system (Orange), with the range denoted to the right of each set of points. Both constructions have proof sizes that are *linear* in the number of key operations, but the constants for our construction are dramatically better. Specifically, the proofs are ≈ 500× smaller. Verification times were ≈ 3.5–8ms.

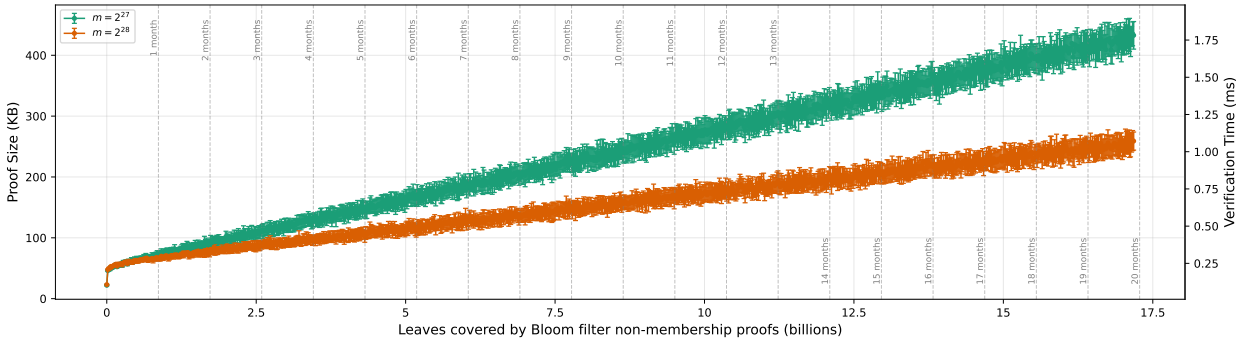


Figure 6: Key query benchmarks. To generate these results, we compute a single forest of size $2^{34}-1$ in which each key is unique. This tree is constructed from epochs of size $\approx 10,000$. Then, every ≈ 2000 epochs, we sample 20 random leaves within that epoch and produce the proofs necessary to demonstrate that that leaf is the latest in the forest with that label. Averages with standard error bars plotted. The x-axis indicates the number of leaves that were added to the forest after the epoch in which the random leaves were sampled, while the left y-axis represents the exact proof size and right y-axis is approximate verification time. Teal for $m = 2^{27}$ and Orange for $m = 2^{28}$.

We believe these benchmarks illustrate that our approach enables mass-scale auditability of epoch transitions, as it consumes a very reasonable amount of bandwidth and computation. In particular, any device that is Wi-Fi connected (either mobile or desktop) can download and verify these proofs without any problem; most web pages are larger than the epoch consistency proofs. Mobile devices that connect to the internet via a cellular connection, however, may not want to consume their monthly data allocations performing this auditing.

Key monitoring. Next, we consider the cost clients incur by monitoring their own keys. Recall that key transparency systems provide security only when clients monitor their own keys (or delegate such a task to a trusted party). At the same time, clients regularly go offline—potentially for long stretches—as part of normal operation; thus, there must be a way for clients to monitor many epochs at once.

In our construction, the bandwidth and verification time required to monitor a key is dependent on the number of new keys and key updates that have occurred since the client last monitored their key material. In Figure 7 we present the proof sizes for a client monitoring their key material as a function of the number of leaves that have been added to the Forget-me-not forest since the client has last monitored their keys. We present two sweeps of the parameter space: an "Average case" sweep which simply divides the number of leaves into roughly 6 hour increments, based on the rate of updates we observe in WhatsApp's deployed system (≈7.2 million key operations). We also run a "Worst case" sweep which only considers forests that have the maximum number of trees, as this requires communicating the largest number of non-membership proofs. The graph shows the length of time it takes for this number of key operations to occur for WhatsApp. The right y-axis is the approximate verification time.

Monitoring keys daily, in the average case, requires $<$ 8KB of communication each day. We see that verifying a month's worth of updates requires $<$ 30KB of communication and can be verified in less than 1ms. Extrapolating (as we have reached the linear regime), we find that a client could monitor a full year of updates in around 300KB.

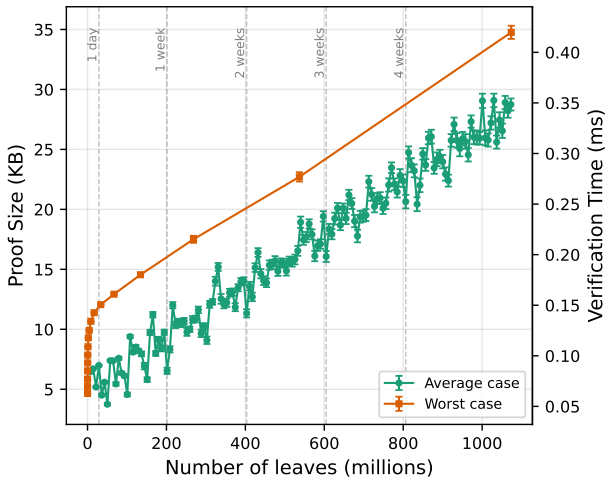


Figure 7: Monitoring proof sizes and approximate verification times as a function of the number of leaves added. Average case (Teal) constructs a forest by adding leaves in 6 hour increments, based on WhatsApp's deployed system. In the worst case sweep (Orange), we consider only forests with $2^\ell - 1$ leaves, which maximizes the number of trees. Each data point is averaged over 100 trials and shows standard error.

The performance of state-of-the-art systems relies only on the number of epochs (not number of updates), making apples-to-apples comparisons challenging. SEEMless [CDGM19] reports that a client who updates and monitors their key material 10 times over the course of a year will consume about 1MB. OPTIKS [LCG+24] generally improves the performance of these operations by a factor of 2. Thus, our approach is slightly better, with two caveats: (1) the asymptotics of existing approaches will eventually outperform our approach, as it is logarithmic in the number of epochs while ours is essentially linear in the long run, and (2) an advantage of our approach is that there is no penalty for monitoring more often (e.g., daily) before the asymptotics of existing approaches start to dominate. Monitoring frequently is significantly better for security, making our approach more secure in practice.

Key queries. Next, we benchmark the performance of query operations. Recall that query proofs demonstrate that a leaf is the latest one with the label, and thus will have linear proof sizes in the long term. In Figure 6 we present proof sizes for queries as a function of the number of leaves added since the leaf in question for $m = 2^{27}$ and $m = 2^{28}$. We find that key queries require a few hundred KB of communication, with $m = 2^{28}$ approximately halving the size of proofs from $m = 2^{27}$. By using random leaf data, our benchmarks represent a worst-case bound on proof sizes; in deployed key transparency systems, there are a bounded number of labels that will be updated, resulting in Bloom filters that are less filled and smaller proof sizes. Nevertheless, this is significantly more than existing approaches: OPTIKS [LCG+24] reports around 6KB key query responses for real-size systems.

We note that querying keys is the rarest key transparency operation, mostly performed when setting up a new device or making a new contact. Additionally, key queries must be performed when a contact updates their key material, but in this case the leaf in question will likely be very recent, resulting in a small proof size. One possible optimization is to have the server refresh every

client's key material every year, bounding the size of key query proofs. With 1 billion keys, this will result in 950 additional key operations per epoch, increasing the size of epoch proofs by roughly 10%.

Choosing parameters. Throughout this section, we have focused on $m = 2^{27}$. As noted above, this choice is somewhat arbitrary, and it may be reasonable to consider larger m in real deployments. Indeed, doubling m will result in key query responses that are about half the size. As shown in Figure 3, doubling m has only a small impact on the size of epoch consistency proofs, particularly for the sizes of key updates that WhatsApp's deployment encounters (2^{12}–2^{14} key updates per epoch). Additionally, because we make use of sparse Merkle trees when Merklizing Bloom filters (in which subtrees containing only 0's or 1's are replaced with a single, well-known hash), doubling m does not have a significant impact on the time or memory requirements for verifying epoch consistency proofs.

There are two drawbacks of increasing m: (1) marginally larger membership proofs (see Section 5), and (2) the worst-case, on-disk storage required to be a stateful verifier of epoch proofs increases. Specifically, when Bloom filters are approximately half filled, they cannot be meaningfully compressed. When $m = 2^{27}$, this results in 16MB of storage, whereas it would be 32MB for $m = 2^{28}$. Of course, larger m means that the Bloom filter will reach this saturation rate later, but in the long term $n \gg m$, so this condition will be reached. We note that once the Bloom filter is fully saturated, the on-disk storage requirements disappear, as we make use of sparse Merkle trees even when the sparsity is with respect to the 0's rather than 1's.

We note that it is also possible to construct Bloom filters that have a false positive rate of 0 at the cost of a processing time that is linear in the *input space* [LCL+17], which offers a different parameter tradeoff for Forget-me-not trees. In short, this technique involves proactively detecting false positives and building a secondary Bloom filter to store these false positives, then recursively detecting false positives in this filter. This could reduce the size of non-membership proofs at the cost of significantly more expensive epoch verification and larger membership proofs. Depending on the deployment setting, this may be an appropriate tradeoff, but in key transparency (where the space of valid labels will be *very* large), this is likely not the optimal tradeoff.

Deployment considerations. The most significant barrier to deployment for Forget-me-not trees is their large memory footprint. If we set $m = 2^{27}$, then the total memory required to hold all the Bloom filters for a tree with $n = 2^{33}$ leaves would be ≈ 144 petabytes! As such, it is necessary to store only parts of the state and compute other parts of the state on the fly, resulting in a time-space tradeoff that must be optimized for real world deployments.

We note, however, that epoch consistency proofs and most monitoring queries are computed only from the roots of trees, and thus do not require large amounts of memory. Key queries may need to descend *anywhere* into a large tree, and, thus, will be most impacted by the time-space tradeoff. We also observe that key query proofs remain largely the same between queries, facilitating caching. Specifically, the membership proof contained within the key query proof is append only—as the leaf is moved deeper into the forest, the prover only *adds* hashes to the pre-existing proofs. Similarly, the non-membership proofs for subtrees within the same tree in the forest will remain fixed, and can thus be reused. As a result, the computational effort required to compute answers to key queries can be efficiently amortized over many queries. Given the caching friendly nature of most queries, future engineering work is required to optimize proof generation, including the careful use of databases (which are omitted completely from our implementation).

Given the significant resources required to deploy Forget-me-not trees at scale, our approach can

be understood as offering the stronger security guarantees associated with mass-scale auditability in exchange for higher computational costs on the server and larger key query responses. We believe that in many key transparency applications, this trade-off may be appropriate.

8 Conclusion

We design the first mass-scale auditable key transparency system from hash functions, reducing the concrete size of epoch consistency proofs by $\approx 500\times$. We do this by carefully embedding Bloom filters into the digests of each node in the Merkle tree. When using our key transparency system, monitoring keys regularly becomes cost-efficient, with total bandwidth consumption slightly lower than state-of-the-art. To enable this performance, responses to key queries grow significantly—but these queries are (relatively) rare events. Our results indicate that a key transparency system built on our approach allows lightweight clients to monitor epoch consistency proofs while keeping the overall system highly performant—all without requiring more advanced cryptographic assumptions.

References

[ABB+26] Ezzudin Alkotob, Giulio Berra, Benjamin Beurdouche, Richard Hansen, Daniel Huigens, Dennis Jackson, Cory Francis Myers, and Michael Rosenberg. Trustworthy javascript for the web. Available at https://rwc26.waict.dev/, 2026.

[ACG+26] Evan Au, Alex Chernyak, Dillon George, Sean Lawlor, Kevin Lewi, Jasleen Malvai, and Ercan Ozturk. GitHub - facebook/akd: An implementation of an auditable key directory. Available at https://github.com/facebook/akd., 2026.

[BCCK25] Joseph Bonneau, Jessica Chen, Miranda Christ, and Ioanna Karantaidou. Merkle mountain ranges are optimal: On witness update frequency for cryptographic accumulators. In Yael Tauman Kalai and Seny F. Kamara, editors, *CRYPTO 2025, Part II*, volume 16001 of *LNCS*, pages 170–202. Springer, Cham, August 2025.

[BFM24] Nicholas Brandt, Mia Filić, and Sam A. Markelon. A formal treatment of key transparency systems with scalability improvements. Cryptology ePrint Archive, Report 2024/1938, 2024.

[Bon16] Joseph Bonneau. EthIKS: Using Ethereum to audit a CONIKS key transparency log. In Jeremy Clark, Sarah Meiklejohn, Peter Y. A. Ryan, Dan S. Wallach, Michael Brenner, and Kurt Rohloff, editors, *FC 2016 Workshops*, volume 9604 of *LNCS*, pages 95–105. Springer, Berlin, Heidelberg, February 2016.

[CDGM19] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. SEEMless: Secure end-to-end encrypted messaging with less trust. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1639–1656. ACM Press, November 2019.

[CW09] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In Fabian Monrose, editor, *USENIX Security 2009*, pages 317–334. USENIX Association, August 2009.

[EA23] Apple Security Engineering and Architecture. Advancing imessage security: imessage contact key verification. Available at https://security.apple.com/blog/imessage-contact-key-verification/, October 2023.

[Eli74] Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM (JACM)*, 21(2):246–260, 1974.

[Fan71] Robert Mario Fano. *On the number of bits required to implement an associative memory*. Massachusetts Institute of Technology, Project MAC, 1971.

[GHMR18] Sanjam Garg, Mohammad Hajiabadi, Mohammad Mahmoody, and Ahmadreza Rahimi. Registration-based encryption: Removing private-key generator from IBE. In Amos Beimel and Stefan Dziembowski, editors, *TCC 2018, Part I*, volume 11239 of *LNCS*, pages 689–718. Springer, Cham, November 2018.

[Gooa] Google. Binary transparency. Available at https://binary.transparency.dev/.

[Goob] Google. Certificate transparency. Available at https://certificate.transparency.dev/.

[Gooc] Google. Trillian: an open-source verifiable log. Available at https://transparency.dev/.

[HHK+21] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. Merkle^2: A low-latency transparency log system. In *2021 IEEE Symposium on Security and Privacy*, pages 285–303. IEEE Computer Society Press, May 2021.

[HSBB26] Hossein Hafezi, Alireza Shirzad, Benedikt Bünz, and Joseph Bonneau. IronDict: Transparent dictionaries from polynomial commitments. In Benjamin Stock and Elissa M. Redmiles, editors, *USENIX Secuirty 2026*. USENIX Association, August 2026.

[LCG+23] Julia Len, Melissa Chase, Esha Ghosh, Daniel Jost, Balachandar Kesavan, and Antonio Marcedone. ELEKTRA: Efficient lightweight multi-dEvice key TRAnsparency. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 2915–2929. ACM Press, November 2023.

[LCG+24] Julia Len, Melissa Chase, Esha Ghosh, Kim Laine, and Radames Cruz Moreno. OPTIKS: An optimized key transparency system. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024.

[LCL+17] James Larisch, David R. Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. CRLite: A scalable system for pushing all TLS revocations to all browsers. In *2017 IEEE Symposium on Security and Privacy*, pages 539–556. IEEE Computer Society Press, May 2017.

[LL23] Sean Lawlor and Kevin Lewi. Deploying key transparency at whatsapp. Available at https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/, April 2023.

[MBB+15] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: Bringing key transparency to end users. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 383–398. USENIX Association, August 2015.

[MFZE+26] Harjasleen Malvai, Francesca Falzon, Andrew Zitek-Estrada, Sarah Meiklejohn, and Joseph Bonneau. SoK: Cryptographic authenticated dictionaries. In *NDSS 2025*. The Internet Society, February 2026.

[MKKS+23] Harjasleen Malvai, Lefteris Kokoris-Kogias, Alberto Sonnino, Esha Ghosh, Ercan Oztürk, Kevin Lewi, and Sean F. Lawlor. Parakeet: Practical key transparency for end-to-end encrypted messaging. In *NDSS 2023*. The Internet Society, February 2023.

[Pro] Google Transparency Project. Compact ranges. Available at https://github.com/transparency-dev/merkle/blob/main/docs/compact_ranges.md.

[RWC] Whatsapp key transparency. Real World Cryptography 2024. Video available at https://www.youtube.com/watch?v=kDdGsKUMYbE.

[RY16] Leonid Reyzin and Sophia Yakoubov. Efficient asynchronous accumulators for distributed PKI. In Vassilis Zikas and Roberto De Prisco, editors, *SCN 16*, volume 9841 of *LNCS*, pages 292–309. Springer, Cham, August / September 2016.

[TFZ+22] Nirvan Tyagi, Ben Fisch, Andrew Zitek, Joseph Bonneau, and Stefano Tessaro. VeRSA: Verifiable registries with efficient client audits from RSA authenticated dictionaries. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2793–2807. ACM Press, November 2022.

[Tod13] P. Todd. Merkle mountain ranges. opentimestamps. Available at https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md, March 2013.

A Proofs for Forget-me-not Trees

We now prove that the construction given above satisfies correctness, binding, and merge consistency. Throughout, we assume that H is a collision-resistant hash function, that the Merkle tree MT used to Merklize the Bloom filters is a standard binding Merkle tree (i.e., no adversary can produce two distinct data values that verify against the same root in the same index), and that the Bloom filter BF satisfies the no-false-negatives property (see Section 4.1).

Theorem A.1 (Correctness)**.** *The Forget-me-not tree construction satisfies the three correctness properties: membership, non-membership, and merge correctness.*

Correctness is clear by construction. Membership and non-membership simply follow directly from the no-false-negatives property. Merge correctness holds because the logical-or of two Bloom filters can only produce false positives. In the case that there are false positives, the recursion will eventually reveal the leaf nodes which do not contain the data in question.

Theorem A.2 (Binding)**.** *If H is collision-resistant and MT is a binding Merkle tree, then the Forget-me-not tree construction satisfies binding.*

Proof. Our proof proceeds by contradiction. Assume there exists a p.p.t. adversary $\mathcal{A}$ that outputs $(\mathsf{digest}, \mathsf{data}, \pi_1, \pi_2)$ such that both $\mathsf{VerifyMembership}(\mathsf{digest}, \mathsf{data}, \pi_1) = 1$ and $\mathsf{VerifyNonMembership}(\mathsf{digest}, \mathsf{data}, \pi_2) = 1$ with non-negligible probability. We show that this implies $\mathcal{A}$ can produce a collision on H with non-negligible probability.

Since $\mathsf{VerifyMembership}$ accepts π_1, the proof π_1 defines a unique path from a leaf with hash $H(\mathsf{data})$ up to digest. At each internal node along this path, the proof contains a Bloom filter Merkle proof π_{BF} demonstrating that all k bits at positions $\{h_i(\mathsf{data})\}_{i\in[k]}$ are set to 1. In particular, these Merkle proofs authenticate against the Bloom Merkle root r_{BF} committed in each node's label.

Since $\mathsf{VerifyNonMembership}$ accepts π_2, the proof π_2 defines a set of nodes such that each leaf in the tree has an ancestor in this set and each element of the set is either a leaf type showing that the data is not embedded in the leaf or an internal bloom node showing that one of $\{h_i(\mathsf{data})\}_{i\in[k]}$ is 0. In the former case, a collision in the hash function can be extracted from π_1 and π_2, as the leaf node is opened to two different data values. In the latter case, this implies a break in the binding for the Merklization of the Bloom filters. □

Theorem A.3 (Merge Consistency)**.** *If H is collision-resistant,* MT *is a binding Merkle tree, and the Bloom filter* BF *satisfies the no-false-negatives property, then the Forget-me-not tree construction satisfies merge consistency.*

Proof. We prove the first condition (membership in the merged tree despite non-membership in a subtree). Suppose for contradiction that there exists a p.p.t. adversary $\mathcal{A}$ that, with non-negligible probability, outputs $(\mathsf{digest}_1, \mathsf{digest}_2, \mathsf{digest}, \mathsf{data}, \pi_{\text{mem}}, \pi_{\text{non-mem},1}, \pi_{\text{non-mem},2}, \pi_{\text{merge}})$ such that:

(i) $\mathsf{VerifyMerge}(\mathsf{digest}_1, \mathsf{digest}_2, \mathsf{digest}, \pi_{\text{merge}}) = 1$,

(ii) $\mathsf{VerifyMembership}(\mathsf{digest}, \mathsf{data}, \pi_{\text{mem}}) = 1$,

(iii) $\mathsf{VerifyNonMembership}(\mathsf{digest}_1, \mathsf{data}, \pi_{\text{non-mem},1}) = 1$, and,

(iv) $\mathsf{VerifyNonMembership}(\mathsf{digest}_2, \mathsf{data}, \pi_{\text{non-mem},2}) = 1$.

We can use the binding property proved above as a lemma to show that there must be a contradiction. The path for the membership proof must pass through either the subtree with digest digest_1 or digest_2. Without loss of generality, say that the path goes through digest_1. Note that a membership proof for any given root also contains a membership proof that would verify with respect to the child through which the path goes. Thus, we can extract a valid membership proof with respect to digest_1, but this violates binding.

The second condition (non-membership in the merged tree despite membership in a subtree) is slightly more complex. Suppose for contradiction that there exists a p.p.t. adversary $\mathcal{A}$ that, with non-negligible probability, outputs $(\mathsf{digest}_1, \mathsf{digest}_2, \mathsf{digest}, \mathsf{data}, \pi_{\text{mem}}, \pi_{\text{non-mem}}, \pi_{\text{merge}})$ such that with non-negligible probability:

(i) $\mathsf{VerifyMerge}(\mathsf{digest}_1, \mathsf{digest}_2, \mathsf{digest}, \pi_{\text{merge}}) = 1$,

(ii) $\mathsf{VerifyNonMembership}(\mathsf{digest}, \mathsf{data}, \pi_{\text{non-mem}}) = 1$, and

(iii) $\exists\ j \in \{1, 2\}$ such that $\mathsf{VerifyMembership}(\mathsf{digest}_j, \mathsf{data}, \pi_{\text{mem}}) = 1$

There are two cases to consider: either the set for (ii) contains the new root or contains only other nodes.

Case 1: $\pi_{non\text{-}mem}$ contains the root: WLOG, assume that π_{mem} provides a path through the node with digest digest_1. This means that π_{mem} contains a Merkle proof showing that all the bits at the indexes $\{h_i(\mathsf{data})\}_{i \in [k]}$ are set to 1 within the Bloom filter embedded into the node with digest digest_1. Because π_{merge} checks that the new root's Bloom filter is honestly computed, all bits in the children's Bloom filters that are set to 1 are also set to 1 in the root's filter. But, this means that producing a Merkle proof that one of these indexes is 0 for the root node breaks the Merkle binding property.

Case 2: $\pi_{non\text{-}mem}$ contains only other nodes: This case reduces to the same argument as the first condition, as there must be a shared node between the membership and non-membership proofs, thus breaking binding.

Therefore, in all cases, $\mathcal{A}$'s success implies a contradiction. □

B Epoch-Aware Construction

B.1 Formal definitions for forest construction and epoch awareness

In Section 5, we defined the Forget-me-not Tree over arbitrary data inputs and assumed that the number of leaves is a perfect power of two. Here, we formalize the modifications described in Section 6.1: (1) the data structure operates over $(\mathsf{label}, \mathsf{val})$ pairs, where only label is inserted into the Bloom filter; (2) an arbitrary number of leaves is supported via a Merkle forest representation; (3) an epoch counter is embedded in every node label; and (4) a new epoch-aware non-membership proof enables efficient queries for the most recent version of a label.

Definition. A *Forget-me-not forest* $F = \{t_1, \ldots, t_\ell\}$ is an ordered set of perfect binary Forget-me-not trees with sizes $2^{s_1} > 2^{s_2} > \cdots > 2^{s_\ell}$, corresponding to the binary decomposition of $n = 2^{s_1} + \cdots + 2^{s_\ell}$. The digest of a forest is the ordered tuple of root digests $\mathsf{digest} = (\mathsf{digest}_1, \ldots, \mathsf{digest}_\ell)$. Each node additionally stores an epoch field indicating the epoch in which it was created.

A Forget-me-not Forest is the following tuple of algorithms:

- $\mathsf{FMNForest.Merklize}((\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n), \mathsf{epoch}) \to (\mathsf{digest}, \mathsf{st})$ takes in a sequence of label-value pairs $(\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n)$ and an epoch epoch, and outputs a forest digest digest and state st.
- $\mathsf{FMNForest.ProveMembership}(\mathsf{digest}, \mathsf{st}, \mathsf{label}, \mathsf{val}, \mathsf{epoch}) \to \{\pi, \perp\}$ takes in a forest digest digest, state st, a label label, a value val, and an epoch epoch and outputs either a proof π or $\perp$ if $(\mathsf{label}, \mathsf{val})$ is not in the forest with epoch epoch.
- $\mathsf{FMNForest.VerifyMembership}(\mathsf{digest}, \mathsf{label}, \mathsf{val}, \mathsf{epoch}, \pi) \to \{0, 1\}$ takes in a forest digest digest, a label label, a value val, an epoch epoch, and a proof π and outputs either 0 or 1.
- $\mathsf{FMNForest.ProveNonMembership}(\mathsf{digest}, \mathsf{st}, \mathsf{label}, \mathsf{epoch}) \to \{\pi, \perp\}$ takes in a forest digest digest, state st, a label label, and an epoch epoch, and outputs a proof π.
- $\mathsf{FMNForest.VerifyNonMembership}(\mathsf{digest}, \mathsf{label}, \mathsf{epoch}, \pi) \to \{0, 1\}$ takes in a label label, a digest digest, an epoch epoch, and a proof π and outputs either 0 or 1.
- $\mathsf{FMNForest.MergeAndProve}(\mathsf{digest}_1, \mathsf{st}_1, \mathsf{digest}_2, \mathsf{st}_2, \mathsf{epoch}) \to (\mathsf{digest}, \mathsf{st}, \pi)$ takes in two digests $\mathsf{digest}_1, \mathsf{digest}_2$, their corresponding states $\mathsf{st}_1, \mathsf{st}_2$, and an epoch epoch and outputs a new digest digest, a new state st, and a proof π that the merge was performed correctly.
- $\mathsf{FMNForest.VerifyMerge}(\mathsf{digest}_1, \mathsf{digest}_2, \mathsf{digest}, \mathsf{epoch}, \pi) \to \{0, 1\}$ takes in two forest digests $\mathsf{digest}_1, \mathsf{digest}_2$, a merged forest digest, an epoch epoch and a proof π and outputs either 0 or 1.

These algorithms provide the following properties:

Correctness. The Forget-me-not Forest construction satisfies the following correctness properties:

(1) *Epoch-Aware Membership:* For all sequences of label-value pairs $(\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n)$, all epochs epoch, and all $i \in [n]$,

$$\Pr\left[\mathsf{VerifyMembership}(\mathsf{digest}, \mathsf{label}_i, \mathsf{val}_i, \mathsf{epoch}, \pi) = 1 \;\middle|\; \begin{array}{c}(\mathsf{digest}, \mathsf{st}) \leftarrow \mathsf{Merklize}((\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n), \mathsf{epoch}) \\ \pi \leftarrow \mathsf{ProveMembership}(\mathsf{digest}, \mathsf{st}, \mathsf{label}_i, \mathsf{val}_i, \mathsf{epoch})\end{array}\right] = 1$$

(2) *Epoch-Aware Non-Membership:* For all sequences of label-value pairs, all epochs $\mathsf{epoch}, \mathsf{epoch}'$ and all $\mathsf{label}^* \notin \{\mathsf{label}_1, \ldots, \mathsf{label}_n\}$

$$\Pr\Big[\mathsf{VerifyNonMembership}(\mathsf{digest}, \mathsf{label}^*, \mathsf{epoch}', \pi) = 1 \;\Big|\; \begin{array}{c}(\mathsf{digest}, \mathsf{st}) \leftarrow \mathsf{Merklize}((\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n), \mathsf{epoch}) \\ \pi \leftarrow \mathsf{ProveNonMembership}(\mathsf{digest}, \mathsf{st}, \mathsf{label}^*, \mathsf{epoch})\end{array}\Big] = 1$$

(3) *Merge:* For all honestly constructed forests with digests $\mathsf{digest}_1, \mathsf{digest}_2$ and states $\mathsf{st}_1, \mathsf{st}_2$,

$$\Pr\left[\mathsf{VerifyMerge}(\mathsf{digest}_1, \mathsf{digest}_2, \mathsf{digest}, \mathsf{epoch}, \pi) = 1 \mid (\mathsf{digest}, \mathsf{st}, \pi) \leftarrow \mathsf{MergeAndProve}(\mathsf{digest}_1, \mathsf{st}_1, \mathsf{digest}_2, \mathsf{st}_2, \mathsf{epoch})\right] = 1$$

Binding. We require that a malicious party that computes digest for the Forget-me-not forest cannot produce both a membership and non-membership proof for any label label such that the membership occurs in an epoch equal to or later than the epoch used for the non-membership proof. Formally, for all p.p.t. adversaries $\mathcal{A}$

$$\Pr\left[\begin{array}{c}\mathsf{VerifyMembership}(\mathsf{digest}, \mathsf{label}, \mathsf{val}, \mathsf{epoch}_{\text{mem}}, \pi_1) = 1 \text{ and} \\ \mathsf{VerifyNonMembership}(\mathsf{digest}, \mathsf{label}, \mathsf{epoch}_{\text{non-mem}}, \pi_2) = 1 \text{ and} \\ \mathsf{epoch}_{\text{non-mem}} \leq \mathsf{epoch}_{\text{mem}}\end{array} \;\middle|\; (\mathsf{digest}, \mathsf{label}, \mathsf{val}, \mathsf{epoch}_{\text{mem}}, \mathsf{epoch}_{\text{non-mem}}, \pi_1, \pi_2) \leftarrow \mathcal{A}(1^\lambda)\right] \leq \mathsf{negl}(\lambda)$$

Merge consistency. When we merge two trees in an epoch epoch, the new resulting digest must respect the membership and non-membership properties of the subtrees. Formally, for all p.p.t. adversaries $\mathcal{A}$:

$$\Pr\left[\begin{array}{c}\mathsf{VerifyMerge}(\mathsf{digest}_1, \mathsf{digest}_2, \mathsf{digest}, \mathsf{epoch}, \pi_{\text{merge}}) = 1 \text{ and} \\ \mathsf{VerifyMembership}(\mathsf{digest}, \mathsf{label}, \mathsf{val}, \mathsf{epoch}_{\text{mem}}, \pi_{\text{mem}}) = 1 \text{ and} \\ \mathsf{VerifyNonMembership}(\mathsf{digest}_1, \mathsf{label}, \mathsf{epoch}_{\text{non-mem},1}, \pi_{\text{non},1}) = 1 \text{ and} \\ \mathsf{VerifyNonMembership}(\mathsf{digest}_2, \mathsf{label}, \mathsf{epoch}_{\text{non-mem},2}, \pi_{\text{non},2}) = 1 \text{ and} \\ \mathsf{epoch}_{\text{non-mem},1} \leq \mathsf{epoch}_{\text{mem}} \text{ and } \mathsf{epoch}_{\text{non-mem},2} \leq \mathsf{epoch}_{\text{mem}}\end{array} \;\middle|\; \begin{array}{c}(\mathsf{digest}_1, \mathsf{digest}_2, \mathsf{digest}, \mathsf{label}, \mathsf{val}, \\ \mathsf{epoch}, \mathsf{epoch}_{\text{mem}}, \mathsf{epoch}_{\text{non-mem},1}, \\ \mathsf{epoch}_{\text{non-mem},2}, \pi_{\text{merge}}, \pi_{\text{mem}}, \\ \pi_{\text{non},1}, \pi_{\text{non},2}) \leftarrow \mathcal{A}(1^\lambda)\end{array}\right] \leq \mathsf{negl}(\lambda)$$

and symmetrically,

$$\Pr\left[\begin{array}{c}\mathsf{VerifyMerge}(\mathsf{digest}_1, \mathsf{digest}_2, \mathsf{digest}, \mathsf{epoch}, \pi_{\text{merge}}) = 1 \text{ and} \\ \mathsf{VerifyNonMembership}(\mathsf{digest}, \mathsf{label}, \mathsf{epoch}_{\text{non-mem}}, \pi_{\text{non-mem}}) = 1 \text{ and} \\ \left(\begin{array}{c}\mathsf{VerifyMembership}(\mathsf{digest}_1, \mathsf{label}, \mathsf{val}, \mathsf{epoch}_{\text{mem}}, \pi_{\text{mem}}) = 1 \text{ or} \\ \mathsf{VerifyMembership}(\mathsf{digest}_2, \mathsf{label}, \mathsf{val}, \mathsf{epoch}_{\text{mem}}, \pi_{\text{mem}}) = 1\end{array}\right) \\ \text{and } \mathsf{epoch}_{\text{non-mem}} \leq \mathsf{epoch}_{\text{mem}}\end{array} \;\middle|\; \begin{array}{c}(\mathsf{digest}_1, \mathsf{digest}_2, \mathsf{digest}, \mathsf{label}, \mathsf{val}, \\ \mathsf{epoch}, \mathsf{epoch}_{\text{mem}}, \mathsf{epoch}_{\text{non-mem}}, \\ \pi_{\text{merge}}, \pi_{\text{mem}}, \\ \pi_{\text{non-mem}}) \leftarrow \mathcal{A}(1^\lambda)\end{array}\right] \leq \mathsf{negl}(\lambda)$$

B.2 Construction

We give formal descriptions of the Forget-me-not forest algorithms in Algorithms 7 to 10. These are direct modifications of the single-tree algorithms in Algorithms 1 to 4, extended to support label-value pairs, epochs, and the forest representation.

Algorithm 7 Building a Forget-me-not Forest

procedure $\mathsf{Merge}(t_{\text{left}}, t_{\text{right}}, \mathsf{epoch})$
  $\texttt{self.f} \leftarrow \mathsf{BF.Merge}(t_{\text{left}}.\texttt{f}, t_{\text{right}}.\texttt{f})$
  $\texttt{self.digest} \leftarrow H(H(t_{\text{left}}.\texttt{digest} \| t_{\text{right}}.\texttt{digest}) \| \mathsf{MT.Merklize}(\texttt{self.f}) \| \mathsf{epoch})$
  $\texttt{self.epoch} \leftarrow \mathsf{epoch}$
  $\texttt{self.left} \leftarrow t_{\text{left}}$, $\texttt{self.right} \leftarrow t_{\text{right}}$
  return self
end procedure

procedure $\mathsf{TreeBuilder}((\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n), \mathsf{epoch})$
  if $n = 1$ **then**
    $\mathsf{f} \leftarrow \mathsf{BF.Setup}()$
    $\texttt{self.f} \leftarrow \mathsf{BF.Add}(\mathsf{f}, \mathsf{label}_1)$ ▷ Only label added to the Bloom filter
    $\texttt{self.digest} \leftarrow H(H(\mathsf{label}_1 \| \mathsf{val}_1) \| \mathsf{MT.Merklize}(\texttt{self.f}) \| \mathsf{epoch})$
    $\texttt{self.epoch} \leftarrow \mathsf{epoch}$
    $\texttt{self.left} \leftarrow \perp$, $\texttt{self.right} \leftarrow \perp$
    return self
  else
    $t_{\text{left}} \leftarrow \mathsf{TreeBuilder}((\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_{\frac{n}{2}}, \mathsf{val}_{\frac{n}{2}}), \mathsf{epoch})$
    $t_{\text{right}} \leftarrow \mathsf{TreeBuilder}((\mathsf{label}_{\frac{n}{2}+1}, \mathsf{val}_{\frac{n}{2}+1}), \ldots, (\mathsf{label}_n, \mathsf{val}_n), \mathsf{epoch})$
    return $\mathsf{Merge}(t_{\text{left}}, t_{\text{right}}, \mathsf{epoch})$
  end if
end procedure

procedure $\mathsf{FMNForest.Merklize}((\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n), \mathsf{epoch})$
  Let $n = 2^{s_1} + \cdots + 2^{s_\ell}$ with $s_1 > \cdots > s_\ell \geq 0$
  $\mathsf{idx} \leftarrow 1$
  for $j = 1, \ldots, \ell$ **do**
    $t_j \leftarrow \mathsf{FMNForest.TreeBuilder}((\mathsf{label}_{\mathsf{idx}}, \mathsf{val}_{\mathsf{idx}}), \ldots, (\mathsf{label}_{\mathsf{idx}+2^{s_j}-1}, \mathsf{val}_{\mathsf{idx}+2^{s_j}-1}), \mathsf{epoch})$
    $\mathsf{idx} \leftarrow \mathsf{idx} + 2^{s_j}$
  end for
  return $(t_1.\texttt{digest}, \ldots, t_\ell.\texttt{digest})$, $\{t_1, \ldots, t_\ell\}$
end procedure

B.3 Proofs

We now prove that the Forget-me-not Forest construction satisfies correctness, binding, and merge consistency. Throughout, we assume that H is a collision-resistant hash function, that the Merkle tree MT used to Merklize the Bloom filters is a standard binding Merkle tree (i.e., no adversary can produce two distinct values that verify against the same root at the same index), and that the Bloom filter BF satisfies the no-false-negatives property (see Section 4.1). These are the same assumptions as in Section 5; the proofs here generalize those in the base construction to account for label-value pairs, epochs, the epoch invariant, and the forest representation.

Theorem B.1 (Forest Correctness)**.** *The Forget-me-not Forest construction satisfies membership correctness, epoch-aware non-membership correctness, and merge correctness.*

Correctness follows by construction, generalizing the argument for the base Forget-me-not Tree (Section 5). For membership, the only changes are the leaf hash format $H(\mathsf{label} \| \mathsf{val} \| \mathsf{epoch})$ and the epoch invariant check; since all nodes in a single $\mathsf{Merklize}$ call share the same epoch, the invariant $\mathsf{epoch}_{\mathsf{par}} \geq \mathsf{epoch}_{\mathsf{child}}$ holds trivially, and the Bloom filter checks follow from the no-false-negatives property exactly as before. For epoch-aware non-membership, a depth-first traversal processes each tree in the forest, and the recursive $\mathsf{VerifyNode}$ reconstructs every node digest in exactly the form produced by the builder (Algorithm 7). At each internal node, the proof either terminates via epoch (the verifier checks $\mathsf{epoch}_{\mathsf{node}} < \mathsf{epoch}_{\mathsf{target}}$, which holds by construction), terminates via

Algorithm 8 Forest Membership Proofs

procedure FMNForest.ProveMembership(digest, st, label, val, epoch)
  $\mathtt{f} \leftarrow \mathsf{BF.Setup}(), \mathtt{f} \leftarrow \mathsf{BF.Add}(\mathtt{f}, \mathsf{label})$
  $d \leftarrow H(H(\mathsf{label} \| \mathsf{val}) \| \mathsf{MT.Merklize}(\mathtt{f}) \| \mathsf{epoch})$
  Find tree $t_j \in \mathsf{st}$ and leaf ℓ in t_j with index index where $\ell.\mathtt{digest} = d$; **return** $\perp$ if none
  $\pi \leftarrow (j, \mathsf{index}, \ell.\mathtt{epoch})$, $\mathtt{node} \leftarrow \ell$
  while node is not the root of t_j **do**
    Let par be the parent of node and sib be its sibling
    $\forall i \in [k],\ (d_{\mathsf{BF}}, \pi_{\mathsf{BF},i}) \leftarrow \mathsf{MT.Prove}(\mathtt{par.f}, h_i(\mathsf{label}), 1)$ ▷ Prove on label, not val
    $\pi \leftarrow \pi \| (\mathtt{sib.digest}, d_{\mathsf{BF}}, \pi_{\mathsf{BF},1}, \ldots, \pi_{\mathsf{BF},k}, \mathtt{node.epoch})$
    $\mathtt{node} \leftarrow \mathtt{par}$
  end while
  return π
end procedure

procedure FMNForest.VerifyMembership(digest, label, val, epoch, π)
  Parse the tree index j, the index index, and leaf epoch $\mathsf{epoch}_{\mathtt{current}}$ from π
  $\mathtt{f} \leftarrow \mathsf{BF.Setup}(), \mathtt{f} \leftarrow \mathsf{BF.Add}(\mathtt{f}, \mathsf{label})$
  $d \leftarrow H(H(\mathsf{label} \| \mathsf{val}) \| \mathsf{MT.Merklize}(\mathtt{f}) \| \mathsf{epoch})$
  assert$(\mathsf{epoch} = \mathsf{epoch}_{\mathtt{current}})$
  for $(d_{\mathtt{sib}}, d_{\mathsf{BF}}, \pi_{\mathsf{BF},1}, \ldots, \pi_{\mathsf{BF},k}, \mathsf{epoch}_{\mathtt{node}}) \in \pi$ **do**
    assert$(\mathsf{epoch}_{\mathtt{node}} \geq \mathsf{epoch}_{\mathtt{current}})$
    assert$(\mathsf{MT.Verify}(d_{\mathsf{BF}}, \pi_{\mathsf{BF},i}, h_i(\mathsf{label}), 1) = 1, i \in [k])$
    if $\mathtt{dir} = \mathtt{left}$ **then** ▷ Compute dir based on index
      $d \leftarrow H(H(d \| d_{\mathtt{sib}}) \| d_{\mathsf{BF}} \| \mathsf{epoch}_{\mathtt{node}})$
    else
      $d \leftarrow H(H(d_{\mathtt{sib}} \| d) \| d_{\mathsf{BF}} \| \mathsf{epoch}_{\mathtt{node}})$
    end if
    $\mathsf{epoch}_{\mathtt{current}} \leftarrow \mathsf{epoch}_{\mathtt{node}}$
  end for
  return $d = \mathsf{digest}[j]$
end procedure

bloom (a zero bit at some $j \in \{h_i(\mathsf{label}^*)\}_{i \in [k]}$ exists since label^* was never added), or recurses into children. At a leaf, the verifier recomputes the leaf digest $H(H(\mathsf{label}_\ell \| \mathsf{val}_\ell) \| \mathsf{MT.Merklize}(\mathtt{f}) \| \mathsf{epoch}_\ell)$ and checks $\mathsf{label}_\ell \neq \mathsf{label}^*$, which holds since label^* is not among the inserted labels. For merge, the forest merge (Algorithm 10) performs binary addition of tree sizes; each pairwise tree merge and the resulting digest are recomputed identically by the verifier.

Theorem B.2 (Forest Binding)**.** *If H is collision-resistant and* MT *is a binding Merkle tree, then the Forget-me-not Forest construction satisfies binding.*

Proof. Our proof proceeds by contradiction. Assume there exists a p.p.t. adversary $\mathcal{A}$ that, with non-negligible probability, outputs $(\mathsf{digest}, \mathsf{label}, \mathsf{val}, \mathsf{epoch}_{\text{mem}}, \mathsf{epoch}_{\text{non-mem}}, \pi_1, \pi_2)$ such that the membership proof verifies, the non-membership proof verifies, and the epoch used for the membership proof $\mathsf{epoch}_{\text{mem}}$ is later than the epoch used for non-membership $\mathsf{epoch}_{\text{non-mem}}$.

Since FMNForest.VerifyMembership accepts π_1, the proof π_1 identifies a tree index j and defines a path from a leaf with the appropriate hash up to the j^{th} root in digest. At each internal node along this path, the proof contains Bloom filter Merkle proofs demonstrating that all k bits at positions $\{h_i(\mathsf{label})\}_{i \in [k]}$ are set to 1. Moreover, the epoch invariant check ensures that every node along this path has epoch greater or equal to its child. We also know that the epoch of the leaf is $\mathsf{epoch}_{\text{mem}}$, so the epoch of each node along this path must be $\geq \mathsf{epoch}_{\text{mem}}$. Since FMNForest.VerifyNonMembership accepts π_2, the proof π_2 includes a set of nodes such that each leaf in the tree has an ancestor in the set and for each node in the set either (1) the epoch of the node is $< \mathsf{epoch}_{\text{non-mem}}$, or (2) there is a Merkle proof demonstrating that at least one of the k bits at positions $\{h_i(\mathsf{label})\}_{i \in [k]}$ is 0.

Algorithm 9 Epoch-Aware Non-Membership Proofs

procedure FMNForest.ProveNonMembership(digest, st, label, $\text{epoch}_{\text{target}}$)
  $\pi \leftarrow \varepsilon$
  for each tree $t_j \in$ st **do** ▷ Process each forest tree in order
    $S \leftarrow \{t_j.\texttt{root}\}$
    while $S \neq \emptyset$ **do**
      node $\leftarrow S$.POP()
      if node is a leaf **then**
        $\pi \leftarrow \pi \| (\texttt{type=leaf}, \texttt{node.label}, \texttt{node.val}, \texttt{node.epoch})$
      else if $\texttt{node.epoch} < \text{epoch}_{\text{target}}$ **then** ▷ Epoch early termination
        $d_{\texttt{inner}} \leftarrow H(\texttt{node.left.digest} \| \texttt{node.right.digest})$
        $d_{\text{BF}} \leftarrow \text{MT.Merklize}(\texttt{node.f})$
        $\pi \leftarrow \pi \| (\texttt{type=epoch}, d_{\texttt{inner}}, d_{\text{BF}}, \texttt{node.epoch})$
      else if $\text{BF.Contains}(\texttt{node.f}, \text{label}) = 0$ **then**
        Pick $j \in \{h_i(\text{label})\}_{i \in [k]}$ such that $\texttt{node.f}[j] = 0$
        $\pi_{\text{BF}} \leftarrow \text{MT.Prove}(\texttt{node.f}, j, 0)$
        $d_{\texttt{inner}} \leftarrow H(\texttt{node.left.digest} \| \texttt{node.right.digest})$
        $\pi \leftarrow \pi \| (\texttt{type=bloom}, d_{\texttt{inner}}, j, \pi_{\text{BF}}, \texttt{node.epoch})$
      else ▷ False positive: recurse into children
        $d_{\text{BF}} \leftarrow \text{MT.Merklize}(\texttt{node.f})$
        $\pi \leftarrow \pi \| (\texttt{type=recurse}, d_{\text{BF}}, \texttt{node.epoch})$
        $S.\text{PUSH}(\texttt{node.right}),\ S.\text{PUSH}(\texttt{node.left})$
      end if
    end while
  end for
  return π
end procedure

procedure FMNForest.VerifyNonMembership(digest, label, $\text{epoch}_{\text{target}}, \pi$)
  Parse π as entries $e_1, \ldots, e_n$ and digest as $(\text{digest}[1], \ldots, \text{digest}[\ell])$
  $i \leftarrow 1$
  for $j = 1, \ldots, \ell$ **do** ▷ One subtree per forest root
    $(d, i) \leftarrow \text{VerifyNode}(\pi, i, \text{label}, \text{epoch}_{\text{target}})$
    assert$(d = \text{digest}[j])$
  end for
  assert$(i = n + 1)$ ▷ All proof entries consumed
  return 1
end procedure

procedure VerifyNode(π, i, label, $\text{epoch}_{\text{target}}$)
  if $e_i = (\texttt{type=leaf}, \text{label}_\ell, \text{val}_\ell, \text{epoch}_\ell)$ **then**
    assert$(\text{label}_\ell \neq \text{label})$
    $\texttt{f} \leftarrow \text{BF.Setup}(),\ \texttt{f} \leftarrow \text{BF.Add}(\texttt{f}, \text{label}_\ell)$
    $d_\ell \leftarrow H(H(\text{label}_\ell \| \text{val}_\ell) \| \text{MT.Merklize}(\texttt{f}) \| \text{epoch}_\ell)$
    return $(d_\ell, i + 1)$
  else if $e_i = (\texttt{type=epoch}, d_{\texttt{inner}}, d_{\text{BF}}, \text{epoch}_{\text{node}})$ **then**
    assert$(\text{epoch}_{\text{node}} < \text{epoch}_{\text{target}})$
    return $(H(d_{\texttt{inner}} \| d_{\text{BF}} \| \text{epoch}_{\text{node}}), i + 1)$
  else if $e_i = (\texttt{type=bloom}, d_{\texttt{inner}}, j, \pi_{\text{BF}}, \text{epoch}_{\text{node}})$ **then**
    assert$(j \in \{h_{i'}(\text{label})\}_{i' \in [k]})$
    assert$(\text{MT.Verify}(\pi_{\text{BF}}.\texttt{digest}, \pi_{\text{BF}}.\pi, j, 0) = 1)$
    return $(H(d_{\texttt{inner}} \| \pi_{\text{BF}}.\texttt{digest} \| \text{epoch}_{\text{node}}), i + 1)$
  else ▷ $e_i = (\texttt{type=recurse}, d_{\text{BF}}, \text{epoch}_{\text{node}})$
    $(d_{\texttt{left}}, i') \leftarrow \text{VerifyNode}(\pi, i + 1, \text{label}, \text{epoch}_{\text{target}})$
    $(d_{\texttt{right}}, i'') \leftarrow \text{VerifyNode}(\pi, i', \text{label}, \text{epoch}_{\text{target}})$
    return $(H(H(d_{\texttt{left}} \| d_{\texttt{right}}) \| d_{\text{BF}} \| \text{epoch}_{\text{node}}), i'')$
  end if
end procedure

Because every leaf has an ancestor in the set in π_2 and there is a valid path from the leaf to the root in π_1, there must be at least one node shared between π_1 and π_2. There are 3 cases for this node:

Algorithm 10 Forest Merge Proofs

procedure FMNForest.MergeAndProve($\mathsf{digest}_1, \mathsf{st}_1, \mathsf{digest}_2, \mathsf{st}_2, \mathsf{epoch}$)
  Let $F_1 = \{t_{1,1}, \ldots, t_{1,a}\}$ and $F_2 = \{t_{2,1}, \ldots, t_{2,b}\}$ be the two forests
  $F \leftarrow \emptyset,\ \pi \leftarrow \varepsilon,\ \texttt{carry} \leftarrow \perp$
  for each tree size 2^s from smallest to largest **do**
    Collect all trees of size 2^s: from F_1, from F_2, and `carry` (if any)
    if exactly one tree t of size 2^s **then**
      $F \leftarrow F \cup \{t\}$
    else if exactly two trees t, t' of size 2^s **then**
      $\texttt{carry} \leftarrow \mathsf{Merge}(t, t', \mathsf{epoch})$
      $\pi_t \leftarrow (t.\texttt{f}, H(t.\texttt{left.digest} \| t.\texttt{right.digest}), t.\mathsf{epoch})$
      $\pi_{t'} \leftarrow (t'.\texttt{f}, H(t'.\texttt{left.digest} \| t'.\texttt{right.digest}), t'.\mathsf{epoch})$
      $\pi \leftarrow \pi \| (\pi_t, \pi_{t'})$
    else if three trees t, t', t'' of size 2^s **then** ▷ Two from forests + carry
      $F \leftarrow F \cup \{t''\}$ ▷ Keep one, merge the other two
      $\texttt{carry} \leftarrow \mathsf{Merge}(t, t', \mathsf{epoch})$
      Append the merge proof for (t, t') to π as above
    end if
  end for
  if $\texttt{carry} \neq \perp$ **then**
    $F \leftarrow F \cup \{\texttt{carry}\}$
  end if
  Order F by decreasing tree size
  return ((root digests of F), F, π)
end procedure

procedure FMNForest.VerifyMerge($\mathsf{digest}_1, \mathsf{digest}_2, \mathsf{digest}, \mathsf{epoch}, \pi$)
  Parse the tree sizes from digest_1 and digest_2
  $\texttt{carry} \leftarrow \perp$
  for each tree size 2^s from smallest to largest **do**
    Collect root digests of size 2^s: from digest_1, digest_2, and `carry`
    if exactly one digest d of size 2^s **then**
      Record d in the output forest at size 2^s
    else if exactly two digests d, d' of size 2^s **then**
      Parse the next entry of π as $(\mathsf{f}, d_{\texttt{inner}}, \mathsf{epoch}_t, \mathsf{f}', d'_{\texttt{inner}}, \mathsf{epoch}_{t'})$
      $d_{\mathsf{BF}} \leftarrow \mathsf{MT.Merklize}(\mathsf{f}),\ d'_{\mathsf{BF}} \leftarrow \mathsf{MT.Merklize}(\mathsf{f}')$
      assert$(d = H(d_{\texttt{inner}} \| d_{\mathsf{BF}} \| \mathsf{epoch}_t))$
      assert$(d' = H(d'_{\texttt{inner}} \| d'_{\mathsf{BF}} \| \mathsf{epoch}_{t'}))$
      assert$(\mathsf{epoch} \geq \mathsf{epoch}_t$ and $\mathsf{epoch} \geq \mathsf{epoch}_{t'})$ ▷ Epoch monotonicity
      $d_{\text{merged}} \leftarrow \mathsf{MT.Merklize}(\mathsf{BF.Merge}(\mathsf{f}, \mathsf{f}'))$
      $\texttt{carry} \leftarrow H(H(d \| d') \| d_{\text{merged}} \| \mathsf{epoch})$
    else if three digests d, d', d'' of size 2^s **then**
      Record d'' in the output forest at size 2^s
      Verify (d, d') and compute `carry` as above
    end if
  end for
  if $\texttt{carry} \neq \perp$ **then**
    Record `carry` in the output forest at the appropriate size
  end if
  return output forest digests = digest
end procedure

Case 1: The node in π_2 is a ***`type=epoch`*** *entry.* In this case, the accepting non-membership proof mandates that the epoch associated with the node $\mathsf{epoch} < \mathsf{epoch}_{\text{non-mem}}$. The accepting membership proof mandates that the epoch associated with the node $\mathsf{epoch} \geq \mathsf{epoch}_{\text{mem}}$. But, we also have by assumption that $\mathsf{epoch}_{\text{non-mem}} \leq \mathsf{epoch}_{\text{mem}}$. Substituting into the first equation, we get $\mathsf{epoch} < \mathsf{epoch}_{\text{mem}}$, implying

$$\mathsf{epoch} < \mathsf{epoch}_{\text{mem}} \text{ and } \mathsf{epoch} \geq \mathsf{epoch}_{\text{mem}}$$

which cannot be satisfied, resulting in a contradiction.

Case 2: The node in π_2 is a `type=leaf` *entry.* This implies that the adversary output a hash collision, as in π_1 we check that the node digest corresponds to a hash including label and in π_2 we check that the node digest corresponds to a hash for any label besides label. Thus, there must be a collision. By the collision resistance of the hash function, this can only happen with negligible probability, so this results in a contradiction.

Case 3: The node in π_2 is a `type=bloom` *entry.* Because the membership proof is accepting, then π_1 contains a Merkle proof that all k bits at indices $\{h_i(\mathsf{label})\}_{i\in[k]}$ are 1, but because the non-membership proof is accepting then there exists a k such that the bit at $h_k(\mathsf{label})$ is 0. This breaks the binding property of the Merkle tree. But, this can only happen with negligible probability, causing a contradiction.

Thus, in all cases there is a contradiction, indicating the initial assumption cannot be true. □

Theorem B.3 (Forest Merge Consistency)**.** *If H is collision-resistant, MT is a binding Merkle tree, and the Bloom filter BF satisfies the no-false-negatives property, then the Forget-me-not Forest construction satisfies merge consistency properties.*

Proof. We prove each case separately.

Case 1 (membership in merged forest despite non-membership in both input forests). Suppose for contradiction that all conditions hold with non-negligible probability. The merge proof π_{merge} certifies that digest is a valid merge of digest_1 and digest_2, indicating that the epoch in the root node is not less than the epochs in either child and that the Bloom filter Merklized in the new root is the OR of the Bloom filters in the roots of each subtree. Moreover, we have π_{mem}, $\pi_{\text{non-mem},1}$, $\pi_{\text{non-mem},2}$, each of which is accepting with the conditions that $\mathsf{epoch}_{\text{non-mem},1} \leq \mathsf{epoch}_{\text{mem}}$ and $\mathsf{epoch}_{\text{non-mem},2} \leq \mathsf{epoch}_{\text{mem}}$.

To prove this, we can use Forest Binding as a lemma. Notice that a membership proof with respect to digest contains either a valid membership proof with respect either to digest_1 or digest_2; WLOG assume it is digest_1. But, by Forest Binding, an adversary cannot produce such a proof with non-negligible probability. Thus, we arrive at a contradiction.

Case 2 (non-membership in merged forest despite an accepting membership in either sub-forest). Suppose for contradiction that all conditions hold with non-negligible probability. The merge proof π_{merge} certifies that digest is a valid merge of digest_1 and digest_2, indicating that the epoch in the root node is not less than the epochs in either child and that the Bloom filter Merklized in the new root is the OR of the Bloom filters in the roots of each subtree. Moreover, we have π_{mem} and $\pi_{\text{non-mem}}$, each of which is accepting with the conditions that $\mathsf{epoch}_{\text{non-mem}} \leq \mathsf{epoch}_{\text{mem}}$.

As above, we can use Forest binding to simplify the argument. Notably, the set of nodes in the non-membership proof either contains exactly the root of the merged tree or nodes that are contained within the subtrees rooted by the node with digests digest_1 or digest_2. In either of the latter cases, we can trivially extract a valid non-membership proof for the respective subtree, thereby violating forest binding. Therefore, we need only consider the first case.

Because the root cannot be a leaf, we only consider two cases: the epoch case and the Bloom filter case:

1. *The root is a* `type=epoch` *entry.* If this is an epoch entry, then it must be that the epoch hashed into the root $\mathsf{epoch} < \mathsf{epoch}_{\text{non-mem}}$. WLOG assume that π_{mem} has a path through the node with digest digest_1. Because π_{mem} is accepting, we also know that epoch for the node with

digest digest_1, denoted as epoch_1, must satisfy $\mathsf{epoch}_1 \geq \mathsf{epoch}_{\text{mem}}$. Moreover, because π_{merge} is accepting, we know that $\mathsf{epoch} \geq \mathsf{epoch}_1$. So we now have the following constraints on epoch:

- $\mathsf{epoch} < \mathsf{epoch}_{\text{non-mem}} \leq \mathsf{epoch}_{\text{mem}}$, and
- $\mathsf{epoch} \geq \mathsf{epoch}_1 \geq \mathsf{epoch}_{\text{mem}}$

Thus, there is a contradiction.

2. *The root is a* `type=bloom` *entry.* If this is a Bloom entry, then there is some k such that the index $h_k(\mathsf{label})$ is 0 within the Bloom filter. WLOG assume that π_{mem} has a path through the node with digest digest_1. Because π_{mem} is accepting, we know that all the indexes $\{h_i(\mathsf{label})\}_{i\in[k]}$ must be 1 in Bloom filter embedded into the node with digest digest_1. Moreover, because π_{merge} verified, we know that the Bloom filter of the root is the OR of the Bloom filters in each of the children nodes. As long as the Merkle tree is binding, this yields a contradiction: the bits cannot be 0 in the root but 1 in the children and have π_{merge} verify.

Thus, in all cases there is a contradiction. □

C Proofs for Transparent Dictionaries

We now prove that the Transparent Dictionary construction given in Algorithms 5 and 6 satisfies the completeness and soundness definitions from Section 4.3. All proofs reduce to the security properties of the underlying Forget-me-not Forest construction established in Theorems B.1 to B.3.

Theorem C.1 (Completeness)**.** *If the underlying Forget-me-not Forest construction satisfies forest correctness (Theorem B.1) and forest merge consistency (Theorem B.3), then the Transparent Dictionary construction satisfies completeness (see Section 4.3).*

Completeness holds by construction:

- *Consistency correctness*: If no new tuples with the label label are injected, then the non-membership proof from the underlying Forget-me-not forest will verify.
- *Audit correctness*: This holds by the correctness of the merging for the underlying Forget-me-not trees.

Theorem C.2 (Lookup Soundness)**.** *If the underlying Forget-me-not Forest construction satisfies forest binding (Theorem B.2), then the Transparent Dictionary construction satisfies lookup soundness (see Section 4.3).*

Proof. Suppose for contradiction that a p.p.t. adversary produces two valid lookup proofs π_0 and π_1 for the same label and digest digest_i with $v_0 \neq v_1$ with non-negligible probability. By the structure of $\mathsf{Client.VerifyLookup}$ (Algorithm 6), each proof π_b (for $b \in \{0, 1\}$) consists of:

- A membership proof $\pi_{\text{mem},b}$ for (label, v_b) against the forest digest digest_i, certifying a leaf with epoch epoch_b, and
- Epoch-aware non-membership proofs for label with $\mathsf{epoch}_{\mathsf{target}} = \mathsf{epoch}_b$ against each subtree root that is a sibling along the membership path or another tree in the forest.

Without loss of generality, assume $\mathsf{epoch}_0 \leq \mathsf{epoch}_1$. The leaf $(\mathsf{label}, v_0, \mathsf{epoch}_0)$ lies somewhere in the forest committed by digest_i—specifically, it is stored in some subtree of tree t_{j_0}. Regardless of whether $j_0 = j_1$ (both lookups in the same tree) or $j_0 \neq j_1$ (different trees), the leaf containing $(\mathsf{label}, v_0, \mathsf{epoch}_0)$ is covered by one of the epoch-aware non-membership proofs in π_1: either it lies in a sibling subtree along π_1's membership path (when $j_0 = j_1$) or it lies in a different tree in the forest (when $j_0 \neq j_1$).

Let digest^* be the root digest of the subtree containing the leaf $(\mathsf{label}, v_0, \mathsf{epoch}_0)$. From π_1, we have a valid epoch-aware non-membership proof for label against digest^* with $\mathsf{epoch}_{\mathsf{target}} = \mathsf{epoch}_1$. From π_0, we can extract a valid membership proof for (label, v_0) against digest^* with leaf epoch epoch_0. But, this implies we have both a membership and non-membership proof for the same label, which happens with only negligible probability by Theorem B.2.

□

Theorem C.3 (Consistency Soundness)**.** *If the underlying Forget-me-not Forest construction satisfies forest binding (Theorem B.2) and forest merge consistency (Theorem B.3), and a valid audit proof was verified by an honest auditor for each epoch transition from i to j, then consistency soundness holds (see Section 4.3).*

Proof. Suppose for contradiction that the server outputs some label label along with two values $v_0 \neq v_1$, an epoch $i < t \leq j$ and two corresponding lookup proofs π_0, π_1. Moreover, assume that the server has produced a valid consistency proof π_{consis} with respect to digest_i and digest_j.

By construction, the non-membership proof in π_{consis} covers all leaves that were added after epoch i. But, we assumed that $\mathsf{VerifyLookup}(\mathsf{digest}_t, \mathsf{label}, v_1, \pi_1) = 1$. This implies that the server produced a membership proof from the root of digest_t to a leaf containing the tuple (label, v_1).

Recall that an honest verifier has checked the Update proofs for each epoch between i and j. Verifying this proof means that the new trees must guarantee merge consistency and that the epoch of the new root is at least as late as the epoch of the old root.

Consider the digest_t, for which the server has produced π_1. The node with this leaf must be covered by the non-membership proof in π_{consis} and there must be a membership proof within π_1. This is a contradiction. Moreover, by merge consistency a prover cannot produce such a pair of proofs after any number of merges. Thus, we reach a contradiction.

□

D Use of AI

All ideas for this work were developed independently by the authors, without any input from AI. The only optimization proposed by AI was the use of the Elias-Fano compression scheme rather than a "standard" compression algorithm like `gzip`.

All code in our implementation was written using Claude, but audited by the research team. Specifically, the research team carefully read the *verification* algorithms for all the generated proofs to ensure that the right information was being serialized into all proofs. During this process, the research team found multiple bugs in the generated code, including bugs that affected both efficiency (e.g., serialized proofs were about 2× larger than necessary) and security (e.g., the verifier did not correctly check the invariant that parents' epochs are non-decreasing relative to their children's). While we cannot be certain that all bugs were caught, uncovering meaningful bugs is an indication that the audit process was valuable. Additionally, the research team developed scripts to estimate

the size of all proof types prior to Claude's implementation. The final proof sizes were within 10% of the estimates (serialization overhead and differences in compression performance), indicating that the implementation is likely correct. We believe that the quality of this implementation is comparable to the implementation done by an early-phase PhD student.

Finally, AI was used as a copy-editing tool when drafting this paper. Most of the text was written initially by the authors, with copy editing provided by AI. The exception is that the proofs were initially drafted by AI, but then (rather heavily) edited and re-written by the authors. The tikz visualizations in Figure 1 and Figure 4 were generated using Claude with careful supervision and feedback from the authors.