Merkle^2: A Low-Latency Transparency Log System

Yuncong Hu, Kian Hooshmand*, Harika Kalidhindi*, Seung Jin Yang*, Raluca Ada Popa
University of California, Berkeley
{*yuncong_hu, kianhooshmand, jrharika, seungjin, raluca.popa*}*@berkeley.edu*

***Abstract*—Transparency logs are designed to help users audit untrusted servers. For example, Certificate Transparency (CT) enables users to detect when a compromised Certificate Authority (CA) has issued a fake certificate. Practical state-of-the-art transparency log systems, however, suffer from high monitoring costs when used for low-latency applications. To reduce monitoring costs, such systems often require users to wait an hour or more for their updates to take effect, inhibiting low-latency applications. We propose Merkle^2, a transparency log system that supports both efficient monitoring and low-latency updates. To achieve this goal, we construct a new multi-dimensional, authenticated data structure that nests two types of Merkle trees, hence the name of our system, Merkle^2. Using this data structure, we then design a transparency log system with efficient monitoring and lookup protocols that enables low-latency updates. In particular, all the operations in Merkle^2 are independent of update intervals and are (poly)logarithmic to the number of entries in the log. Merkle^2 not only has excellent asymptotics when compared to prior work, but is also efficient in practice. Our evaluation shows that Merkle^2 propagates updates in as little as 1 second and can support $100\times$ more users than state-of-the-art transparency logs.**

I. Introduction

Interest in transparency logs [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11] has increased in recent years because they promise a trustworthy PKI [12], [13], [14] or certificate infrastructure [3], [15]. For example, *certificate transparency* (CT) [3], [15] enables building accountable PKIs for web applications and is widely deployed. As of March 2021, CT has publicly logged over 12 billion certificates [16]; Google Chrome requires web certificates issued after April 30, 2018 to appear in a CT log [17]. Besides CT, there have been significant efforts into *key transparency* [1], [7], [9], [18], [19] for managing the public keys of end-users and *software transparency* [20], [21], [22], [23], [24], [25] for securing software updates.

Transparency logs provide a consistent, immutable, and append-only log: anybody reading the log entries will see the same entries in the same order, nobody can modify the data already in the log, and parties can only append new data. One of their distinctive features is that they combine aspects of blockchains/ledgers [26], [27], [28], [29], [30], [31], [32], [33] with aspects of traditional centralized hosting. Like blockchains and ledgers, transparency logs rely on *decentralized verification*, enabling anyone to verify their integrity. At the same time, they are hosted traditionally by a *central service provider*, such as Google [3], [5]. Due to guarantees provided by the log and decentralized verification by third parties, the service provider cannot modify or fork the log without detection. Additionally, centralized hosting enables these logs to be significantly more efficient than Bitcoin-like blockchains; they provide higher throughput and lower latency while avoiding expensive proof of work or the expensive replication of the ledger state at many users.

Common transparency logs are append-only logs that provide an efficient *dictionary* for key-value pairs stored in the log. State-of-the-art transparency logs like CONIKS [1] provide the following crucial properties for applications: efficient membership and non-membership proof, and monitoring proof. In particular, when users look up key-value pairs, the server can provide a succinct proof of membership or non-membership to convince users that it returns the correct lookup result. For example, in CT [3], the browser only downloads logarithmic-sized data to check whether a particular website's certificate is in the log. However, unlike CONIKS, CT cannot provide succinct non-membership proofs, so CT cannot support efficient revocation for certificates.

A major impediment to the wider adoption of transparency logs is their high update latency, precluding their use in many low-latency applications. To understand what impacts update latency, one must first understand *monitoring*, a key component of transparency logs. Monitoring allows other parties to monitor the state of an untrusted server and the results it returns to users. The server periodically – every *epoch* – publishes a *digest* summarizing the state of the system. Transparency logs rely on auditors [1], [2], [5], [9] (third-parties or individual users) to keep track of digests published by the server and gossip with each other to prevent server equivocation. Data owners and users who look up data from the log retrieve the digest from the auditors. Given the digest, data owners can check the integrity of their data, and users can check the correctness of the lookup results from the server.

A key challenge in existing, state-of-the-art, practical transparency logs like CONIKS [1] or Key Transparency (KT) [5] is that *every* data owner must monitor their data for *every* published digest. For example, Google's KT proposal [19], [34] cites 1 second as a desirable epoch interval (enabling a variety of applications). For those transparency logs supporting a desired target of 1 billion users, handling each user monitoring every second is too large of a cost for the server both in

*Seung Jin Yang, Harika Kalidhindi, and Kian Hooshmand contributed equally to this work. They are listed in alphabetical order by last name.

bandwidth and computation power. For example, if CONIKS runs with a desirable epoch interval of 1 second [19], [34], every year every user has to download about 65 GB of data to check their data. For 1 billion users, the server has to provide about 65 exabytes of data.

This dependence of the server's cost proportional to the product of users and epochs is what drives existing proposals to set infrequent epochs, e.g., of hours or days. In turn, long epoch intervals affect responsiveness and user experience in applications requiring low-latency updates [18]. We consider PKI as an example. Prior transparency logs [1] have users wait **an hour** to be able to start using the service, as it takes an epoch for new public keys to appear on the log enabling other users to look them up. However, a study [35] shows that the tolerable waiting time for web users is only **two seconds**. For example, users may not want to wait an hour before being able to register and set up IoT devices that generate SK-PK pairs [2], [36], [37], [38]. Also, key owners may not want to wait an hour to revoke compromised keys as the attacker may use the compromised key to steal data or information. Applications like intrusion detection systems [39], [40], [41] aim to stop incidents and revoke malicious accounts immediately.

To reduce the monitoring cost, researchers proposed transparency logs based on heavy cryptographic primitives such as recursive SNARKs [42], [43], [44] or bilinear accumulators [9]. However, these works result in high append latency and memory usage. Other work [2], [8], [10], [11] has auditors check every operation on behalf of users, which results in a high overhead on auditors. We elaborate in Sections IX and X.

Hence, this paper asks the question: Is it possible to build a transparency log system that supports both efficient monitoring and low-latency updates? We propose Merkle^2, a low-latency transparency log system, to answer this question. At the core of Merkle^2 is a new data structure designed to support efficient append, monitoring, and lookup protocols. As a result, the server can publish digests frequently, and data owners do not need to check each digest published by the server. Moreover, data owners only download polylogarithmic-sized data for monitoring throughout the system life.

We implemented Merkle^2 and evaluated it on Amazon EC2. Merkle^2 can sustain an epoch size of 1 second, enabling low-latency applications. For such an epoch size, Merkle^2 can support up to 8×10^6 users per server machine (Amazon EC2 r5.2xlarge instance), which is 100x greater than CONIKS. For this significant increase in monitoring efficiency, the cost of append and lookup in Merkle^2 increases only slightly; as a result, for large epoch size, when the monitoring cost of CONIKS is acceptable, CONIKS may perform slightly better than Merkle^2. For example, when the epoch size is 1 hour, CONIKS can support 2x more users than Merkle^2. However, such a large epoch size is difficult to accommodate in various low-latency applications. In Section VII, we apply Merkle^2 to certificate and key transparency applications and show the benefits it brings in these settings compared to existing transparency logs. We compare Merkle^2's asymptotic complexity with prior systems in Table I.

Works	Storage Cost	Append Cost	Monitoring Cost		Lookup Cost
			Auditor	Owner	
AAD [9]	n	$\log^3(n)$	$\log(n)$		$\log^2(n)$
ECT [8]	n	$\log(n)$	$\mathbf{P}\log(n)$		$\log(n)$
CONIKS [1]	$\mathbf{E}\ n$	$\log(n)$		$\mathbf{E}\log(n)$	$\log(n)$
CONIKS	$n\log(n)$	$\log(n)$		$\mathbf{E}\log(n)$	$\log(n)$
Merkle^2	$n\log(n)$	$\log^2(n)$	$\log(n)$	$\log^2(n)$	$\log^2(n)$

TABLE I: Asymptotic costs of the server in Merkle^2 against other systems. n refers to the number of entries in the log, is the security parameter for AAD, E is the number of epochs, and P is the number of appends between epochs. Red indicates the worst performance in the category. For the storage cost, we measure the number of nodes in data structures throughout the system life. For the monitoring cost, the auditor column refers to the size of proof provided to each auditor per epoch; the owner column refers to the size of proof provided to each data owner for each log entry throughout the system life. We do not consider "collective verification" for ECT since it relies on a different threat model. The original CONIKS design copies and reconstructs the whole data structure in each epoch to enable the data owners, who go offline, to verify their data in epochs they missed. With CONIKS , we optimize CONIKS by leveraging persistent data structures [45], which we discuss in Section IX.

A. Overview of our techniques

Background. Merkle^2 is built upon Merkle trees [46]. Two types of Merkle trees are common in transparency logs: prefix trees (whose leaves are ordered in lexicographic order) as in CONIKS [1] and chronological trees (whose leaves are ordered by time of append) as in CT [3]. We elaborate in Appendix A-C. In prior transparency logs, the server stores data in Merkle trees and publishes the root hash as the digest in every epoch. When users access the data, the server provides an authentication path of the corresponding leaf as proof.

Our data structure. The advantages and limitations of chronological trees and prefix trees are complementary: CT, which is based on chronological trees, does not require users to monitor each digest but cannot provide efficient lookup or revocation; CONIKS can support efficient lookup, but not efficient monitoring. Therefore, a natural question arises: is there a way to combine them to obtain both benefits?

We solve this problem by leveraging ideas from "multi-dimensional" data structures [47]. Merkle^2's data structure (Section IV) consists of nested Merkle trees, with chronological trees in the outer layer. Each internal node of the chronological layer corresponds to a prefix tree, hence the name for our system, Merkle^2. The hash of each data block is stored in a leaf node of a chronological tree, and, for each node from that leaf to the root of the chronological tree, in its corresponding prefix tree. We provide a "pre-build" technique (Section IV-B) to avoid high append time in the worst case.

We show that Merkle^2's data structure has many convenient properties that enable us to design efficient monitoring and lookup protocols. For example, each data block is stored in only $O(\log(n))$ prefix trees, where n is the number of leaves in the chronological layer. Those prefix trees allow us to look up

data blocks based on indices like CONIKS. Also, the structure of the chronological layer allows us to design a monitoring protocol like CT so that data owners do not have to check every digest, but can simply only check the latest digest.

Transparency log system design. By leveraging Merkle^2's data structure, we design the monitoring (Section V) and lookup (Section VI) protocols of our transparency log system.

The first problem we want to avoid is data owners having to monitor their data in every digest. To address this problem, we adapt the consistency proof in CT to provide an extension proof for Merkle^2's nested trees. The consistency proof in CT allows auditors to ensure no certificate is removed or modified when the server publishes a new digest. Using it, domain owners do not need to check every digest in CT. We observe that the consistency proof preserves not only the integrity of leaves, but also the integrity of internal nodes. Because each internal node contains a prefix tree, it is paramount that no leaf node values or internal node contents are changed. Thus, we design an extension proof that allows auditors to ensure the integrity of all existing nodes in future system states. We further show that our extension proof allows data owners to verify *only* the latest digest instead of every digest in history, and be assured that they will see earlier modifications to their data. So far, this proof only ensures that the prefix tree root hash in the internal node remains unmodified, but it does not check the actual content in the prefix tree. Therefore, Merkle^2 has each data owner check contents of $O(n)$ prefix trees.

To further reduce the monitoring cost of each data owner, we require that data owners check only $O(\log(n))$ prefix trees to ensure the membership of their data block. Moreover, because of the extension proof guarantee, the data owner needs to check each of these prefix trees *only once* throughout the life of the system. However, by requiring the data owner to check only $O(\log(n))$ prefix trees, we do not prevent attackers from adding corrupted data blocks for an index that does not belong to them. To solve this problem, we co-design *signature chains*, which enable users to verify data block ownership in lookup results. The security of the signature chain relies on the chronological order maintained by chronological trees.

Finally, we design a lookup protocol for users to verify lookup results efficiently. The protocol involves (non-)membership proofs from only $O(\log(n))$ prefix trees that cover all data blocks in the entire system. Moreover, we design an optimized protocol for looking up only the latest append.

II. System Overview

In this section, we describe the architecture and API of Merkle^2. Merkle^2 is a transparency log system consisting of key-value pairs. We refer to these as **ID-value pairs** to avoid confusion with cryptographic keys. As in CONIKS [1] and Google Key Transparency [5], Merkle^2 indexes data blocks on the ledger based on their ID to support efficient ID lookup.

A. System architecture

Merkle^2's system setup (in Fig. 1) is similar to that of prior transparency logs. Recall that in transparency log systems, time is split into *epochs*. The system consists of a logical server, auditors, and clients, whose roles are described below.

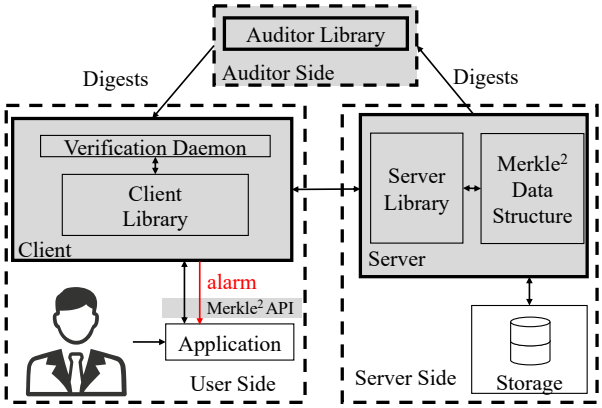

Fig. 1: System overview of Merkle^2. Shaded areas indicate components introduced by Merkle^2.

Server. The (logical) server stores users' ID-value pairs and is responsible for maintaining Merkle^2's data structure and servicing client requests. The server produces proofs for clients and auditors to monitor the system. At the end of each epoch, the server publishes a digest to the auditors summarizing the state of Merkle^2. Every response provided by the server will be signed.

Auditors. Auditors are responsible for verifying that the server provides to clients and other auditors a consistent view of the state. At the end of every epoch, each auditor requests a server-signed digest of the overall state from the server along with a short proof. By verifying the proof, the auditor confirms that the state transition from the previous epoch is valid. It is critical that auditors gossip with each other about the digests to make sure they share a consistent view of the system. If two auditors discover different digests for the same epoch, they can prove server's misbehavior by presenting the signature. Anyone can volunteer to serve as an auditor. We present the monitoring protocol for auditors in Section V. Clients fetch digests from multiple auditors and cross check them.

Clients. Users run the client software, including a client library and a verification daemon. Users' applications interact with the client library to append and look up ID-value pairs through the API shown in Section II-B. Each user is responsible for monitoring their own ID-value pairs on the server. We define the owner of an ID as the person who appends the first value for that ID in the system. Only the ID owner is allowed to append new values for that ID. A user can become the owner of multiple IDs. The verification daemon runs as a separate process that regularly monitors the ID-value pairs of the ID owner. After the first append, the daemon will periodically come online and send monitoring requests to the server.

B. Merkle^2's API

We now explain the API that Merkle^2 provides to application developers wanting to use a transparency log.

$\mathbf{append}(\langle \mathsf{ID}; \mathsf{val} \rangle)$: Appends a new value val for ID to the log. If there does not exist a value for ID before the append, the

user will be identified as the owner of ID. Otherwise, only the owner of ID is allowed to append a new value. We describe in Section V-C how Merkle^2 enforces this condition. The server adds $\langle \mathsf{ID}; \mathsf{val} \rangle$ to both persistent storage and Merkle^2's data structure. We discuss how to maintain Merkle^2's data structure in Section IV-B. The append will not take effect until the server publishes a new digest in the next epoch. For each append, the verification daemon will periodically come online and monitor $\langle \mathsf{ID}; \mathsf{val} \rangle$ on behalf of the ID owner to ensure its integrity. Section V explains the monitoring protocol in detail.

lookup(ID): Looks up all values for ID. The server should return all values for ID appended by the ID owner so far, sorted in chronological order by append time. The lookup result does not contain values appended after the current epoch; but, in this case, the server can notify the user to send another lookup request in the next epoch. We also introduce an extended API that supports **lookup_latest**, which allows users to look up the latest value for ID, because for many applications the most recent value is the only one of interest. This value is the latest append for ID in epochs before the lookup operation. To verify a lookup result, clients must fetch the latest digest from auditors and the lookup proof from the server. We discuss the lookup protocol in Section VI.

III. Threat model and security guarantee

Now, we describe Merkle^2's threat model and guarantees intuitively and then formalize them in Guarantee 1. Merkle^2 protects the system against a malicious attacker who has compromised the server. A hypothetical attacker can modify data or control the protocol running at the server. In particular, it can fork the views of different users [48] or return old data. Merkle^2 guarantees that honest users and auditors will detect these kinds of active attacks.

As in most transparency logs [3], [2], [1], we assume that at least some of the auditors are trusted, which means they will verify the digest and proof published at the end of each epoch and detect forks. Intuitively, the requirement is that any user should be able to reach at least one connected and trusted auditor. Each ID owner is responsible for monitoring its own ID-value pairs. Unlike systems that rely on strong ID owners [1], ID owners in Merkle^2 do not need to monitor every epoch. They can choose a monitoring frequency, like once per day or once per week, and monitor updates in between monitoring periods by checking only the latest epoch. If an ID owner monitors the digest in epoch p without detecting a violation, honest users performing a lookup for ID before epoch p (inclusive) are guaranteed to receive the correct values or to detect a violation.

To define Merkle^2's guarantee, we introduce the following conventions. If the owner appends an ID-value pair in epoch E, other users can look it up starting with epoch $E+1$. The server assigns a *position number* to each ID-value pair to indicate its global order in the system. When users append or lookup, they also receive the position number from the server.

We further define the notion of a *correct lookup result*, as follows. We denote by S_E^{ID} the ordered list of ID-value pairs and corresponding position numbers that the ID owner appended for ID before epoch E, and $S_E^{\emptyset \mathsf{ID}}$ the ordered list that the user received as the lookup result for ID in epoch E. A correct lookup result for ID in epoch E means that the two lists S_E^{ID} and $S_E^{\emptyset \mathsf{ID}}$ are identical.

Merkle^2 provides the following guarantee, for which we provide a proof sketch in Appendix A.

Guarantee 1. *Assume that the hash function used by Merkle^2 is a collision-resistant hash function [49] and that the signature scheme is existentially unforgeable [49].*

For any set of users U, for any set of honest auditors A, for any set of append, lookup and monitoring operations by users in U, for any set of honest users $C \in U$, for any ID *whose owner is in C, let E_1 be the first epoch in which* ID*'s owner appends the first value for* ID*, and let $E_2 > E_1$, for each lookup operation for* ID *performed by a user u in C during epochs $E_1+1 ::: E_2$, if*

Connectivity conditions:

1) *users in C can reach the server and at least one of the auditors in A;*
2) *auditors in A can reach the server and all other auditors in A;*
3) *in every epoch $e \geq 1$, the server outputs the digest of epoch $e-1$ to all the auditors in A.*

Honesty conditions:

4) *the auditors in A follow the monitoring protocol in Section V to gossip with each other and check digests in epochs $1 ::: E_2$;*
5) *the owner of* ID *follows the monitoring protocol in Section V to check its ID-value pairs in epoch E_2.*
6) *whenever a user in C looks up* ID*, it follows the lookup protocol in Section VI for* ID*;*

then, if user u did not receive the correct lookup result for ID during $E_1+1 ::: E_2$, then at least one of the following parties has detected a server violation: users in C (including the ID owner), or auditors in A.

IV. Merkle^2's data structure

In this section, we explain Merkle^2's data structure.

A. Data structure layout

Fig. 2 depicts Merkle^2's data structure. For clarity, we use usernames as IDs, so users append values associated with their usernames in the form of ID-value pairs. Merkle^2's data structure consists of several top-level trees called chronological trees (sorted by time), and for every internal node in this tree, there exists a prefix tree (sorted by IDs). We now elaborate on each type of tree and on how they are nested.

The **Chronological Tree** stores users' ID-value pairs at its leaves from left to right in chronological order. For example, if Alice adds an ID-value pair into the system before Bob, her pair will appear on the left side of Bob's pair. Each ID-value pair is given a *position number*, which indicates the position of the leaf in the tree. For example, in Fig. 2, Alice has two values Val_0 and Val_3, which are assigned to leaves with

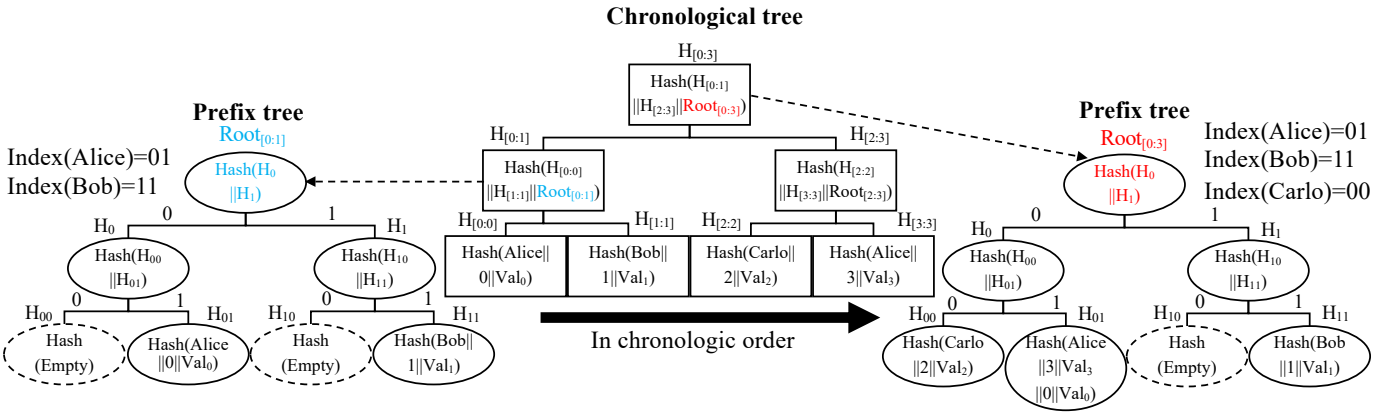

Fig. 2: The center tree is the chronological tree in which internal nodes store root hashes of prefix trees. We denote H the hash value of the node, and Root the root hash of the Merkle prefix tree. Each leaf in the chronological tree has a position number (from 0 to 3). We denote by [X:Y] the node that covers leaves with position number from X to Y. The other two trees are prefix trees with the Merkle root $\mathsf{Root}_{[0;1]}$ and $\mathsf{Root}_{[0;3]}$ respectively. We denote by Index the index of leaves in prefix trees.

position numbers 0 and 3, respectively. The ID-value pair can later be referenced by its position number as a leaf node. Each internal node of the chronological tree has a corresponding prefix tree, as shown in Fig. 2. The hash of each internal node in the chronological tree is the hash of the following triple:

the hashes of its two children in the chronological tree, and

the root hash of the prefix tree corresponding to this node.

This allows the root hash of a chronological tree to summarize the states of all the prefix trees within that chronological tree.

The **Prefix Tree** stores users' ID-value pairs, arranged in lexicographic order by ID. The prefix tree associated with an internal node stores all ID-value pairs that appear inside the subtree rooted at that node. For example, in Fig. 2, the prefix tree corresponding to node [0:1] of the chronological tree stores all children in the subtree of [0:1]. Thus, the prefix tree of [0:1] (depicted on the left of Fig. 2) stores ID-value pairs $\mathsf{Alice}||\mathsf{Val}_0$ and $\mathsf{Bob}||\mathsf{Val}_1$. Meanwhile, the prefix tree on the right in Fig. 2 stores all of the ID-value pairs, because it is associated with the root node [0:3]. If a user appends multiple values, they will all be stored under the same leaf node in the prefix tree because they share the same ID. Since Alice appends two values (Val_0 and Val_3), both are stored in the same leaf node of the prefix tree. The hash of the internal node in a prefix tree is computed as in a typical Merkle tree, where the parent node hash is the hash of left and right child hashes concatenated together ($\mathsf{H}(\mathsf{leftChildHash}||\mathsf{rightChildHash})$).

The reason why we chose the chronological tree as the outer layer is that auditors can check a succinct proof for all the appends between epochs. We elaborate in Section V-A. In contrast, if we use the prefix tree as the outer layer, the size of the proof, which auditors need to check, might be linear to the number of appends between epochs. That is also the reason why CT [3] chooses the chronological tree. Other systems such as CONIKS [1] avoid this overhead by asking each ID owner to monitor every epoch.

The forest of chronological trees. Merkle^2's data structure consists of a forest of chronological trees. Each such tree is full; that is, no leaf is missing. This property is maintained so

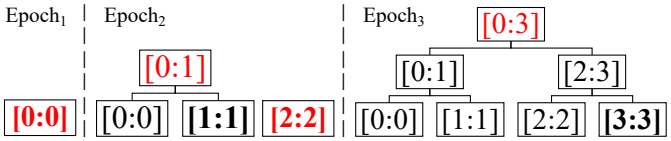

Fig. 3: A forest transition starting from one leaf to four leaves. The red nodes indicate the Merkle roots in the digest. Leaves in bold indicate ID-value pairs added in each epoch.

that as ID-value pairs get appended in epoch_{i+1}, we preserve the structure of the chronological trees from epoch_i. We ensure that in epoch_{i+1}, we only add parents to the existing trees of epoch_i or add separate trees altogether. This helps greatly with our extension proof in Section V-A. Thus, leaves are added by extending Merkle^2's data structure to the right; as more leaves are added, new internal nodes (and therefore new roots) are created whenever we can construct a larger, full binary tree (as discussed in Section IV-B).

Fig. 3 illustrates the transition of Merkle^2's data structure after a set of appends. The internal node [0:1] is automatically created because leaves [0:0] and [1:1] can be stored under a full binary tree. Note that each internal node ([0:1], [2:3], [0:3]) contains the root hash of a prefix tree as shown in Fig. 2. This forest design ensures that the roots of an old forest remain as internal nodes in any later version of the forest, and their hashes will still be the same. For example, the hashes of [0:1] and [2:2] will not change between epoch_2 and epoch_3.

This construction enables us to design a succinct global extension proof and, as shown later, enables users to check *only* the latest version. Since the latest version is an extension of all of the versions before it, no others need to be checked. The **digest** of Merkle^2's data structure contains: (1) the root hashes of the chronological trees in the forest; (2) the total number of leaves. The digest stores $O(\log(n))$ root hashes because n leaves can be stored under $O(\log(n))$ full binary trees, and there are $O(\log(n))$ chronological trees in the forest.

Storage complexity analysis. At first glance, the fact that each internal node of the chronological tree is associated with a prefix tree implies $O(n^2 \log(n))$ storage as there are $O(n)$

internal nodes in the chronological tree and each prefix tree may require $O(n\log(n))$ nodes. However, we observe that the size of most prefix trees is much smaller than $O(n\log(n))$ since each prefix tree only stores a small portion of ID-value pairs. Moreover, each prefix tree can be compressed to require only $O(p)$ nodes where p is the number of ID-value pairs in the prefix tree. We describe the compression algorithm in Appendix C. Now, we consider the storage costs for all prefix trees at each height of the chronological forest. The number of ID-value pairs of all the prefix trees in the same height is $O(n)$; thus, there are $O(n)$ prefix tree nodes at each height of the chronological forest. And there are $O(\log(n))$ levels in total. Therefore, Merkle^2's data structure with n ID-value pairs requires only $O(n\log(n))$ storage.

B. Appending ID-value pairs

To append a new ID-value pair, Merkle^2's data structure first extends the forest by creating a new leaf node containing the ID-value pair. As mentioned above, the ID-value pair is assigned a position number according to the leaf position. Then, Merkle^2's data structure recursively merges the rightmost two chronological trees into one big chronological tree if they have the same size. This process repeats until it is no longer possible to merge the last two trees. For each root node created in the merging process, a corresponding prefix tree must be built by inserting all the ID-value pairs that occur under that root node.

For example, in epoch$_3$ of Fig. 3, the leaf [3:3] is added and the ID-value pair is assigned a position number 3. Nodes [2:3] and [0:3] are created to merge equally-sized chronological trees. The prefix tree of [2:3] is created by adding the ID-value pairs of [2:2] and [3:3]. And, the prefix tree of [0:3] is created by adding all ID-value pairs appended so far.

We now analyze the complexity of appending an ID-value pair. A single append results in the creation of only $O(\log(n))$ internal nodes, but Merkle^2's data structure has to build a new prefix tree for each new internal node. The bottleneck is building new prefix trees, since some prefix trees may have $O(n)$ leaves. However, every leaf node has at most $O(\log(n))$ ancestor nodes, which means each ID-value pair is inserted into $O(\log(n))$ prefix trees. The cost of inserting an ID-value pair into a prefix tree is $O(\log(n))$. Thus, the amortized cost of appending a new ID-value pair is $O(\log^2(n))$.

However, this solution is still impractical for a low-latency system. Suppose there are 2^{20}-1 ID-value pairs in the system; the next append combines all roots into a singular chronological tree, under node [0:2^{20}-1]. Building the corresponding prefix tree, which contains at most 2^{20} leaves, incurs a $O(n)$ cost. Thus, although the amortized cost is $O(\log^2(n))$, some appends results in linear-time operations in the worst case.

To solve this problem, we introduce the pre-build strategy. We first fix the maximum number of ID-value pairs supported by Merkle^2's data structure. This number can be sufficiently large, such as 2^{32}. To append an ID-value pair, Merkle^2's data structure inserts it into all possible prefix trees, including those that do not exist yet but are supposed to be built in the future. In other words, we pre-build prefix trees that may be used in the future. For example, if Merkle^2's data structure supports at most 2^{32} ID-value pairs, we will add an ID-value pair to 32 prefix trees, which correspond to all the existing and future ancestor nodes of the leaf. The cost of each append is still $O(\log^2(n))$, where n now is the maximum number of ID-value pairs, but we avoid the high latency of the worst case operations. We provide more details in Appendix B.

V. Monitoring protocol in Merkle^2

In this section, we show how Merkle^2 performs monitoring efficiently. ID owners are responsible for monitoring their own ID-value pairs, while auditors keep track of digests published by the server in each epoch. The goal of our monitoring protocol is to:

avoid the need to monitor in every single epoch;
enable efficient monitoring in any given epoch.

Intuition. To address the former problem, we design an *efficient extension proof* that allows auditors to prove that every epoch is an extension of the previous one. This way, when an ID owner verifies a monitoring proof for epoch t, the ID owner is implicitly also verifying epochs $t-1;\ldots;1$. Thus, the ID owner need not monitor each digest, only the latest one.

For the latter property, we carefully design a monitoring proof and co-design *signature chains* that enable an ID owner to verify that their values have not been tampered with.

A. Extension proofs

In prior transparency log [1], each ID owner must monitor their ID-value pairs in every epoch as there is no guaranteed relationship between the server's state in different epochs. For example, at epoch$_t$, the server could switch to a corrupted state s' (having some corrupted value for some ID) for this epoch alone and then switch back to the correct state s in epoch$_{t+1}$. Thus, the server is able to equivocate, and ID owners will never detect it if they do not audit the equivocated epoch$_t$.

As a first step in solving this problem, Merkle^2 maintains the invariant that a system state in epoch$_t$ is an extension of the state in epoch$_{t-1}$. In other words, every epoch is an extension of those before it, with the existing ID-value pairs in the same chronological order as before with all the new ID-value pairs occurring after the existing ones. Our extension proof, thus, is designed to be used by auditors to verify these requirements between system states in different epochs.

Each state of the system s_x can be summarized by the root hashes of the trees in its chronological forest. For example, in Fig. 3, the state of the system in epoch$_2$ can be represented by the hashes of nodes [0:1] and [2:2]. To prove that a state s_y is an extension of a state s_x, we must prove that all chronological roots of s_x are contained within those of s_y. By providing the minimum set of hashes necessary, it is possible to compute the root hashes of s_y from those of s_x, thereby proving the extension relationship between the two states.

For example, in Fig. 3, the extension proof between epoch$_2$ and epoch$_3$ contains the following hashes: the chronological tree node hash, $H_{[3:3]}$ and the prefix tree root hashes, $\mathsf{Root}_{[2:3]}; \mathsf{Root}_{[0:3]}$. Given the hashes $H_{[0:1]}; H_{[2:2]}$ from the

old epoch's digest, the auditor can check if the hash $H_{[0:3]}$ in the new digest is computed correctly as follows:

1) compute $H_{[2:3]}$ using $H_{[2:2]}; H_{[3:3]}; Root_{[2:3]}$;
2) compute $H_{[0:3]}$ using $H_{[0:1]}; H_{[2:3]}; Root_{[0:3]}$;
3) check if $H_{[0:3]}$ matches the root hash in the new digest.

At the end of each epoch, auditors receive the new digest and the extension proof from the server. After verifying the extension proof, auditors gossip with each other to ensure that they share a consistent view of the new digest. To reduce server load, auditors can also share extension proofs amongst themselves, since they are checking the same extension proofs.

Extension proofs prevent attackers from removing or modifying existing nodes in Merkle^2's data structure. Once an internal node is created, it will be a part of all future epochs, as they are extensions of the current state. Moreover, because each internal node contains a corresponding prefix tree, all prefix trees that exist in an earlier epoch will remain the *same* for all future epochs. Thus, once an ID owner monitors epoch_t and goes offline for m epochs, the ID owner only has to monitor the latest epoch (epoch_{t+m}), and implicitly verifies the system states in $\text{epoch}_{t+m}; ...; \text{epoch}_{t+1}$. In contrast, CONIKS requires owners to monitor system states in all epochs.

The Merkle^2 extension proof is similar to the consistency proof in CT [3]; thus, the extension proof provides a similar property. There are a few key differences though. A difference is that the extension proof contains root hashes of prefix trees in the path due to the nested Merkle tree design. Another difference is that the consistency proof ensures that leaves are append-only and thus guarantees CT's security. Still, the extension proof alone does not suffice for Merkle^2's security goal because the attacker might compromise newly added prefix trees. Thus, we need monitoring proofs for ID owners to check prefix trees as explained in Section V-B.

Complexity analysis. The size of the extension proof from a state s_x into a state s_y is dependent on the number of hashes required to construct the root hashes of s_y from those of s_x. Because the depth of any chronological root is $O(\log(n))$, there are $O(\log(n))$ ancestor node hashes required to prove the inclusion of the roots of s_x in those of s_y. Thus, the extension proof between two epochs contains $O(\log(n))$ hashes.

B. Monitoring proofs

Monitoring proofs enable ID owners to check contents of prefix trees in Merkle^2's data structure. For concreteness of exposition, consider that Bob wants to monitor his ID, denoted ID_{Bob}. A strawman design for the monitoring proof has Bob check every prefix tree for ID-value pairs matching ID_{Bob}. This way, for each prefix tree, the server provides a (non-)membership proof for ID_{Bob}, which can convince Bob that there are no unwanted changes. Unfortunately, the cost of this strawman is quasilinear in the number of ID-value pairs.

Instead, Merkle^2 requires ID owners to check only the prefix trees that are supposed to store their ID-value pairs. ID owners keep track of position numbers assigned to their ID-value pair; thus, they can infer which prefix trees to check. Given an ID-value pair $\langle \text{ID}; \text{val} \rangle$, we denote by $\mathbf{v}$ the leaf node that stores it in the chronological tree. The only prefix trees that will contain $\langle \text{ID}; \text{val} \rangle$ are those that correspond to the ancestors of $\mathbf{v}$ in the chronological tree. Thus, for each of these prefix trees, the server generates a membership proof for ID to ensure it exists within the prefix tree. Once checked, each membership proof will generate the prefix root hash it belongs to. Note that this mechanism by itself does not prevent a compromised server from adding ID-value pairs (the attacker can add values to the prefix trees that ID owners do not check); our signature chain co-design (Section V-C) addresses this aspect.

The ID owner is not done yet, however, because she must still verify that the generated prefix root hashes are correct. The digest provided by the server only contains the root hashes of the chronological forest. Thus, we must provide the minimum set of hashes so that the ID owner can reproduce the chronological root hash and compare it with the digest. Notice that the generated prefix tree root hashes each correspond to an ancestor of $\mathbf{v}$ in the chronological tree. If we provide the authentication path for $\mathbf{v}$, the ID owner will have enough information to reproduce a digest root hash. Therefore, the monitoring proof for $\langle \text{ID}; \text{val} \rangle$ corresponding to leaf node $\mathbf{v}$ in the chronological tree consists of the following:

membership proofs for prefix trees in ancestor nodes of $\mathbf{v}$;
the authentication path in the chronological tree for $\mathbf{v}$.

The verification process works as follows. Given $\langle \text{ID}; \text{val} \rangle$, its monitoring proof, and the digest, the ID owner begins by computing the root hashes of the prefix trees using the membership proofs. In conjunction with the authentication path, the verifier can reconstruct the hashes of every ancestor of node $\mathbf{v}$ until eventually reaching the root of the chronological tree. Then, the verifier can compare the computed root hash with the corresponding root hash in the digest. For example, in Fig. 2, Bob wants to monitor the ID-value pair $\text{Bob}||\text{Val}_1$. First, Bob computes the prefix root hashes $\text{Root}_{[0:1]}; \text{Root}_{[0:3]}$ using the membership proofs for the prefix trees corresponding to nodes [0:1], [0:3]. Then, together with the authentication path ($H_{[0:0]}; H_{[2:3]}$), Bob can reconstruct and verify the chronological tree root hash $H_{[0:3]}$ with the corresponding hash in the digest.

The protocol described thus far allows ID owners to efficiently monitor Merkle^2 in a particular epoch. Because of the extension proofs in Section V-A, the server cannot modify existing content of Merkle^2 in future epochs. Therefore, if a prefix tree has already been verified by an ID owner, it does not need to be verified again. When ID owners come online and request a monitoring proof, they can specify the latest epoch they have already monitored and only download membership proofs from prefix trees added since that should contain that ID-value pair. This way, a single prefix tree is only checked once by the same ID owner.

Complexity analysis. We now analyze the size of monitoring proofs. For each ID-value pair, we observe that the ID owner only needs to check $O(\log(n))$ prefix trees, because each leaf node in the chronological tree has at most $O(\log(n))$ ancestor nodes. For each of those prefix trees, the membership proof is of size $O(\log(n))$. Note that the ID owner can skip and

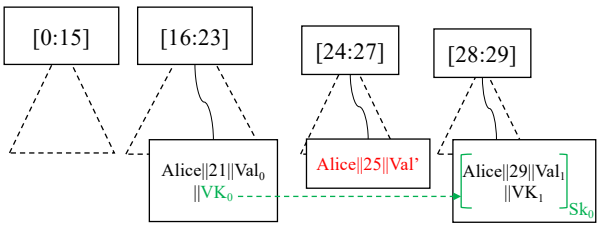

Fig. 4: Alice's values are Val_0 and Val_1. The attacker may add values for the ID "Alice" in the chronological trees [24:27]. The signature chain prevents such attacks.

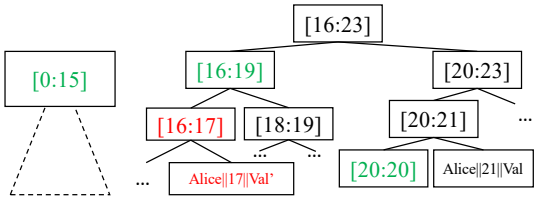

Fig. 5: Alice appends the first value Val for her ID at position 21. Someone (either another honest user or the attacker) has appended Val' for ID "Alice" at position 17 already. Alice will verify the first-value proof, which contains non-membership proofs for ID "Alice" in the prefix trees at the green nodes.

cache prefix trees if they have been checked before. Overall, for each ID-value pair, the ID owner downloads monitoring proofs of total size $O(\log^2(n))$ throughout the system's life.

C. Signature chains design

The monitoring proof discussed in Section V-B only guarantees that the attacker cannot remove ID-value pairs from Merkle^2's data structure. It does not prevent the compromised server from adding ID-value pairs (e.g. to prefix trees the ID owners never check). For example, in Fig. 4, Alice cannot detect that the attacker inserted $\mathsf{Alice}||\mathsf{Val}'$ at position 25, because Alice does not have a value inside the chronological tree rooted at node [24:27], so the monitoring proof will not include a membership proof for the prefix tree at node [24:27].

To prevent attackers from adding corrupted ID-value pairs, Merkle^2 co-designs signature chains as follows. The ID owner attaches a verifying key to each ID-value pair in Merkle^2. And, on append, each new ID-value pair, its position, and the new verifying key are signed by the verifying key attached to the previous ID-value pair for the owner. Users can then verify the signature on an ID-value pair with the same verifying key. By verifying the signature chain, users can confirm that all the ID-value pairs are indeed appended by the ID owner. In the example of Fig. 4, although the attacker can still add $\mathsf{Alice}||\mathsf{Val}'$ without being caught by Alice, other users will not accept the corrupted pair because the attacker cannot produce a valid signature. Notice that the attacker may try to hide the end of the chain during lookup. The monitoring proof ensures that the attacker cannot hide the owner's values without detection.

The protocol described thus far is still insecure because the first value is not signed. An attacker may insert a corrupt ID-value pair and try to convince users that it is the first value for that ID; thus, the attacker could circumvent the signature chain. We observe, however, that if the honest ID owner already has inserted values for that ID in Merkle^2, the attacker cannot convince other users that the falsified value is the first for that ID. This holds true because the monitoring proof ensures that the attacker cannot remove existing values without being detected. For example, in Fig. 4, the attacker cannot hide $\mathsf{Alice}||21||\mathsf{Val}_0$ and claim $\mathsf{Alice}||25||\mathsf{Val}'$ as the first pair for "Alice"; as shown in our lookup protocol in Section VI, other users looking up Alice's values will verify the non-membership proof for the prefix tree in [16:23]. Meanwhile, Alice will also check the membership proof for [16:23] by verifying the monitoring proof. The server cannot provide both a membership and non-membership proof for a leaf node associated with the ID "Alice" for the same prefix tree.

First-value checking. The ID owner must ensure that it indeed appends the first ID-value pair for that ID in Merkle^2, otherwise others may have obtained ownership for that ID already. It is not feasible to check all the leaves appended before it. Instead, we can leverage non-membership proofs from prefix trees to prove that no value exists for that particular ID. For example, if the ID-value pair is added at position x, there exists a minimum set of chronological trees $\mathsf{Ct}_1; \ldots; \mathsf{Ct}_n$ that cover the previous $x-1$ leaves (there are only $O(\log(x))$ chronological trees in this covering set). For the prefix tree corresponding to every chronological root of Ct_i, we can generate a non-membership proof for the ID. The non-membership proofs allow the ID owner to compute the root hashes of the prefix trees; in order to compute the root hashes of the chronological roots, $\mathsf{Ct}_1; \ldots; \mathsf{Ct}_n$, the first-value proof also contains the minimum set of node hashes needed to do so. This way, the ID owner can compute the hashes of the chronological roots and compare them against the digest.

For example, in Fig. 5, Alice verifies the non-membership proofs of the prefix trees for nodes [0:15], [16:19], and [20:20] to ensure that they do not contain values for "Alice." If the non-membership proofs are not valid and there exists a value for "Alice," Alice will know that she does not have ownership of that ID. If another honest user appends $\mathsf{Alice}||17||\mathsf{Val}'$ before Alice appends $\mathsf{Alice}||21||\mathsf{val}$, then the attacker cannot hide $\mathsf{Alice}||17||\mathsf{Val}'$ from Alice because when the honest user performs the monitoring protocol, it verifies the membership proof of the prefix tree of node [16:19]. Since the server cannot provide both a membership and non-membership proof for the same leaf node in the same prefix tree, the misbehavior will be detected. In addition to the non-membership proofs, the server must also provide the hashes necessary for Alice to compute the root hashes of the digest. For example, Alice can compute prefix root hash of node [16:19] using the non-membership proof provided, but she still needs $\mathsf{H}_{[16:17]}, \mathsf{H}_{[18:19]}$ to compute $\mathsf{H}_{[16:19]}$. With the other hashes, Alice can finally compute the chronological root hash, $\mathsf{H}_{[16:23]}$, to see if it matches the digest.

It is possible for a malicious server to add $\mathsf{Alice}||17||\mathsf{Val}'$ to the chronological tree but remove it from the prefix tree at node [16:19]. However, once Alice appends the first value and successfully verifies the first-value proof, the attacker can no longer add Val' back to any prefix trees associated with its

ancestor nodes without detection. New ancestor nodes added in the future will also be ancestors of $\mathsf{Alice}||21|\mathsf{val}$, so they will be checked in monitoring proofs by Alice. Thus, the attacker may be the owner of ID "Alice" before Alice joins the system; but, after Alice appends her first value, any value appended previously by the attacker will not be accepted by honest users, and the attacker no longer has the ability to append values for "Alice" without a valid signature.

Complexity analysis. If there are $O(n)$ ID-value pairs in the system, the covering set contains $O(\log(n))$ trees. Thus, there are $O(\log(n))$ total non-membership proofs to verify, each with a size of $O(\log(n))$, so the total cost of the first-value proof is $O(\log^2(n))$. ID owners only need to check the first-value proof once due to extension proofs.

VI. Lookup protocol in Merkle^2

In this section, we present the construction of lookup proofs in Merkle^2. We denote by $\mathsf{Ct}_1; \cdots; \mathsf{Ct}_n$ the chronological trees in the forest of Merkle^2's data structure, and $\mathsf{Pt}_1; \cdots; \mathsf{Pt}_n$ the prefix trees at their roots, which we also refer to as *root prefix trees*. The lookup proof must be able to convince users that the lookup result contains *all* the values for a given ID. By providing (non-)membership proofs for all the root prefix trees, the server can prove the (non-)membership of ID-value pairs for the given ID in all of Merkle^2. Further, the signature chain helps users verify the authenticity of the lookup result.

Based on these ideas, the lookup protocol for an ID works as follows. For each ID-value pair except the first one, the lookup proof contains a signature signed by the ID owner. For each prefix tree Pt_i, we generate a proof, $\ _i$; if there exists a value for ID in Pt_i, $\ _i$ is a membership proof. Otherwise, $\ _i$ is a non-membership proof. The lookup proof also contains the hashes $\mathsf{LH}_i; \mathsf{RH}_i$ of the left and right children of root node Ct_i, which allows users to compute the root hashes and compare them to those in the digest. Finally, the lookup proof contains the signature chain and the following tuples: $(\langle\ _1; \mathsf{LH}_1; \mathsf{RH}_1\rangle; \cdots; \langle\ _n; \mathsf{LH}_n; \mathsf{RH}_n\rangle)$. Intuitively, the lookup proof captures which chronological roots contain values for ID and which do not.

In the example of Fig. 4, the lookup proof for Alice's values contains the signature chain and (non-)membership proofs for prefix trees [0:15], [16:23], [24:27], and [28:29]. The proofs for [16:23] and [28:29] are membership proofs of the leaf values associated with Alice. The proofs for [0:15] and [24:27] are used to prove non-membership of Alice's ID. Finally, the lookup proof also contains any hashes which are necessary to compute the root hashes; for example, $\mathsf{H}_{[0:7]}$ and $\mathsf{H}_{[8:15]}$ are used to compute the root hash $\mathsf{H}_{[0:15]}$.

During lookup proof verification, the user possesses the following: 1) the lookup result, which contains all ID-value pairs for the target ID; 2) the latest digest from auditors that contains the root hashes of each chronological tree in the forest; 3) the lookup proof corresponding to the lookup result.

The user proceeds as follows:

1) verifies the signature chain using the verifying keys provided in each ID-value pair;
2) for each ID-value pair, finds which chronological tree Ct_i and corresponding prefix tree Pt_i it belongs to;
3) computes the root hash Root_i for each Pt_i using $\ _i$ and the ID-value pairs in Pt_i based on the results from step 2;
4) computes the root hash for Ct_i using Root_i and $\langle \mathsf{LH}_i; \mathsf{RH}_i \rangle$

If the signature chain is valid and the computed root hashes match those in the digest, the user can accept the lookup result.

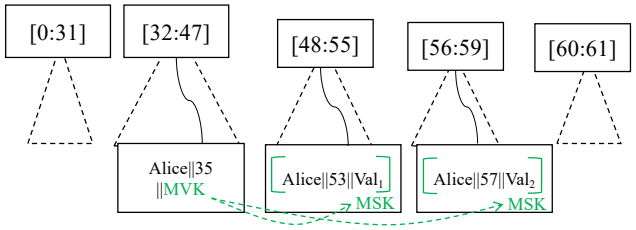

Fig. 6: Alice appends the master verifying key as the first value. Following values are signed by the master signing key.

Complexity analysis. The signature chain is of length $O(\grave{})$ where $\grave{}$ is the number of values. The lookup proof contains (non-)membership proofs from $O(\log(n))$ prefix trees, and each (non-)membership proof is of size $O(\log(n))$. Therefore, the overall lookup proof is of size $O(\grave{} + \log^2(n))$.

A. Lookup for the latest ID-value pair

We provide an optimized protocol to look up the *latest* (i.e., most recent) value for an ID. In many applications, users may want the latest value instead of all values. We make use of master keys, shown in Fig. 6, to replace signature chains since the size of the chain is linear to the number of values. ID owners generate a pair of master keys and append the master verifying key as their first ID-value pair in the log. ID owners must also ensure the master verifying key is the first value by verifying the first-value proof. All future pairs appended by the ID owner will be signed using the master signing key.

Instead of downloading all the ID-value pairs and signature chain for a given ID, users can download only the latest value and the master verifying key to verify the signature. Users also need (non-)membership proofs from prefix trees to ensure that the master verifying key is in fact the first value for the given ID and that the lookup result is in fact the latest one.

For example, in Fig. 6, users can verify that Alice's master verifying key MVK is the first value for "Alice" by verifying the non-membership proof for the prefix tree [0:31] and the membership proof for the prefix tree [32:47]. Similarly, users can also verify that $\mathsf{Alice}||57||\mathsf{Val}_2$ is the latest value by verifying the membership proof for the prefix tree [56:59] and the non-membership proof for the prefix tree [60:61].

We assume ID owners will not change their master keys in the system. If required, Merkle^2 can permit ID owners to change their master keys as follows. ID owners can revoke master keys by appending and signing the new master key using the old master key. However, allowing changes to master keys will increase the lookup cost because the other users now also need to check the signature chain of the master keys. Instead, users can use existing techniques for backing up master keys on multiple devices securely just as they would

do for their secret keys for end-to-end encrypted services, for example by using secret sharing [50], [51], [52]. The cost of this modified lookup protocol is $O(\log^2(n))$ since there is no longer a signature chain cost. Users can also cache master keys to further reduce the cost of a lookup.

VII. Applications of Merkle^2

In this section, we discuss two applications of Merkle^2: a ledger for web certificates and for a public key infrastructure.

A. Transparency log for web certificates

The security issues, where CAs have been compromised and issued certificates incorrectly [53], [54], [55], [56], [57], prompted the design of web certificate management systems using transparency logs, where the owner of a certificate can verify the integrity of their own certificate and hold CAs accountable for corrupted certificates [15], [17], [8], [3], [4].

We now describe how to use Merkle^2 for certificate management in place of existing systems [15], [3]. The log server runs Merkle^2's server to manage certificates for each domain name. The ID is the domain name and the values are the web certificates for that domain. There may be more than one certificate for the same domain. Instead of storing certificates as different domain-certificate pairs, we bundle multiple certificates together as a single value, and each append will be the hash of all the certificates for the domain name. Merkle^2 also supports revocation efficiently by allowing the domain owner to append the hash of all the unrevoked certificates for the same domain name. If a certificate is not in the latest append for this domain name, it is not valid (e.g., it was revoked). In the end, web browsers can simply retrieve the latest value for the domain to check whether a certificate is valid.

Benefits of our system. We compare Merkle^2 to existing proposals. Deployed CT systems [15], [3] do not support revocation. Enhanced certificate transparency (ECT) [8] aims to solve this problem. ECT also uses prefix trees (they use a similar design called lexicographic trees) and chronological trees, but ECT keeps these trees *separate*; thus, ECT requires auditors to verify the relation between these two trees by scanning linearly through all entries in the log. Hence, auditors require $O(n)$ time and space to perform their monitoring. In contrast, Merkle^2 provides a way to *nest* the two trees to reap their benefits *simultaneously* through our "multi-dimensional" design, which reduces the cost of auditors to only $O(\log(n))$. We give a concrete performance comparison in Section IX.

B. Transparency log for public keys

Merkle^2 can also be used for a transparent public-key infrastructure as an alternative to CONIKS [1] or KT [5]. We use end-to-end encrypted email systems [58] as a real-world example. In this application, an ID in Merkle^2 corresponds to a user's email, e.g. alice@org:com, and the value corresponds to the public key of the user, e.g., $\mathsf{PK}_{\mathsf{Alice}}$. To join the system, Alice appends the first public key for her email address alice@org:com. To revoke a public key, Alice appends a new public key for her email address. If Bob wants to send an encrypted email to Alice, he looks up the latest public key for alice@org:com and uses it to encrypt and send the email. For company communications, the organization can monitor all the email-PK pairs. For a personal account, the client running on the user's devices can monitor the transparency log regularly.

Benefits of our system. We compare Merkle^2 to CONIKS [1], a state-of-the-art key transparency system. Monitoring in CONIKS is inefficient as key owners must check each digest published by the server. In contrast, each key owner in Merkle^2 only monitors $O(\log^2(n))$ data throughout the system's life, where n is the number of keys in the log. As we will show in Section IX, Merkle^2 can support short enough epoch periods (such as 1 second) to be considered low-latency [19], [34].

Privacy concerns. CONIKS [1] shows that verifiable random functions (VRFs) [59] can reduce the leakage of IDs to malicious users. The same technique can be applied to Merkle^2. Instead of using an ID directly, users compute indices using the output of VRFs on the ID; the server also includes the information needed to verify the VRF result in the reply.

VIII. Implementation

We implement a prototype of Merkle^2 in Go. It consists of four parts, as in Fig. 1: the server ($\approx$ 800 LoC), auditor ($\approx$ 200 LoC), client library ($\approx$ 450 LoC), and verification daemon ($\approx$ 600 LoC), which all depend on a set of core Merkle^2 data structure libraries ($\approx$ 2400 LoC). The Merkle^2 library is available at https://github.com/ucbrise/MerkleSquare.

Our server implementation backs up ID-value pairs in persistent storage in case the server fails using LevelDB [60], which has been used in previous transparency log systems [1], [5]. To provide a 128-bit security level, we used SHA-3 [61] as the hash function and Ed25519 signatures [62] (this is the only public key operation in the system). We did not implement VRFs [59] since privacy is not the focus of this paper. We limit the chronological tree height in Merkle^2 to 31 to support the pre-build strategy, which means it can store up to 2^{31} ID-value pairs and each append will be added to 31 prefix trees.

Concurrency control. Merkle^2's server can serve requests in parallel, relying on concurrency control of Merkle^2's data structure and LevelDB. Merkle^2's data structure prohibits concurrent appends since two appends may affect the same prefix tree. For each append, the server sends a position number to the user, and the user should reply with the signature. If the user withholds or does not reply with the signature within a short time bound, the server rejects the append. Lookups and monitoring can be concurrent with appends because required hashes have been computed in the past epochs.

IX. Evaluation

Experiment setups. We ran our experiments on Amazon EC2 instances; the microbenchmarks and system server were run on a r5.2xlarge instance. The auditor and client services were run on a r5a.xlarge and c4.8xlarge instance, respectively.

Baselines. We chose three state-of-the-art transparency logs to compare with: CONIKS [1], AAD [9], and ECT [8]. We compare Merkle^2's complexity with these baselines in Table I.

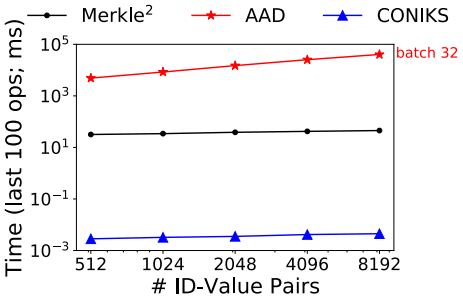

Fig. 7: Append time.

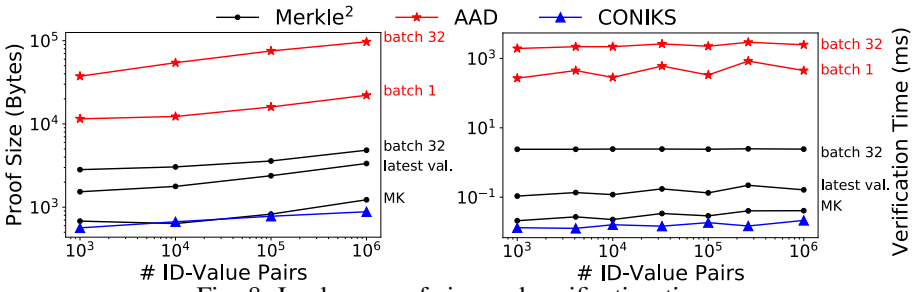

Fig. 8: Lookup proof size and verification time.

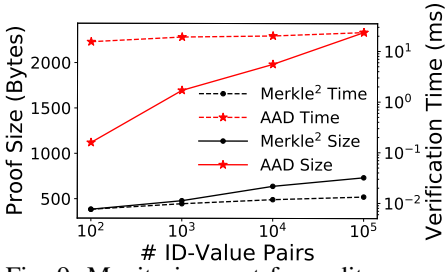

Fig. 9: Monitoring cost for auditors in AAD and Merkle^2.

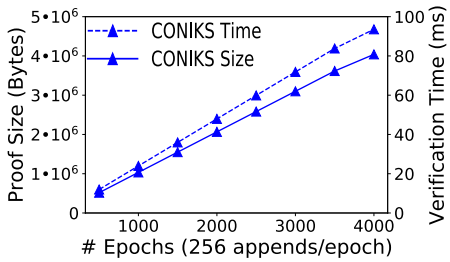

Fig. 10: Monitoring cost for ID owners in CONIKS.

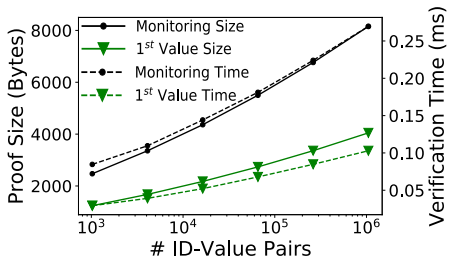

Fig. 11: Monitoring cost for ID owners in Merkle^2.

We compare Merkle^2 with CONIKS via both microbenchmarks and end-to-end system performance. The original CONIKS implementation [63] is quite incomplete; for example, it does not support monitoring or persistent storage. Moreover, since CONIKS is not designed for short epochs, it copies and reconstructs the entire Merkle tree in each epoch; this incurs a large time and space cost, which would unfairly disadvantage CONIKS in comparison to Merkle^2. To produce a fair comparison, we enhanced the CONIKS design to use persistent data structures [45], avoiding the overhead of copying the entire tree. We also implemented the monitoring functionality from its paper, and wrapped the CONIKS data structure into a server system. We disabled VRFs [59] in the CONIKS implementation since we do not focus on privacy. Similarly to Merkle^2, the modified CONIKS implementation can process lookups and monitoring during appends, but does not allow concurrent appends.

AAD is an asymptotically efficient transparency log built on top of bilinear accumulators [64], but its constants are large. We compare with AAD's microbenchmarks results from their paper and repository [9], [65] because our setup cannot support running experiments for AAD. For example, it takes more than 20 hours to generate the public parameters necessary, and as shown in [9], the experiments were run on a r4:16xlarge instance, which is much more powerful than our machines.

We also compare Merkle^2 with ECT, for the use case of transparent web certificates. Since there is no ECT implementation available, we use the numbers provided in their paper and other online statistics for the comparison.

A. Microbenchmarks

We compare Merkle^2's core protocol to AAD [9] and CONIKS [1] via microbenchmarks for individual operations.

Append time. The append time comparison (depicted in Fig. 7) measures the total time taken for the last 100 appends for a given number of ID-value pairs. AAD only provides benchmarks up to $2^{13} = 8192$ ID-value pairs because of how long larger scale appends would take [9]. AAD supports a batching mechanism to speed up append operations; in our graph, we include only the result for the 32 ID-value pair batch as the time for a single append is too high. In spite of the batching, AAD is still significantly slower than Merkle^2 and CONIKS because it uses bilinear accumulators. CONIKS, on the other hand, is faster than Merkle^2; a single append in CONIKS inserts only an ID-value pair into a single prefix tree. In Merkle^2, we measured the append cost with the pre-build strategy. Because we set the maximum chronological tree height to 31, each append is added to 31 prefix trees. If there are 2^{20} ID-value pairs in the system, it takes 3 ms for the last 100 appends in CONIKS, while in Merkle^2 it takes 151 ms.

Monitoring cost. In Fig. 9, we contrast the monitoring costs of auditors in AAD and in Merkle^2. We compare proof size and verification time for extension proofs between system states containing 10^i and 10^{i+1} ID-value pairs. Clearly, the monitoring costs of auditors in AAD is higher than those of auditors in Merkle^2 for both proof size and verification time. Because system states in different epochs in CONIKS share no defined relationship, CONIKS does not have extension proofs, so it was excluded from this experiment.

Now, we compare the monitoring costs of ID owners in CONIKS with those of ID owners in Merkle^2. In Fig. 10, we add 2^{20} ID-value pairs into a CONIKS system. Then, we vary the number of epochs the ID owner must monitor because CONIKS requires each ID owner to monitor every epoch. The monitoring costs of CONIKS grow linearly with the number of epochs. In contrast, Merkle^2's monitoring proof depends only on the number of existing ID-value pairs; it is independent of the number of epochs since ID owners can simply monitor the latest epoch. In Merkle^2, an ID owner must check the

first-value proof when they join the system and then regularly monitor their ID-value pairs; the costs are depicted in Fig. 11. As discussed in Section V-B, ID owners can cache monitoring proofs and need to verify the first-value proof only once; thus, the monitoring cost of ID owners in Merkle^2 is significantly lower than those of ID owners in CONIKS.

Lookup cost. In Fig. 8, we compare the average lookup proof size and verification time for all three systems. The lookup cost in Merkle^2 and AAD may increase when there are more values for the target ID, so we measure the results for both single ID-value pairs and batches of 32. Lookup costs fluctuate as performance depends on the underlying structure of the system. In Merkle^2, for example, $2^{12}-1$ ID-value pairs result in more root prefix trees than 10^4 ID-value pairs. What first appears to be a discrepancy is actually an odd feature of the system. The batched lookup proof, however, is not affected because the cost is dominated by the signature chain.

In Merkle^2, the master key and latest value lookups are more efficient than the batched lookup because they do not require signature chains. Note that the master key lookup cost of Merkle^2 is close to that of CONIKS. This occurs due to the forest design in Merkle^2, where the root prefix trees at the beginning of the forest span more ID-values than the others. Because master keys require non-membership proofs beginning from the left of the forest, there are fewer (non-)membership proofs in the master key proof than in the latest value proof. In most cases, the master key is covered by the largest chronological tree. Thus it contains only a single membership proof and has a cost close to that of CONIKS.

Memory usage. Fig. 12 compares the memory cost of Merkle^2 and CONIKS. We fix the number of appends per epoch to be 256. We also limit the chronological tree height in Merkle^2 to 31 to support the pre-build strategy.

For 2^{20} ID-value pairs, Merkle^2 consumed about 22 GiB of RAM, and CONIKS consumed about 6.3 GiB of RAM. We obtain the memory usage of AAD from its paper [9], and it consumed 263 GiB of RAM. AAD additionally required 64 GiB of RAM for public parameters.

The original CONIKS implementation copies the prefix tree for each epoch and results in $O(E \cdot n)$ nodes where E is the number of epochs, which is prohibitively large for shorter epochs. Instead, we improve CONIKS by leveraging persistent data structures [45] to avoid copying prefix trees in each epoch. Each insertion in the persistent prefix tree creates $O(\log(n))$ nodes; thus, CONIKS has to store $O(n\log(n))$ nodes in total. Merkle^2's asymptotic storage cost is the same as CONIKS, but the memory usage is higher due to the larger constants. We discuss how to optimize Merkle^2's storage in Section IX-D.

B. End-to-end system evaluation

In this section, we evaluate Merkle^2's system-level performance with that of CONIKS. Note that we do not use VRFs [59] in either Merkle^2 or CONIKS as we are interested in the main system cost. We also limit the chronological tree height in Merkle^2 to 31. In the experiment, we insert 10^6 ID-value pairs into each system before running the benchmarks.

Operation		Merkle^2	CONIKS
Append		4.39	3.02
Lookup	Master key	1.62	1.28
	Latest value	2.31	
Owner Monitor	1 epoch	2.33	1.29
	10 epochs		2.32
	100 epochs		10.62
	1000 epochs		88.03
Auditor Monitor		0.25	0.22

TABLE II: Latency (in ms).

Operation		Merkle^2	CONIKS
Append		42B	42B
Lookup	Master key	3.8KB	1.6KB
	Latest value	9.8KB	
Owner Monitor	1 epoch	22.9KB	2.1KB
	10 epochs		21KB
	100 epochs		209.6KB
	1000 epochs		2.1MB
Auditor Monitor		654B	370B

TABLE III: Message size of server's responses.

End-to-end performance. We analyze the end-to-end performance from the client's perspective, which includes the proof verification and communication with the server and auditor. Table II shows the average latency and Table III shows the message size of server responses for 10^3 operations. For Merkle^2, we measure only the latest-value lookup protocol as it is more efficient and useful in real-world applications. And, we separate the cost of looking up the master verifying key and latest ID-value pair, since users can store and reuse master keys as discussion in Section VI-A. The result shows that appends and lookups in Merkle^2 are more expensive. The cost for the master key lookup is cheaper than that of the latest-value lookup; this occurs because the proof for the latter is smaller due to the forest design in Merkle^2.

To measure the cost for ID owners to monitor, we vary the number of epochs the user is offline since last monitoring. As mentioned, ID owners in CONIKS must check every digest published, which makes the cost grow significantly when a user is offline for some time. In contrast, Merkle^2 does not see the same increase because ID owners can check only the latest digest. Based on results in Table II and Table III for CONIKS, we can estimate the cost of an ID owner who wishes to verify all digests in a given month. Suppose the epoch is 1 second, as is desired by Google KT [19]; in a month, there will be $30 \cdot 24 \cdot 60 \cdot 60 = 2592000$ epochs. Then, the ID owner needs to download $\frac{2592000}{1000} \cdot 2.1\text{MB} \approx 5.4\text{GB}$ of data and spend $\frac{2592000}{1000} \cdot 88\text{ms} \approx 3.8\text{mins}$ to verify it. This cost is problematic for the server who has to incur this cost for every data owner. In contrast, ID owners in Merkle^2 only download 22.9KB of data, which is independent of the number of epochs.

We measure the cost for auditors to check the new digest. Auditors in Merkle^2 check the extension proof, whereas auditors in CONIKS fetch only the digest from the server. The result shows that if there are 2^8 appends between epochs, auditors in Merkle^2 are as lightweight as those in CONIKS. Note that auditors in Merkle^2 can gossip the extension proof with each other to further reduce the bandwidth of the server.

Throughput. Next, we measure the throughput of frequent

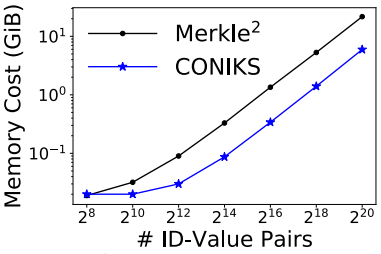

Fig. 12: Memory cost.

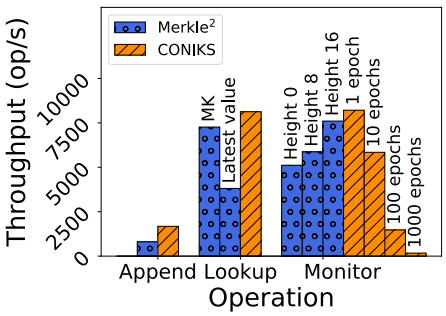

Fig. 13: Server throughput.

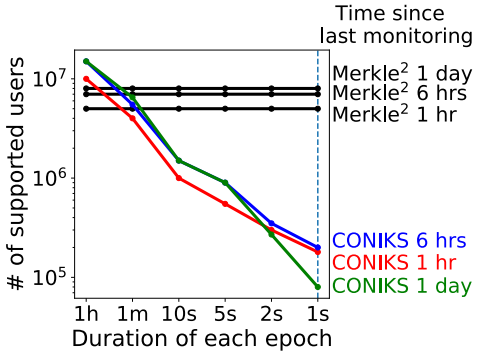

Fig. 14: Users supported by one server.

operations in Merkle^2 and CONIKS, depicted in Fig. 13. In the experiments, we randomly choose an ID-value pair for lookup and monitoring. CONIKS can support more append and lookup operations because its append and lookup are more efficient than those of Merkle^2. The throughput of the master key lookup is much higher than that of the latest-value lookup because the master key lookup proof is smaller than the latest value lookup proof, as shown in Table III.

For the monitoring throughput benchmark, we fix every ID owner's monitoring frequency; for CONIKS, there is a fixed number of digests the ID owners must monitor. The results clearly show that the server's performance decreases significantly when ID owners monitor larger numbers of epochs. In Merkle^2, the monitoring cost is independent of the number of epochs, and ID owners can cache and skip membership proofs of prefix trees that are checked in the past. To illustrate the saving provided by this caching mechanism, let i be the height of the highest node associated with the prefix tree that is in ID owners' cache. ID owners do not need to download membership proofs for prefix trees at nodes below height i. In Fig. 13, we vary the height i to show the throughput. "Height 0" shows the worst case result, as it means ID owners need to check all the prefix trees. The results show that the throughput increases when ID owners cache more monitoring proofs.

C. Performance in applications

In this section, we compare Merkle^2's performance in the applications described in Section VII.

Web certificate management. To compare with ECT, we estimate the cost of ECT based on numbers in their paper and with the help of online statistics. Recent certificate statistics [66] show that about $5,002,599$ certificates are appended every day since June 2018. Additionally, auditors in ECT require 2 KB of data for each append [8]. Thus, ECT auditors have to download $5002599 * 2\text{KB} \approx 9.5\text{GB}$ of data per day for monitoring. In contrast, Merkle^2 auditors require only $\log(10^9) * 32\text{B} * 2 \approx 1.9\text{KB}$ of data for monitoring.

Public keys management. As shown in previous sections, Merkle^2 supports efficient monitoring but sacrifices some performance for append and lookup operations. To understand the benefits of Merkle^2, we must run our benchmarks under workloads matching the target application. Thus, we compare Merkle^2 and CONIKS in the following scenario.

We consider the real-world application of encrypted emails using available statistics; in particular, there are 200 appends per second for 1 billion users [19], [34], and each user sends 42 emails per day [67]. Users may also cache public keys for emails sent within a short period of time, such as 1 hour. Using the Enron email dataset [68], we find that users need to perform a key lookup for about 62% of emails. In summary, each user may send $42 \cdot 62\% = 26$ lookup requests per day. We control the number of monitoring requests by adjusting the average monitoring frequency. Based on these results, we generate the workload for different numbers of users under different epoch intervals and monitoring frequencies. To keep the experiment time reasonable, we add only 10^6 ID-value pairs into the system before running them.

Fig. 14 depicts the number of supported users by a single server machine for both systems. The result shows that the epoch interval does not affect the performance of Merkle^2. When users monitor the server more frequently, Merkle^2's performance decreases because the server has to serve more requests. CONIKS's performance is significantly worse than Merkle^2 when the epoch interval is short because of its expensive monitoring protocol. Users may increase their monitoring frequency to avoid larger computations as a result of being offline for many epochs. However, this increases the number of monitoring requests that the server must handle, which may decrease its performance. On the other hand, when the epoch interval becomes 1 hour, CONIKS starts to outperform Merkle^2 because its appends and lookups are more efficient. We find that the monitoring caching mechanism does not improve the performance of Merkle^2 when the monitoring interval is long (more than 1 hour) because lookups dominate the workload in this application. The caching mechanism is more helpful in applications with more frequent monitoring. In conclusion, Merkle^2 significantly outperforms CONIKS when the epoch interval is small (for example, one second), which is desirable for a low-latency key transparency system.

D. Limitations and future work

These experiments suggest Merkle^2 cannot outperform CONIKS when the epoch interval is long because appends and lookups become the bottleneck. Merkle^2 leaves as future work improving its append and lookup efficiency. For example, we can use one server to pre-build large prefix trees that

are supposed to be used in the future; thus, another server can cache smaller prefix trees and serve users' requests more efficiently. Also, we can batch lookup proofs for different users and leverage proxies to save the bandwidth of the server.

Given that data is replicated in Merkle^2's data structure, memory usage is also a concern. We notice that prefix trees associated with the right children in chronological trees are not needed; thus, the server can save half of the space by skipping these prefix trees. We leave these optimizations to future work.

X. Related works

Transparency logs. We have already compared extensively with CT [15], [3], ECT [8], CONIKS [1], and AAD [9].

Recently, there has been extensive research into improving the performance of transparency logs. One school of thought [2], [8], [10], attempts to use prefix trees and chronological trees in parallel for efficient lookup and state monitoring, respectively. Unfortunately, as in ECT, auditors and owners must still verify all operations in the chronological tree to verify that the prefix tree is built correctly. WAVE [2] relies on strong auditors to monitor on behalf of users, which incurs a large burden on the auditors. SEEMless [11] is the first for proposing persistent data structures to optimize CONIKS. However, it relies on strong auditors to monitor the append-only property of the Persistent Patricia Trie, which still incurs a super-linear cost on the auditors. ECT and DTKI [10] also rely on users to collectively verify server states; collective verification, however, assumes enough honest users in the system, which limits its use in real-world applications. Google KT [19] recently proposed a new design that requires users performing a lookup on a value to only perform verification of the same digest as the value owner. To ensure that the value owner and user both verify the same digest, KT uses a meet-in-the-middle algorithm. As a result, the monitoring cost of the owner becomes $O(\log(E)\log(n))$, where E is the number of epochs. However, the cost of a lookup also increases to $O(\log(E)\log(n))$. We do not perform any comparison to KT in our evaluations as it is still in its early stages, and there is no protocol detail or implementation available for benchmarking. AKI [69] has the server maintain prefix trees, as in CONIKS; it distributes the monitoring workload to auditors, ID owners, users, and other third parties. Unfortunately, AKI operates on the assumption that no parties collude. ARPKI [70] and PoliCert [71] extend the security of AKI by protecting against attackers controlling n-1 out of n parties.

Another approach is to use recursive SNARKs [42], [43], [44], [72] or cryptographic accumulators [64], [73]. However, these solutions are too expensive to be practical.

Several gossip protocols [24], [4], [74], [25] are designed to ensure a consistent view of digests among users and auditors. Merkle^2 can use these to share digests and extension proofs. Software transparency logs [20], [21], [22], [23], [24], [25] are designed for securing software updates. Merkle^2 can be used to improve the performance of these systems.

Previous works also formalize the security guarantees of CT [75], [76] and those of general transparency logs [75], [9]. The security guarantees of Merkle^2 can be analyzed under the same model; we leave this for future work.

Authenticated data structures. Authenticated data structures [77], [9] are at the core of transparency logs. Some works [78], [79] focus on improving the performance of Merkle tree implementations, which can also be applied to Merkle^2. Cryptographic accumulators [64], [73], [73] can also be used for building authenticated data structures; however, these cryptographic primitives have a large overhead.

Blockchains and consensus protocols. Decentralized ledgers can be built on top of blockchains using consensus protocols [29], [30], [31], [32], [33], [80], which have seen widespread adoption in cryptocurrencies [26], [27], [28], [80]. Merkle^2 is more efficient and lightweight than blockchain-based systems because it is hosted centrally; this way, there is no need for expensive consensus protocols or data replication. At the same time, Merkle^2 loses the availability guarantee of blockchain systems, since a malicious server can deny service.

Another approach is to use blockchains to provide efficient auditing mechanisms for transparency logs [7], [25], [81], [82], [83]. However, the performance of this approach is limited by the underlying blockchain protocol. Some works [84], [85], [86] leverage trusted hardware to improve the performance of blockchain systems. However, existing hardware, like Intel SGX, are susceptible to side-channel attacks [87].

File sharing with an untrusted server. Many systems [48], [88], [89], [90] allow users to share files on untrusted storage. The focus of these works is to provide a file sharing functionality instead of an immutable append-only log. SUNDR [48] and Venus [90] achieve weaker consistency guarantee than Merkle^2. Verena [88] relies on two "non-colluding" servers. Ghostor [89] relies on either a blockchain or transparency log, so Merkle^2 can be used as a foundation for Ghostor.

XI. Conclusion

In this paper, we present Merkle^2, a low-latency transparency log system. Merkle^2 contributes a novel authenticated data structure and the system design leveraging it, which achieve efficient append, lookup, and monitoring protocols. For epochs as a short as 1 second, a Merkle^2 server can serve $100\times$ more users than CONIKS. Merkle^2 has applications to both web certificate and public key transparency.

XII. Acknowledgment

We thank the anonymous reviewers and our shepherd, Henry Corrigan-Gibbs, for their invaluable feedback. We would also like to thank Gary Belvin from the Google Key Transparency Group and colleagues from the RISELab Security Group for giving us feedback on early drafts.

This research was supported by the NSF CISE Expeditions Award CCF-1730628, NSF Career 1943347, as well as gifts from the Sloan Foundation, Bakar, Okawa, Amazon Web Services, AntGroup, Capital One, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

REFERENCES

[1] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: bringing key transparency to end users," in *USENIX Security*, 2015.

[2] M. P. Andersen, S. Kumar, M. AbdelBaky, G. Fierro, J. Kolb, H. Kim, D. E. Culler, and R. A. Popa, "WAVE: A decentralized authorization framework with transitive delegation," in *USENIX Security*, 2019.

[3] B. Laurie, "Certificate transparency," *CACM*, 2014.

[4] L. Chuat, P. Szalachowski, A. Perrig, B. Laurie, and E. Messeri, "Efficient gossip protocols for verifying the consistency of certificate logs," in *CNS*, 2015.

[5] R. Hurst and G. Belvin, "Security through transparency," https://security.googleblog.com/2017/01/security-through-transparency.html.

[6] B. Laurie and E. Kasper, "Revocation transparency," *Google Research*, 2012.

[7] J. Bonneau, "Ethiks: Using ethereum to audit a coniks key transparency log," in *FC*, 2016.

[8] M. D. Ryan, "Enhanced certificate transparency and end-to-end encrypted mail." in *NDSS*, 2014.

[9] A. Tomescu, V. Bhupatiraju, D. Papadopoulos, C. Papamanthou, N. Triandopoulos, and S. Devadas, "Transparency logs via append-only authenticated dictionaries," in *CCS*, 2019.

[10] J. Yu, V. Cheval, and M. Ryan, "Dtki: A new formalized pki with verifiable trusted parties," *The Computer Journal*, 2016.

[11] M. Chase, A. Deshpande, E. Ghosh, and H. Malvai, "Seemless: Secure end-to-end encrypted messaging with less trust," in *CCS*, 2019.

[12] R. Perlman, "An overview of pki trust models," *IEEE network*, 1999.

[13] C. Adams and S. Lloyd, *Understanding public-key infrastructure: concepts, standards, and deployment considerations*, 1999.

[14] U. Maurer, "Modelling a public-key infrastructure," in *ESORICS*, 1996.

[15] A. Langley, E. Kasper, and B. Laurie, "Rfc 6962-certificate transparency," 2013.

[16] Google, "Https encryption on the web," transparencyreport.google.com/https/certificates.

[17] "Certificate transparency in chrome," https://chromium.github.io/ct-policy/#chromium-certificate-transparency-policy.

[18] Google, "Key transparency contributions," github.com/google/keytransparency/blob/master/docs/design-improvements.md.

[19] ——, "Key transparency new design," github.com/google/keytransparency/blob/master/docs/design_new.md.

[20] M. Al-Bassam and S. Meiklejohn, "Contour: A practical system for binary transparency," in *ESORICS workshops*, 2018.

[21] S. Fahl, S. Dechand, H. Perl, F. Fischer, J. Smrcek, and M. Smith, "Hey, nsa: Stay away from my market! future proofing app markets against powerful attackers," in *CCS*, 2014.

[22] B. Hof and G. Carle, "Software distribution transparency and auditability," *arXiv:1711.07278*, 2017.

[23] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford, "CHAINIAC: Proactive software-update transparency via collectively signed skipchains and verified builds," in *USENIX Security*, 2017.

[24] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, "Keeping authorities" honest or bust" with decentralized witness cosigning," in *S&P*, 2016.

[25] A. Tomescu and S. Devadas, "Catena: Efficient non-equivocation via bitcoin," in *S&P*, 2017.

[26] "Bitcoin," https://bitcoin.org/.

[27] "Ethereum," https://ethereum.org/.

[28] "Zcash," https://z.cash/.

[29] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Tech. Rep., 2019.

[30] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2014.

[31] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI*, 1999.

[32] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *PODC*, 2019.

[33] R. Pass and E. Shi, "Thunderella: Blockchains with optimistic instant confirmation," in *Eurocrypt*, 2018.

[34] Google, "New design doc," github.com/google/keytransparency/pull/1469.

[35] F. F. Nah, "A study on tolerable waiting time: how long are web users willing to wait?" *Behaviour & Information Technology*, 2004.

[36] M. Singh, M. Rajan, V. Shivraj, and P. Balamuralidhar, "Secure mqtt for internet of things (iot)," in *CSNT*, 2015.

[37] A. L. M. Neto, A. L. Souza, I. Cunha, M. Nogueira, I. O. Nunes, L. Cotta, N. Gentille, A. A. Loureiro, D. F. Aranha, H. K. Patil, and L. B. Oliveira, "Aot: Authentication and access control for the entire iot device life-cycle," in *SenSys*, 2016.

[38] S. Kumar, Y. Hu, M. P. Andersen, R. A. Popa, and D. E. Culler, "*f*JEDI*g*: Many-to-many end-to-end encryption and key delegation for IoT," in *USENIX Security*, 2019.

[39] S. Raza, L. Wallgren, and T. Voigt, "Svelte: Real-time intrusion detection in the internet of things," *Ad hoc networks*, 2013.

[40] W. Lee, S. J. Stolfo, P. K. Chan, E. Eskin, W. Fan, M. Miller, S. Hershkop, and J. Zhang, "Real time data mining-based intrusion detection," in *DISCEX*, 2001.

[41] K. Scarfone and P. Mell, "Guide to intrusion detection and prevention systems (idps)," NIST, Tech. Rep., 2012.

[42] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Scalable zero knowledge via cycles of elliptic curves," *Algorithmica*, 2017.

[43] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "Recursive composition and bootstrapping for snarks and proof-carrying data," in *STOC*, 2013.

[44] A. Chiesa, D. Ojha, and N. Spooner, "Fractal: Post-quantum and transparent recursive proofs from holography," in *Eurocrypt*, 2020.

[45] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," in *STOC*, 1986.

[46] R. Merkle, "Merkle tree patent," 1979.

[47] S. S. Iyengar, R. L. Kashyap, V. K. Vaishnavi, and N. S. V. Rao, "Multidimensional data structures: Review and outlook," *ADV COMPUT*, 1988.

[48] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (SUNDR)," in *OSDI*, 2004.

[49] O. Goldreich, *Foundations of cryptography: volume 1, basic tools*, 2007.

[50] "Secret Double Octopus — passwordless high assurance authentication," https://doubleoctopus.com, apr. 21, 2019.

[51] A. Shamir, "How to share a secret," *CACM*, 1979.

[52] F. Wang, J. Mickens, N. Zeldovich, and V. Vaikuntanathan, "Sieve: Cryptographically enforced access control for user data in untrusted clouds," in *NSDI*, 2016.

[53] B. Laurie, "Improving ssl certificate security," security.googleblog.com/2011/04/improving-ssl-certificate-security.html.

[54] H. Adkins, "An update on attempted man-in-the-middle attacks," security.googleblog.com/2011/08/update-on-attempted-man-in-middle.html.

[55] A. Whalley, "Distrusting wosign and startcom certificates," security.googleblog.com/2016/10/distrusting-wosign-and-startcom.html.

[56] R. Mandalia, "Security breach in ca networks -comodo, diginotar, globalsign," blog.isc2.org/isc2_blog/2012/04/test.html.

[57] A. Niemann and J. Brendel, "A survey on ca compromises," 2013.

[58] P. F. encrypted email, https://protonmail.com/.

[59] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions," in *FOCS*, 1999.

[60] S. Ghemawat and J. Dean, "Leveldb," https://github.com/google/leveldb.

[61] M. J. Dworkin, "Sha-3 standard: Permutation-based hash and extendable-output functions," Tech. Rep., 2015.

[62] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Yang, "High-speed high-security signatures," *JCEN*, 2012.

[63] "CONIKS Go," github.com/coniks-sys/coniks-go.

[64] L. Nguyen, "Accumulators from bilinear pairings and applications," in *CT-RSA*, 2005.

[65] "Libaad," github.com/alinush/libaad-ccs2019/tree/master/experiments.

[66] B. Li, J. Lin, F. Li, Q. Wang, Q. Li, J. Jing, and C. Wang, "Certificate transparency in the wild: Exploring the reliability of monitors," in *CCS*, 2019.

[67] S. Radicati, "Email statistics report, 2014-2018," www.radicati.com/wp/wp-content/uploads/2014/01/Email-Statistics-Report-2014-2018-Executive-Summary.pdf.

[68] W. W. Cohen, "Enron email dataset," www.cs.cmu.edu/ enron.

[69] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor, "Accountable key infrastructure (aki) a proposal for a public-key validation infrastructure," in *WWW*, 2013.

[70] D. Basin, C. Cremers, T. Kim, A. Perrig, R. Sasse, and P. Szalachowski, "Arpki: Attack resilient public-key infrastructure," in *CCS*, 2014.

[71] P. Szalachowski, S. Matsumoto, and A. Perrig, "Policert: Secure and flexible tls certificate management," in *CCS*, 2014.
[72] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward, "Marlin: Preprocessing zksnarks with universal and updatable srs," in *Eurocrypt*, 2020.
[73] D. Boneh, B. Bünz, and B. Fisch, "Batching techniques for accumulators with applications to iops and stateless blockchains," in *Crypto*, 2019.
[74] R. Dahlberg, T. Pulls, J. Vestin, T. Høiland-Jørgensen, and A. Kassler, "Aggregation-based gossip for certificate transparency," 2019.
[75] M. Chase and S. Meiklejohn, "Transparency overlays and applications," in *CCS*, 2016.
[76] B. Dowling, F. Günther, U. Herath, and D. Stebila, "Secure logging schemes and certificate transparency," in *ESORICS*, 2016.
[77] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Eurocrypt*, 1987.
[78] R. Dahlberg, T. Pulls, and R. Peeters, "Efficient sparse merkle trees," in *NordSec*, 2016.
[79] J. Kalidhindi, A. Kazorian, A. Khera, and C. Pari, "Angela: A sparse, distributed, and highly concurrent merkle tree."
[80] M. Lokhava, G. Losa, D. Mazières, G. Hoare, N. Barry, E. Gafni, J. Jove, R. Malinowsky, and J. McCaleb, "Fast and secure global payments with stellar," in *SOSP*, 2019.
[81] M. Ali, J. Nelson, R. Shea, and M. J. Freedman, "Blockstack: A global naming and storage system secured by blockchains," in *USENIX ATC*, 2016.
[82] S. Matsumoto and R. M. Reischuk, "Ikp: Turning a pki around with decentralized automated incentives," in *S&P*, 2017.
[83] Z. Wang, J. Lin, Q. Cai, Q. Wang, D. Zha, and J. Jing, "Blockchain-based certificate transparency and revocation transparency," *TDSC*, 2020.
[84] M. Milutinovic, W. He, H. Wu, and M. Kanwal, "Proof of luck: An efficient blockchain consensus protocol," in *SysTEX*, 2016.
[85] F. Zhang, I. Eyal, R. Escriva, A. Juels, and R. Van Renesse, "REM: Resource-efficient mining for blockchains," in *USENIX Security*, 2017.
[86] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *EuroS&P*, 2019.
[87] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *USENIX Security*, 2018.
[88] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun, "Verena: End-to-end integrity protection for web applications," in *S&P*, 2016.
[89] Y. Hu, S. Kumar, and R. A. Popa, "Ghostor: Toward a secure data-sharing system from decentralized trust," in *NSDI*, 2020.
[90] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket, "Venus: Verification for untrusted cloud storage," in *CCSW*, 2010.
[91] R. De La Briandais, "File searching using variable length keys," in *WJCC*, 1959.

APPENDIX A
SECURITY OF MERKLE^2

Before we can prove the security of Merkle^2, we must first consider the properties of the data structure. Our first step is thus to define the security properties of the prefix trees and chronological trees that make up Merkle^2's data structure in Appendix A-A and Appendix A-B. Armed with these properties, we prove Guarantee 1 in Appendix A-C.

A. Prefix trees

In this section, we briefly introduce prefix trees [1] used in Merkle^2. The prefix tree is a binary trie [91] built over a set of ID-value pairs S. Each node in the prefix tree is labeled with an index that is defined recursively: the root node is labeled with an empty index string; given a node with index p, its left and right children are labeled with indices $p +^0 0^0$ and $p +^0 1^0$, respectively. The leaf node with prefix p stores the hash of all the ID-value pairs $\langle \mathsf{ID}; \mathsf{val} \rangle$ in S for which ID is equal to the index p. The internal node hash will be the hash of its left and right children hashes. We use the term *empty node* to denote any node with an index that is not a prefix of any ID in S. The empty node has a special hash to avoid a collision in the same location [1]. We define the authentication path in prefix trees below.

Definition A.1 (Authentication paths in prefix trees)**.** *Given a prefix tree $\mathcal{T}$ and a node v with index $\mathcal{I}$, the authentication path of v contains node hashes that are on the co-path of v.*

The server proves membership by presenting the authentication path for the leaf node with index matching ID. If the user successfully recomputes the root hash, it ensures that the server returns all the values for ID in S. The server can prove that there is no value for ID in S by presenting the authentication path for an empty node, whose index is the prefix of ID.

Given a prefix tree root hash Root, an ID, the node hash H (H can be the hash of the empty node with index that is the prefix of ID), and an authentication path Path, we denote by $\mathsf{Pre\ xTree{:}Check}(\mathsf{Root}; \mathsf{ID}; \mathsf{H}; \mathsf{Path})$ the process to check the authentication path. $\mathsf{Pre\ xTree{:}Check}(\mathsf{Root}; \mathsf{ID}; \mathsf{H}; \mathsf{Path}) = 1$ means that users successfully recompute the root hash Root. We define (non-)membership security of prefix trees below.

Guarantee A.2 ((Non-)Membership security of prefix trees)**.** *Assume that the hash function used in the prefix tree is a collision-resistant hash function [49]. For all polynomial-time adversaries $\mathcal{A}$ there exists a negligible function $(\cdot)$ such that:*

$$\Pr\left[\begin{array}{l}(\mathsf{Root}; \mathsf{ID}; \mathsf{H}_1; \mathsf{Path}_1; \mathsf{H}_2; \mathsf{Path}_2) \leftarrow \mathcal{A}(1^{\ }) : \\ \quad \mathsf{H}_1 \neq \mathsf{H}_2 \\ \quad \wedge \\ \quad \mathsf{Pre\ xTree{:}Check}(\mathsf{Root}; \mathsf{ID}; \mathsf{H}_1; \mathsf{Path}_1) = 1 \\ \quad \wedge \\ \quad \mathsf{Pre\ xTree{:}Check}(\mathsf{Root}; \mathsf{ID}; \mathsf{H}_2; \mathsf{Path}_2) = 1\end{array}\right] = (\ ) :$$

This is a standard security property of Merkle prefix trees [1]; thus, we do not prove it here due to space constraints.

B. Chronological trees

In this section, we introduce chronological trees in Merkle^2. Our chronological trees are different from those in CT [3] since the internal nodes contain root hashes of prefix trees. A chronological tree is a full binary tree, whose leaves are created chronologically. A chronological forest is a set of chronological trees as discussed in Section IV-A. We define the index for a node v as a tuple [L:R], where the subtree rooted at v includes all leaves with position numbers L to R. If v is a leaf, L and R will be the position number of v. The node index enables us to specify the location of node in the chronological forest. We define the authentication path below.

Definition A.3 (Authentication path in chronological trees)**.** *Given a chronological forest $\mathcal{F}$, a chronological tree $\mathcal{T}$ in $\mathcal{F}$, and a node v with index $[\mathsf{L} : \mathsf{R}]$ in $\mathcal{T}$, the authentication path of v consists of the following hashes:*

prefix tree root hashes in all the ancestor nodes of v in $\mathcal{T}$;

the node hashes that are on the co-path of v in $\mathcal{T}$.

The server uses the authentication path to prove the membership of both leaf and internal nodes. That is, given a prefix tree root hash, the server can prove that it is stored in a node in the chronological forest. For example, the server first proves a node v has the node hash H_v by presenting the authentication path of v. Then, the server can provide the node hashes $\mathsf{H}_{\mathsf{Left}}; \mathsf{H}_{\mathsf{Right}}$ of left and right children of v, and the user checks by recomputing H_v using the prefix tree root hash.

Given a digest $\mathcal{D}$ of a chronological forest (which contains hashes of all the root nodes in the forest and the forest size), a node index $\mathcal{I}$, the node hash H, and an authentication path Path, we denote by $\mathsf{ChronForest{:}Check}(\mathcal{D}; \mathcal{I}; \mathsf{H}; \mathsf{Path})$ the process to check the authentication path. The user first finds the chronological tree root hash Root in $\mathcal{D}$ based on the forest size and the index $\mathcal{I}$. If $\mathsf{ChronForest{:}Check}(\mathcal{D}; \mathcal{I}; \mathsf{H}; \mathsf{Path}) = 1$, then the user successfully recomputed the root hash Root and ensured that the node with index $\mathcal{I}$ has node hash H. If the node with index $\mathcal{I}$ is the root node, then Path is empty, and the user will compare H with Root directly. We define membership security of chronological trees below.

Guarantee A.4 (Membership security of chronological trees)**.** *Assume that the hash function used in chronological trees is a collision-resistant hash function [49]. For all polynomial-time adversaries $\mathcal{A}$ there exists a negligible function $(\cdot)$ such that:*

$$\Pr\left[\begin{array}{c}(\mathcal{D}; \mathcal{I}; \mathsf{H}_1; \mathsf{Path}_1; \mathsf{H}_2; \mathsf{Path}_2) \leftarrow \mathcal{A}(1^\lambda): \\ \mathsf{H}_1 \neq \mathsf{H}_2 \\ \wedge \\ \mathsf{ChronForest{:}Check}(\mathcal{D}; \mathcal{I}; \mathsf{H}_1; \mathsf{Path}_1) = 1 \\ \wedge \\ \mathsf{ChronForest{:}Check}(\mathcal{D}; \mathcal{I}; \mathsf{H}_2; \mathsf{Path}_2) = 1\end{array}\right] = \mu(\lambda):$$

One can prove the above security guarantee via a straightforward extension to the membership security proof in CT [3]. We do not include it here due to space constraints.

The chronological tree in Merkle^2 can also provide the extension proof for updates. At the end of each epoch, the server provides the extension proof to auditors to show that the update does not modify existing node hashes. Given two digests $\mathcal{D}_1; \mathcal{D}_2$ and an extension proof π, we denote by $\mathsf{Extension{:}Check}(\mathcal{D}_1; \mathcal{D}_2; \pi)$ the process to check the extension proof. Auditors first check the forest size in $\mathcal{D}_2$ is greater than the size in $\mathcal{D}_1$, and then try to recompute root hashes in $\mathcal{D}_2$ using $\mathcal{D}_1$ and π. If $\mathsf{Extension{:}Check}(\mathcal{D}_1; \mathcal{D}_2; \pi) = 1$, then auditors will accept the update to the chronological forest. We define the append-only property of chronological trees below.

Guarantee A.5 (Append-only security of chronological trees)**.** *Assume that the hash function used in chronological trees is a collision-resistant hash function [49]. For all polynomial-time adversaries $\mathcal{A}$ there exists a polynomial-time extractor $\mathcal{E}$ and a negligible function $(\cdot)$ such that:*

$$\Pr\left[\begin{array}{c}(\mathcal{D}_1; \mathcal{D}_2; \pi; \mathcal{I}; \mathsf{H}; \mathsf{Path}_1) \leftarrow \mathcal{A}(1^\lambda); \\ \mathsf{Path}_2 \leftarrow \mathcal{E}(1^\lambda; \mathcal{D}_1; \mathcal{D}_2; \pi; \mathcal{I}; \mathsf{H}; \mathsf{Path}_1): \\ \mathsf{Extension{:}Check}(\mathcal{D}_1; \mathcal{D}_2; \pi) = 1 \\ \wedge \\ \mathsf{ChronForest{:}Check}(\mathcal{D}_1; \mathcal{I}; \mathsf{H}; \mathsf{Path}_1) = 1 \\ \Downarrow \\ \mathsf{ChronForest{:}Check}(\mathcal{D}_2; \mathcal{I}; \mathsf{H}; \mathsf{Path}_2) = 1\end{array}\right] = 1 - \mu(\lambda):$$

Proof. We show how to construct the authentication path Path_2 by using Path_1 and π. Root_1 and Root_2 each denotes the root hash of the tree, from $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively, that contains the node v with index $\mathcal{I}$. There are two cases:
Case 1. $\mathsf{Root}_1 = \mathsf{Root}_2$. We can use Path_1 as the authentication path between the node v and the root hash Root_2.
Case 2. $\mathsf{Root}_1 \neq \mathsf{Root}_2$. The extension proof π between $\mathcal{D}_1$ and $\mathcal{D}_2$ must contain necessary hashes that enable us to compute Root_2 from $\mathcal{D}_1$; thus, we can construct an authentication path Path' between Root_1 and Root_2 by obtaining hashes from $\mathcal{D}_1$ and π. Finally, we can construct Path_2 by merging Path_1 and Path' since Path_1 enables us to compute Root_1 from H, and Path' enables us to compute Root_2 from Root_1. □

We prove a lemma that is useful in Merkle^2. Although auditors only check the extension proof between neighboring epochs, there exist a extension proof between any two epochs.

Lemma A.6. *Assume that the hash function used in chronological trees is a collision-resistant hash function [49]. For all polynomial-time adversaries $\mathcal{A}$ there exists a polynomial-time extractor $\mathcal{E}$ and a negligible function $(\cdot)$ such that:*

$$\Pr\left[\begin{array}{c}(\mathcal{D}_1; \mathcal{D}_2; \mathcal{D}_3; \pi_{12}; \pi_{23}) \leftarrow \mathcal{A}(1^\lambda); \\ \pi_{13} \leftarrow \mathcal{E}(1^\lambda; \mathcal{D}_1; \mathcal{D}_2; \mathcal{D}_3; \pi_{12}; \pi_{23}): \\ \mathsf{Extension{:}Check}(\mathcal{D}_1; \mathcal{D}_2; \pi_{12}) = 1 \\ \wedge \\ \mathsf{Extension{:}Check}(\mathcal{D}_2; \mathcal{D}_3; \pi_{23}) = 1 \\ \Downarrow \\ \mathsf{Extension{:}Check}(\mathcal{D}_1; \mathcal{D}_3; \pi_{13}) = 1\end{array}\right] = 1 - \mu(\lambda):$$

Proof. The proof is similar to the proof of Guarantee A.5. We can construct the extension proof π_{13} by obtaining necessary hashes from π_{12} and π_{23}. □

C. Security proof of Guarantee 1

We will perform a reduction to show that if there exists an adversary $\mathcal{B}$ that can compromise lookup results without being detected, then there exists an adversary $\mathcal{A}$ that can violate the security properties of prefix trees or chronological trees, the collision-resistance of the hash function, or the existential unforgeability of the signature scheme.

Since $\mathcal{B}$ is not detected, $\mathcal{B}$ can violate the property in Guarantee 1 for a user u in C who looks up ID in epoch e, where $E_1 < e \leq E_2$, while remaining undetected by the owner r of ID, the user u, and auditors in A. We denote by S_e the ordered list of ID-value pairs and their position numbers that are appended by r before epoch e, and S_e^0 the ordered

list received by u as the lookup result in epoch e. Because $S_e \neq S_e^0$, $\mathcal{B}$'s attack must fall into one of three cases.

1) The first element in S_e and S_e^0 are identical, but there exist an element that is in S_e^0 but NOT in S_e.
2) The first element in S_e and S_e^0 are identical, but there exist an element that is NOT in S_e^0 but in S_e.
3) The first element in S_e and S_e^0 are NOT identical.

We will show that no matter which of the above three cases describes $\mathcal{B}$'s attack, $\mathcal{A}$ can either break the security of Merkle trees, find a hash collision, or forge the signature. We denote by p the position number of the first element elem in S_e, and p^0 the position number of the first element elem^0 in S_e^0.

Case 1. In this case, the first ID-value pair in the lookup result is correct ($\mathsf{p} = \mathsf{p}^0$ and $\mathsf{elem} = \mathsf{elem}^0$), but there exist other ID-value pairs that are not appended by the ID owner r. Due to the signature chain design, each ID-value pair is associated with a verifying key.

The fact that the first elements of S_e and S_e^0 are identical does not imply that the user can retrieve the correct verifying key associated with the first element in the Merkle^2's design; thus, we begin by proving that the verifying key that the user u received for the first ID-value pair is the one appended by the owner, r. The lookup protocol has the user u check the prefix tree of a node in the chronological forest with the index [L:R] in epoch e, where $\mathsf{L} \leq \mathsf{p} \leq \mathsf{R}$ and [L:R] is the root node in epoch e (if $\mathsf{L} = \mathsf{R}$, the user u will check the node hash directly). The monitoring protocol also has the owner r check the prefix tree of the node [L:R] in epoch E_2.

We denote by $\mathsf{H}_{[\mathsf{L:R}]}$ the hash of the node in the chronological forest with index [L:R]. If $e = E_2$, the hash $\mathsf{H}_{[\mathsf{L:R}]}$ is the root hash in the digest $\mathcal{D}_{E_2}(\mathcal{D}_e)$. By our assumptions in Guarantee 1, both the owner r and the user u retrieve the same digest of epoch E_2 (which is equal to e) from honest auditors in A; thus, they will see the same hash for [L:R]. If $e < E_2$, because auditors in A check digests in epochs $e :: : E_2$, by Guarantee A.5 and Lemma A.6, there exists an authentication path for the node [L:R] and digest $\mathcal{D}_{E_2}$. By Guarantee A.4, the owner r and the user u will see the same node hash $\mathsf{H}_{[\mathsf{L:R}]}$.

If the user u and the owner r see different verifying keys, they will obtain different prefix tree root hashes for [L:R]; otherwise, Guarantee A.2 will be violated. Recall that they also obtain the same node hash $\mathsf{H}_{[\mathsf{L:R}]}$. Therefore, if $\mathcal{B}$ cheated without being detected by the user u and the owner r, we will successfully find a collision for $\mathsf{H}_{[\mathsf{L:R}]}$.

Now, we conclude that the user u received the correct verifying key associated with the first ID-value pair. We denote by elem^{00} the first element in S_e^0 but not in S_e. The user u will check the signature chain when receiving the lookup result; thus, elem^{00} must be associated with a signature that can be verified using the verifying key associated with the previous element. Because elem^{00} is the first compromised element in S_e^0, the attacker must provide a valid signature without knowing the signing key associated with the previous element. This violates the existential unforgeability of the signature scheme.

Case 2. In this case, the first ID-value pair in the lookup result is correct, but there exists an ID-value pair that the user u did not receive but was appended by the owner, r. We denote by p the position number of the ID-value pair $\langle \mathsf{ID}; \mathsf{val} \rangle$ that is in S_e but not in S_e^0. Similar to the previous case, both the user u and the owner r are supposed to check the prefix tree associated with the node [L:R] where $\mathsf{L} \leq \mathsf{p} \leq \mathsf{R}$ and [L:R] is the root node in epoch e. Using the same argument, the owner r and the user u will see the same node hash $\mathsf{H}_{[\mathsf{L:R}]}$. However, because the user u did not receive $\langle \mathsf{ID}; \mathsf{val} \rangle$, u must check the non-membership proof of the prefix tree [L:R] as specified in the lookup protocol. u and r will obtain different prefix tree root hashes for [L:R] due to Guarantee A.2. Therefore, we can find a collision for $\mathsf{H}_{[\mathsf{L:R}]}$.

Case 3. In this case, the attacker $\mathcal{B}$ circumvents the signature chain by forging the first ID-value pair in S_e^0 ($\mathsf{elem} \neq \mathsf{elem}^0$). If $\mathsf{p} \leq \mathsf{p}^0$, the server must hide elem, otherwise the user u will notice elem for the same ID before elem^0. Then, we can use a similar argument as in case 1 to find a collision for the hash $\mathsf{H}_{[\mathsf{L:R}]}$, where $\mathsf{L} \leq \mathsf{p} \leq \mathsf{R}$ and [L:R] is the root in epoch e.

We denote by $[\mathsf{L}^0{:}\mathsf{R}^0]$ the node index where $\mathsf{L}^0 \leq \mathsf{p}^0 \leq \mathsf{R}^0$ and $[\mathsf{L}^0{:}\mathsf{R}^0]$ is the root in epoch e. We now consider the case where $\mathsf{p}^0 < \mathsf{p} \leq \mathsf{R}^0$. The monitoring protocol has the owner check the prefix tree of $[\mathsf{L}^0{:}\mathsf{R}^0]$, which is supposed to contain elem. However, elem in not S_e^0; thus, we can apply the same argument as in case 1 to find a collision for $\mathsf{H}_{[\mathsf{L}':\mathsf{R}']}$.

What about $\mathsf{p}^0 \leq \mathsf{R}^0 < \mathsf{p}$? The owner r will check the prefix tree associated with $[\mathsf{L}^0{:}\mathsf{R}^0]$ in the first-value proof. For simplicity, we assume the owner r checks the first-value proof in epoch $E_1 + 1$. The owner will choose a minimum set of chronological trees $\mathsf{Ct}_1; : : : ; \mathsf{Ct}_n$ to cover leaf nodes from 0 to $\mathsf{p} - 1$. Because $[\mathsf{L}^0{:}\mathsf{R}^0]$ is the root node in epoch e and $\mathsf{R}^0 < \mathsf{p}$, there must exist a chronological tree Ct_i whose root node is $[\mathsf{L}^0{:}\mathsf{R}^0]$. Through the first-value proof, the owner r will check the prefix tree associated with the root node of Ct_i; it is supposed not to contain elem^0. However, the user u can find elem^0 in the prefix tree of $[\mathsf{L}^0{:}\mathsf{R}^0]$. By Guarantee A.2 and Guarantee A.5, the user u and the owner r cannot obtain the same prefix tree root hash. But, the user u and the owner r obtain the same node hash $\mathsf{H}_{[\mathsf{L}':\mathsf{R}']}$. Therefore, we find a collision for $\mathsf{H}_{[\mathsf{L}':\mathsf{R}']}$.

APPENDIX B
PRE-BUILD STRATEGY

In this section, we describe the pre-build strategy in Merkle^2. The pre-build strategy amortizes the construction of prefix trees so that they are incrementally built as ID-value pairs are appended to Merkle^2. In other words, before a chronological tree internal node N_i is even created, Merkle^2 maintains its prefix tree P_i; as ID-value pairs are appended to Merkle^2, they are appended to P_i if they will be in the subtree rooted at N_i. In the process, the cost of building any prefix tree P_i is amortized across all appends to Merkle^2. Thus, when creating N_i, there is no additional overhead for building P_i.

For every internal node N_i—even those not yet created—in the ancestor chain of the ID-value pair, we must append the

ID-value pair to the corresponding prefix tree P_i. Because we have not set a limit on the maximum height of Merkle^2, each ID-value pair would have infinite ancestor nodes; thus, we choose a maximum height H so that the root to leaf path in Merkle^2 may not exceed H. Now, each ID-value pair will be a part of at most H prefix trees; so, we perform H appends to pre-built prefix trees in the ancestor chain, resulting in $O(H \log n)$ operations on each append. Because $H = O(\log(n))$, the cost of each append is still $O(\log^2(n))$.

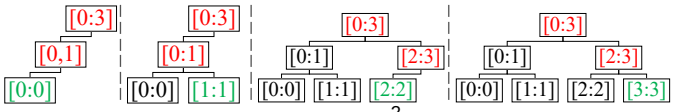

Fig. 15: The progression of Merkle^2 through 4 appends. The maximum height is set to H = 2. The green nodes represent the most recent appends to Merkle^2. Red indicates the internal nodes whose prefix trees the ID-value pair was appended to.

Fig. 15 illustrates the progression of Merkle^2 through 4 appends. The first ID-value pair is appended to the green node [0:0]. To maintain the pre-build strategy, the ID-value pair is also appended to the prefix trees of nodes [0:1] and [0:3]. Note that the internal nodes at [0:1] and [0:3] have yet to be created but that the pre-build strategy requires that we maintain their corresponding prefix trees. The next ID-value pair is appended to the node [1:1]. It is also added to the prefix trees of internal nodes [0:1] and [0:3] since they are in its ancestor chain. After the second append, the subtree rooted at [0:1] will no longer change because no future ID-value pairs will be added to [0:1]. The third append will be added to the node [2:2] and also to the prefix trees of nodes [2:3] and [0:3]. Finally, the last ID-value pair is appended to the node [3:3] and the prefix trees of nodes [2:3] and [0:3], completing this instance of Merkle^2 with height 2.

Appendix C
Compression Algorithm

In this section, we explain the compression algorithm for prefix trees. We show that a prefix tree with $O(n)$ ID-value pairs can be compressed so that it requires only $O(n)$ nodes.

We have already introduced prefix trees in Appendix A-A. We observe that many of the internal nodes in the prefix tree only have one child, and a chain of such nodes can be represented by a single compressed node instead; thus, in a compressed prefix tree, each node will either have 2 children or 0 children (leaf nodes). Each node will also store a partial prefix representing the compressed path. The internal node hash additionally includes the partial prefix. For example, in Fig. 16, nodes on the path from the root node to the leaf node associated with Alice's value only have one child; thus, they can be compressed as a single node in the compressed prefix tree. The partial prefix "001" represents the compressed path.

Now we explain how to add a new ID-value pair to the compressed prefix tree. If there exists no value for the ID in the prefix tree, we will find a compressed node v whose index only partially matches the ID. We denote by p the shared prefix

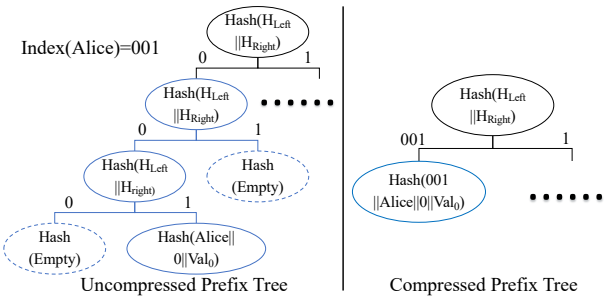

Fig. 16: Blue nodes in the left tree will be compressed into one node in the right tree because they are all nodes (internal or leaf) with at most one child.

between the index of v and the ID. Then, we split the node v and add three new compressed nodes: (1) a new node v' that replaces the old node v with the index p; (2) a child node u_0 of v' with the index matching that of v; (3) a child node u_1 of v' with the index matching the ID. The partial prefix of new nodes can be computed accordingly. The node u_1 will store the hash of the new ID-value pair.

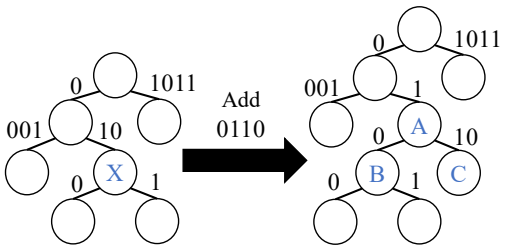

Fig. 17: Node X will be split into three nodes (A, B, C). The new ID-value pair will be stored in node C.

In Fig. 17, if a new ID-value pair with index "0110" is added, we will find the node X with index "010". Because "010" is not the prefix of "0110", we must decompress the node X into three nodes (node A, B, C). The index of node A is "01", which is the prefix of "0110", and node B inherits the index and children of node X. We also add node C as the child of node A to store the new ID-value pair.

Next, we analyze the storage cost of a compressed prefix tree. When adding a new ID-value pair, we only add a constant number (three) of new nodes to the prefix tree; thus, a compressed prefix tree with $O(n)$ appends only requires $O(n)$ nodes. Note that although adding a new ID-value pair creates only three new nodes, it still costs $O(\log(n))$ time per append because we must update the hash of $O(\log(n))$ ancestor nodes.