Balloon: A Forward-Secure Append-Only Persistent Authenticated Data Structure

Tobias Pulls
Dept. of Mathematics and Computer Science
Karlstad University
Universitetsgatan 2, Karlstad, Sweden
tobias.pulls@kau.se

Roel Peeters
KU Leuven, ESAT/COSIC & iMinds
Kasteelpark Arenberg 10 bus 2452
3001 Leuven, Belgium
roel.peeters@esat.kuleuven.be

ABSTRACT

We present Balloon, a forward-secure append-only persistent authenticated data structure. Balloon is designed for an initially trusted author that generates events to be stored in a data structure (the Balloon) kept by an untrusted server, and clients that query this server for events intended for them based on keys and snapshots. The data structure is persistent such that clients can query keys for the current or past versions of the data structure based upon snapshots, which are generated by the author as new events are inserted. The data structure is authenticated in the sense that the server can verifiably prove all operations with respect to snapshots created by the author. No event inserted into the data structure prior to the compromise of the author can be modified or deleted without detection. Balloon supports efficient (non-)membership proofs and verifiable inserts by the author, enabling the author to verify the correctness of inserts without having to store a copy of the Balloon. We also sketch how to use Balloon to enable client-specific forward-secure author consistency. In case of author inconsistency, a client can make a publicly verifiable statement that shows that the author was inconsistent with respect to his events.

1. INTRODUCTION

This paper is motivated by the lack of an appropriate data structure that would enable to relax the trust assumptions needed in transparency logging. In the setting of transparency logging, an *author* logs messages intended for *clients* through a *server*: the author sends messages to the server, and clients poll the server for messages intended for it. In previous work [22], we constructed a cryptographic scheme in this setting for which we proved a number of security and privacy properties of that scheme in the *forward security* model: *both* the author and the server are assumed to be initially trusted and will become compromised at some point in time. Any messages logged *before* this compromise remain secure and private. To reduce the level of trust needed in the server, we also presented a secure hardware extension for the server [27]. With this extension the server does not need to be trusted, but the hardware needs to be.

In this paper, we propose a novel *append-only authenticated data structure* that allows the server to be untrusted without the need for trusted hardware. Our data structure, which is called Balloon[1], allows for efficient proofs of both membership and non-membership. As such, the server is forced to provide a verifiable reply to all queries. Hence the server needs to be honest or risk being caught when cheating. Since Balloon is append-only, we can greatly improve the efficiency in comparison with other authenticated data structures, such as persistent authenticated dictionaries [1].

As already discussed, Balloon can be used for transparency logging to make data processing by service providers transparent to data subjects whose personal data are being processed in a secure and privacy-friendly way. Balloon can also be also used as part of a secure logging system, similar to the history tree system by Crosby and Wallach [9]. In Certificate Transparency (CT) [15], Balloon can be used to provide efficient non-membership proofs, which are highly relevant in relation to certificate revocation for CT [14, 15, 21].

We make the following contributions:

- A novel append-only authenticated data structure called Balloon that allows for both efficient membership and non-membership proofs while keeping the storage and memory requirements minimal (Section 4).
- Efficient verifiable inserts into our append-only authenticated data structure that enable both the author and monitors to ensure consistency of the data structure (Section 4). Verifiable inserts can have applications for monitors in, e.g., [3, 13, 14, 15, 24, 28].
- A thorough security evaluation of Balloon and its algorithms (Section 5).
- How to use an authenticated data structure, like a Balloon, to achieve client-specific forward-secure author consistency and how the client can make author inconsistencies publicly verifiable (Section 6).

The rest of the paper is structured as follows. Section 2 defines our setting, adversary model, and high-level security and privacy requirements. Section 3 presents the intuition behind our idea and the two key building blocks. Section 7 presents related work and compares Balloon to prior work. Section 8 concludes the paper. Of independent interest, the appendix shows why probabilistic proofs are insufficient for ensuring consistency of a Balloon without greatly increasing the burden on the prover.

[1]Like an ideal balloon, a vacuum balloon, our balloon is filled with nothingness (seemingly random data) without giving in to any external pressures (attackers) to collapse (remove or replace nothingness with something).

2. SETTING

We have three types of entities: an author A, a server S, and one or more clients C. The author is trusted to insert events into a data structure kept by the server, where an event e is defined as being composed of a key k and a value v, such that $e = (k, v)$. Each insert into the data structure, kept by the server, produces a snapshot s that covers all data stored at the server up to that point. Clients query the server for events based on keys and snapshots, and the server proves the correctness of the replies to clients based on signatures made by the author as part of the snapshots. The author is assumed to either gossip snapshots to clients or that all snapshots are available publicly, e.g., on a website that clients continuously poll[2].

Figure 1 illustrates our setting, giving an example with one author, one server, and three distinct clients. For transparency logging, the author could be a data processor and the clients data subjects. For syslog, the author could be an intrusion detection system (IDS), the server a collector, and clients different security information and event management (SIEM) systems. For increasing transparency in certificate issuance, the author could be a certificate authority (CA) and clients browsers.

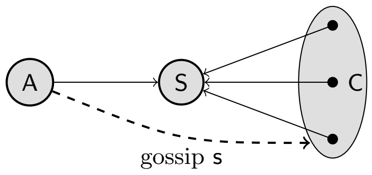

Figure 1: Our setting with one author A inserting events into a data structure kept by an untrusted server S. A gossips snapshots s to one or more clients C, who query S for events and may challenge S to prove the consistency of snapshots.

For achieving client-specific forward-secure author consistency, as described in Section 6, one key assumption is that the author and each client share a secret. The secret is used by the author to generate deterministic keys for events intended for specific clients. We assume that the secret is unique per client, and generated uniformly at random from a sufficiently large sample space: $\texttt{secret} \overset{R}{\leftarrow} \{0,1\}^{\phi}$ with 2^{ϕ} being the intended security level. How this secret is agreed upon is out-of-scope of this paper, but could, e.g., be generated using an authenticated key-agreement protocol [6].

2.1 Adversary Model and Assumptions

The author A is assumed to be initially trusted, but will turn into an active adversary at some point in time. Our goal is to protect events sent by A prior to compromise, commonly known as *forward security* [5]: events that are sent now are secure from a future compromise of A. Unlike A, we consider the server S to be an active adversary from the beginning. We assume that at least one client C is honest. For communication, we assume a secure channel between A, S, and all C. We explicitly consider availability out of scope, that is, A and S will always reply, however, their replies may be malicious.

[2]While exceptionally important for security, we consider the exact gossip mechanism out of scope of our work, as done in closely related work, such as [9, 11, 15].

We assume idealised cryptographic building blocks in the form of a hash function $\texttt{Hash}(\cdot)$ and signature scheme that is used to sign a message m and verify the resulting signature:

$$\{\top, \bot\} \leftarrow \texttt{Verify}_{\texttt{vk}}(\texttt{Sign}_{\texttt{sk}}(m), m) \quad (1)$$

The signature scheme should be existentially unforgeable under known message attack [12]. The hash function should be collision and pre-image resistant [23]. In Section 5 we show that these properties are sufficient for fulfilling the security and privacy requirements in our setting, as defined below. The message authentication ($\texttt{MAC}$) function, used in Section 6, should be existentially unforgeable [4].

2.2 Requirements

We identify the following high-level requirements related to the data structure kept at the server:

R1. Minimal trust in the server Even if an adversary controls the server, the server should be unable to cheat without violating any of our assumptions stated in Section 2.1.

R2. (Non-)membership proofs The server must always provide a reply (either a membership or a non-membership proof) that is provably consistent with a statement made by an honest author. (Non-)membership proofs for keys should be verifiable both by the author and the clients. The (non-)membership proofs keep the server honest, i.e., force the server to follow the protocol.

R3. Publicly verifiable integrity & deletion detection The data structure should be publicly verifiable in the sense that anyone can detect any modifications or deletions between snapshots.

R4. Minor overhead at the author The author should be able to discard most of the data that is stored at the server. In some settings the author may be greatly resource-constrained, e.g., as a log relay or sensor.

R5. Event privacy When the server proves the correctness of a reply to a query, the resulting proof should not leak any information about events other than the events which are being queried for (if applicable).

Requirements **R1**–**R3** focus on the verifiability of safely outsourcing events to an untrusted server by the author. **R4** is a matter of efficiency. Finally, **R5** impacts all other requirements in the sense that no proof from the server should leak information about events to an adversary that is not in control of the server.

3. OUR IDEA

The intuition behind Balloon is that S keeps two tree-based authenticated data structures, that we merge into one data structure (the Balloon), over all events created by A. Periodically, A sends new events to S. S updates its data structure and proves the correctness of the update to A. This convinces A to sign the roots of the two composing data structures, forming a snapshot, and the snapshot is gossiped to clients and shared with the server. Clients can then query S for events based on keys and snapshots.

Balloon is composed of two data structures: a history tree and a hash treap. The reason for this is simple: a history tree, as defined by Crosby and Wallach [9], can provide efficient membership proofs (for any snapshot), while a hash treap can provide efficient non-membership proofs. A similar approach of composing two data structures has been used by, e.g., ECT [24], DTKI [28] and Revocation Transparency [14]. In Section 7 we compare our solution to these solutions. Next, we present our two composing data structures before defining how they are merged together into one data structure in Section 4 to form a Balloon.

3.1 History Tree

A tamper-evident history system, as defined by Crosby and Wallach [9], consists of a *history tree* data structure and five algorithms. A history tree is in essence a *versioned* Merkle tree [18] (hash tree). Each leaf node in the tree is the hash of an event, while interior nodes are labeled with the hash of its children nodes in the subtree rooted at that node. The root of the tree fixes the content of the entire tree. Different versions of history trees, produced as events are added, can be proven to make consistent claims about the past. The five algorithms, adjusted to our terminology, were initially defined by Crosby and Wallach [9], as follows:

- $\texttt{H.Add}(e) \rightarrow c_i$. Given an event e the system appends it to the history tree H as the i:th event and then outputs a commitment[3] c_i.
- $\texttt{H.MembershipGen}(i, c_j) \rightarrow (P, e_i)$. Generates a membership proof P for the i:th event with respect to commitment c_j, where $i \le j$, from the history tree H. The algorithm outputs P and the event e_i.
- $\texttt{H.IncGen}(c_i, c_j) \rightarrow P$. Generates an incremental proof P between c_i and c_j, where $i \le j$, from the history tree H. Outputs P.
- $\texttt{P.MembershipVerify}(i, c_j, e'_i) \rightarrow \{\texttt{true}, \texttt{false}\}$. Verifies that P proves that e'_i is the i:th event in the history defined by c_j (where $i \le j$). Outputs $\texttt{true}$ if true, otherwise $\texttt{false}$.
- $\texttt{P.IncVerify}(c'_i, c_j) \rightarrow \{\texttt{true}, \texttt{false}\}$. Verifies that P proves that c_j fixes every event fixed by c'_i (where $i \le j$). Outputs $\texttt{true}$ if true, otherwise $\texttt{false}$.

Proof generation works in essence by creating a *pruned tree* containing just the nodes needed to convince the verifier. The verifier traverses the pruned tree and is provided with the values needed to compute the relevant hashes. Every other value can safely be discarded due to the properties of the cryptographic hash function [9]. For membership proofs, the pruned tree in the proof forms an *authenticated path* from the root of the tree in the commitment c_j to the i:th event (stored in the i:th leaf). Incremental proofs between two commitments c_i and c_j are generated by creating a pruned tree that enables the verifier to compute *both* c_i and c_j from the same pruned tree. Similar to membership queries, this boils down to authenticated paths. While not mentioned by Crosby and Wallach, the security of authenticated paths in Merkle trees are well studied. Coronado [7] shows that the security of the authenticated path in the Merkle tree, as part of the Merkle signature scheme, reduces to finding a collision for the underlying hash function.

[3]A commitment c_i is the root of the history tree for the i:th event, signed by the system.

3.2 Hash Treap

A treap is a type of randomised binary search tree [2], where a binary search tree is balanced using heap priorities. Each node in a treap has a key, value, priority, and a left and a right child. A treap has three important properties:

1. Traversing the treap in order gives the sorted order of the keys;
2. Treaps are structured according to the nodes' priorities, where each node's children have lower priorities;
3. Given a deterministic attribution of priorities to nodes, a treap is *set unique* and *history independent*, i.e., its structure is unique for a given set of nodes, regardless of the order in which nodes were inserted, and the structure does not leak any information about the order in which nodes were inserted.

When a node is inserted in a treap, its position in the treap is first determined by a binary search. Once the position is found, the node is inserted in place, and then rotated upwards towards the root until its priority is consistent with the heap priority. When the priorities are assigned to nodes using a cryptographic hash function, the tree becomes probabilistically balanced with an expected depth of $\log n$, where n is the number of nodes in the treap. Inserting a node takes expected $\mathcal{O}(\log n)$ operations and results in expected $\mathcal{O}(1)$ rotations to preserve the properties of the treap [8]. Given a treap, it is straightforward to build a hash treap: have each node calculate the hash of its own attributes[4] together with the hash of its children. Since the hash treap is a Merkle tree, its root fixes the entire hash treap. The concept of turning treaps into Merkle trees for authenticating the treap has been used for example in the context of persistent authenticated dictionaries [10] and authentication of certificate revocation lists [21].

We define the following algorithms on our hash treap:

- $\texttt{T.Add}(k, v) \rightarrow r$. Given a unique key k and value v, where $|k| > 0$ and $|v| > 0$, the system inserts them into the hash treap T and then outputs the updated hash of the root r. The add is done with priority $\texttt{Hash}(k)$, which results in a deterministic treap. After the new node is in place, the hash of each node along the path from the root has its internal hash updated. The hash of a node is $\texttt{Hash}(v||k||\text{left.hash}||\text{right.hash})$.
- $\texttt{T.AuthPath}(k) \rightarrow (P, v)$. Generates an authenticated path P from the root of the treap T to the key k. The algorithm outputs P and, in case of when a node with key k was found, the associated value v. For each node i in P, k_i and v_i need to be provided to verify the hash in the authenticated path.
- $\texttt{P.AuthPathVerify}(k, v) \rightarrow \{\texttt{true}, \texttt{false}\}$. Verifies that P proves that k is a non-member if $v \overset{?}{=} \bot$ or otherwise a member.

Now that we have explained our idea and its two core building blocks, we can define a Balloon more precisely and the algorithms on a Balloon.

[4]The priority can safely be discarded since it is derived solely from the key and implicit in the structure of the treap.

4. BALLOON

Here we define a Balloon as a data structure composed of a hash treap and history tree, and a number of algorithms on a Balloon. Since a Balloon is append-only, we decided to only keep the latest version of the hash treap and consequently make the membership queries in the hash treap only against this latest version. To link both underlying data structures together, we insert events as follows: the hash of the event is first inserted into the history tree, the position (leaf index) in the history tree is then used as value together with the hash of the event key as key to insert a node to the hash treap. Note that events are inserted in batches or sets, and that only one snapshot will be generated for each set, as such the numbering of the snapshots might be non-consecutive, e.g., s_{10} is the next snapshot after s_5 if 5 events where added in the last insert.

We present the following algorithms for Balloon:

- $\texttt{B.Insert}(E) \rightarrow P$. Given a set of events E with unique keys and non-zero values the system appends it to the Balloon B, and then outputs proof P.

- $\texttt{P.VerifyInsert}(E, \mathsf{s}_l) \rightarrow \{\mathsf{s}_{l+|E|}, \texttt{false}\}$. Verifies that P proves that events E were correctly inserted into the Balloon with the latest verified snapshot s_l. Outputs the next snapshot $\mathsf{s}_{l+|E|}$ if the proof is correct, `false` otherwise.

- $\texttt{B.MembershipQuery}(k, \mathsf{s}_j) \rightarrow (P, \mathsf{s}_l, e_i)$. Generates a (non-)membership proof P for the event with key k from snapshot s_j for the Balloon B. The algorithm outputs P, the latest snapshot s_l, and, in the case of membership, the event e_i, where $i \leq j \leq l$.

- $\texttt{P.MembershipVerify}(k, \mathsf{s}_j, \mathsf{s}_l, i, e'_i) \rightarrow \{\texttt{true}, \texttt{false}\}$. Verifies that P proves the (non-)membership of the event e'_i with key k in s_l and, if member, that e'_i is the i:th event in s_j, where $i \leq j \leq l$.

To define our algorithms we make use of the high-level algorithms previously defined on history trees and hash treaps in Section 3, and two other algorithms: `BuildPrunedTree` and `root`. The `BuildPrunedTree` algorithm takes *one or more* authenticated paths in the same hash treap or history tree and constructs a *single* pruned Merkle tree based on the provided paths. The tree is *pruned* in the sense that several paths in the original tree the authenticated paths were generated in are replaced by the hashes of branches not visited along the paths. In case of operating on output from our `T.AuthPath`, the pruned tree qualifies as a pruned hash treap, since the output of `T.AuthPath` contains the key (k in Section 3.2) and priority (derived from k) used to structure the treap. This fact is later used for our verifiable insert algorithm. The `root` algorithm takes either a (pruned) Merkle tree or a commitment from a history tree as input and returns the hash of the root node that fixes the entire tree represented by the input.

Next, in Section 4.1, we describe how membership queries are performed (`B.MembershipQuery` and `P.MembershipVerify`). Section 4.2 presents how new events are inserted into the hash treap and history tree, resulting in new snapshots created by the author (`B.Insert` and `P.VerifyInsert`). Section 4.3 discusses hash treap and history tree consistency.

4.1 Membership Query

A membership query is performed between a client/author and a server. Algorithm 1 defines how the server, given a key k and a snapshot s_j, performs a membership query for an event with the provided key. First, in step 1, the server generates an authenticated path in the hash treap for the hash of the provided key. In steps 2–3, the proof of non-membership is complete if there was no event with the provided key, indicated by the empty value, or if the event was inserted *after* the provided snapshot was created. The proof, in step 3 (and in step 5), includes the latest snapshot s_l since the authenticated path in the hash treap is performed on the current hash treap at the server, not past versions, and the value of the key in the hash treap, which is the index (position) of the event in the history tree (if applicable). Step 4 generates a membership proof for the event at the position indicated by the value in the hash treap. This proof is performed on the commitment c_j, as part of s_j, since this was the snapshot the client queried for. Note that for a *membership* proof the hash treap is irrelevant. We include searching in the hash treap regardless since it provides us with an expected $\mathcal{O}(\log n)$ lookup time for the index of the event in the history tree, instead of assuming some other data structure facilitating this lookup. Finally, step 5 returns the full membership proof, including the event e_i in question together with the two proofs in the hash treap and history tree. In the history tree we store the hash of the event, and not the event itself. Where the server actually stores events is out of scope for our work, as noted by Crosby and Wallach [8].

Algorithm 1 A membership query for a key in a snapshot.

```
Require: The history tree $H$, the hash treap $T$, a key $k$, where
    $|k| > 0$, and a snapshot $\mathsf{s}_j$.
Ensure: A proof $P$ of the (non-)membership of an event with
    key $k$, the latest snapshot $\mathsf{s}_l$, and, if an event with the key
    exists, the event $e_i$, where $i \leq j \leq l$ and $e.k = k$.
1: $(P_T, i) \leftarrow T.\texttt{AuthPath}(\texttt{Hash}(k))$
2: if $i \overset{?}{=} \perp \vee i > j$ then     ▷ key not in treap or after snapshot
3:     return $(P \leftarrow (P_T, i), \mathsf{s}_l)$     ▷ $\mathsf{s}_l$ cached Algorithm 3
4: $(P_H, e_i) \leftarrow H.\texttt{MembershipGen}(i, c_j)$     ▷ $c_j$ is part of $\mathsf{s}_j$
5: return $(P \leftarrow (P_T, i, P_H), \mathsf{s}_l, e_i)$
```

Algorithm 2 describes how a client/author verifies a proof from a membership query, as defined in Algorithm 1. First, in steps 1–2, the client/author verifies the provided authenticated path in the hash treap and that the root of the authenticated path matches the root provided in the latest snapshot s_l (r_l is part of s_l). Steps 3–4 check if the value in the hash treap indicates a non-membership proof, either by the value being empty or the value indicating that the event was added *after* queried for snapshot s_j. Finally, step 5 verifies that the membership proof is correct in the history tree and that the root of the authenticated path in the history tree matches the commitment c_j (as part of s_j). Note that when we require as input to Algorithm 2 (and later also to Algorithms 3 and 4) one or more snapshots s, we assume that the verifier will indeed verify that each snapshot is valid, i.e., that it is signed by the author and of expected length.

Algorithm 2 Verify a membership query proof.

```
Require: A proof $P$, a key $k$, snapshots $\mathsf{s}_j$ and $\mathsf{s}_l$ where $j \leq l$,
    an optional event $e_i'$.
Ensure: true if the query is correct, false otherwise.
1: if $P_T$.AuthPathVerify(Hash($k$), $i$) $\neq$ true $\vee$ root($P_T$) $\neq r_l$
   then                                        ▷ $i$ is part of $P$, $r_l$ of $\mathsf{s}_l$
2:     return false               ▷ invalid path or root in treap proof
3: if $i \overset{?}{=} \perp \vee i > j$ then          ▷ no key in treap or event after $\mathsf{s}_j$
4:     return true
5: return root($P_H$) $\overset{?}{=} c_j \wedge$ $P_H$.MembershipVerify($i, c_j, e_i'$)    ▷
   $c_j$ is part of $\mathsf{s}_j$
```

4.2 Verifiable Insert

A verifiable insert is performed by the author and the server. The intuition is that the server will provide only the necessary pruned trees of the history tree and hash treap such that the author can perform the insert itself into the pruned trees and calculate the resulting roots. This convinces the author to sign the updated resulting roots of the history tree and hash treap, resulting in a new snapshot.

Algorithm 3 defines how the server inserts a set of events E, provided by the author, into the hash treap and history tree such that the author can verify the correctness of the insertion. A function Ω, that deterministically orders events, is part of the input to the algorithm. This is needed because the history tree is, unlike the hash treap, *not* history independent and we need to order the events in the history tree. As an added bonus, using Ω prevents leaking the order events were generated by the author for each run of the algorithm (see requirement **R5**). Step 1 generates a membership proof for the latest event inserted into the history tree. This authenticated path in the tree fixes all events inserted into the history tree this far [8]. Next, steps 2–3 generate $|E|$ authenticated paths in the hash treap and store the resulting pruned hash treap. This fixes the path in the hash treap to all events in E. The generation can be done in parallel, since the treap is not modified, and redundant nodes as part of the resulting pruned hash treap can safely be omitted. Steps 4–6 inserts all events into the hash treap and the history tree. Step 5 inserts the hash of the event into the history tree, and the commitment c_i is returned for the newly inserted event with index i. Step 6 inserts the hash of the event key as the key in the hash treap, and the value is the index of the hash of the event in the history tree. This merges the hash treap and history tree into one data structure. Finally, steps 7–9 add the updated roots of the two trees to the proof and return the proof.

Algorithm 4 verifies a proof from Algorithm 3 and is run by the author. First, in steps 1–2, the author verifies that the roots of the pruned history tree and hash treap matches the last verified roots by the author. Steps 3–4 check the membership proof in the history tree. Step 5 builds a pruned hash treap and history tree based on the nodes included in the proof. Steps 6–8 add all events to the pruned history tree and hash treap, similar to steps 4–6 in Algorithm 3. For the history tree, events are added chronologically and the authenticated path fixes all past events. For the hash treap, the authenticated paths also contain the hashed keys (and therefore also the priorities) of the nodes along the relevant paths. Last, but most importantly, step 9 compares the provided updated roots of the history tree and hash treap with the calculated roots.

Algorithm 3 Verifiably insert a set of events.

```
Require: A set of events $E$, the history tree $H$, the hash treap
    $T$, a function $\Omega$ that deterministically orders events, and the
    latest snapshot $\mathsf{s}_l$.
Ensure: The events have been inserted and a proof $P$ of correctness has been created.
1: $(P_H, P_e) \leftarrow H$.MembershipGen($l, c_l$)  ▷ $c_l$ is part of $\mathsf{s}_l$, $l$ is the
   latest index
2: for all $e \in E$ do                      ▷ This can be done in parallel
3:     $P_T \leftarrow P_T \cup T$.AuthPath(Hash($e_k$))
4: for all $e \in E$ in the order generated by $\Omega$ do
5:     $c_i \leftarrow H$.Add(Hash($e$))   ▷ the value is Hash($e$), the key the
   position $i$ in $H$
6:     $T$.Add(Hash($e_k$), $i$)  ▷ key is Hash($e_k$), value the position $i$
7: $P_{H_r} \leftarrow c_i$              ▷ Only the latest $c_i$ from adding events
8: $P_{T_r} \leftarrow$ root($T$)                         ▷ The root of the treap
9: return $P$
```

Algorithm 4 Verify the correctness of an insertion proof.

```
Require: A set of events $E$, a function $\Omega$ that deterministically
    orders events, a proof $P$, the last verified hash treap root $T_r$
    and history tree commitment $c_l$.
Ensure: true if the proof is correct, false otherwise.
1: if root($P_H$) $\neq c_l \vee$ root($P_T$) $\neq T_r$ then
2:     return false
3: if $P_H$.MembershipVerify($l, c_l, P_e$) $\neq \top$ then
4:     return false
5: $H \leftarrow$ BuildPrunedTree($P_H$), $T \leftarrow$ BuildPrunedTree($P_T$)
6: for all $e \in E$ in the order generated by $\Omega$ do
7:     $c_i \leftarrow H$.Add(Hash($e$))
8:     $T$.Add(Hash($e_k$), $i$)
9: return root($H$) $\overset{?}{=}$ root($P_{H_r}$) $\wedge$ root($T$) $\overset{?}{=} P_{T_r}$
```

If Algorithm 4 succeeds, the author creates a snapshot:

$$s_i \leftarrow \mathtt{Sign}_{\mathsf{sk}}\big(c_i || \mathtt{root}(T)\big) \tag{2}$$

The new snapshot is gossiped to clients and shared with the server. Now that we have the basic functionality of querying and updating a Balloon in place, we look at *consistency*, a key consideration for security in the setting when the author may become compromised.

4.3 Ensuring Consistency

Efficient incremental proofs, realised by the `IncGen` and `IncVerify` algorithms, are a key feature of history trees [8]. Anyone can challenge the server to provide a proof that one commitment as part of a snapshot is *consistent* with another commitment as part of another snapshot. Consistency means that all commitments make consistent claims about the past, i.e., the data structure is append-only. While these proofs give strong assurance about the consistency of commitments on a history tree, these say nothing about the signed roots of a hash treap as part of snapshots. We are not aware of any efficient algorithms for incremental proofs in hash treaps (or any lexicographically sorted data structure), it appears to be an open problem [11], and we note it as interesting future work. In the appendix, we show why one cannot efficiently use probabilistic proofs of consistency for a Balloon. In absence of efficient (both for the server and verifier in terms of computation and size) incremental proofs in hash treaps, we rely on a concept from, e.g., Certificate Transparency: monitors [15].

We assume that a subset of clients, or any third party, will take on a role referred to as a “monitor”, “auditor”, or "validator" in ,e.g., [3, 13, 14, 15, 24, 28]. A monitor continuously monitors all data stored at a server and ensures that all snapshots issued by an author are consistent. Note that a monitor is not directly concerned about *what* is stored in a server, just that the snapshots are consistent (ensuring the append-only property of the Balloon). A monitor can request *all* hashes of events and hashes of event keys as part of a snapshot from the server. This, together with the function Ω[5] for a Balloon, enables the monitor to continuously reconstruct both the hash treap and history tree starting from scratch. When a monitor has verified the consistency of all snapshots generated so far, then the monitor has an identical copy of the history tree and hash treap as the server. At this point in time, the monitor can take the same approach as the author for verifiably inserts: the monitor can remove all data and only keep the latest verified snapshot. When new events are inserted, the server can forward the insertion proof P to the monitors as well as to the author. Figure 2 illustrates this setting. The monitor will also need hashes from each new event, E', which contains the hashes of each event and its key. With this, monitors can use a version of Algorithm 4, denoted Algorithm 4' in Figure 2, to verify updates (steps 7 and 8 need to be modified). This saves storage at monitors at the cost of the extra bandwidth needed for sending also the insertion proofs P to the monitors.

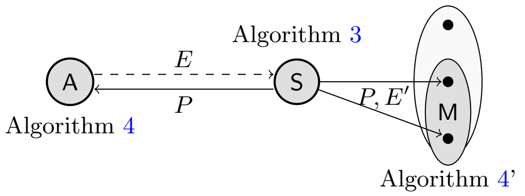

Figure 2: Clients may take on the role of a monitor, M, and receive proofs (P) and hashes of events (E') from the server generated by Algorithm 3 as part of the verifiable insert of new events by the author. This enables monitors to continuously verify snapshots without a copy of the Balloon.

It may be in the interest of server to store one or more proofs and events (P, E') for the latest x snapshots to facilitate monitors that only periodically verifies consistency. The server can of course opt to recompute P and/or E' for any snapshot, but calculating past versions of a hash treap are relatively expensive [11]. We note that there are with a high-probability redundancies between subsequent insertion proofs, just like there are redundant nodes in the authenticated paths of the hash treap when inserting events (Algorithm 3). This redundancy could be used to make subsequent proofs more efficient for both authors and monitors that are willing to use some storage to keep past proofs. We leave this to future work.

4.4 Event Privacy

A membership query for a Balloon involves an authenticated path in the hash treap. Along the authenticated path in the hash treap, each node contains a key and value. The key is the hash of an event key (due to step 3 in Algorithm 3), and the value the index of an event in the history tree. Thanks to hashing event keys, an authenticated path in a hash treap does not reveal event keys that can be used to query the server for valid events. However, the authenticated path reveals the relationship between *other* hashed event keys and their values, i.e., *when* other events have been inserted into the Balloon, relative to other events as they are represented in Balloon. In settings where the server may be considered semi-trusted, it may be of interest to hide this information. Also note that there is little point in trying to hide values in the hash treap from monitors, since monitors will learn the relationship between internal nodes as a consequence of reconstructing the Balloon.

One can hide the relationship between nodes in authenticated paths by making the following changes in the algorithms on the hash treap specified in Section 3.2:

- $\texttt{T.Add}(k, v) \to r$. A node key is calculated as $k' \leftarrow \texttt{Hash}(k)$ and the priority $\texttt{Hash}(k')$. The hash for a node is $\texttt{Hash}\big(\texttt{Hash}(v||k)||k'||\text{left.hash}||\text{right.hash}\big)$.
- $\texttt{T.AuthPath}(k) \to (P, v)$. For each node i in P, $\texttt{Hash}(v_i||k_i)$ and k'_i are provided, not v_i or k_i.
- $\texttt{P.AuthPathVerify}(k, v) \to \{\texttt{true}, \texttt{false}\}$. Verifies the authenticated path using the updated form of calculating the hash of a node. If a membership proof $(v \neq \perp)$, $\texttt{Hash}(v||k)$ of the node with the queried for key is calculated and compared to the output included in P.

The main change above involves in each node hashing the value v together with k, $\texttt{Hash}(v||k)$, to hide the value. It is not sufficient to hash the value alone, since it is an incremental index of leaf nodes in the history tree. Presumably, keys in the hash treap have sufficient entropy to prevent brute-force attacks[6]. We therefore hash the value with the key k, and use the hash of the key k' as the node key in the hash treap. The verification algorithm, $\texttt{P.AuthPathVerify}$, verifies that both k and v are correct in authentication proofs. This only reveals a verifiable v for events which a client queries for, in line with our event privacy requirement.

5. EVALUATION

We will first do a security evaluation of the two main components of a Balloon: membership queries and verifiable insert. Verifiable insert is of great importance to prove consistency of our data structure with its past (and indirectly, that the two underlying trees are consistent with each other). Finally, we will evaluate how the requirements, as set in Section 2.2 are met.

5.1 Membership Queries

An adversary is said to forge a response to a membership query for a given key and snapshot if it can provide either:

1. a non-membership proof for an event with the given key k that was added to the Balloon before the given snapshot s_i, or
2. a membership proof for an event with the given key k that was not added to the Balloon before the given snapshot s_i.

[5]Note that for monitors, Ω has to produce the same order given only the hashes of events and their keys, as when given the original events.

[6]This is at least the case for our client-specific forward author consistency extension, as described in Section 6.

A snapshot is said to be correct if the snapshot is consistent with previous snapshots and the author's signature on it verifies using `Verify`.

THEOREM 1. *Assuming a collision resistant hash function and correct snapshots, membership proofs in a Balloon are universal unforgeable. I.e., there exists no polynomially computationally bounded adversary $\mathcal{A}$ that can forge a membership proof for a given key and snapshot with a non-negligible success probability.*

PROOF. We will start by showing that there exists no polynomially computationally bounded adversary that can create a non-membership proof for an event with the given key that was added to the Balloon before the given snapshot. Recall that a non-membership proof consists of the latest snapshot s_l (containing the root of the latest hash treap), an authenticated path for the root of the latest hash treap to the hash of the given key $\texttt{Hash}(k)$, and if the key is present in the latest hash treap, the corresponding value j.

Under the assumption that the latest snapshot, and thus the root of the latest hash treap, is correct, the hash treap is a fixed Merkle tree. An adversary can only succeed by constructing a different authenticated path for the correct root of the latest hash treap to the given key to show that either a) $\texttt{Hash}(k)$ is not present in the treap; or b) the value j corresponding to $\texttt{Hash}(k)$ is greater than i, meaning that the event with the given key k has been added after snapshot s_i. Coronado [7] showed that the security of authenticated paths in Merkle trees can be reduced to the collision resistance of underlying the hash function.

Now, we only need to show that there also exists no polynomially computationally bounded adversary that can create a membership proof for an event with the given key that was not added to the Balloon before the given snapshot. A membership proof consists of the latest snapshot s_l, an authenticated path in the hash treap, the corresponding value in the hash treap $j \leq i$, an authenticated path in the history tree from the root as present in the given snapshot s_i to the index (position) j and the event e_j.

Assuming that the given snapshot is correct, the adversary needs to construct two different authenticated paths (one in the hash treap and one in the history tree) with a number of constraints to show that: a) there is a node in the hash treap with key $\texttt{Hash}(k)$ and value $j \leq i$, b) there is a node in the history tree with key j for which the adversary can also return an event e_j. Similar to the first case, this can be reduced to the collision resistance of the hash function. □

If snapshots can no longer be assumed to be correct due to author compromise at some point in time, one needs to differentiate between proofs of membership and proofs of non-membership. The former are done against a given snapshot s_i, the latter against the latest snapshot s_l. For the proofs of membership one has forward security: provided with a snapshot s_i, received through gossip before the time of author compromise, the proof of membership can be trusted. However, after author compromise, one cannot trust any more the proofs of non-membership nor the proofs of membership against a snapshot that was generated after author compromise, unless one is convinced of the consistency of the snapshots.

5.2 Consistent Snapshots

In a Balloon, it is the author (verifiable insert) and the monitors (ensuring consistency) that determine whether or not a snapshot is consistent with all previous snapshots. This has an influence on the attacker model: as long as the author is honest and verifies the insert before signing the new snapshot, correctness of the snapshot can be reduced to the existential unforgeability of the author's signature. From the moment the author turns malicious, both the monitors and the gossiping mechanism (to ensure that monitors have access to every snapshot) play an essential role to ensure the continued correctness of the snapshots.

LEMMA 1. *Assuming a collision resistant hash function, for a given snapshot that is consistent with the past, inserting all events into a Balloon, snapshot by snapshot, according to the sorting function Ω up till a given snapshot will result in an identical Balloon, fixed by the same roots as present in the given snapshot.*

A history tree is determined by the events and the order these were inserted into the tree. The order is deterministically set by Ω within each snapshot. For the hash treap the order is irrelevant because the data structure is set unique and history independent.

THEOREM 2. *Assuming a collision resistant hash function, verifiable insert ensures that the roots of snapshots are consistent with the past.*

PROOF. By induction. First, we show that the first snapshot is consistent with the events added with verifiable insert, by applying Lemma 1.

Then we show that using the current snapshot, verifiable insert ensures the consistency of the next snapshot. In the history tree, the membership proof for the latest added event fixes the entire tree up to the time of the current snapshot. By doing membership queries in the hash treap for the keys of the new events to insert, one gets authenticated paths in the hash treap that form a relevant pruned version of the hash treap. With this pruned hash treap, one can insert the events into the hash treap and compute its new root. This follows from the fact that the position to insert each a new node is determined by a binary search (which path the membership query proves), and that balancing the treap using the heap priority only rotates nodes along the authenticated path. We showed that both pruned sub-trees fix the Balloon with respect to the currents snapshot. Due to Lemma 1, we get that adding the events of the next snapshot to the Balloon according to the sorting function Ω will result in a consistent Balloon that is fixed by its roots. □

Monitors do not work directly on the event key and value but instead are supplied with the hash of the event key and the hash of the entire event. Because of the way inserting events to Balloon is done, these hashes can also serve as input, be it for a slightly modified Algorithm 4'.

5.3 Requirements

We fulfil our requirements as follows:

R1. Minimal trust in the server This follows directly from the fact that neither the membership queries nor the verifiable insert rely on any trust assumptions at the server. It is also the author that signs the snapshots.

R2. (Non-)membership proofs Since the server must always provide a reply (either a membership or a non-membership proof) that is provably consistent with a statement made by the author, it must follow the protocol.

R3. Publicly verifiable integrity & deletion detection The data structure is publicly verifiable, in the sense that any modifications or deletions between snapshots are detected. This follows directly from Lemma 1.

R4. Minor overhead at the author With the verifiable insert, the author only needs to store the latest snapshot. As an added benefit, this also holds for monitors, as soon as they have catched up with the author.

R5. Event privacy A Balloon achieves computational f-Zero Knowledge as defined by Naor and Ziv [20], where $f(R)$ is some information about the set which can be tolerated to leak to resolvers (in our case, clients). Given a hash function that is pre-image resistant, no keys that can be used to query the server for an event leak from the server's replies to membership queries or from the verifiable inserts run by clients. This is because Balloon uses the hash of the event key as the key for nodes in hash treaps and the hash of the event in the history tree. Note however that we do leak the cardinality of the events, both in total and per snapshot. The total number of events can be determined by querying the server for random keys to eventually retrieve a full copy of the Merkle tree formed by the hash treap. By observing how the tree changes over time the number of events inserted per snapshot, as they take place, will be apparent.

Clients acting as monitors also learn the relationship between nodes in the hash treap and the history tree, i.e., the values of all nodes in the hash treap and the indices they specify in the history tree. For clients that do not act as monitors, this information leakage can be prevented by modifying the hash treap, as described in Section 4.4.

6. CLIENT-SPECIFIC FORWARD AUTHOR CONSISTENCY

Balloon can be used to provide client-specific forward author consistency similar to [22], i.e., integrity protection with deletion detection for the chain of events generated for a specific client, up to the time of compromise of the author. As such one can be assured of the data structure's consistency, be it only for the subset of events one is interested in, without having to rely on monitors. To be able to provide client-specific consistency, the author has to:

- generate the keys for the events in a client-specific, deterministic, forward-secure manner;
- provide the client upon request with a signed statement on the value of the key for the next client-specific event, the latest snapshot, and the identity of the client; and
- sign the initial secret together with the identity of the client on setup.

This will ensure that the author remains honest, otherwise the author will have signed two conflicting statements (the author also signs each snapshot on its own) that can be used to prove to a third party that the author is inconsistent.

6.1 Generating Client-Specific Keys

The client-specific keys for events will be generated based on the client's identifier (ID_i) and the $\texttt{secret}_i$. The server only keeps the latest value of the intermediate key $k'_{i,j}$, that can be used to generate the next client-specific key $k_{i,j}$. The j-th key for client i with identity ID_i is generated as follows:

$$k_{i,j} = \texttt{MAC}_{k'_{i,j}}(ID_i) \text{ with } \begin{cases} k'_{i,j} = \texttt{Hash}(k'_{i,j-1}) \\ k'_{i,0} = \texttt{secret}_i \end{cases} .$$

Bellare and Yee [5] showed that this type of construction provides forward security.

6.2 Verifying Consistency

The client can verify the data structure's consistency with respect to events created for it by the author, up till the point in time of author compromise, as follows. The client will generate its keys, starting from the shared secret, in the same way as the author. For each key, the client will query the server and receive an event together with a membership proof up to a certain key for which it will receive a non-membership proof. At this point, the client will request a signed statement from the author on the value of its next key. This key needs to correspond with the key on which the non-membership proof was received[7].

6.3 Proving Author Inconsistency

When the client-specific consistency verification fails, the client might want to make a complaint about the author to a third party. One has to differentiate between two cases:

1. There is an event in the Balloon for the key and snapshot from the signed statement; or
2. There is no such event with the indicated key.

Case 1

A proof of author inconsistency towards a third party is straightforward:

- The signed statement on the key and snapshot s_l; and
- A membership proof for the given key in snapshot s_i, where $i \leq l$.

The membership proof can only be with respect to a snapshot before or the same snapshot as the signed statement. If the membership proof is with respect to a snapshot after the statement snapshot, then the event could have been inserted after the statement, and therefore the author is not necessarely inconsistent.

Case 2

We have to distinguish between two sub-cases:

a. There exists at least one event for the client; or

b. There is no event for the client in the Balloon.

[7]One has to make sure that both the proof of non-membership and the signed statement are with respect to the same latest snapshot.

For case 2a, the inconsistency proof can leverage the existance of an event for the client and the structure in which client-specific keys are generated. The proof of inconsistency is then:

- The signed statement on an incorrect value and snapshot s_l;
- A membership proof for the last key that exists for the client in snapshot s_i, where $i \leq l$;
- A non-membership proof for the key, that follows on the last existing key for the client, in snapshot s_j, where $j \geq l$; and
- The last intermideate key and the client's identity needed to calculate the last key and the one following on the last key.

For the membership proof, the proof has to be in a snapshot s_i where $i \leq l$, because otherwise the event was inserted after the statement. For the non-membership proof, the proof has to be in a snapshot s_j where $j \geq l$, because otherwise the event might be somewhere in a snapshot s_k where $i < k \leq l$. The event key and identifier needed to calculate the event keys for the last event and its successor (the non-membership proof) are required to link the membership and non-membership proofs together in succession and to the same identifier as for the client in the signed statement.

For case 2b, the inconsistency proof leverages the initial signature by the author on the secret and identity of the client used to generate event keys. The proof of inconsistency is then:

- The signed statement on an incorrect value and snapshot s_l;
- A non-membership proof for the first event of the client in snapshot s_i, where $i \geq l$; and
- The signed secret and identity from the author.

For the non-membership proof, the proof has to be in a snapshot s_i where $i \geq l$, because otherwise the event might be somewhere in a snapshot s_k where $i < k \leq l$. The signed secret from the author links the statement and non-membership proof together to the same client.

Note that by proving to a third party that the author has been inconsistent, for some cases, the client leaks all future event keys to the third party. However, given that the author is inconsistent, the loss of privacy for event existence is of secondary importance, since presumably the author is already malicious.

Theorem 3. *Assuming a pre-image and collision resistant hash function, an existentially unforgeable MAC algorithm and an existentially unforgeable digital signature scheme being used, Balloon with its extension provides client-specific forward author consistency. Moreover, in case of an inconsistent author, the client can show this to a third party.*

7. RELATED WORK

Certificate Transparency [15] uses a nearly identical[8] data structure and operations as in the tamper-evident history system by Crosby & Wallach [9] (summarised in Section 3.1). The setting of Certificate Transparency and Crosby & Wallach's history system is fundamentally different from ours: a number of *minimally trusted* authors insert data into a history tree kept by a server, and clients query the server for data and acts as auditors to challenge the server to prove consistency between commitments. Clients play a similar role in our setting, however, in our setting it is the author's signatures as part of snapshots that are being challenged, not the server's. Our verifiable insert algorithms, described in Section 4.2, enable the author to verify that the server updated the data structure correctly, without the author storing a copy of the entire data structure. Author verification is not needed in the setting of Certificate Transparency or Crosby & Wallach's history system, since it is the server signing, not authors.

Balloon is an authenticated data structure [16, 19, 26], closely related to authenticated dictionaries [21] and persistent authenticated dictionaries (PADs) [1]. In a PAD, clients can query a server based on key and snapshot, as in Balloon. However, in a PAD it is possible for the author to delete keys. By allowing clients to query past versions of the PAD, the server needs to be able to construct past versions of the PAD to calculate proofs. This is relatively costly, and even the most efficient solution[9] for tree-based PADs according to Crosby and Wallach [11] incurs $\mathcal{O}(\log n)$ expected storage per insert. Table 1 compares this PAD with Balloon. Balloon only requires constant total storage size at the author, thanks to verifiable inserts, at the cost of a logarithmic insert size due to the transport of verifiable insert proofs[10]. Performing a membership query for past snapshots is more efficient in Balloon, since we do not have to calculate past versions of our data structure, while the PAD has to query the version cache of each node along the path (each cache of size v). We can make this improvement in Balloon due to not allowing deletion of keys (i.e. Balloon is append only). The insert storage size at the server is also more efficient in Balloon thanks to not having to maintain a version cache.

For PADs, Crosby and Wallach [11] note that red-black trees are more efficient than treaps due to their worst-case instead of expected logarithmic bounds on several important operations. We opt for using a treap due to its relative simplicity and its history independence property, which is important for our event privacy requirement (see Section 2.2).

Miller et al. [19] present a generic method for authenticating operations on any data structure that can be defined by standard type constructors. In essence, their generic solution works by having the prover provide the authenticated paths (if viewed as a directed acyclic graph) in the data structure that are traversed by the prover when performing an operation. The verifier can then perform the same operation on only the authenticated paths provided in the

[8]The only difference is how non-full trees are handled, as noted in Section 2.1 of [15].

[9]Using Sarnak-Tarjan versioned nodes with a cache-everywhere strategy for calculated hash values [25, 10].

[10]Verifying this proof is what causes the insert time at the author. For the PAD, the insert time is due to updating the data structure.

Table 1: A comparison between Balloon and a PAD based upon a red-black tree using Sarnak-Tarjan versioned nodes with a cache-everywhere strategy for calculated hash values [10, 11, 25]. Values are expected complexity. Note that red-black trees also have the same worst-case complexity, while Balloon due to being a probabilistic data structure does not. The number of events in the data structure is n and v is the size of the version cache.

	Total Storage Size (A)	Query Time (current)	Query Time (past)	Insert Storage Size (S)	Insert Time (A)	Insert Time (S)	Insert Proof Size
Balloon	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Tree-based PAD	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log v \cdot \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$

proof. The verifier only has to store the latest correct digest that fixes all the contents in the data structure, like our snapshots. Our verifiable insert algorithms are based on the same principle, i.e., we provide authenticated paths in the hash treap and history tree that correspond to the part of the data structures that the verifier needs to insert the data itself and calculate the updated roots.

In Revocation Transparency, Laurie and Kasper [14] present two options for data structures for certificate revocation: a sorted list and a sparse Merkle tree. The sorted list is, as noted by Laurie and Kasper, inferior to the sparse Merkle tree approach, so we only discuss sparse Merkle trees. Sparse Merkle trees create a Merkle tree with 2^N leafs, where N is the bit output length of a hash algorithm $\texttt{Hash}(\cdot)$. A leaf is set to 1 if the certificate with the hash value fixed by the path to the leaf from the root of the tree is revoked, and 0 if not. While the tree in general is too big to store or compute on its own, the observation that most leafs are zero (i.e., the tree is sparse), means that most authenticated paths in the tree are redundant, and only paths including non-zero leafs need to be computed (and potentially stored) on their own. We are unaware of any thorough work on sparse Merkle trees beyond the idea presented by Laurie and Kasper [14]. At first glance, it appears like sparse Merkle trees could replace the hash treap in a Balloon with similar size/time complexity operations. We note this as future work.

Enhanced Certificate Transparency (ECT) by Ryan [24] extends CT by using two data structures: one chronologically sorted and one lexicographically sorted. The chronologically sorted data structure corresponds to a history tree (like CT). The lexicographically sorted data structure is similar to our hash treap. In ECT, each new operation on a subject (like a certificate) is inserted into the chronological data structure together with the hash that fixes the entire lexicographical data structure *after* it has been updated. Each entry in the lexicographical data structure contains the operations for a subject, such as a domain name or other form of identity. Updating the lexicographical data structure involves adding the operation to *the list of operations* performed on the subject of the operation in the lexicographical data structure. An operation to e.g. revoke a certificate for a subject does not remove any data from the data structure, but simply records the operation by appending it to the list of performed operations. Checking consistency between the two data structures then comes down to two options: random checking or auditors that check everything. Auditors correspond to monitors in CT or validators in e.g. AKI and ARPKI [13, 3]. Random checking involves probabilistic proofs that e.g. browsers can make. The random checking verifies that a random operation recorded in the chronological data structure has been correctly performed in the lexicographical data structure. To check non-current versions of the lexicographical data structure, this requires the prover to generate past versions of the lexicographical data structure, as for PADs, which is relatively costly [11]. Distributed transparent key infrastructure (DTKI) [28] builds upon the same data structures as ECT. ETC and DTKI only supports probabilistic proofs of the current lexicographic data structure. This is safe, assuming that the probability that each version of the lexicographic data structure (presented to any client) is fully verified is high enough to deter an attacker.

CONIKS [17] is a key management system, in a setting similar to CT, where minimally trusted clients manage their public keys in directories at untrusted key servers. To ensure consistency of the directory, clients download and verify their keys in every commitment (version) of the directory, somewhat similar to how our clients in Balloon can ensure consistency of their own events using client-specific forward author consistency. This requires the key server to be able to reconstruct past versions of the directory. Keys for entries in the directory are predictable (.e.g. an email address), leading to privacy problems similar to our event privacy requirement. CONIKS uses verifiable unpredictable functions, based on verifiable random functions, to prevent key enumeration. This could be used for Balloon in case event keys are predictable. Finally, CONIKS link commitments together into a commitment chain, together with a specified (instead of assumed, as in our and related work) gossiping mechanism that greatly increases the probability that an attacker creating inconsistent snapshots is caught.

8. CONCLUSIONS

This paper presented Balloon, an authenticated data structure composed of a history tree and a hash treap. With Balloon, a forward-secure author can safely outsource storage of events intended for different clients to an untrusted server. Our security evaluation shows that different parts of Balloon are secure under the modest assumptions of a collision resistant hash function and an unforgeable signature algorithm. Balloon is a more efficient solution in our setting than using a PAD, as summarised by Table 1.

The idea behind verifiable inserts for Balloon may be of interest, e.g., for monitors in several schemes focused on certificate issuance transparency. Minimising storage at monitors, at the cost of increased bandwidth, may be beneficial in some settings such as for exceedingly large data structures.

Using Balloon for client-specific forward author consistency, with publicly verifiable proofs of author inconsistency, is a promising approach for tacking the problems of data structure and snapshot consistency. Monitors can be seen as prohibitively expensive and probabilistic proofs as too unlikely to catch an attacker performing a targeted attack. In the appendix we present a negative result on the use of probabilistic proofs for consistency in a Balloon.

Moving forward, our intent is to use Balloon as a building block for an improved version of our prior work on distributed privacy-preserving transparency logging [22], providing client-specific forward author consistency. Another interesting avenue of future work is investigating sparse Merkle trees as a replacement for the hash treap in Balloon.

Acknowledgements

We would like to thank Simone Fischer-Hübner, Stefan Lindskog, Leonardo Martucci, Jenni Reuben, Philipp Winter, and Jiangshan Yu for their valuable feedback. Tobias Pulls has received funding from the Seventh Framework Programme for Research of the European Community under grant agreement no. 317550.

References

[1] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In G. I. Davida and Y. Frankel, editors, *ISC*, volume 2200 of *Lecture Notes in Computer Science*, pages 379–393. Springer, 2001.

[2] C. R. Aragon and R. Seidel. Randomized Search Trees. In *FOCS*, pages 540–545. IEEE Computer Society, 1989.

[3] D. A. Basin, C. J. F. Cremers, T. H. Kim, A. Perrig, R. Sasse, and P. Szalachowski. ARPKI: attack resilient public-key infrastructure. In G. Ahn, M. Yung, and N. Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 382–393. ACM, 2014.

[4] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In N. Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[5] M. Bellare and B. Yee. Forward-security in private-key cryptography. In *Topics in Cryptology—CT-RSA 2003*, pages 1–18. Springer, 2003.

[6] S. Blake-Wilson, D. Johnson, and A. Menezes. Key agreement protocols and their security analysis. In M. Darnell, editor, *Cryptography and Coding, 6th IMA International Conference, Cirencester, UK, December 17-19, 1997, Proceedings*, volume 1355 of *Lecture Notes in Computer Science*, pages 30–45. Springer, 1997.

[7] C. Coronado. On the security and the efficiency of the Merkle signature scheme. *IACR Cryptology ePrint Archive*, 2005:192, 2005.

[8] S. A. Crosby. *Efficient tamper-evident data structures for untrusted servers.* PhD thesis, Rice University, Houston, TX, USA, 2010.

[9] S. A. Crosby and D. S. Wallach. Efficient Data Structures For Tamper-Evident Logging. In *USENIX Security Symposium*, pages 317–334. USENIX Association, 2009.

[10] S. A. Crosby and D. S. Wallach. Super-Efficient Aggregating History-Independent Persistent Authenticated Dictionaries. In M. Backes and P. Ning, editors, *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 671–688. Springer, 2009.

[11] S. A. Crosby and D. S. Wallach. Authenticated dictionaries: Real-world costs and trade-offs. *ACM Trans. Inf. Syst. Secur.*, 14(2):17, 2011.

[12] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

[13] T. H. Kim, L. Huang, A. Perrig, C. Jackson, and V. D. Gligor. Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure. In D. Schwabe, V. A. F. Almeida, H. Glaser, R. A. Baeza-Yates, and S. B. Moon, editors, *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013*, pages 679–690. International World Wide Web Conferences Steering Committee / ACM, 2013.

[14] B. Laurie and E. Kasper. Revocation transparency. September 2012. http://www.links.org/files/RevocationTransparency.pdf.

[15] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962, June 2013.

[16] C. U. Martel, G. Nuckolls, P. T. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A General Model for Authenticated Data Structures. *Algorithmica*, 39(1):21–41, 2004.

[17] M. S. Melara, A. Blankstein, J. Bonneau, M. J. Freedman, and E. W. Felten. CONIKS: A privacy-preserving consistent key service for secure end-to-end communication. Cryptology ePrint Archive, Report 2014/1004, 2014. http://eprint.iacr.org/.

[18] R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In C. Pomerance, editor, *CRYPTO*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987.

[19] A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated data structures, generically. In S. Jagannathan and P. Sewell, editors, *POPL*, pages 411–424. ACM, 2014.

[20] M. Naor and A. Ziv. Primary-secondary-resolver membership proof systems. Cryptology ePrint Archive, Report 2014/905, 2014. http://eprint.iacr.org/.

[21] K. Nissim and M. Naor. Certificate revocation and certificate update. In A. D. Rubin, editor, *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998.* USENIX Association, 1998.

[22] T. Pulls, R. Peeters, and K. Wouters. Distributed privacy-preserving transparency logging. In A.-R. Sadeghi and S. Foresti, editors, *WPES*, pages 83–94. ACM, 2013.

[23] P. Rogaway and T. Shrimpton. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In B. K. Roy and W. Meier, editors, *FSE*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2004.

[24] M. D. Ryan. Enhanced certificate transparency (how johnny could encrypt). *IACR Cryptology ePrint Archive*, 2013:595, 2013.

[25] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986.

[26] R. Tamassia. Authenticated data structures. In *Algorithms-ESA 2003*, pages 2–5. Springer, 2003.

[27] J. Vliegen, K. Wouters, C. Grahn, and T. Pulls. Hardware strengthening a distributed logging scheme. In *15th Euromicro Conference on Digital System Design, DSD 2012, Cesme, Izmir, Turkey, September 5-8, 2012*, pages 171–176. IEEE, 2012.

[28] J. Yu, V. Cheval, and M. Ryan. DTKI: a new formalized PKI with no trusted parties. *CoRR*, abs/1408.1023, 2014.

Negative Result on Probabilistic Consistency

Here we present a negative result from our attempt at ensuring consistency of a Balloon with probabilistic proofs. Probabilistic proofs are compelling, because they may enable more resource-constrained clients en-mass to verify consistency, removing the need for monitors that perform the relatively expensive role of downloading all events at a server. Assume the following pair of algorithms:

- `B.IncGen`$(\mathsf{s}_i, \mathsf{s}_j, r) \rightarrow P$. Generates a probabilistic incremental proof P using randomness r between s_i and s_j, where $i \leq j$, from the Balloon B. Outputs P.
- `P.IncVerify`$(s'_i, \mathsf{s}_j, r) \rightarrow \{\texttt{true}, \texttt{false}\}$. Verifies that P probabilistically proves that s_j fixes every event fixed by s'_l, where $i \leq j$, using randomness r.

Our attempt

Our envisioned `B.IncGen` algorithm shows consistency in two steps:

- First, it uses the `H.IncGen` algorithm from the history tree. This ensures that the snapshots are consistent on the history tree.
- Secondly, it selects deterministically and uniformly based on r a number of events E from the history tree. Which events to select from depends on the two snapshots. For each selected event, the algorithm performs a `B.MembershipQuery` for the event key to show that the event is part of the hash treap and points to the index of the event in the history tree.

The `P.IncVerify` algorithm checks the incremental proof in the history tree, each membership query, and that the events E were selected correctly based on r. Next, we explain an attack, why it works, and possible lessons learned.

Attack

Here we explain an attack on our attempt that allows an attacker to hide an arbitrary event that was inserted *before author compromise*. The attacker takes control over both the author and server just after snapshot s_t. Assume that the attacker wants to remove an event e_i from Balloon, where $i \leq t$.

The attacker does the following:

1. Remove the event identifier of e_i from the hash treap and insert a random key. Do any rebalancing of the treap if needed.
2. Insert a new event with a random key and value using `B.Insert`.
3. Create the new snapshot, s_{t+1} .

We know that the snapshot s_{t+1} is inconsistent with all other prior snapshots, s_p, where $p \leq t$.

Now, we show how the attacker can avoid being detected by `P.IncVerify` in the case that the verifier challenges the server (and therefore the attacker) to probabilistically prove the consistency between s_p and s_{t+1}, AND that the randomness provided by the verifies selects the event e_i that was modified by the attacker[11]. The attacker can provide a valid incremental proof in the history tree, using `H.IncGen`, since the history tree has not been modified. However, the attacker cannot create a valid membership query for the modified event, since its key was removed from the hash treap in s_{t+1}. To avoid detection, the attacker creates a *new snapshot* s_l that contains the event in question in the hash treap and performs the `B.MembershipQueries` using this new snapshot for the hash treap.

Lessons learnt

This attack succeeds because the attacker can, once having compromised the author and server, a) create snapshots at will; and b) membership queries are always performed on the current version of the hash treap.

In settings where snapshots are generated periodically, e.g., once a day, the probability of the attacker getting caught in this way is non-neglible given a sufficient number of queries. However, as long as the attacker can create snapshots at will, the probability that it will be detected with probabilistic incremental proofs is zero, as long as:

- it cannot be challenged to generate past versions of the hash treap; and
- there are no monitors or another mechanism like client-specific forward author consistency, presented in Section 6, that prevent the attacker from modifying or deleting events.

[11] Note that this case only happens with a very small probability.