Efficient Sparse Merkle Trees
Caching Strategies and Secure (Non-)Membership Proofs

Rasmus Dahlberg[1], Tobias Pulls[1], and Roel Peeters[2]

[1] Karlstad University, Dept. of Mathematics and Computer Science, Sweden
[2] KU Leuven, ESAT/COSIC & iMinds, Belgium

Abstract. A sparse Merkle tree is an authenticated data structure based on a perfect Merkle tree of intractable size. It contains a distinct leaf for every possible output from a cryptographic hash function, and can be simulated efficiently because the tree is sparse (i.e., most leaves are empty). We are the first to provide complete, succinct, and recursive definitions of a sparse Merkle tree and related operations. We show that our definitions enable efficient space-time trade-offs for different caching strategies, and that verifiable audit paths can be generated to prove (non-)membership in practically constant time ($<$ 4 ms) when using SHA-512/256. This is despite a limited amount of space for the cache—smaller than the size of the underlying data structure—and full security in the multi-instance setting.

1 Introduction

Secure HTTPS connections rely on the users' browsers to obtain authentic domain-to-key bindings during set-up. For that purpose, trusted third parties called certificate authorities are used to vouch for the integrity of public keys by issuing X.509 certificates. While this may appear to solve the initial problem of establishing trust, several new complications arise. Considering that there are hundreds of certificate authorities, all of which are capable of issuing certificates for any domain, it is challenging to concisely observe what has been issued for whom [11]. As such, a misissued or maliciously issued certificate could remain unnoticed forever, or more likely until an attack against a domain has taken place. Naturally that raises an important question: *who watches the watchmen*?

Google's Certificate Transparency (CT) project proposes public logs based on append-only Merkle trees [17]. The basic idea is that an SSL/TLS certificate must be included in some log to be trusted by a browser, and because the infrastructure is public anyone can audit or monitor these logs to ensure correct behaviour [6,15]. Thus, CT allows clients to determine whether a certificate was valid at some point in time, but inclusion in the log cannot guarantee that it is current. For instance, what if a certificate has to be revoked due to a compromised private key or an entire certificate authority [14,26]? Since the log is append-only, it must be chronological. Therefore certificates can neither be removed nor can the absence of a revocation certificate be proven efficiently [12].

Certificate Revocation (RT) [22] is a proposed extension to CT by Laurie and Kasper [16]. The aim is to provide a separate mechanism that proves certificates unrevoked, and requires an authenticated data structure supporting efficient non-membership proofs [31]. As is, there are at least two approaches towards such proofs. One is based on sorted Merkle trees, and the other on tuple-based signed statements on the form "Key k_i has the value v_i; there are no keys in the interval (k_i, k_{j+1})" [10,16]. We consider the former in terms of a sparse Merkle tree (SMT), whose scope goes far beyond RT. For example, SMTs can be used as key building blocks in a wide area of applications, ranging from dynamic authenticated dictionaries to secure messaging applications [10,19,27,29].

After introducing some necessary preliminaries (Section 2) and the approach taken here (Section 3), our contributions are as follows. First, building on an interesting proposal started by Laurie and Kasper [16], we define efficient caching strategies and complete recursive definitions for an SMT (Section 4). Second, we evaluate the security of our definitions in the multi-instance setting, comparing our design decisions with those made in CONIKS [19] (Section 5). Third, we examine three caching strategies experimentally for an SMT, showing different space-time trade-offs (Section 6). Finally, we discuss related work (Section 7) and end with conclusions (Section 8).

2 Preliminaries

We start by describing background regarding Merkle trees and audit paths, then we present cryptographic assumptions that our security evaluation relies on.

2.1 Merkle Trees

A Merkle tree [20] is a binary tree that incorporates the use of cryptographic hash functions. One or many attributes are inserted into the leaves, and every node derives a digest which is recursively dependent on all attributes in its subtree. That is, leaves compute the hash of their own attributes, and parents derive the hash of their children's digests concatenated left-to-right. As further described in Section 5, certain digests must also be encoded with additional constants. This is to prevent indistinguishability between different types of nodes [8,19].

A Merkle tree without a proper encoding is shown in Figure 1. It contains eight attributes ρ–ω, and the *root digest* $r \leftarrow d_0^3$ serves as a reference to prove membership by presenting an *audit path* [17]. For instance, dashed nodes are necessary to authenticate the third left-most leaf containing attribute τ. More generally, an audit path comprises all siblings along the path down to the leaf being authenticated. Combined with a retrieved attribute that represents a membership proof, which is valid if it reconstructs the root digest r' such that $r' = r$. Therefore a proof is only as convincing as r, but trust can be established using, e.g., digital signatures or by periodically publishing roots in a newspaper.

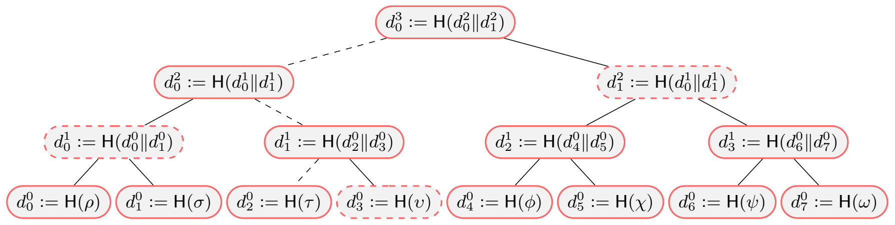

$d_0^3 := \mathsf{H}(d_0^2 \| d_1^2)$
$d_0^2 := \mathsf{H}(d_0^1 \| d_1^1)$
$d_1^2 := \mathsf{H}(d_0^1 \| d_1^1)$
$d_0^1 := \mathsf{H}(d_0^0 \| d_1^0)$
$d_1^1 := \mathsf{H}(d_2^0 \| d_3^0)$
$d_2^1 := \mathsf{H}(d_4^0 \| d_5^0)$
$d_3^1 := \mathsf{H}(d_6^0 \| d_7^0)$
$d_0^0 := \mathsf{H}(\rho)$
$d_1^0 := \mathsf{H}(\sigma)$
$d_2^0 := \mathsf{H}(\tau)$
$d_3^0 := \mathsf{H}(\upsilon)$
$d_4^0 := \mathsf{H}(\phi)$
$d_5^0 := \mathsf{H}(\chi)$
$d_6^0 := \mathsf{H}(\psi)$
$d_7^0 := \mathsf{H}(\omega)$

Fig. 1: A Merkle tree containing attributes ρ–ω. The digest rooted at height h and index i is abbreviated d_i^h.

2.2 Setting and Cryptographic Assumptions

Inspired by Katz [13] and Melara *et al.* [19], we consider a computationally bound adversary in the multi-instance setting. This means that there are several distinct SMTs, and the adversary should not gain any advantage in terms of necessary computation if she attempts to attack all SMTs at once. In other words, despite the adversary's multi-instance advantage, the goal is to provide full λ-bit security for each SMT. For security we rely on a collision and pre-image resistant hash function H with digests of $N := 2\lambda$-bits, as well as Lemma 1.

Lemma 1. *The security of an audit path reduces to the collision resistance of the underlying hash function* H.

Proof. This follows directly from the work of Merkle [20] and Blum *et al.* [5].

3 Sparse Merkle Trees

First we introduce non-membership proofs that are based on sorted Merkle trees, then the notion of an SMT and our approach is incrementally described.

3.1 Non-Membership Proofs and High-Level Properties

In RT-like applications it is crucial to prove a value absent [16,29,32]. This forms the notion of a non-membership proof, and requires lexicographically sorted tree structures that prevent all nodes from being enumerated. Previously this has been accomplished by viewing balanced binary search trees, e.g., treaps and red-black trees, as Merkle trees [10]. This involves rules that rotate nodes upon insertion and removal, and means to fix the tree structure by trustworthy roots. Non-membership is proven by generating an audit path through binary search, and a verifying party accepts it to be valid if there is no evidence that the tree structure is unsorted or that the root is improperly reconstructed. In other words, non-membership is proven efficiently due to a balanced tree structure, and proofs are convincing because the structure is fixed by a trustworthy root.

While an SMT also relies on the structure of the tree together with being a Merkle tree, it is different in that it requires neither balancing techniques nor certain constants when encoding digests. This is due to an intractably large Merkle tree that reserves a unique leaf ℓ for every conceivable key digest. The hash of a key k determines ℓ, and k is a (non-)member if the attribute $a \in \ell$ is set to a_0/a_1. Therefore the resulting tree structure contains 2^N leaves at all times, and (non-)membership can be proven by presenting an audit path for leaf $\mathsf{H}(k)$. This set-up also implies *history independence*: a unique set of keys produce a deterministic root digest, regardless of the order in which keys are inserted or removed. Notably history independence is not necessarily provided by balanced Merkle trees (e.g., the case for deterministic treaps, but not for red-black trees).

3.2 Tractable Representations

Considering the intractably large size of an SMT, it is challenging to represent efficiently in memory. To begin with, the only reason why this is feasible traces back to the key observation that SMTs are *sparse*. That is, the majority of all leaves are associated with attribute a_0 and represent non-members. Such leaves are henceforth referred to as empty, and yield a construction where the empty subtrees rooted at height h derive identical *default digests*. The basic principle is that an empty leaf computes $d_*^0 \leftarrow \mathsf{H}(a_0)$, a node rooted at an empty subtree with height one derives $d_*^1 \leftarrow \mathsf{H}(d_*^0 \| d_*^0)$, and so on. Since these default digest can be precomputed, they need neither be associated with explicit nodes nor be derived recursively by visiting all leaves. Instead, referring to Figure 2, it suffices to process the filled nodes whose digests depend on existing keys.

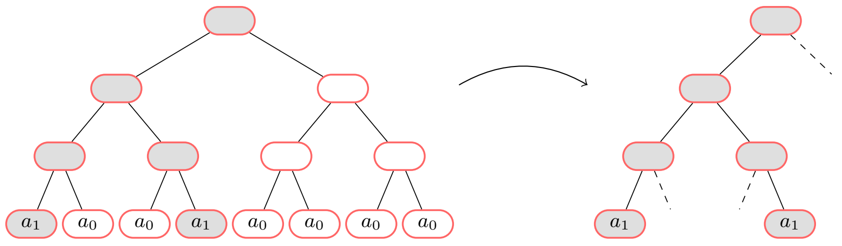


Fig. 2: An illustration of how default nodes can be discarded to attain a tractable representation of an SMT.

3.3 Earlier Proposals

Different approaches can be used to provide efficient representations of an SMT. Bauer [3] has proposed an explicit pruned tree structure where all the non-empty attributes are elevated upwards through their ancestors. The elevation stops when the root of a subtree containing a single non-empty leaf is reached, and all descendants to such roots are discarded. The original SMT can be reconstructed by recording indices for the non-empty leaves in each subtree, but will require excessive amounts of memory unless they are evenly spread out. Hence, while the proposal is neat, we find the approach started by Laurie and Kasper [16] more generally applicable. It is based on maintaining a collection of keys $\mathcal{K}$, and the collection is authenticated by simulating an SMT. As is, however, their proposal is incomplete and cannot, e.g., derive (non-)membership proofs efficiently. This is due to deriving subtrees' digests over and over again—an issue we solve in the following sections by introducing *relative information*.

3.4 Our Approach

We define the SMT as a composition of a non-authenticated data structure $\mathcal{D}$ and some relative information δ. The data structure contains unique keys k (or more accurately the keys' digests), and must support insertion, removal, look-up, and splitting. Splitting refers to dividing $\mathcal{D}$ in two based on a key, and relative information a collection of digests that are recorded by a caching strategy. The resulting SMT is simulated in the sense that there is no explicit tree structure, which is possible because every $k \in \mathcal{D}$ can be mapped to its associated subtrees recursively. For instance, as shown in Figure 3, a root digest can be obtained by starting a traversal from the root down to all the non-empty leaves. The *base* is initially set to all zeros and refers to the left-most leaf in a subtree. It remains the same on left-traversals, must be updated by setting the appropriate bit to one on right traversals, and is used to determine the *split index*. The split index is the key upon which $\mathcal{D}$ is divided on and refers to the left-most leaf in the right subtree. Thus, as formalized in Section 4.3, it is an upper exclusive and lower inclusive bound for the keys in the left and right subtrees, respectively.

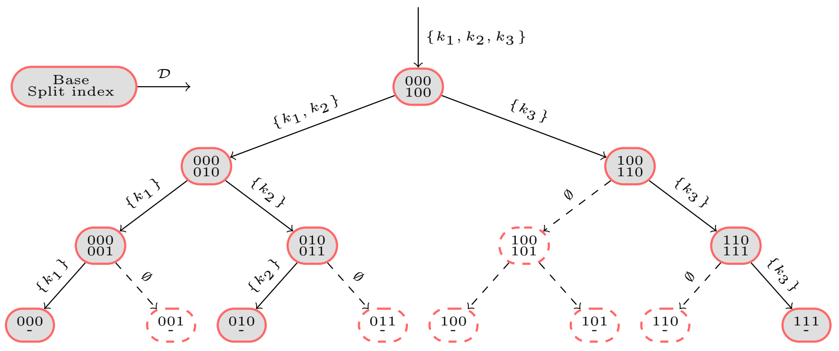


Fig. 3: An illustration of a recursive traversal to obtain the root digest; $k_1 = 000$, $k_2 = 010$, and $k_3 = 111$.

Clearly it is inefficient to obtain a subtree's digest by repeatedly visiting all the non-empty leaves. Therefore relative information is necessary: a collection of cached digests with the sole purpose of preventing such inefficiency. For instance, a naïve caching strategy could record every digest that is non-default. Although that requires excessive amounts of memory, it would ensure that all siblings' digests are available upon generating audit paths. Consequently, the number of splits will be constant, and (non-)membership can be proven with the same time complexity as the underlying split operation. Our aim when defining caching strategies is to preserve this property while reducing memory requirements.

4 Efficient Representations

First we define caching strategies that are based on capturing *branches*, then our proposal is formalized by presenting complete recurrences for an efficient SMT.

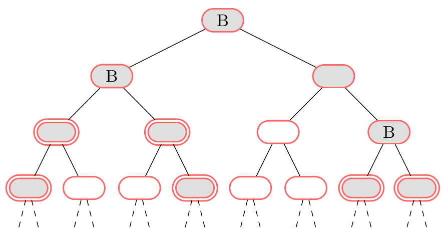
4.1 Caching Strategies

During the design of a caching strategy it is important to consider expected and worst case scenarios. The former is straight forward since the output of H is uniformly distributed, whereas the latter is strategy and use-case dependent. In the average case, however, non-empty leaves will be evenly spread out. Therefore a cluster of non-default digests will be formed at the first $\lceil \log n \rceil + 1$ layers, and intuitively these digests are important to prevent traversals down to the leaves. The digests rooted at layers below the dense threshold are of lesser importance due to the sparse property, but can be vital if non-empty leaves are *clumped*.

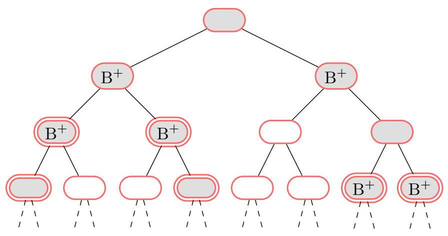
An intuitive caching strategy that we omit is to record the higher $\lceil \log n \rceil + 1$ layers of the SMT. While the dense part would be captured in the average case, forcing leaves to clump at some subtree is trivial for an adversary that selects the keys. Consequently, the majority of all the non-default digests cannot be captured, and the resulting cache will be useless if (non-)membership proofs are issued for that subtree. For this reason our caching strategies aim to bound the number of recursive traversals down to the leaves by a constant, evolving around capturing branches: parents whose children derive non-default digests [24].

B cache. Figure 4a depicts the B cache which captures every digest rooted at a branch. It contains $n-1$ digests at all times, and requires at most N traversals down to either a branch or a leaf upon generating audit paths. The former follows from the observation that all but the first insertion yield a single branch, and the latter (i.e., the worst case) is discussed in Section 5.3.

B$^-$ cache. By discarding $\mathrm{f}(n)$ branches from the B cache, memory requirements can be reduced at the cost of additional computation. This forms the notion of B$^-$, which provides trade-offs depending on how $\mathrm{f}(n)$ is implemented. We examine a probabilistic approach where a branch is captured with probability p, meaning $\mathrm{f}(n)$ is roughly $n(1-p)$. Other variations for $\mathrm{f}(n)$ include ignoring every other layer, as well as defining an upper bound for how many branches to ignore.



(a) B cache



(b) B$^+$ cache

Fig. 4: Captured digests as circled subtrees contain a single non-empty leaf.

B$^+$ cache. The drawback of using a B cache is that, in the average case, only the higher $\lceil \log n \rceil$ layers will be captured[3]. In other words, since the dense part also spans layer $\lceil \log n \rceil + 1$, we are missing out on some performance. B$^+$ aims to solve this issue by capturing branches together with their children. While the resulting cache cover the entire dense part of an SMT, we also bound the worst case memory requirements by $2n$ due to discarding branches (Figure 4b). The difference is negligible with regard to time because the children are cached.

4.2 The Cache Routine

Implementation wise our caching strategies are convenient. To process an interior digest, a cache function that accepts the left and the right child digests can be used. Upon invocation it computes the interior digest d, examines if both children are non-default, deletes the previous branch (if applicable), caches in case of a new branch, and outputs d. While this protocol merely concerns the B cache, it extends perfectly to B$^-$ and B$^+$. Therefore these caching strategies are practical to mix: start off with B$^+$, switch to B as memory requirements grow larger, and finally migrate to B$^-$ with shrinking probability p. For instance, this could be interesting in real-world settings where memory is a limited resource.

4.3 Recurrences

Let h be the height of a subtree, b the base of a node, and $\mathcal{D}$ a data structure containing unique keys' digests $\mathsf{H}(k)$. Further denote by α_i the $i^{th} \geq 0$ left-most bit in α, $\alpha_{i=\beta}$ the assignment of that bit to $\beta \in \{0,1\}$, and by colon (:) list concatenation. Finally, define the bit in the base that is set on right traversals[4] as $j := N - h$, the split index as $s := b_{j=1}$, and $\mathcal{D}$ divided on s for relation R as $\mathcal{D}_s^R := \{k | k \in \mathcal{D} \wedge kRs\}$. Our recurrences are shown in Figure 5 and interpreted as follows:

[3] Note that this drawback applies in an even larger extent for B$^-$ since B$^- \subset$ B.
[4] This bit refers to the depth of a subtree.

$$\xi_*^h := \begin{cases} \mathtt{LH}_*^*(a_0) & \text{, if } h = 0 \\ \mathtt{IH}_*^h\big(\xi_*^{h-1},\, \xi_*^{h-1}\big) & \text{, else .} \end{cases} \tag{1}$$

$$\mathtt{R}_b^h(\mathcal{D}) := \begin{cases} \delta_b^h & \text{, if available} \\ \xi_*^h & \text{, elif } |\mathcal{D}| = 0 \\ \mathtt{LH}_b^*(a_1) & \text{, elif } |\mathcal{D}| = 1 \land h = 0 \\ \mathtt{IH}_b^h\big(\mathtt{R}_b^{h-1}(\mathcal{D}_s^<),\, \mathtt{R}_s^{h-1}(\mathcal{D}_s^\geq)\big) & \text{, else .} \end{cases} \tag{2}$$

$$\mathtt{A}_b^h(\mathcal{D}, k) := \begin{cases} \emptyset & \text{, if } h = 0 \\ \mathtt{R}_s^{h-1}(\mathcal{D}_s^\geq) : \mathtt{A}_b^{h-1}(\mathcal{D}_s^<, k) & \text{, elif } k_j = 0 \\ \mathtt{R}_b^{h-1}(\mathcal{D}_s^<) : \mathtt{A}_s^{h-1}(\mathcal{D}_s^\geq, k) & \text{, else .} \end{cases} \tag{3}$$

$$\mathtt{B}_b^h(P, k, a) := \begin{cases} \mathtt{LH}_b^*(a) & \text{, if } h = 0 \\ \mathtt{IH}_b^h\big(\mathtt{B}_b^{h-1}(P, k, a),\, P[j]\big) & \text{, elif } k_j = 0 \\ \mathtt{IH}_b^h\big(P[j],\, \mathtt{B}_s^{h-1}(P, k, a)\big) & \text{, else .} \end{cases} \tag{4}$$

$$\mathtt{U}_b^h(\mathcal{D}, \mathcal{K}, a) := \begin{cases} \mathtt{LH}_b^*(a) & \text{, if } h = 0 \\ \mathtt{C}_b^h\big(\mathtt{R}_b^{h-1}(\mathcal{D}_s^<),\, \mathtt{U}_s^{h-1}(\mathcal{D}_s^\geq, \mathcal{K}, a)\big) & \text{, elif } \big|\mathcal{K}_s^<\big| = 0 \land \big|\mathcal{K}_s^\geq\big| \neq 0 \\ \mathtt{C}_b^h\big(\mathtt{U}_b^{h-1}(\mathcal{D}_s^<, \mathcal{K}, a),\, \mathtt{R}_s^{h-1}(\mathcal{D}_s^\geq)\big) & \text{, elif } \big|\mathcal{K}_s^<\big| \neq 0 \land \big|\mathcal{K}_s^\geq\big| = 0 \\ \mathtt{C}_b^h\big(\mathtt{U}_b^{h-1}(\mathcal{D}_s^<, \mathcal{K}_s^<, a),\, \mathtt{U}_s^{h-1}(\mathcal{D}_s^\geq, \mathcal{K}_s^\geq, a)\big) & \text{, else .} \end{cases} \tag{5}$$

Fig. 5: Recurrences that derive default digests (ξ), root digests ($\mathtt{R}$), audit paths ($\mathtt{A}$), reconstructed root digests ($\mathtt{B}$), and relative information ($\mathtt{U}$).

- Given a height h, (1) derives the default digest d_*^h. The leaf hash (LH) and interior hash (IH) functions serve the purpose of encoding digests securely, as further described in Section 5.
- Given a height h, a base b, and a collection of keys $\mathcal{D}$, (2) derives the digest d_b^h. The base case occurs if there is relative information available, if a default digest is applicable, or if a non-empty leaf is reached. Otherwise, (2) performs two recursive calls with $\mathcal{D}$ divided on s, b updated on right traversals, and h reduced by one.
- Given a height h, a base b, a collection of keys $\mathcal{D}$, and a key k for leaf ℓ, (3) generates an audit path for ℓ. Note that all siblings' digests are gathered by list concatenation and invoking (2) after reaching ℓ.
- Given a height h, a base b, an attribute $a \in \{a_0, a_1\}$, and an audit path P for key k, (4) reconstructs the root digest by traversing the tree structure down to the leaf being authenticated. Every sibling's digest is obtained from $P[j]$.
- Given a height h, a base b, a collection of keys $\mathcal{D}$, a subset of keys $\mathcal{K} \subset \mathcal{D}$ where $\mathcal{K} \neq \emptyset$, and an attribute $a \in \{a_0, a_1\}$, (5) outputs the new root digest and updates the relative information. This is achieved by visiting all leaves $\ell \in \mathcal{K}$, also invoking the cache function (C) to compute the interior digest d_b^h and ensure that the relative information is up-to-date.

The size of an audit path is $\mathcal{O}(1)$, but can be further reduced by discarding default digests. That yields a *sparse audit path*, and necessitates encoding of an N-bitmap to determine whether a digest is (non-)default. We omit the details of such a recurrence since it is trivially added when (3)–(4) is provided.

5 Security of Audit Paths

Consider a single SMT and assume that the hash function is fixed. Then it follows that the size of an audit path is fixed by N due to the structure of tree, and consequently we can distinguish between leaves and interior nodes. This means that, for the case of a single SMT with a fixed hash function, no special encoding is necessary to distinguish between nodes, and that the security of an audit path reduces to the collision resistance of the underlying hash function (Lemma 1).

Next, to prevent an adversary from gaining any advantage when brute-forcing several SMTs in parallel, we consider the full (concrete) security of an audit path in the multi-instance setting. Thereafter we relate our encoding of nodes to CONIKS [19], and examine the impact of caching strategies for security.

5.1 The Merkle Prefix Tree in CONIKS

CONIKS, as described more broadly in Section 7, is a key verification service that uses a Merkle prefix tree (MPT) to authenticate users' key bindings [19]. An MPT can be seen as a dynamically sized and explicit SMT where empty subtrees are replaced by empty nodes. Key-bindings are mapped to unique indices i by a hash function, and every (non-)empty leaf in the tree is associated with a depth ℓ and an ℓ-bit unique prefix j of i. The encoding of an empty node is defined by (6).

$$\mathsf{H}(C_{\text{empty}} \| C_{\text{tw}} \| j \| \ell) \tag{6}$$

C_{empty} is a constant for empty leaves and C_{tw} a tree-wide constant. The encoding of a non-empty node containing the payload p is defined by (7).

$$\mathsf{H}(C_{\text{leaf}} \| C_{\text{tw}} \| i \| \ell \| p) \tag{7}$$

C_{leaf} is a constant for non-empty leaves. Finally, the encoding for an interior node is defined by (8).

$$\mathsf{H}(d_{\text{right}} \| d_{\text{left}}) \tag{8}$$

The constants C_{empty} and C_{leaf} serve the purpose of preventing indistinguishability between (non-)empty leaves, and the tree-wide constant C_{tw} provides some protection against an adversary in the multi-instance setting. That is, if all MPTs use distinct tree-wide constants, no nodes' pre-images can be valid across different trees. Similarly, no nodes' pre-images can be valid across multiple locations because the leaves' digests are uniquely encoded by $j\|\ell$ and $i\|\ell$ (the location of an interior node is implicit due to the children it commits to). Thus, as oppose to searching collisions across multiple trees and locations in parallel, an adversary must target a particular tree and location. However, we also need to consider different versions of trees generated by updates. To accomplish full λ-bit security for an instance of the MPT, a new tree-wide constant must be selected after each update to prevent parallel attacks through past versions of the tree structure. For all updates, the entire MPT has to be recomputed.

5.2 A Secure Encoding for Sparse Merkle Trees

Figure 6 defines a secure encoding for an SMT in the multi-instance setting. We prevent attacks across distinct trees by introducing a tree-wide constant C_{tw}, but we do not protect against attacks on different versions of the same tree because C_{tw} is reused between updates. For attacks within particular tree structures, we include unique identifiers in every *non-empty subtree*. This differs with respect to MPTs, but is necessary to preserve the sparse property of an SMT: if unique prefixes were included in all empty subtrees, then there would no longer be any default digests. As shown in (10), we solve this issue and retain security by moving the encoding of an empty node into the non-empty parent. An interior node that is non-default will still commit properly to a certain location encoded by base and height[5], and since the digest of an empty node is publicly known even for an MPT no security is lost. Furthermore, note that we do not encode the attributes a_0 and a_1 explicitly in (9). Inclusion of the base suffices to distinguish between (non-)empty leaves.

$$\mathtt{LH}_b^*(a) := \begin{cases} \mathsf{H}(C_{\text{tw}}) & \text{, if } a = a_0 \\ \mathsf{H}(C_{\text{tw}} \| b) & \text{, else }. \end{cases} \tag{9}$$

$$\mathtt{IH}_b^h\left(d_{\text{left}},\, d_{\text{right}}\right) := \begin{cases} \mathsf{H}(d_{\text{left}} \| d_{\text{right}}) & \text{, if } d_{\text{left}} = d_{\text{right}} = \xi_*^{h-1} \\ \mathsf{H}(d_{\text{left}} \| d_{\text{right}} \| b \| h) & \text{, else }. \end{cases} \tag{10}$$

Fig. 6: Secure node encodings for an SMT.

5.3 Security Aspects of Caching Strategies

Often, one distinguishes between best, worst, and average case complexities. For instance, a hash table has amortized constant look-up time, but can degrade to a linear construction if all entries hash to the same bucket. Likewise, a binary search tree that is probabilistically balanced is in danger of breaking down into a linked list. Critics might claim that such attacks are of theoretical interest alone, but Crossby and Wallach [7] has already presented a class of denial of service attacks that exploits such degradations. Thus, showing the importance of evaluating worst case behaviour and their likelihood to ever occur.

[5] The height is necessary since the base is fixed-size and traversals to the left of an SMT does not modify the base.

Therefore, consider the B cache. In the worst case, if there are merely N keys, an adversary could force an almost perfect spine of branches as depicted in Figure 7. Whenever membership proofs are issued for the leaves on that spine, the large majority of all the non-default digests must be computed because the siblings are not captured by the cache. While this is not an issue for a small SMT, the worst case efficiency actually increases as the tree grows: new insertions yield additional branches, and it is more efficient to stop traversals at a branch than at a leaf. In other words, there are two scenarios each time a sibling's digest is requested. First, the digest is default and can be requested in constant time. Second, the digest is non-default and can be derived by traversing the tree down to a branch or leaf. In either case, regardless of how an adversary selects the keys, *at most* N traversals are necessary (one per layer). This analysis applies to B$^+$ with minor modifications, and for B$^-$ one can show that the number of traversals will be bounded by f(n). As such, to prevent an adversary from causing inefficiency, f(n) must be constant or unpredictable to the adversary.

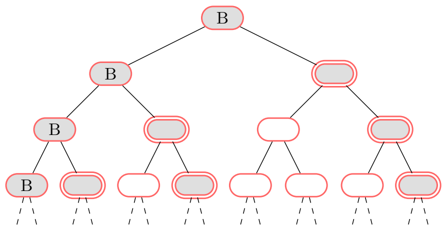


Fig. 7: A branch spine, potentially caused by an adversary.

A similar analysis for worst case behaviour applies during updates. This follows from the observation that (3) and (5) traverse the tree down to the leaves, invoking (2) on each layer.

6 Performance

We evaluate performance and space-time trade-offs experimentally with a proof of concept implementation in Go[6]. We used SHA-512/256 as the hash function[7], a non-authenticated data structure that supports splitting in logarithmic time, and relative information that is maintained in constant time (a hash table). All experiments use an Intel(R) Core(TM) i7-4790 CPU @ 3.60 GHz with 2x8 GB DDR4 RAM, as well as Go's built-in benchmarking tool.

The B$^-$ cache was implemented probabilistically such that a branch is captured with probability p (uniformly at random). While this is inefficient in the worst case, it does provide resilience against an adversary that selects the keys.

[6] Source code available at `https://github.com/pylls/gosmt`.

[7] SHA-512 truncated to 256-bit output, resulting in an SMT with 2^{256} leaves [23].

Denoted by $\mathrm{B}^{\text{-}}_{p}$, we evaluate $\mathrm{B}^{\text{-}}$ for $p \in \{0.5 \ldots 0.9\}$. We also include B, B^{+}, and a hash treap in our experiments. For relevant operations, the expected $\mathcal{O}(\log n)$ time complexity of a hash treap makes it a good representation of other related authenticated data structures that are explicitly stored in memory.

Figure 8a shows the size of the authenticated data structure as a function of the data structure being authenticated. There is essentially no distinction between the two for a hash treap[8], and in the case of an SMT this is the relation between δ and $\mathcal{D}$. For 2^{20} keys, the hash treap needs 960 MiB, the B^{+} cache 512 MiB, the B cache 256 MiB, and the $\mathrm{B}^{\text{-}}_{0.5}$ cache 128 MiB. It is evident that the different caches double in size, and that the size of a hash treap is roughly eight times larger than that of a $\mathrm{B}^{\text{-}}_{0.5}$ cache. Furthermore, it should be noted that the $\mathrm{B}^{\text{-}}_{p}$ caches with $p \in \{0.6 \ldots 0.9\}$ have sizes evenly distributed in $[\mathrm{B}^{\text{-}}_{0.5}, \mathrm{B}]$.

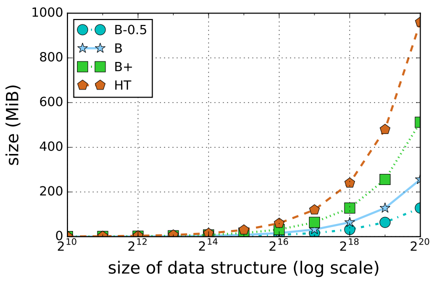
Figure 8b shows the time required to generate an audit path. Since the full structure is in memory for the hash treap, it is just a matter of copying the nodes along the path in negligible time (0.003 ms). Similarly, for B^{+} and B, we see consistent results that are less than 1 ms regardless of how large $\mathcal{D}$ is. This is because both caching strategies ensure that the vital non-default digests are cached, whereas additional recursive traversals down to either branches or leaves are necessary for $\mathrm{B}^{\text{-}}_{p}$. Finally, we also observe the impact of selecting p. While $p > 0.6$ gives an expected time that is less than 4 ms, $p = 0.5$ behaves erratically. This follows from the high probability that a siblings' digests must be derived instead of being found in the cache, as is also evident to a smaller extent for $p = 0.6$.

Figure 8c shows the time it takes to update m keys in a data structure containing $n = 2^{15}$ keys. All approaches scale as $\mathcal{O}(m \log n)$, with the hash treap being significantly faster. Similarly, Figure 8d shows the time it takes to update $m = 256$ keys as a function of the size n. The B^{+} cache consistently needs less than 20 ms, as oppose to the hash treap which needs 9.5 ms for $n = 2^{20}$. Considering that a hash treap consumes twice as much memory, this is indeed an interesting trade-off. For the remaining caching strategies, p together with the relation between n and m determines the probability of having cache misses. Simplified, larger p yields less variance and greater efficiency in terms of time.

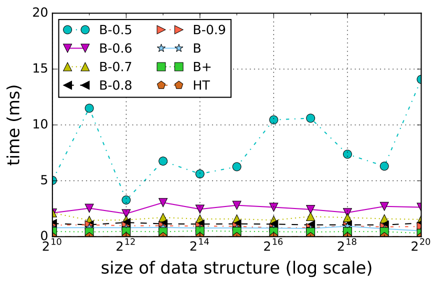
7 Related Work

Google considers three categories of authenticated data structures when adding transparency to a trust model: verifiable logs, maps, and log-backed maps [12]. While CT relies on verifiable logs to support efficient consistency and membership proofs, verifiable maps based on SMTs are proposed in RT to prove non-membership. This is not without issues, however. All operations must be enumerated to determine whether a map's state is correct. Therefore, the former two categories are combined into a verifiable log-backed map where consistency issues can be detected by the verifiable log, (non-)membership can be proven by

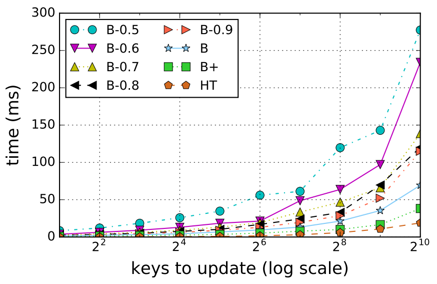
[8] The size refers to the nodes of the tree together with the children's pointers.



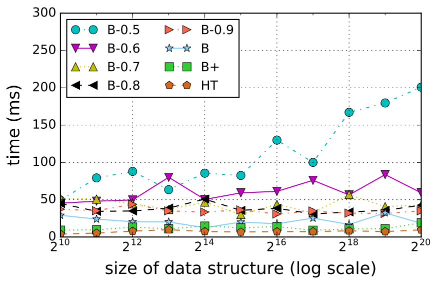
(a) Size of authenticated data structure.



(b) Time to generate an audit path.



(c) Time to update in a 2^{15} data structure.



(d) Time to update 256 keys.

Fig. 8: Space-time trade-offs for a hash treap (abbreviated HT) and different caching strategies for SMTs.

the verifiable map, and full audits can ensure complete correct behaviour. As such, using an efficient verifiable map based on our extension of an SMT, the combination of CT and RT can prove whether a certificate's status is current. Other CT-like proposals that an SMT could be applicable to include Distributed Transparent Key Infrastructure [32] and Enhanced Certificate Transparency [29].

Verifiable maps are closely related to persistent authenticated dictionaries (PADs) [10]. While both are dynamic, the difference is that a PAD supports (non-)membership queries to current *and* past versions of the data structure. Hence, if our representation of an SMT is first extended to a key-value store[9], adding some form of persistency would yield a PAD. Crosby and Wallach [9] investigate caching strategies for tree-based PADs in conjunction with Sarnak and Tarjan [30] versioned nodes. We could use a similar approach, or the path copying approach of Anagnostopoulos et al. [1], for the *cache* in our SMT.

CONIKS is a privacy-preserving key-management service that allows clients to monitor their own key-bindings efficiently [19]. An MPT (See Section 5.1) is used for the purpose of verifiability, but prior to deriving a unique index i the key-bindings are first transformed by a *verifiable unpredictable function* [21].

[9] It suffices to add a payload in the non-empty leaves.

While that prevents audit paths from leaking user information, it cannot conceal the total number of users. CONIKS solves this issue and others (e.g., ensuring *fork consistency* [18]) by defining a protocol on top of an MPT. It appears that an SMT could be a viable and attractive replacement if viewed as a dictionary.

The issue of proving non-membership is not only evident in CT and RT. For instance, in privacy-preserving transparency logging [28], Balloon is a provably secure append-only data structure that supports historical (non-)membership queries [27]. This is accomplished by inheriting a scheme for authenticated data structures defined by Papamanthou *et al.* [25], as well as combining a history tree [8] and a hash treap [10,27]. The former is essentially a verifiable log, and the latter a treap [2,4] viewed as a Merkle tree. While hash treaps and SMTs share many properties, e.g., efficient (non-)membership proofs an history independent representations, there are some striking differences. To begin with, hash treaps store attributes in each node. As oppose to an SMT, information regarding these attributes must be provided in audit paths due to encoding digests differently (possibly leaking valuable information). There will also be exactly n nodes at all times, and efficiency relies on a probabilistic balance. In these regards an SMT is flexible: the variable parameters $\mathcal{D}$ and δ determine if/when efficiency is provided, and memory requirements can be reduced to less than n if need be.

Finally, this work is an extension of the Bachelor's thesis by Dahlberg [24]. Apart from improving terminology, we define recursions for batch updates and reconstruction of root digests, as well as caching strategies based on branches. We also add a security evaluation for full security in the multi-instance setting, provide a publicly available implementation that uses a memory safe language, and compare our results with a related authenticated data structure.

8 Conclusion

Our definition of an SMT builds upon and extends the principles provided by Laurie and Kasper [16]. The proposal is generic in the sense that an arbitrary data structure supporting insertion, removal, look-up, and splitting can be used, and different caching strategies (B, B^{-}, and B^{+}) provide fine-grained control over consumed space contra run time. In other words, rather than having an explicit tree structure, the resulting SMT is simulated. While this comes as the cost of additional computation when compared to other explicit tree-based data structures, our benchmarks and worst case analysis imply that our definitions are efficient regardless of how an adversary selects the keys. We also prove our definitions secure in the multi-instance setting.

There is nothing that prevents further space-time trade-offs as the tree evolves. In principal, the relation $\text{B}^{-} \subset \text{B} \subset \text{B}^{+}$ holds. Therefore it is simple to go from one strategy to another depending on, e.g., how much memory is available at the time being. This is a major difference with respect to an explicit tree structure, which has no construction that is alike. Furthermore, the succinct recursions used to simulate an SMT yield limited implementation complexity, and history independence is a prevalent property if parallelized and distributed solutions are considered for large-scale applications.

Acknowledgements We would like to thank Stefan Lindskog for his valuable feedback. Rasmus Dahlberg and Tobias Pulls have received funding from the HITS research profile funded by the Swedish Knowledge Foundation.

References

1. Anagnostopoulos, A., Goodrich, M.T., Tamassia, R.: Persistent authenticated dictionaries and their applications. In: Davida, G.I., Frankel, Y. (eds.) Information Security, 4th International Conference, ISC 2001, Malaga, Spain, October 1-3, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2200, pp. 379–393. Springer (2001)
2. Aragon, C.R., Seidel, R.: Randomized search trees. In: 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989. pp. 540–545. IEEE Computer Society (1989)
3. Bauer, M.: Proofs of zero knowledge. CoRR cs.CR/0406058 (2004), http://arxiv.org/abs/cs.CR/0406058
4. Blelloch, G.E., Reid-Miller, M.: Fast set operations using treaps. In: SPAA. pp. 16–26 (1998)
5. Blum, M., Evans, W.S., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. Algorithmica 12(2/3), 225–244 (1994)
6. Chuat, L., Szalachowski, P., Perrig, A., Laurie, B., Messeri, E.: Efficient gossip protocols for verifying the consistency of certificate logs. In: 2015 IEEE Conference on Communications and Network Security, CNS 2015, Florence, Italy, September 28-30, 2015. pp. 415–423. IEEE (2015)
7. Crosby, S.A., Wallach, D.S.: Denial of service via algorithmic complexity attacks. In: Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003. USENIX Association (2003), https://www.usenix.org/conference/12th-usenix-security-symposium/denial-service-algorithmic-complexity-attacks
8. Crosby, S.A., Wallach, D.S.: Efficient data structures for tamper-evident logging. In: Monrose, F. (ed.) 18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings. pp. 317–334. USENIX Association (2009)
9. Crosby, S.A., Wallach, D.S.: Super-efficient aggregating history-independent persistent authenticated dictionaries. In: Backes, M., Ning, P. (eds.) Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5789, pp. 671–688. Springer (2009)
10. Crosby, S.A., Wallach, D.S.: Authenticated dictionaries: Real-world costs and trade-offs. ACM Trans. Inf. Syst. Secur. 14(2), 17 (2011)
11. Eckersley, P.: How secure is https today? how often is it attacked? Electronic Frontier Foundation (October 2011), https://www.eff.org/deeplinks/2011/10/how-secure-https-today
12. Eijdenberg, A., Laurie, B., Cutter, A.: Verifiable data structures. Google Research (November 2015), https://github.com/google/trillian/blob/master/docs/VerifiableDataStructures.pdf
13. Katz, J.: Analysis of a proposed hash-based signature standard (2014)
14. Langely, A.: Enhancing digital certificate security. Google Research (January 2013), https://security.googleblog.com/2013/01/enhancing-digital-certificate-security.html

15. Laurie, B.: Certificate transparency. ACM Queue 12(8), 10–19 (2014)
16. Laurie, B., Kasper, E.: Revocation transparency. Google Research (September 2012), http://www.links.org/files/RevocationTransparency.pdf
17. Laurie, B., Langley, A., Kasper, E.: Certificate transparency. RFC 6926 (June 2013), https://tools.ietf.org/html/rfc6962, internet Requests for Comments
18. Li, J., Krohn, M.N., Mazières, D., Shasha, D.: Secure untrusted data repository (SUNDR). In: Brewer, E.A., Chen, P. (eds.) 6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004. pp. 121–136. USENIX Association (2004)
19. Melara, M.S., Blankstein, A., Bonneau, J., Felten, E.W., Freedman, M.J.: CONIKS: bringing key transparency to end users. In: Jung, J., Holz, T. (eds.) 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015. pp. 383–398. USENIX Association (2015)
20. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings. Lecture Notes in Computer Science, vol. 293, pp. 369–378. Springer (1987)
21. Micali, S., Rabin, M.O., Vadhan, S.P.: Verifiable random functions. In: 40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA. pp. 120–130. IEEE Computer Society (1999)
22. Naor, M., Nissim, K.: Certificate revocation and certificate update. IEEE Journal on Selected Areas in Communications 18(4), 561–570 (2000)
23. National Institute of Standards and Technology: FIPS PUB 180-4: Secure Hash Standard. Federal Information Processing Standards Publication 180-4, U.S. Department of Commerce (March 2012), http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf
24. Östersjö/Dahlberg, R.: Sparse Merkle Trees: Definitions and Space-Time Trade-Offs with Applications for Balloon. Bachelor's thesis, Karlstad University (June 2016)
25. Papamanthou, C., Tamassia, R., Triandopoulos, N.: Optimal verification of operations on dynamic sets. In: Rogaway, P. (ed.) Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6841, pp. 91–110. Springer (2011)
26. Prins, R.: Diginotar certificate authority breach — "operation black tulip". Fox-IT (September 2011)
27. Pulls, T., Peeters, R.: Balloon: A forward-secure append-only persistent authenticated data structure. In: Pernul, G., Ryan, P.Y.A., Weippl, E.R. (eds.) Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9327, pp. 622–641. Springer (2015)
28. Pulls, T., Peeters, R.: Insynd: Privacy-preserving transparency logging using balloons. ESORICS (2016), to appear
29. Ryan, M.D.: Enhanced certificate transparency and end-to-end encrypted mail. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014. The Internet Society (2014)
30. Sarnak, N., Tarjan, R.E.: Planar point location using persistent search trees. Commun. ACM 29(7), 669–679 (1986)

31. Tamassia, R.: Authenticated data structures. In: Battista, G.D., Zwick, U. (eds.) Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2832, pp. 2–5. Springer (2003)
32. Yu, J., Cheval, V., Ryan, M.: DTKI: a new formalized PKI with no trusted parties. CoRR abs/1408.1023 (2014), http://arxiv.org/abs/1408.1023