Super-efficient Aggregating History-independent Persistent Authenticated Dictionaries[⋆]

Scott A. Crosby and Dan S. Wallach

Rice University, {scrosby,dwallach}@cs.rice.edu

Abstract. Authenticated dictionaries allow users to send lookup requests to an untrusted server and get authenticated answers. Persistent authenticated dictionaries (PADs) add queries against historical versions. We consider a variety of different trust models for PADs and we present several extensions, including support for aggregation and a rich query language, as well as hiding information about the order in which PADs were constructed. We consider variations on tree-like data structures as well as a design that improves efficiency by speculative future predictions. We improve on prior constructions and feature two designs that can authenticate historical queries with constant storage per update and several designs that can return constant-sized authentication results.

1 Introduction

This paper considers data being stored in a cryptographic and tamper evident fashion. The earliest example of such a data structure was the Merkle tree [26], where each tree node contains a cryptographic hash of its childrens' contents. Consequently, the root node's hash value fixes the values of the entire tree. Hash-based data structures have been used in a variety of different systems, including smartcards [17], outsourced databases [41], distributed filesystems [29, 24, 35, 16], graph and geometric searching [19], tamper-evident logging [11, 12, 37], and many others. These systems are often built around the *authenticated dictionary* [31, 23] abstraction, which supports ordinary dictionary operations, with lookups returning the answer and a proof of its correctness.

In systems where data changes values over time, such as stock ticker data, revision control systems [38], or public key infrastructure, participants will want to query historical versions or *snapshots* of the repository as well as the most recent version. Persistent data structures were developed to support these features and have been extensively studied [8, 22], particularly with respect to functional programming [33, 4].

Persistent authenticated dictionaries (PADs) combine these features and were introduced by Anagnostopoulos et al. [1], using applicative (i.e., functional or mutation-free) red-black trees and skiplists, requiring $O(\log n)$ storage per update.

In Sect. 2 we discuss threat models and features that PADs may support. In Sect. 3, we show how to adapt Sarnak and Tarjan's construction [36] in order to build PADs

[⋆] The authors wish to thank the anonymous referees for their helpful comments and feedback. We also thank the program chairs for allowing us to expand our paper beyond its original length to better address the referees' concerns. This research was funded, in part, by NSF grants CNS-0524211 and CNS-0509297.

with lower storage overheads, including a design with constant storage per update. In Sect. 4 we develop *super-efficient* PADs based around a different design principle, offering constant-sized authentication results, as well as constant storage per update. In Sect. 5 we summarize the expected running times of our algorithms. Finally, in Sect. 6 we describe future work and conclusions.

2 Definitions and models

In this paper, we focus on authenticating set-membership and non-membership queries over a dynamic set, stored on an untrusted server. To prevent the server from lying about the data being stored, the author includes authentication information permitting lookup responses to be verified.

The authenticated dictionary [31] abstraction supports the ordinary dictionary operations, $\textsc{Insert}(\textsc{Key}, \textsc{Val})$ and $\textsc{Delete}(\textsc{Key})$, which *update* the contents. Lookups, $\textsc{Lookup}(\textsc{Key}) \rightarrow (\textsc{Val}, P)$ return both the answer or $\square$ if no such key exists, and a *membership proof* P of the correctness of their result. Ultimately, a server must prove that a given query result is consistent with some external data, such as an author's signature on the tree's root hash.

Authenticated dictionaries become persistent [1] when they allow the author to take snapshots of the contents of the dictionary. Queries can be on the current version, or any historical snapshot. PADs ideally have efficient storage of all the snapshots, presumably sharing state from one snapshot to the next.

2.1 Threat model

We make typical assumptions for the security of cryptographic primitives. We assume that we have idealized cryptographic one-way hash functions (i.e., collisions never occur and the input can never be derived from the output), and that public key cryptography systems' semantics are similarly idealized. We also assume the existence of a trusted PKI or other means to identify the public key associated with an author.

In this paper, we consider a trust model with three parties: a trusted *author* with limited storage and possibly intermittent connectivity, an untrusted *server* with significant storage and a consistent online connection, and multiple *clients* who perform queries and have limited storage.

The author asks the server to insert or remove (key, value) pairs, providing any necessary authentication information. When clients contact the server they will verify the resulting proof which will include validating the consistency of the server's data structure as well as the author's digital signature.

We also consider scenarios where the author of a PAD is not trusted, which can be relevant to a variety of financial auditing and regulatory compliance scenarios. For instance, the author may wish to maliciously change past values of the PAD, possibly in collusion with the server. Or, the author may be responsible for collecting and aggregating records, such as a list of bank accounts and balances and attempt to misbehave. Fortunately, if the author ever signs inconsistent answers or it improperly aggregates records, its misbehavior can be caught by clients and auditors.

2.2 Features

An authenticated dictionary (persistent or not) may support many features. In this section, we describe features supported by the dictionaries we investigate.

Super-efficiency. The proof returned on a lookup request is constant-sized. Our tuple-based PADs, described in Sect. 4 offer super-efficiency.

Partial persistence. The PADs we consider are actually partially persistent, meaning that although any version of the authenticated dictionary may be queried, only the latest version can be modified.[1] Whenever we use the term "persistent" in this paper, we really mean "partially persistent." To this end, we offer two additional operations, $\textsc{Snapshot}() \rightarrow \textsc{Version}$ is used to take a snapshot of the current contents of the dictionary and returns a version number. $\textsc{Lookup}\mathrm{V}(\textsc{Key}, \textsc{Version}) \rightarrow (\textsc{Val}, P)$ looks up the value, if any, associated with a key in a historical snapshot and returns a proof P of the correctness of the result. Snapshots can be taken at any time. For simplicity when we evaluate costs, we will assume a snapshot is taken after every update.

History independence. Some data structures can hide information as to the order in which they were constructed. For instance, if data items are stored, sorted in an array, no information would remain as to the insertion order. History independence can derive from randomization; Micciancio [28] shows a 2-3 tree whose structure depends on coin tosses, not the keys' insertion order.

History independence can also derive from data structures that have a canonical or unique representation [32]. To this end, our data structures are "set-unique" [2], meaning that a given set of keys in the dictionary has a unique and canonical representation (see Sect. 3.2). Our tree-based PAD designs and some of our tuple-based PADs are history-independent.

In a persistent dictionary, history independence means that if multiple updates occur between two adjacent snapshots, the client learns nothing as to the order in which the updates occurred and the server learns nothing if it receives the updates as a batch. In addition, it must not be possible for a client to learn anything about the keys in one snapshot, given query responses from any other snapshots.

Aggregates. Any tree data structure may include aggregates that summarize the children of a given node (e.g., capturing their minimum and maximum values or their sum). These aggregates are valuable on their own and may be used for searching or other applications (see Sect. 3.1). Our tree-based PADs support aggregates.

Root authenticators. For each snapshot, it would be beneficial if there was a single value that fixes or commits the entire dictionary at that particular time. This value can then be stored and replicated efficiently by clients, stored in a time-stamping system [21, 9], or tamper-evident log [11, 12, 37]. Root authenticators simplify the process of discovering when an untrusted author or server may be lying about the past. Mistrusting

[1] In the persistency literature [13], the term "persistent" is reserved for data structures where any version, present or past, may be updated, thus forming a tree of versions. Path copying trees, described in Sect. 3.3, are an example of such a data structure. Confluently persistent data structures permit merge operations between snapshots [15].

clients need only to discover that the author has signed different root authenticators for the same snapshot. They need not look any deeper.

3 Tree-based PADs

In this section, we describe how we can build PADs with balanced search trees. Tree-based PADs have membership proof sizes, update sizes and membership proof verification times that are logarithmic in the number of keys in the dictionary. Tree-based PADs offer a range of query time and storage-space tradeoffs. In this section, we first describe the three components from which we build our tree-based PADs: Merkle trees, treaps, and persistent binary search trees. We then show how to combine them.

3.1 Merkle trees

Given a search tree, where each node contains a key, value, and two child pointers, we can build an authenticated dictionary by building a Merkle tree [26]. For each node x, we assign a *subtree authenticator* $x.H$ with the following recurrence: $x.H = H(x.key, H(x.val), x.left.H, x.right.H)$. H denotes a cryptographic hash function. The *root authenticator*, $root.H$, authenticates the whole tree. It may then be published or signed by the author. Merkle trees also support a feature called Merkle aggregation where nodes in a search tree can be annotated with additional data that may be accumulated up the tree. (More on aggregation below.)

A *membership proof*, seen in Fig. 1 and returned on a Lookup request is a proof that a key k_q is in the tree. It consists of a pruned tree containing the search path to k_q. Subtree authenticators for the sibling nodes on the search path are included in the proof as well as subtree authenticators of the children of the node containing k_q, if k_q is found. From this pruned tree, the root authenticator is reconstructed and compared to the given root authenticator. We can prove that a key is not in the tree by showing that the unique in-order location where that key would otherwise be stored is empty.

For a balanced search tree, a membership proof has size $O(\log n)$, and can be generated in $O(\log n)$ time if the subtree authenticators are precomputed. Conventional implementations of authenticated search trees implement a logical *subtree authenticator cache* storing the subtree authenticator for each node in the node itself. Note that this cache is optional, because the server could certainly recompute any hash on the fly from the existing tree. Without a cache, generating a membership proof requires $O(n)$ time for recomputing subtree authenticators of elided subtrees. Of course, the cache has obvious performance benefits. In Sect. 3.3, we will consider how, where, and when these subtree authenticators are cached and investigate tradeoffs in caching strategies.

Merkle aggregation. Merkle aggregation [11] was originally applied to annotating events in a Merkle tree storing a tamper-evident log. These annotations are then aggregated up to the root of the tree where they may be directly queried or used to perform authenticated searches. For example, in a log of bank transactions, annotations could be flags for notable transactions, dollar values aggregated by sum, or time intervals aggregated by min and max bounds. To prevent tampering, the annotations of a node are included in the subtree authenticator of its parent. If the author is not trusted, these annotations can be checked by auditors to verify the author's proper behavior.

We extend Merkle aggregation to binary search trees that include keys and values in interior nodes. We let the *subtree aggregate* of a node x be $x.A$, Γ be a function that computes the annotation associated with a key and value pair, and $\oplus$ be a function that aggregates. If we define $x.* = H(x.H, x.A)$, then we can describe the Merkle aggregation over a search tree with the formulas: $x.A = \Gamma(x.key, x.val) \oplus x.left.A \oplus x.right.A$ and $x.H = H(x.key, H(x.val), x.left.*, x.right.*)$. Wherever a host previously stored or included the hash of a node in a proof, it will now include the node's hash and aggregate, which can be cached or recomputed as-needed.

3.2 Treap

Our tree-based dictionaries are based on treaps [3], a randomized search tree implementing a dictionary. The expected cost of an insert, delete, or lookup is $O(\log n)$. Treaps support efficient set union, difference, and intersection operations [6]. We could have used any other balanced search tree that supports $O(1)$ expected (not amortized) node mutations per update, such as AVL or red-black trees [20], but we preferred treaps for their set-uniqueness properties (discussed further below).

Each node in a treap is given a key, value, priority, and left and right child pointers. Nodes in a treap obey the standard search-key order; a node's key always compares greater than all of the keys in its left subtree and less than all of the keys in its right subtree. In addition, each node in a treap obeys the heap property on its priorities; a node's priority is always less than the priorities of its descendants. Operations that mutate the tree will perform rotations to preserve the heap property on the priorities. When the priorities are assigned at random, the resulting tree will be probabilistically balanced. Furthermore, given an assignment of priorities to nodes, a treap on a given set is unique.[2] We exploit this uniqueness by creating *deterministic treaps*, assigning priorities using a cryptographic digest of the key, creating a set-unique representation.

Assuming that the cryptographic digest is a random oracle, in expectation, each insert and delete only mutates $O(1)$ nodes, consisting of one node having a child pointer modified and $O(1)$ rotations. The expected path to a key in the treap is $O(\log n)$.

Benefits of a set-unique representation. Deterministic treaps are set-unique, which means that all authenticated dictionaries with the same contents have identical tree structures. If we build Merkle trees from these treaps, then any two authenticated dictionaries with identical contents will have identical root hashes. Set-uniqueness makes our treaps history independent. The root hash that authenticates a treap leaks no information about the insertion order of the keys or the past contents of the treap, which may be valuable, for example, with electronic vote storage or with zero-knowledge proofs.

History-independence is also useful if an dictionary is used to store or synchronize replicated state in a distributed system. Updates may arrive to replicas out-of-order, perhaps through multicast or gossip protocols. Also, by using a set-unique authenticated data structure, we can efficiently determine if two replicas are inconsistent.

History independence makes it easier to recover from backups or create replicas. If a host tries to recover the dictionary contents from a backup or another replica, history independence assures that the recovered dictionary has the same root hash. Were a

[2] Proof sketch: If all priorities are unique for a given set of keys, then there exists one unique minimum-priority node, which becomes the root. This uniquely divides the set of keys in the treap into two sets, those less than and greater than the key, stored in the left and right subtrees, respectively. By induction, we can assume that the subtrees are also unique.

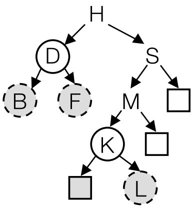

Fig. 1. Graphical notation for a membership proof for M or a non-membership proof for N. Circles denote the roots of elided subtrees whose children, grayed out, need not be included.

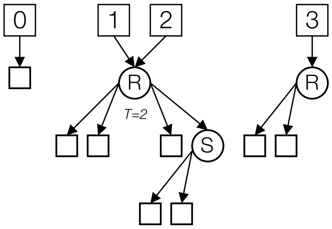

Fig. 2. Four snapshots in a Sarnak-Tarjan versioned-node tree, starting with an empty tree, then inserting R, then inserting S, then deleting S. We show the archived children to the left of a node and the current children to the right. Note that R is modified in-place for snapshot 2, but cloned for snapshot 3.

non-set-unique data structure, such as red-black trees used, the different insertion order between the original dictionary and that used when recovering would likely lead to different root hashes even though the recovered dictionary had the same contents.

3.3 Persistent binary search trees

Persistent search tree data structures extend ordinary search tree data structures to support lookups in past snapshots or versions. In this section we summarize the algorithms proposed by Sarnak and Tarjan [36], who considered approaches for persistent red-black search trees, and apply their techniques to treaps.

Logically, a persistent dictionary built with search trees is simply a forest of trees, i.e., a separate tree for each snapshot. The root of each of these trees is stored in a *snapshot array*, indexed by snapshot version. Historical snapshots are frozen and immutable. The most recent, or *current* snapshot can be updated in place to include inserted or removed keys. Whenever a snapshot is taken, a new root is added to the snapshot array and that snapshot is thereafter immutable.

Three strategies Sarnak and Tarjan proposed for representing the logical forest are *copy everything*, *path copying*, and *versioned nodes*. They range from $O(n)$ space to $O(1)$ space per update. Note that these different physical representations store the same logical forest. The simplest, *copy everything*, copies the entire treap on every snapshot and costs $O(n)$ storage for a snapshot containing n keys.

Path copying uses a standard applicative treap, avoiding the redundant storage of subtrees that are identical across snapshots. Nodes in a path-copying treap are immutable. Where the normal, mutating treap algorithm would modify a node's children pointers, an applicative treap instead makes a modified clone of the node with the new children pointers. The parent node will also be cloned, with the clone pointing at the new child. This propagates up to the root, creating a new root. Each update to the treap will create $O(1)$ new nodes and $O(\log n)$ cloned nodes. Storage per update is $O(\log n)$ when a snapshot is taken after every update.

Versioned nodes are Sarnak and Tarjan's final technique for implementing partially persistent search trees and can represent the logical forest with $O(1)$ storage per update. We will first explain how versioned node trees work and then, in Sect. 3.4, we will show how to build these techniques into treaps with Merkle hashes.

Rather than allocating new nodes, as with path copying, versioned nodes may contain pointers to older children as well as the current children. While we could have an infinite set of old children pointers, versioned nodes only track two sets of children (*archived* and *current*) and a *timestamp* T. The archived pointers archive one prior version, with T used to indicate the snapshot time at which the update occurred so that LookupV's know whether to use the archived or current children pointers. A versioned node cannot have its children updated twice. If a node x's children need to be updated a second time, it will be cloned, as in path copying. The clone's children will be set to the new children. x's parent must also be updated to point to the new clone, which may recursively cause it to be cloned as well if its archived pointers were already in use. In Fig. 2 we present an example of a versioned node tree.

Each update to a treap requires an expected $O(1)$ rotations, each of which requires updating the children of 2 versioned nodes, requiring a total of $O(1)$ storage per update. To support multiple updates within a single snapshot, we include a last-modified version number in each versioned node. If the children pointers of a node are updated several times within the same snapshot, we may update them in place. As with path copying trees, saving a copy of the root node in the snapshot array is sufficient to find the data for subsequent queries.

3.4 Making treaps persistent and authenticated

A persistent treap is just a forest of individual treaps, one for each snapshot, each of which is an independent authenticated dictionary with the proscribed structure of a treap. As each snapshot is an ordinary search tree, tree-based PADs naturally extend to support queries of a given value's successor, predecessor, and so forth. The choice of how we represent the logical forest of treaps, described in Sect. 3.3 is completely invisible to clients and has no effect on the algorithms to generate membership proofs in historical snapshots or on the root authenticator for a snapshot. However, different representations do have different performance and storage cost tradeoffs.

In order to generate membership proofs in a snapshot, the server has to be able to generate subtree authenticators. If *copy everything* is used to represent the forest of treaps, membership proofs can be computed in $O(\log n)$ time. Each node occurs in exactly one snapshot and each node can cache its subtree authenticator. When *path copying* is used to represent the forest of treaps, each node is immutable once created. The subtree rooted at that node is fixed and the subtree authenticator is constant and can be cached directly on that node. Membership proofs can be computed in $O(\log n)$ time and updates cost $O(\log n)$ storage. PADs based on path-copying red-black trees were proposed by Anagnostopoulos et al. [1].

Caching subtree authenticators in Sarnak-Tarjan versioned nodes adds extra complexity. Unlike before, the descendants of a node are no longer immutable and the subtree authenticator of a node is no longer constant for all snapshots in which it occurs. For example, in Fig. 2, the node containing R in the version 1 and 2 trees has different authenticators in snapshots 1 and 2. In this section, we present novel techniques for building authenticated data-structures out of persistent data structures based on versioned nodes by controlling when and how subtree authenticators are recomputed or cached. In these designs, each update costs $O(1)$ storage to create new versioned nodes plus whatever overhead is used for caching subtree authenticators.

In our designs, we store subtree authenticators for the current snapshot, mutating it in place on each update to the treap. This *ephemeral subtree authenticator* can be used to generate membership proofs for the current snapshot in $O(\log n)$ time. For historical snapshots, however, it cannot be used.

For historical snapshots, a simple solution is to not cache any subtree authenticators at all. In this *cache nothing* case, the server can calculate the subtree authenticator for a node on-the-fly from its descendants and generate a membership proofs in $O(n)$ time. Obviously, we want to generate proofs faster than that. By spending additional space to cache the changing subtree authenticators, we can reduce the cost of generating membership proofs.

Each versioned node can cache the changing authenticator for every version in a *versioned reference* which can be stored as an append-only resizable vector of pairs containing version number transition points v_i and values r_i, $((v_1, r_1), (v_2, r_2), \ldots (v_k, r_k)))$. The reference is undefined for $v < v_1$. The reference is r_1 for $v_1 \leq v < v_2$, r_2 for $v_2 \leq v < v_3$, and so forth. The reference is r_k for versions $\geq v_k$. $r_i = \square$ means that the cache is invalid and the subtree authenticator must be recomputed by visiting the node's children. Lookups by version number use binary search over the vector in $O(\log k)$ time.

Note that in this cache design, the most recently cached subtree authenticator remains valid forever. If a cached subtree authenticator is about to becomes stale, the authenticator cache must be either updated with the new subtree authenticator, or explicitly invalidated for the next snapshot. Note that if the authenticator cache is invalidated for the next snapshot, it remains valid for prior snapshots. Similar updates will also be necessary for the authenticator caches in the modified node's ancestors.

Our first caching option, *cache everything*, ensures that the authenticator cache always hits. On each update to the treap, we update the cache for each node in the path to the root. This means that we lose the $O(1)$ benefit of using versioned nodes, because we must pay a $O(\log n)$ cost to maintain the cached authenticators. Generating a membership proof will cost $O(\log v \cdot \log n)$ time for $O(\log n)$ binary searches in the subtree authenticator cache. In the example presented in Fig. 2, the nodes containing R in the version 1 and 3 trees have 2 and 1 cached authenticators respectively. The node containing S has 1 cached subtree authenticator.

Although PADs implemented by versioned nodes implemented using the cache-everything strategy have the same big-O space usage as PADs implemented by trees that use path copying, the constant factors are smaller. Appending another hash and timestamp threshold to $O(\log n)$ versioned references implemented by resizable arrays is much more concise than cloning $O(\log n)$ nodes.

We are not required to cache every subtree authenticator. Authenticators may be recomputed as needed, offering a diverse set of choices for caching strategies and time-space tradeoffs. Caching strategies may be generic, or exploit spacial or temporal locality, as long as a cached authenticator is updated or invalidated in any snapshot where a descendant changes. Caching strategies may also purge authenticators at any time to save space. Although many application-specific strategies are possible, we will only present one generic caching strategy with provable bounds.

Our *median layer cache* offers $O(1)$ storage per update while generating membership proofs in historical snapshots in $O(\sqrt{n}\log v)$ time by permanently caching subtree authenticators on exactly those nodes at depth D chosen to be close to the median layer $\frac{\log_2 n}{2}$ in the tree. As nodes enter or leave the median layer, or the median layer itself changes, we maintain the invariant that for each snapshot, the versioned nodes in the median layer for that particular snapshot have cached authenticators.

When an update occurs, in the typical case where only leaves' values change, we update the subtree authenticator cache in the ancestor median layer node. In addition, all other ancestors of the changed node potentially have stale authenticators, forcing us to explicitly invalidate their caches for the upcoming snapshot. In the atypical case, many nodes may enter or leave the median layer at a time, due to changes of the number of keys in the tree or rotations among the first D layers of the tree. However, only $O(1)$ expected additional storage per-update is required to account for these effects.

Computing membership proofs for the median layer treap can be done in $O(\sqrt{n}\log v)$ time. Generating a membership proof requires calculating $O(\log n)$ subtree authenticators at depths $d = 1$, $d = 2$, and so forth. (Recall that $D = \log_2 \sqrt{n}$.) There are three cases for computing any one single subtree authenticator. The subtree authenticator for a node at depth $d = D$ is cached and can used directly.

Computing a subtree authenticator for a node x at depth $d < D$ (i.e., x is higher than the median layer, closer to the root), requires recursing down until hitting nodes at the median layer, then using the cached authenticators. This recursion will visit at most $2^{D-d} = O\left(\frac{\sqrt{n}}{2^d}\right)$ nodes. Computing a subtree authenticator for a node x at depth $d > D$ (i.e., x is below the median layer, closer to the leaves) requires visiting every descendant of x. In expectation, a node at depth $d > D$ has $O\left(\frac{n}{2^d}\right) = O\left(\frac{\sqrt{n}}{2^{d-D}}\right)$ descendants.

4 Tuple-based PADs

Previously, we described how to design PADs based on Merkle trees. In this section, we develop a novel alternative foundation. These designs are super-efficient, yielding constant-sized query response proofs instead of the $O(\log n)$ proofs from tree-based PADs. In addition, these PADs offer different features, functionality, and efficiency choices.

This class of techniques uses a *tuple representation* of a dictionary. If a dictionary has keys $k_1 \ldots k_n$, with $k_i < k_{i+1}$ and corresponding values $c_1 \ldots c_n$, we subdivide the entire key-ID space into disjoint intervals $[k_0, k_1), [k_1, k_2)$, and so forth. Each interval $[k_j, k_{j+1})$ contains a single dictionary key at k_j with value c_j and indicates that there is no other key elsewhere in the interval. Let this be represented as the tuple $([k_j, k_{j+1}), c_j)$, which we can formally read as: "Key k_j has value c_j, and there are no keys in the dictionary in the interval (k_j, k_{j+1})." Keys could be integers, strings, hash values, or any type that admits a total ordering. In order to cover the key-ID space before the first key k_1 and after the last key k_n in the dictionary, we include two sentinels, $([k_{\min}, k_1), \square)$ and $([k_n, k_{\max}), c_n)$ where $k_{\min}$ and $k_{\max}$ denote the lowest and highest key-IDs respectively.

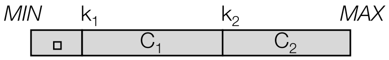

Fig. 3. We graphically show 2 keys and 3 tuples. Tuple $([k_j, k_{j+1}), c_j)$ is represented as a rectangle from k_j to k_{j+1} containing c_j.

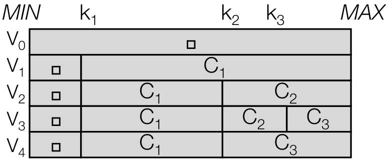

Fig. 4. Example of a Tuple PAD containing 5 snapshots. From top to bottom, starting with an empty PAD, inserting k_1, c_1, inserting k_2, c_2, inserting k_3, c_3, and removing k_2. Each rectangle corresponds to a signed tuple.

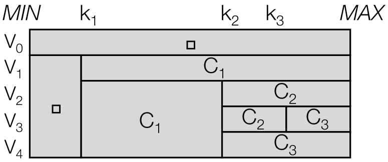

Fig. 5. Example of tuple-superseding representation of Fig. 4, showing the space savings when tuples can span many version numbers. As before, each rectangle corresponds to a signed tuple.

An alternative would use a circular key-ID space rather than the sentinels. Figure 3 illustrates the tuples composing a dictionary.

If each tuple is individually signed by an author to form an authenticated dictionary, then the server can prove the presence or absence of a key k_q from the authenticated dictionary by returning the one signed tuple $T = ([k_j, k_{j+1}), c_j)$ that *matches* k_q by being responsible for the section of the key-space containing k_q, or, more formally, having $k_q \in [k_j, k_{j+1})$. The key k_q is in the dictionary with value c_j if $k_q = k_j$ and $c_j \neq \square$ ($\square$ denotes no key). If $k_q \neq k_j$, the client may conclude k_q is absent from the dictionary. This representation offers super-efficient, $O(1)$, membership proofs for its authenticated dictionary. This representation also offers super-efficient proofs of non-membership.

Now that we have explained the tuple representation of a single authenticated dictionary, the challenges are how to add persistence, how to efficiently store the tuples and their signatures, how to reduce the number of tuples that need to be signed, and finally how to authenticate tuples without individually signing each one.

4.1 PADs based on individually signed tuples

In a solitary PAD, each tuple is individually signed by the author. The author signs $n + 1$ tuples for each snapshot. To support persistency, tuples include a version number and have the form: $(v_\alpha, [k_j, k_{j+1}), c_j)$, which can be read as "In version v_α, key k_j has contents c_j, and there is no key in the dictionary with a key between k_j and k_{j+1}." Figure 4 graphically shows such a PAD. The server can prove the membership or non-membership of any key k_q in snapshot v_q in the PAD by returning one signed tuple $T = (v_q, [k_j, k_{j+1}), c_j)$ that matches the lookup request by having $k_q \in [k_j, k_{j+1})$. This design is super-efficient, persistent and history independent, but does not have a root authenticator or support Merkle aggregation.

Updates are clearly expensive. The author must sign each tuple individually on each snapshot and send the signatures to the server, which must then store them. The per-snapshot computation, storage, and communications costs are $O(n)$.

Optimizing storage by coalescing tuples. We can reduce the tuple storage costs by exploiting redundancy between snapshots. If we assume that a snapshot is generated after every update, all but at most two of the signed tuples in snapshot v_α will have the same keys and values in snapshot $v_\alpha+1$. This is because an insert into the dictionary will split the range of the prior tuple into two ranges. Removing a key will require deleting a tuple and replacing its predecessor tuple with a new one with an expanded range.

Most tuples may remain unchanged across many snapshots. Instead of storing each of the tuples, $(v_\alpha, [k_j, k_{j+1}), c_j)$, $(v_\alpha + 1, [k_j, k_{j+1}), c_j)$, ... $(v_\alpha + \delta, [k_j, k_{j+1}), c_j)$, and signatures on each of these tuples, the server may store one *coalesced tuple* $([v_\alpha, v_\alpha + \delta], [k_j, k_{j+1}), c_j, \mathit{SIGS})$ that encodes that the key space from k_j to k_{j+1} did not change from snapshot v_α to $v_\alpha + \delta$. In each coalesced tuple, *SIGS* stores the $\delta + 1$ signatures signing each individual snapshot's tuple. The coalesced tuple, itself, is never signed.

Upon a lookup query for k_q at time v_q, the server find the tuple $T = ([v_\alpha, v_\alpha + \delta], [k_j, k_{j+1}), c_j, \mathit{SIGS})$ that matches k_q and v_q by having $v_q \in [v_\alpha, v_\alpha + \delta]$ and $k_q \in [k_j, k_{j+1})$, from which it regenerates the tuple $(v_q, [k_j, k_{j+1}), c_j)$, which the author signed earlier.

Storing tuples with a persistent search tree. Our next challenge is how to store coalesced tuples and signatures so that they may be easily found during lookups. We need a data structure that can store the varying set of coalesced tuples representing each snapshot, and for any given snapshot version, we need to be able to find the tuple containing a search key. This can be easily done with a persistent search tree that supports predecessor queries, such as the $O(1)$ persistent search tree data structure described in Sect. 3.3.

Each snapshot in the PAD has a corresponding snapshot in the persistent search tree *PST* for storing the tuples representing that snapshot. Whenever an update occurs, the author will indicate which tuples are *new* (i.e., their key interval or value was not in the prior snapshot), and which tuples are to be *deleted* (i.e., their key interval or value is not in the new snapshot). The remaining tuples are *refreshed.* At most two tuples will be deleted and one tuple will be new. The author transmits signatures on every new or refreshed tuple.

When a tuple $([v_\alpha, v_\beta], [k_j, k_{j+1}), c_j, \mathit{SIGS})$ is to be deleted from snapshot $v_\beta + 1$, the server removes that tuple from the next snapshot of *PST*. When a tuple is to be added to snapshot $v_\beta + 1$, the server inserts $([v_\beta + 1, v_\beta + 1], [k_j, k_{j+1}), c_j, \mathit{SIG})$ into *PST*. If a tuple $T = ([v_\alpha, v_\beta], [k_j, k_{j+1}), c_j)$ is refreshed, the server appends the author's signature to T and updates the ending snapshot version to $v_\beta + 1$.

This data-structure requires $O(1)$ storage per update for managing the coalesced tuples representing the PAD and can find the matching coalesced tuple and signature for any key in any snapshot in logarithmic time. Unfortunately, the additional costs of $O(n)$ signatures for every snapshot must also be included in the communication and storage costs. Reducing these costs is the challenge in building tuple-based PADs.

4.2 Optimizing storage: Tuple superseding

We now show how to reduce storage costs on the server from $O(n)$ to $O(1)$ signatures per snapshot. Previously, authors signed tuples of the form $(v_\alpha, [k_j, k_{j+1}), c_j)$ for each snapshot. With tuple superseding, the author signs a coalesced tuple of the form $([v_\alpha, v_\beta], [k_j, k_{j+1}), c_j)$ attesting that for all snapshots in $[v_\alpha, v_\beta]$, key k_j has value c_j and there is no key in the interval (k_j, k_{j+1}). Figure 5 shows the benefits of tuple superseding, when a signature can span many version numbers. Clients authenticating a response to a query k_q in snapshot v_q will receive a tuple of the form $([v_\alpha, v_\beta], [k_j, k_{j+1}), c_j)$. They will verify that its signature is valid and that $k_q \in [k_j, k_{j+1})$ and $v_q \in [v_\alpha, v_\beta]$.

For tuples that are refreshed, the server will receive a tuple $([v_\alpha, v_\beta + 1], [k_j, k_{j+1}), c_j)$, signed by the author. This newly signed tuple supersedes the signed tuple $([v_\alpha, v_\beta], [k_j, k_{j+1}), c_j)$ already possessed by the server and can transparently replace it. Although the author must sign $O(n)$ tuples and send them to the server for each snapshot, all but $O(1)$ of them refresh existing tuples. Only the $O(1)$ new tuples and their signatures add to storage on the server. When tuple superseding is used, the PAD is no longer history independent because the signed tuples describe keys in earlier snapshots.

Iterated hash functions. Public key signatures are notably slow to generate and verify. In contrast, cryptographic hash functions are very fast. With a light-weight signature [27] implemented by iterated hash functions, we can indicate that a tuple is refreshed. Rather than signing each superseded tuple, the author now only signs the tuple: $(v_\alpha, H^m(R), [k_j, k_{j+1}), c_j)$ where $H^m(R)$ represent the result of iterating a hash function m times on a random nonce R. The author can indicate that a tuple is refreshed in successive snapshots by releasing successive preimages of $H^m(R)$ which it can incrementally generate in $O(1)$ time and $O(\log m)$ space. A client will need to verify at most m hashes, which will still be significantly cheaper than the cost of verifying the digital signature for reasonable values of m.

4.3 Optimizing signatures via speculation

We now show how a novel application of *speculation* in authenticated data structures can signifigantly reduce the number of signatures. In our original design, the author was required to sign every tuple to refresh it for a new snapshot, at a cost proportional to the number of keys in that snapshot. We can improve on this by dividing the PAD P into two generations: a young generation G_0 that contains keys that are recently modified, and an old generation G_1 that contains all other keys. Tuples in the old generation G_1 are speculatively signed with version intervals that stretch into the future, but are only considered when there is a proof that the key is not set in the younger generation. (Sect. 4.1 noted that it's trivial to prove the absence of a key by returning the signed tuple for the interval containing that key.) Effectively, G_0 contains "patch" tuples that can correct erroneous speculations in G_1. Tuples now include generation markers, g_0 or g_1, to indicate which generation they're in. In Fig. 6 we present such a speculative PAD with an epoch of 3 snapshots.

A snapshot of G_0 must be taken every time a snapshot is taken of P, which requires signing every new or refreshed tuple in G_0. To reduce these costs, we keep the size of G_0 small by dividing time into *epochs*. Every E_1 times a snapshot is taken of P, we migrate all of the entries from G_0 into G_1, take a snapshot of G_1, and erase G_0. With a snapshot taken after every update, this ensures that G_0 contains at most $E_1 + 1$ tuples.

When an insert into P is requested, the author inserts the tuple representing the key and value into G_0. When a removal of k_j from P is requested, G_0 is updated to store the tuple $(g_0, [v_\beta, v_\beta], [k_j, k_{j+1}), \square)$, indicating that key k_j is not in the PAD in version v_β.

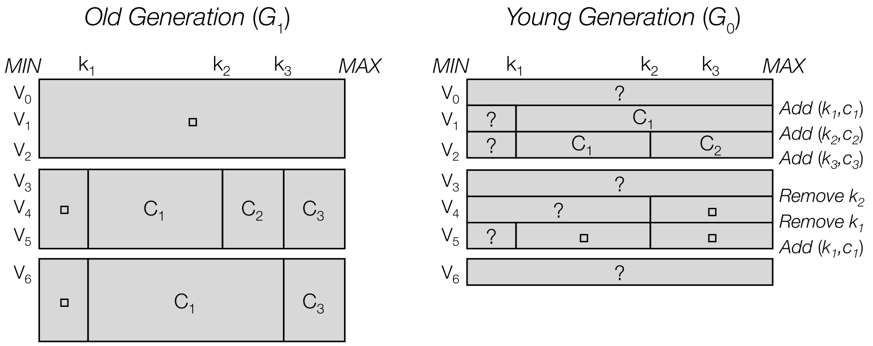

Fig. 6. Example of a PAD using speculation with an epoch of 3 snapshots. Lookups examine the young generation first. Because we did not use a circular ID-space the sentinal tuple in the young generation uses a key of ? to indicate that the older generation must be examined for $k_q = k_{MIN}$.

Tuples in G_0 have the form $(g_0, [v_\beta, v_\beta], [k_j, k_{j+1}), \square)$, indicating the one version that they are valid for, while tuples in G_1 have the form, $(g_1, [v_\gamma, v_\gamma + E_1 - 1], [k_j, k'_{j+1}), c'_j)$, indicating that they are valid for the duration of an epoch. At the start of every epoch, the author enumerates every key-value pair in the current snapshot in G_0, and inserts them into G_1. During this process, the author may find opportunities to merge tuples representing deleted keys. If a tuple $(g_0, [v_\beta - 1, v_\beta - 1], [k_j, k_{j+1}), \square)$ representing a removed key is migrated, it may force the deletion of a tuple, $(g_1, [v_\beta - E_1, v_\beta - 1], [k_j, k'_{j+1}), c'_j)$, in G_1 from the next epoch. After migrating keys into G_1, the author speculatively signs each tuple in G_1 as valid for the entire duration of the future epoch.

On a lookup of key k_q in snapshot v_q, the server returns two signed tuples: $(g_0, v_\beta, [k_j, k_{j+1}), c_j)$ with $v_q = v_\beta$ and $k_q \in [k_j, k_{j+1})$ and $(g_1, [v_\gamma, v_\gamma + E_1 - 1], [k'_j, k'_{j+1}), c'_j)$ with $v_q \in [v_\gamma, v_\gamma + E_1 - 1]$ and $k_q \in [k'_j, k'_{j+1})$. There are two cases. If $k_q = k_j$, then the key is in G_0 with value c_j, with $c_j = \square$ denoting a deleted key. Otherwise, if $k_q \in (k_j, k_{j+1})$, we must examine G_1. If $k_q = k'_j$, then the key is in G_1 with value c'_j. Otherwise, if $k_q \in (k'_j, k'_{j+1})$ then the lookup key is not in the snapshot.

Speculation can reduce the number of signatures required by the author from $O(n)$ to $O(\sqrt{n})$ for each update if a snapshot is taken after every update. The author must sign $E_1 + 1$ tuples in G_0 each time P has a snapshot taken, and, once every E_1 snapshots, the author must sign all $n + 1$ tuples in G_1. The amortized number of signatures per update is $O(E_1 + n/E_1)$, with a minimum when $E_1 = \sqrt{n}$. If DSA signatures are used, latency can be reduced at the start of an epoch by partially precomputing signatures [30]. This creates a super-efficient, history-independent PAD with $O(\sqrt{n})$ signatures and $O(\sqrt{n})$ storage per update. Note that speculation makes a PAD no longer history independent because the tuples in G_1 describe keys contained in the PAD at the start of the epoch.

More than two generations. Speculative PADs can be extended to more than two generations. As before, generation G_0 is definitive, and later generations are progressively more speculative. Membership proofs will include one tuple per generation.

In the case of 3 generations, we have epochs every E_1 snapshots, when keys are migrated from G_0 to G_1, and every E_2 snapshots, when keys are migrated from G_1 to G_2. If we assume a snapshot after every update, the author must sign an amortized $O\left(\frac{n}{E2} + \frac{E2}{E_1} + E_1\right)$ tuples per update. This is minimized to $O(\sqrt[3]{n})$ when $E_2 = n^{\frac{2}{3}}$ and $E_1 = n^{\frac{1}{3}}$. More generally, if there are C generations, lookup proofs contain C signatures, the author must sign a $O(C\sqrt[C]{n})$ tuples, and the storage per update is $O(C\sqrt[C]{n})$ if tuple superseding is not used.

Speculation and tuple superseding. Speculation reduces the total number of signatures by the author and thus reduces the space required on the server to store them. It can be naturally combined with tuple-superseding (with our without using iterated hashes) to reduce the number of tuples the server must save to $O(C)$ per update.

4.4 Tuple PADs based on RSA accumulators

RSA accumulators [5] are a useful way to authenticate a set with a concise $O(1)$ summary, which can be signed using digital signatures. Dynamic accumulators [10, 18, 34] permit efficient incremental update of accumulator without requiring that it be regenerated. Membership of an element in the set is proved with *witnesses*, which may be computed by the untrusted server. Recent developments include an accumulator supporting efficient non-membership proofs [25] or batch update of witnesses [39, 40]. By storing tuples in a signed accumulator, the update size for a snapshot can be reduced to $O(1)$ while supporting a root authenticator. We leave the complete design and evaluation of such PADs to future work.

5 Evaluation

In this paper we have presented a variety of algorithms for implementing a PAD. In Table 1 we compare our designs to the existing related work and present a comparison of the space usage and amortized expected running time of each algorithm in terms of the number of keys n and number of snapshots v. We assume that a snapshot is taken after every update. For tree-based PADs, query times include the $O(\log v)$ cost to binary search in the authenticator cache. For tuple-based PADs, query times include searching the persistent tree for the tuple. We also note which designs support a root authenticator, Merkle aggregation, and are canonical or history independent.

A modular exponentation, used in signatures, is much more expensive than many cryptographic hashes. A standard big-O bound would not capture these effects. To enable a more accurate comparison, we account for exponentiations used in verifying signatures by using β to denote its cost. Table 1 then describes:

1. **Server storage (per-update).** Storage, per update, on the server.
2. **Membership proof size.** Size of a membership proof sent to a client.
3. **Query time (historical).** Time to make a membership proof for old snapshots.
4. **Query time (current).** Time to make a membership proof for the current snapshot.
5. **Verify time.** Time to verify a membership proof by a client.
6. **Update info.** The size of an update, sent to the server.
7. **Author update time.** Time on the author required to generate an update.
8. **Server update time.** Time on the server required to process an update.

Reference	Storage Size	Query Time (historical)	Query Time (current)	Proof Size	Verify Time	Update Time (author)	Update Time (server)	Update Size	Notes
Path Copy Skiplist [1]	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$\beta + O(\log n)$	$\beta + O(\log n)$	$O(\log n)$	$O(\log n)$	Root.
Path Copy Red-black [1]	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$\beta + O(\log n)$	$\beta + O(\log n)$	$O(\log n)$	$O(\log n)$	Root.
Treap (Path Copy)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$\beta + O(\log n)$	$\beta + O(\log n)$	$O(\log n)$	$O(\log n)$	Canonical. Root. Aggregates.
Treap (Versioned Node) (No Cache)	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$	$\beta + O(\log n)$	$\beta + O(\log n)$	$O(\log n)$	$O(\log n)$	Canonical. Root. Aggregates.
Treap (Versioned Node) (Cache Everywhere)	$O(\log n)$	$O(\log v \cdot \log n)$	$O(\log n)$	$O(\log n)$	$\beta + O(\log n)$	$\beta + O(\log n)$	$O(\log n)$	$O(\log n)$	Canonical. Root. Aggregates.
Treap (Versioned Node) (Median Cache)	$O(1)$	$O(\sqrt{n} \log v)$	$O(\log n)$	$O(\log n)$	$\beta + O(\log n)$	$\beta + O(\log n)$	$O(\log n)$	$O(\log n)$	Canonical. Root. Aggregates.
Solitary Tuple	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$	$\beta + O(1)$	$O(\beta n)$	$O(n)$	$O(n)$	
Solitary Tuple (Speculating)	$O(C \sqrt[C]{n})$	$O(C \log n)$	$O(C \log n)$	$O(C)$	βC	$O(\beta C \cdot \sqrt[C]{n})$	$O(C \sqrt[C]{n})$	$O(C \sqrt[C]{n})$	
Solitary Tuple (Speculating) (+Superseding)	$O(C)$	$O(C \log n)$	$O(C \log n)$	$O(C)$	βC	$O(\beta C \cdot \sqrt[C]{n})$	$O(C \sqrt[C]{n})$	$O(C \sqrt[C]{n})$	
Solitary Tuple (Speculating) (+Superseding+IterHash)	$O(C)$	$O(C \log n)$	$O(C \log n)$	$O(C)$	$(\beta + D)C$	$O(C \sqrt[C]{n}(\frac{\beta}{D} + D))$	$O(C \sqrt[C]{n})$	$O(C \sqrt[C]{n})$	

Table 1. Persistent authenticated dictionaries, comparing techniques assuming a snapshot is taken after every update. Storage sizes are measured per-update. β denotes the cost of an exponentiation used during signature generation. C denotes the number of generations in a speculative PAD and D denotes the maximum hash-chain length. In this table, we report the amortized expected time or space usage. “Canonical” refers to designs that are history-independent.

6 Future work and conclusions

PADs are suitable for a variety of problems, such as in a public key infrastructure where they can efficiently store a constantly-changing set of valid certificates. If a PAD supporting a root authenticator is used, the root authenticator may be stored in a tamper-evident log [11, 12, 37]; the author cannot later modify it without detection. Similarly, the root authenticator could be submitted to a time-stamping service [21, 9] every time a snapshot is taken to prove its existence. PADs can be used to implement many forms of outsourced databases. Using Merkle aggregation, PADs can be used to implement flexible query languages, or in the case of Pari-mutuel gambling, as used in horse racing, to count wagers. With a canonical or history independent representation, PADs can make distributed algorithms more robust.

In this work we developed several new ways of implementing PADs. We presented designs offering constant-sized proofs and lower storage overheads. We also developed speculation as a new technique for designing authenticated data structures. In future work, we will perform an empirical evaluation of each of our algorithms and of their respective costs for each operation in order to guide which algorithm is right for which situation. We will also compare our designs to alternative PAD algorithms [1] and evaluate PADs based on RSA accumulators and other cryptographic techniques.

There are a number of properties we would like to formally prove, including big-O bounds on the storage costs and tighter bounds on lookup time, as well as proving for various threat models that our PAD designs always detect failure or return the correct answer. We leave this to future work.

Future work also includes creating fully persistent authenticated dictionaries based on fully persistent data structures [13] as well as extending our designs to support outsourced storage where a trusted device uses a small amount of trusted storage to detect faults in a larger untrusted storage [7, 14].

If persistence is unnecessary, but authentication is, our techniques should be easily simplified to only preserve the data necessary to authenticate the latest snapshot. We plan to adapting speculation and lightweight signatures to create a dynamic super-efficient authenticated dictionary.

References

1. Anagnostopoulos, A., Goodrich, M. T., and Tamassia, R. Persistent authenticated dictionaries and their applications. In *International Conference on Information Security (ISC)* (Seoul, Korea, Dec. 2001), pp. 379–393.
2. Anderson, A., and Ottmann, T. Faster uniquely represented dictionaries. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science (SFCS)* (San Juan, Puerto Rico, Oct. 1991), pp. 642–649.
3. Aragon, C. R., and Seidel, R. G. Randomized search trees. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (SFCS)* (Oct. 1989), pp. 540–545.
4. Bagwell, P. Fast functional lists, hash-lists, deques and variable length arrays. In *In Implementation of Functional Languages, 14th International Workshop* (Madrid, Spain, Sept. 2002), p. 34.
5. Benaloh, J., and de Mare, M. One-way accumulators: a decentralized alternative to digital signatures. In *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology (EuroCrypt '93)* (Lofthus, Norway, May 1993), pp. 274–285.

6. Blelloch, G. E., and Reid-Miller, M. Fast set operations using treaps. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (Puerto Vallarta, Mexico, June 1998), pp. 16–26.
7. Blum, M., Evans, W., Gemmell, P., Kannan, S., and Naor, M. Checking the correctness of memories. In *Proceedings of the 32nd annual symposium on Foundations of computer science (SFCS)* (San Juan, Puerto Rico, Oct. 1991), pp. 90–99.
8. Brodal, G. S. Partially persistent data structures of bounded degree with constant update time. *Nordic Journal of Computing 3*, 3 (1996), 238–255.
9. Buldas, A., Lipmaa, H., and Schoenmakers, B. Optimally efficient accountable time-stamping. In *International Workshop on Practice and Theory in Public Key Cryptography (PKC)* (Melbourne, Victoria, Australia, Jan. 2000), pp. 293–305.
10. Camenisch, J., and Lysyanskaya, A. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO '02* (Santa Barbara, CA, Aug. 2002), pp. 61–76.
11. Crosby, S. A., and Wallach, D. S. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th USENIX Security Symposium* (Montreal, Canada, Aug. 2009). Preprint available at http://www.cs.rice.edu/~scrosby/pubs/preprints/paper-treehist.pdf.
12. Davis, D., Monrose, F., and Reiter, M. K. Time-scoped searching of encrypted audit logs. In *Information and Communications Security Conference* (Malaga, Spain, Oct. 2004), pp. 532–545.
13. Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. E. Making data structures persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing (STOC)* (Berkeley, CA, May 1986), pp. 109–121.
14. Dwork, C., Naor, M., Rothblum, G. N., and Vaikuntanathan, V. How efficient can memory checking be?. In *Proceedings of the Theory of Cryptography Conference (TCC)* (San Francisco, CA, Mar. 2009), pp. 503–520.
15. Fiat, A., and Kaplan, H. Making data structures confluently persistent. *Journal of Algorithms 48*, 1 (2003), 16–58.
16. Fu, K., Kaashoek, M. F., and Mazières, D. Fast and secure distributed read-only file system. *ACM Transactions on Compututer Systems 20*, 1 (2002), 1–24.
17. Gassend, B., Suh, G., Clarke, D., Dijk, M., and Devadas, S. Caches and hash trees for efficient memory integrity verification. In *The 9th International Symposium on High Performance Computer Architecture (HPCA)* (Anaheim, CA, Feb. 2003).
18. Goodrich, M. T., Tamassia, R., and Hasic, J. An efficient dynamic and distributed cryptographic accumulator. In *Proceedings of the 5th International Conference on Information Security (ISC)* (Sao Paulo, Brazil, Sept. 2002), pp. 372–388.
19. Goodrich, M. T., Tamassia, R., Triandopoulos, N., and Cohen, R. F. Authenticated data structures for graph and geometric searching. In *Topics in Cryptology, The Cryptographers' Track at the RSA Conference (CT-RSA)* (San Francisco, CA, Apr. 2003), pp. 295–313.
20. Guibas, L. J., and Sedgewick, R. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (SFCS)* (Oct. 1978), pp. 8–21.
21. Haber, S., and Stornetta, W. S. How to time-stamp a digital document. In *CRYPTO '98* (Santa Barbara, CA, 1990), pp. 437–455.
22. Kaplan, H. Persistent data structures. In *Handbook on Data Structures and Applications* (2001), D. Mehta and S. Sahni, Eds., CRC Press.
23. Kocher, P. C. On certificate revocation and validation. In *International Conference on Financial Cryptography (FC '98)* (Anguilla, British West Indies, Feb. 1998), pp. 172–177.
24. Li, J., Krohn, M., Mazières, D., and Shasha, D. Secure untrusted data repository (SUNDR). In *Operating Systems Design & Implementation (OSDI)* (San Francisco, CA, Dec. 2004).

25. Li, J., Li, N., and Xue, R. Universal accumulators with efficient nonmembership proofs. In *Proceedings of the 5th International Conference on Applied Cryptography and Network Security (ACNS)* (Zhuhai, China, June 2007), pp. 253–269.
26. Merkle, R. C. A digital signature based on a conventional encryption function. In *CRYPTO '88* (1988), pp. 369–378.
27. Micali, S. Efficient certificate revocation. Tech. Rep. TM-542b, Massachusetts Institute of Technology, Cambridge, MA, 1996. http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail\&id=oai%3Ancstrlh%3Amitai%3AMIT-LCS%2F%2FMIT%2FLCS%2FTM-542b.
28. Micciancio, D. Oblivious data structures: Applications to cryptography. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)* (El Paso, Texas, May 1997), pp. 456–464.
29. Muthitacharoen, A., Morris, R., Gil, T., and Chen, B. Ivy: A read/write peer-to-peer file system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)* (Boston, MA, Dec. 2002).
30. Naccache, D., M'Raihi, D., Vaudenay, S., and Raphaeli, D. Can DSA be improved? Complexity trade-offs with the digital signature standard. In *EuroCrypt* (Perugia, Italy, May 1994), pp. 77 – 85.
31. Naor, M., and Nissim, K. Certificate revocation and certificate update. In *USENIX Security Symposium* (San Antonio, TX, Jan. 1998).
32. Naor, M., and Teague, V. Anti-presistence: history independent data structures. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing (STOC)* (Heraklion, Crete, Greece, July 2001), pp. 492–501.
33. Okasaki, C. *Purely Functional Data Structures*. Cambridge University Press, 1999.
34. Papamanthou, C., Tamassia, R., and Triandopoulos, N. Authenticated hash tables. In *ACM Conference on Computer and Communications Security (CCS '08)* (Alexandria, VA, Oct. 2008), pp. 437–448.
35. Peterson, Z. N. J., Burns, R., Ateniese, G., and Bono, S. Design and implementation of verifiable audit trails for a versioning file system. In *USENIX Conference on File and Storage Technologies* (San Jose, CA, Feb. 2007).
36. Sarnak, N., and Tarjan, R. E. Planar point location using persistent search trees. *Communications of the ACM 29*, 7 (1986), 669–679.
37. Schneier, B., and Kelsey, J. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security 1*, 3 (1999).
38. Shapiro, J. S., and Vanderburgh, J. Access and integrity control in a public-access, high-assurance configuration management system. In *USENIX Security Symposium* (San Francisco, CA, Aug. 2002), pp. 109–120.
39. Wang, P., Wang, H., and Pieprzyk, J. A new dynamic accumulator for batch updates. In *Information and Communications Security, 9th International Conference (ICICS 2007)* (Zhengzhou, China, Dec. 2007), pp. 98–112.
40. Wang, P., Wang, H., and Pieprzyk, J. Improvement of a dynamic accumulator at ICICS 07 and its application in multi-user keyword-based retrieval on encrypted data. In *Asia-Pacific Services Computing Conference* (Yilan, Taiwan, Dec. 2008), pp. 1381–1386.
41. Williams, P., Sion, R., and Shasha, D. The blind stone tablet: Outsourcing durability. In *Sixteenth Annual Network and Distributed Systems Security Symposium (NDSS)* (San Diego, CA, Feb. 2009).