Abstract

Authenticated dictionaries are a widely discussed paradigm to enable verifiable integrity for data storage on untrusted servers, such as today's widely used "cloud computing" resources, allowing a server to provide a "proof," typically in the form of a slice through a cryptographic data structure, that the results of any given query are the correct answer, including that the absence of a query result is correct. Persistent authenticated dictionaries (PADs) further allow queries against older versions of the structure. This research presents implementations of a variety of different PAD algorithms, some based on Merkle tree-style data structures and others based on individually signed "tuple" statements (with and without RSA accumulators). We present system throughput benchmarks, presenting costs in terms of time, storage, and bandwidth as well as considering how much money would be required given standard cloud computing costs. We conclude that Merkle tree PADs are preferable in cases with frequent updates, while tuple-based PADs are preferable with higher query rates. For Merkle tree PADs, red-black trees outperform treaps and skiplists. Applying Sarnak-Tarjan's versioned node strategy, with a cache of old hashes at every node, to red-black trees yields the fastest Merkle tree PAD implementation, notably using half the memory of the more commonly used applicative path copying strategy. For tuple PADs, although we designed and implemented an algorithm using RSA accumulators that offers constant update size, constant storage per update, constant proof size, and sublinear computation per update, we found that RSA accumulators are so expensive that they are never worthwhile. We find that other optimizations in the literature for tuple PADs are more cost-effective.

Authenticated dictionaries: Real-world costs and tradeoffs

Scott A Crosby and Dan S. Wallach
Rice University

December 14, 2010

1 Introduction

The recent growth of cloud computing and software-as-a-service offers an attractive option for storing data "in the cloud" rather than locally, for example, replicating data to improve fault tolerance. Of course, the cloud computing provider may well be untrusted. In situations where one author wants to use cloud services to publish data to multiple consumers, or to store data remotely, data integrity is a vital concern. These situations include outsourced databases [37] and untrusted or distributed filesystems [24, 20, 31, 12]. Similar problems of untrusted remote storage occur in commercial remote backup services, p2p systems, smartcard storage [13, 34], and certificate revocation lists [26].

Many of these designs can be implemented using the *dynamic authenticated dictionary* paradigm (*DAD*) [26, 19]. A DAD is a key/value dictionary that permits updates. Updates are signed by trusted *authors*. The dictionary is stored on untrusted *servers* and queried by *clients*. Query responses are authenticated by the author's digital signature.

An authenticated dictionary can be extended to be a *PAD* or *persistent authenticated dictionary* [2, 11], by allowing queries to older versions as well as the current version, such as in revision control systems [36]. Explicit versioning, plus an external channel to alert clients to the latest version ID, are essential to defeating version rollback attacks.

For this research, we implemented 21 different PAD algorithms, including prior designs based on Merkle trees [2] and our prior work with "tuple-based" PADs [11]. RSA accumulators [7, 10] have also been proposed as a primitive for building authenticated dictionaries [30]. In this paper we designed and implemented such a PAD offering constant update size, constant storage per update, constant proof size, and sublinear computation per update, all by using accumulator techniques. For each algorithm we measured the time, space and communication overheads, determining real-world performance that includes the constant factors of digital signature generation, modular exponentiation, primality testing, serialization, and so forth.

In earlier work, we presented a traditional complexity analysis of these algorithms [11]. Because our current work measures real implementations, we can report performance

in terms of milliseconds, bytes, and dollars, leading to some surprising results. For example, in comparing PADs using RSA accumulators with PADs using other cryptographic data structures built from hashes and traditional digital signatures, we concluded that RSA accumulators are *never* the preferable algorithm, despite their superior asymptotic complexity. Results like this would be difficult or impossible to prove absent running code.

Our collection of PAD algorithms make different tradeoffs of CPU, bandwidth, and storage requirements. The ideal algorithm for any given workload will thus depend on the relative costs of these resources. Rather than guess at these tradeoffs, we instead normalize them using contemporary costs, in U.S. Dollars, charged by Google and Amazon for bandwidth, CPU time, and storage on their EC2 and AppEngine services, respectively. If we assume that Google and Amazon are offering these resources at their marginal cost, i.e., that their rates charged for bandwidth, CPU time, and storage are close to the actual costs to any provider delivering large quantities of these resources, then our evaluation strategy should generalize to other vendors as well. Furthermore, our measurements can be easily extrapolated to allow a system designer to consider a variety of "what if" scenarios (e.g., what if crypto accelerators allowed a huge speedup for crypto algorithms) and know which PAD algorithms are likely to be the fastest or cheapest under their system constraints.

In Section 2, we introduce the properties that a persistent authenticated dictionary possesses and we summarize the two classes of PAD algorithms we investigate. In Section 3, we describe PAD algorithms based on search trees. In Section 4, we describe our prior PAD algorithms based on signed tuples and introduce our new RSA accumulator variation. In Section 5, we describe our PAD implementations and evaluation methodology. Section 6 presents benchmark results for our tree PAD implementations. Section 7 presents benchmark results for our tuple-based PAD implementations, including the RSA accumulator variation. Section 8 presents realistic benchmark results against real-world traces. Section 9 discusses issues relating to scaling these systems up to larger compute clusters. Section 10 discusses how to extrapolate our benchmark results to different scenarios. Finally, conclusions and future work are discussed in Section 11.

2 Background

PAD systems divide the world into three roles. Trusted authors *update* the dictionary by inserting or removing key-value pairs. At any time, a *snapshot* of the contents of the dictionary can be taken, resulting in a new *version* of the PAD. The author then sends an *update blob* to the server containing data and authentication information that is stored in a *repository*, used by the server to respond to lookup requests from clients. Clients send lookup requests containing a lookup key to the server and receive a *lookup proof* of the membership of the key and its corresponding value, or non-membership of that key in the dictionary, signed by the author. What makes a PAD "tamper-evident" or "authenticated" is that a malicious server can neither lie to clients about the existence or non-existence of the stored key, nor lie about the value stored for a key without the cooperation of the author (or without breaking the underlying cryptosystem).

In an outsourced storage model, the authors and client may well be the same entity. In any PAD design, we assume a single author who produces digital signatures and possibly multiple clients who can verify them. The server is untrusted and clients have limited state with which to verify the server's output. Authors are assumed to only know about the latest snapshot, while the server is assumed to store all snapshots.

In the next two subsections, we consider two main structures for PADs: those based on Merkle trees and those based on individually signing records.

2.1 Tree-based PADs

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 [21]. If the root hash of such a tree is signed by the author, a server can prove membership of a key in such a tree by showing a path in the search tree to the key. A server can prove non-membership by showing a path to the unique location in the tree where the key would have been stored.

When implementing a PAD, the author only needs to manage one search tree, that of the latest snapshot. On the server, each snapshot is a logically distinct Merkle tree with a different signed root hash. Rather than storing each snapshot as a distinct tree, we can exploit the similarity between trees across snapshots to implement a more space-efficient repository on the server. The design of the repository may affect the performance of the server or its memory usage, but has no effect on the size of an update or lookup proof.

There are several Merkle tree-based approaches for implementing the repository. In Section 3.3, we describe four different implementations and in Section 6.2, we compare their performance. We could use any balanced search tree that supports $O(1)$ expected (not amortized) node mutations per update, such as AVL [1] or red-black trees [17]. The balanced tree algorithm has an effect on the sizes of an updates and lookup proofs. We like treaps [4] for their set-uniqueness properties but we also implement skiplists (see Section 3.1.2) and red-black trees (see Section 3.1.3).

Combining the choice of repository designs and the choice of balanced tree algorithms, we have 12 different PAD implementations that we can compare. In Section 3.1 we describe our implementation of all three algorithms and in Section 6.1 we compare their performances.

2.1.1 Set-unique representations

Treap and skiplist designs are normally probabilistic, in that the ultimate layout of the data structure depends upon random coin flips. We can determinize these data-structures by using a hash function over the key stored in a node. Our deterministic treaps and skiplists become "set-unique," meaning that all authenticated dictionaries with the same contents will have identical tree structures. If we build Merkle trees from these treaps or skiplists, then any two dictionaries with identical contents will have identical root hashes. Set-uniqueness [3, 27] also makes these data structures history independent [23]. The root hash that authenticates such a treap or skiplist will leak no information about the insertion order of the keys or of the past contents. Such

semantics may be valuable, for example, with electronic vote storage or with zero-knowledge proofs.

History-independence is also useful if a 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 non-set-unique data structure, such as red-black tree, 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.

2.2 Tuple-based PADs

Unlike authenticated dictionaries based around search trees, tuple-based authenticated dictionaries and persistent authenticated dictionaries offer *constant* proof size, regardless of the number of keys in the dictionary. At their core, they are based around signed statements of the form:

"Key k_j has value c_j, and there are no keys in the dictionary in the interval (k_j, k_{j+1})."

These statements are represented as *tuples*, $([k_j, k_{j+1}), c_j)$, and can be used to design an authenticated dictionary. A PAD can be designed by extending these tuples to additionally include version numbers. In Section 4 we describe such a PAD, including optimizations to reduce signature overhead by using speculation, and reduce storage on the server by including version number ranges in tuples; we also propose a new optimization where we use RSA accumulators [7] to reduce communication costs. In total, we implemented 9 variations on tuple PADs and in Section 7 we analyze their performance.

3 Tree PADs

In this section we describe tree-based PADs. Every tree-based PAD is a Merkle tree, but we must make two orthogonal design choices: the tree balancing operations, and how the repository is stored. The repository logically consists of a forest of trees, one for each snapshot, but we wish to share storage across trees to save space. Tree-based PADs based around path copying red-black trees and skiplists were originally designed by Anagnostopoulos et al. [2]. Extensions to support Sarnak-Tarjan trees were presented our prior work [11] which we briefly summarize.

In a Merkle tree, each node x is assigned a *subtree authenticator* $x.H$ with the following recurrence: $x.H = H(x.key, H(x.val), x.left.H, x.right.H)$ where H is a cryptographic hash function.

A *lookup proof*, seen in Figure 3 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 trusted root authenticator signed by the author. We can prove that a key is not in the tree by showing a path to the unique leaf location where that key would otherwise be stored.

For a balanced search tree, a lookup proof has size $O(\log n)$, and can be generated in $O(\log n)$ time.

Conventionally, the subtree authenticators for each node in the tree will be precomputed and stored in the node, but we note that these values can be recomputed whenever needed, on the fly, from the keys and values in the subtree, offering a variety of time, space, and design tradeoffs. As such, keeping the hashes around can be thought of as a form of caching.

Without a cache, generating a lookup proof requires $O(n)$ time for recomputing subtree authenticators of elided subtrees.

3.1 Different tree-balancing algorithms

3.1.1 Treap

Treaps [4] are a randomized search tree that can implement a dictionary with a $O(\log n)$ expected cost of an insert, delete, or lookup. Treaps support efficient set union, difference, and intersection operations [8]. 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.[1] *Deterministic treaps* can be created by assigning priorities using a cryptographic digest of the key, creating a set-unique representation [4].

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 length to a key in the treap is $O(\log n)$. The worst case is $O(n)$, but this is unlikely to ever occur.

3.1.2 Skiplists

Papamanthou et. al. [2] described PADs based on path copying red-black trees and skiplists. In this section, we describe skiplists and how they can represent an authenticated dictionary. A skiplist [32] is a data structure offering logarithmic lookups, inserts, and deletes. A classic skiplist is a singly-linked list except that nodes may have several outgoing links, stored in a variable-sized array, which can skip over a large number of list nodes.

[1] 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 that node's key, stored in the left and right subtrees, respectively. By induction, we can assume that the subtrees are also unique.

An alternative formulation of skiplists exists, shown in Figure 1, where each variable-sized array is represented as a "tower" of nodes where each node has only two outgoing links. This forms a representation of a skiplist resembling a set of parallel sorted linked lists. Each key in the skiplist is assigned a maximum level $L_{\max}$ when it is inserted, and it will be placed in the level-$L_{\max}$ linked list and all lower-level linked lists.

Maximum levels are assigned using an exponential distribution. The level-0 list contains every list node. The level-ith list contains one in every 2^i list nodes on average. In this example, keys $\{3, 6, 9, 15\}$ are at level 0, key $\{8\}$ is at level 1 and keys $\{5, 11\}$ are at level 3. If the level of a key is chosen deterministically from the key, the skiplist over a set of keys will be set-unique. Searching a skiplist involves starting in the upper left and "skipping" many nodes by using the higher level links. Skiplists offer an expected $O(\log n)$ update time and lookup time. Just as with a treap, the worst-case lookup and update time is $O(n)$.

Our applicative tree-representation of skiplists is based on the tower-style skiplist authentication trees as used in Goodrich et. al. [14]. We improve on their constructions in several ways, described below.

During lookups, not every edge in a skiplist is used. Extra edges, represented in grey in Figure 1, are only needed for performing updates. We observe that completely omitting the extra edges lets us store a skiplist as if it were an ordinary binary tree; it can then be made persistent using any technique applicable to a binary tree. To this end, we have redesigned our skiplists to not require these extra edges. In Figure 2 we present our final representation of this skiplist.

In addition to a new formulation of skiplists as binary trees, our lookup proofs improve on prior work in authenticated skiplists. Lookup proofs consist of a path from the root to node containing a lookup key. A lookup proof showing non-membership must prove that the interval between the two neighboring keys where the lookup key would otherwise belong does not contain the lookup key.

In the original formulation of authenticated skiplists, non-membership is proved by including the right siblings of each node in the path from the root to the lookup key. For example, to prove that the key 7 is not in the skiplist in Figure 2, the server includes the bold-faced edges along with the $(-\infty, \infty)$ edge at L_3 and the $(5, 11)$ edge at L_2. When proving non-membership of a lookup key that occurs after a level-0 node without a right sibling, the proof of non-membership uses the right successor key stored in that node.

We can improve on this construction. Observe that in a skiplist, the successor of a level-0 node without a right sibling is always the key stored in the right sibling of the first ancestor of that node with a right sibling. If the lookup proof already contains the right sibling of every node in the lookup path, then the successor node is already included in the proof, removing the need for any nodes to explicitly store the keys of their successors. By removing the non-tree-like behavior of storing successor keys, this construction simplifies the design and implementation of update operations.

We can further optimize the proof when the author is trusted to correctly build the skiplist. Instead of including every right sibling in the lookup proof, we only need to include *one* right sibling. If we want to show that a key K is not in the skiplist, we do a search for K. If we find a level L_0 node N with key $k_1 < K$ and a right child containing $k_2 > K$, then by including both N and its right child, we can prove that K is not in the

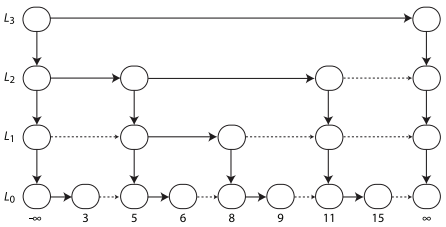

Figure 1: Skiplist representation. Dashed arrows represent redundant edges that are omitted in our implementation.

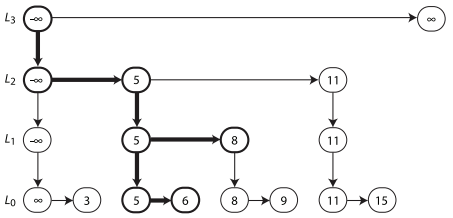

Figure 2: Skiplist query for "7." Highlighted nodes will be included in the result proof to demonstrate that "7" is absent from the result.

skiplist. If N does not have a right child, then the successor key to k_1 is stored in the right sibling of the first ancestor of N that has a right sibling, if that right sibling has key $k_2 > K$, then K is not in the dictionary. In our construction, only this one right sibling in the lookup path needs to be included in the proof whereas before, every right sibling was included. For example, in Figure 2, the level L_0 node 6's first ancestor with a right sibling is the level L_1 node 5, whose right sibling contains an 8. This is 6's successor in the skiplist. The highlighted edges and nodes would suffice to prove that the value 7 is absent from the data structure. This optimization makes our construction of a skiplist lookup proof include approximately half of the number of nodes as prior constructions.

3.1.3 Red-black trees

Authenticated dictionaries can also be built with red-black trees [2], offering $O(1)$ expected node mutations, $O(\log n)$ worst-case update costs, and $O(\log n)$ worst case path length. Red-black trees offer a tighter bound than skiplists or treaps, with a logarithmic worst-case bound, not just a logarithmic expected-case bound[2]. We omit a detailed description of red-black trees here, but we note that red-black trees are not history independent. They should only be used when such semantics are not required.

3.2 Persistent binary search trees

Persistent search tree data structures extend ordinary search tree data structures to support lookups in past snapshots or versions. Persistent data structures have been extensively studied [9, 18], particularly with respect to functional programming [29, 5]. In this section we summarize the algorithms proposed by Sarnak and Tarjan [35], who considered approaches for persistent red-black search trees. Their techniques apply equally well to treaps, red-black trees, or our version of a skiplist, as described above.

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

[2]For simplicity in reporting results in our evaluation, we will gloss over the difference between expected and worst-case bounds.

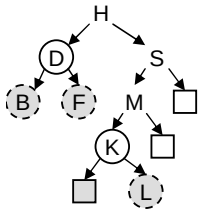

Figure 3: Graphical notation for a lookup proof for M or proving the non-membership of N. Circles denote the roots of elided subtrees whose children, grayed out, need not be included.

snapshot array, indexed by snapshot version. Historical snapshots are frozen and immutable. The most recent, or *current* snapshot can ostensibly be updated in place. Whenever a snapshot is taken, a new root is added to the snapshot array and that snapshot is thereafter immutable.

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

Path copying uses a standard applicative tree, avoiding the redundant storage of subtrees that are identical across snapshots. Nodes in a path-copying tree are immutable. Where the normal, mutating treap, red-black, or skiplist algorithm would modify a node's children pointers, an applicative tree 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. For any of red-black trees, treaps, or skiplists, each update will create $O(1)$ new nodes and $O(\log n)$ cloned nodes in expectation. When a snapshot is taken after every update, skiplists and treaps will use $O(\log n)$ expected storage per update while red-black trees will have a worst-case bound of $O(\log n)$ storage per update.

Versioned nodes are Sarnak and Tarjan's final technique for implementing partially persistent search trees and can represent the logical forest with $O(1)$ expected amortized storage per update. We will first explain how versioned node trees work and then, in Section 3, we will show how to build these techniques into search trees 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 a tree traverser knows 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

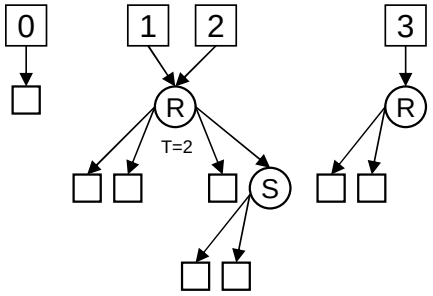

Figure 4: Four snapshots in a Sarnak-Tarjan versioned-node tree, starting with an empty tree, then inserting R, then inserting S, then deleting S. We show the archived children to the left of a node and the current children to the right. Note that R is modified in-place for snapshot 2, but cloned for snapshot 3.

clone, which may recursively cause it to be cloned as well if its archived pointers were already in use. In Figure 4 we present an example of a versioned node tree.

Each update to a treap or red-black tree requires an expected $O(1)$ rotations, each of which requires updating the children of 2 versioned nodes, requiring a total of $O(1)$ expected amortized 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.

3.3 Making trees persistent and authenticated

Although Sarnak-Tarjan trees are a very concise way for a server to store a PAD's snapshots, the server must be able to generate lookup proofs on the demand of clients. Generating responses to lookup requests requires having subtree authenticators for pruned subtrees that are not included in the proof. When using Sarnak-Tarjan versioned-node trees, the subtree authenticator of a node will depend on the snapshot version being used. Consequently, a versioned node cannot simply keep one hash of its children.

Subtree authenticators can always be recomputed from the tree structure by visiting every node in the subtree. This requires no additional storage but this cacheless strategy is inefficient, with $O(n)$ lookup proof generation times. Our prior work presented several caching strategies to either store or dynamically recalculate subtree authenticators [11]. The tradeoffs of different caching strategies are shown in Table 1.

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 subtree authenticator values r_i, $((v_1, r_1), (v_2, r_2), \ldots (v_k, r_k))$. The cache is undefined for $v < v_1$. The cached value is r_1 for $v_1 \leq v < v_2$, r_2 for $v_2 \leq v < v_3$, and so forth. The cached value is r_k for versions $\geq v_k$ through the current version. $r_i = \square$ means that the cache is invalid and the subtree authenticator must be recomputed by visiting the node's children. Lookups by version

Caching strategies	Storage (per update)	Lookup proof (time)
No cache	$O(1)$	$O(n)$
Cache everywhere	$O(\log n)$	$O(\log n)$
Median layer	$O(1)$	$O(\sqrt{n})$

Table 1: Caching strategies for subtree authenticators in a Sarnak-Tarjan tree.

number use binary search over the vector in $O(\log k)$ time.

In the *cache everything* strategy, whenever that node's subtree authenticator changes in a new snapshot and results in logarithmic storage per update and logarithmic time to generate a lookup proof. This offers the same big-O storage as path copying but with lower constant factors because updating a cache is much cheaper than cloning a node.

In the *cache median* strategy, the repository only caches authenticators on the median layer of the tree, i.e., a tree containing n nodes will have depth $\log_2(n)$, so the server caches on all nodes at depth $\frac{\log_2 n}{2}$. Compared to having no cache at all, only constant storage is expected per update, while the time to generate a lookup proof decreases from $O(n)$ to $O(\sqrt{n})$.

3.4 Implementation details

We implemented treaps [4], red-black trees [17], and skiplists [32]. For the server's repository of persistent trees, we implement path copying and the three variations of storing/recomputing subtree authenticators on Sarnak-Tarjan versioned-node trees as discussed above, giving us 12 different tree-based PAD variations to benchmark. We present performance results from native C++ implementations.

Because we are supporting different types of applicative representations, our red-black, skiplist and treap implementations are *only* allowed to "mutate" the children of a node through an abstract interface which, given a node and a pair of new left and right children, returns a node representing the result of applying those changes. The result depends on the underlying repository implementation. With path copying, it will always be a clone. With Sarnak-Tarjan versioned trees, it may or may not be a clone. This requires that the implementations of these algorithms be *bottom-up* and *mutation-free*. In addition, because nodes store keys and values, we must preserve node identity during rotations and other operations, reusing nodes that already store the needed key and value, updating their children through our abstract interface, rather than needlessly cloning those nodes.

4 Tuple-based PADs

In prior work, we presented a new design for implementing PADs with a series of individually signed messages, called *tuple-based PADs* [11]. We briefly introduce our design here, both at its most basic and with all of our optimizations. In Section 4.3, we extend tuple PAD designs to use RSA accumulators, reducing certain operations from

$O(\log n)$ to $O(1)$ time. Our implementation is described in Section 4.4 and benchmarks appear in Sections 7 and 8.

4.1 Basic tuple PADs

Tuple PADs, at their most basic essence, are a list of signed statements made by the author for every version of the dictionary. For each key/value pair (k_i, c_i), at every version number v_n the tuple PAD will contain a signed statement, by the author of the form $(v_n, [k_i, k_{i+1}), c_i)$ which denotes that for the n^{th} version, the key k_i has value c_i and that there is no other key in the dictionary whose value is greater than k_i and less than k_{i+1}. Two additional special-case entries deal with the key-range less than the smallest key and the key-range greater than the largest key. Keys could be integers, strings, hash values, or any type that admits a total ordering.

For a dictionary with m versions, each of which has n key/value pairs, this requires the author to generate $n \times m$ digital signatures, which is clearly quite expensive, but the benefits for the server and client are clear. The client can query the server for a given key k at a particular version v and the server just needs to return the proper signed tuple. If the requested key doesn't exist, the server can return the tuple whose key interval covers the requested key to prove its absence.

Storing tuples with a persistent search tree. Our next challenge is how to store tuples and signatures so that they may be easily found during lookups. We need an auxiliary data structure that can store the varying set of 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 Section 3.2.

Each snapshot in the PAD has a corresponding snapshot in the auxiliary persistent search tree 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.

This data-structure requires $O(1)$ storage per update for managing the tuples representing the PAD and can find the matching 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 Fast tuple PADs

We wished to reduce the author's costs, noting that between any two versions, most of the key/value pairs will not change. In our prior work, we considered a variety of different optimizations and structures which we have now implemented in our benchmarks. We summarize that work here.

Superseding. Let's say the value c_i of a key k_i has remained constant for several versions and the subsequent key k_{i+1} has likewise been unchanged. When the author signs the tuple for the most current version, the author can include a *range* of past version numbers over which the statement is also valid. This allows the server to discard older signatures that refer to the same key, reducing server storage space to $O(1)$ per update. The author must still sign and send $O(n)$ signatures per snapshot to the server.

Our implementation further optimizes this by adopting *lightweight signatures* [22]. Rather than requiring the author to sign the same tuple, again and again, the author first computes an iterated hash $H(H(H(....H(R))))$ for some random number R and signs that along with the initial tuple entry. Subsequent *refreshes* of the tuple need only reveal successive preimages of the hash function, saving the author from the expense of recomputing so many digital signatures.

Speculation. The author cannot predict the future, of course, but it's a safe assumption that most keys aren't changing. We introduced a structure reminiscent of a generational copying garbage collector, where there are now two separate sets of signed tuples. The young generation G_0 contains only keys that are recently modified, while the old generation G_1 contains all other keys. Once every epoch, the author generates a new set of tuples in G_1 and an empty G_0.

The tuples in G_1 contain speculative signatures that cover the range from the time they are inserted until the (future) end of the current epoch. Inserts within a given epoch then do not require any updates to the signatures in G_1. Of course, a client making any given query will require results from both G_0 and G_1, where the young generation describes whether anything relevant has changed relative to what's stored in the old generation.

If we assume a snapshot is taken after every update, then with an epoch length of E, G_0 will have at most $E + 1$ tuples. The author must sign all of the tuples in G_0 each time a snapshot is 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 thus $O(E_1 + n/E_1)$, with a minimum when $E_1 = \sqrt{n}$. Speculation can be generalized to multiple generations. With C generations, the author must sign and communicate $O(C\sqrt[C]{n})$ signatures per update instead of $O(n)$ if a snapshot is taken after every update.

If DSA signatures are used, latency can be reduced at the start of an epoch by partially precomputing signatures [25]. In addition, speculation can also be combined with superseding and lightweight signatures to reduce the storage on the server, to $O(C)$ per update.

4.3 Tuple PADs based on RSA accumulators

RSA accumulators [7] use RSA exponentiation to generate a constant-sized integer that can be used to authenticate set membership. The RSA accumulator is then signed using a traditional digital signature. The server proves that an element is in the set by sending item in question, the accumulator as signed by the author, the author's signature, and a constant-sized *witness*.

Dynamic accumulators [10] permit efficient incremental update of accumulators without requiring that they be regenerated from scratch. Accumulators have been

widely proposed for use in systems such as our (see, e.g., Goodrich et al. [15]). We now present a design that uses a signed accumulator as a concise summary of the set of tuples representing a snapshot, thus allowing for constant-sized lookup proofs and update messages.

Background.

Consider storing a set of e r-bit prime numbers $p_1 \dots p_e$. The accumulator storing these keys works as follows: The author selects an s-bit modulus $N = pq$ and a generator g with $s > 3r$. p and q are strong primes, and g is a quadratic residue mod N. p and q are kept secret. The RSA accumulator A over this set is $g^{p_1 \cdots p_e}$. The accumulator A is then signed. To prove that a key k_i is in the set, the server supplies a witness $W_i = g^{p_1 p_2 \cdots p_{i-1} p_{i+1} \cdots p_e}$. (To prevent keys from having a mathematical relationship with one other, prime numbers must be used to represent the set members.)

The author, with its knowledge of the factorization of N, may insert or remove keys from the accumulator with $O(1)$ exponentiations per update. Witnesses can be computed by an untrusted server without the knowledge of any secrets. The witness for any single key can be computed with $O(e)$ exponentiations and the set of all witnesses can be computed with an $O(e \log e)$ algorithm [6].

A membership proof that prime p_i is in the set, consists of (A, W_i, p_i), and the author's signature on A. The proof is verified by checking the signature on A and that $A = (W_i)^{p_i}$. By the Strong RSA Assumption [6], it is hard for a computationally bounded adversary to find $y > 1$ such that $g^y = A \mod N$ without knowing the factorization of N.

Bari and Pfitzmann [6] observed that we can generate *prime representatives* for arbitrary keys in the random oracle model by cryptographically hashing the key and then appending a fixed number t of extra bits. t is chosen such that there is a prime number in $[2^t(X), 2^t(X+1))$ with high probability. The value of those extra bits is chosen such that the concatenation is a prime number. Inputs for which this is not possible cannot be stored in the RSA accumulator. Papamanthou et. al. [30] recently implemented an authenticated hash table following this design.

In our design, we require that the conversion from a hash value into a prime representative is deterministic. This ensures that the RSA accumulator for a given set is uniquely defined by the inputs to the set and can be recomputed from the keys being inserted. To do this, we follow Bari and Pfitzmann [6], testing successive integers until we find a prime number.

Design. By cryptographically hashing tuples and then converting them into prime representatives, we can use RSA accumulators to authenticate a set of tuples as a single $O(1)$ accumulator that can then be bound to the version number and signed by the author. Define $A(v_q)$ to be the accumulator value for version v_q. $A(v_q)$ authenticates tuples of the form $([k_j, k_{j+1}), c_j)$ containing a key range and a contents. These tuples can omit the version number v_q because it is in the signature over the accumulator.

Each update to a PAD now only requires adding or removing at most $O(1)$ tuples. The accumulator for the next snapshot, $A(v_{q+1})$, can be computed by incrementally modifying $A(v_q)$ at a cost of $O(1)$ exponentiations per dictionary update to add or remove tuples. Updates require $O(1)$ communication; the author sends the key being inserted or removed from the PAD, the new accumulator, and the signature. Storage

increases by only $O(1)$ per update, just to store the changed key. The server could compute witnesses lazily upon lookup requests at a cost of $O(n)$ exponentiations, using no additional storage. Alternatively the server can expend $O(n)$ additional storage per-snapshot for precomputed witnesses. The server can precompute witnesses by itself with $n \log_2 n$ exponentiations. Alternatively, the author can incrementally update the n witnesses in $O(n)$ exponentiations and send them along with the update.

When a server receives a lookup request from a client for key k_q in snapshot v_q, the server returns the accumulator $A(v_q)$, bound to the version number v_q and signed by the author, a tuple $T = ([k_j, k_{j+1}), c_j)$ with $k_q \in [k_j, k_{j+1})$, prime representative p_i, and a witness for tuple i in snapshot (v_q, W_{i,v_q}). The client verifies that the prime representative corresponds to the returned tuple, $\left\lfloor \frac{p_i}{2^t} \right\rfloor = H(T)$, that the accumulator authenticates the tuple, $(W_{i,v_q})^{p_i} = A(v_q)$, and that the signature on the accumulator is valid. For efficiency, instead of sending the full prime representative p_i, only the offset from the hash of the tuple to the prime representative $p_i - 2^t H(T)$ is sent.

Unlike standard accumulator schemes, this representation offers super-efficient proofs of non-membership. The tuple $T = ([k_j, k_{j+1}), c_j)$ attests that there is no key in the interval (k_j, k_{j+1}) is in the set.

Speculation and witness computation. Accumulator-based tuple PADs can be combined with speculation, as described in Section 4.2. This increases the size of a lookup proof to $O(C)$ but reduces the costs of witness computation from $O(n \log n)$ to $O\left((C+1)\sqrt[C]{n}\right)$ exponentiations per update. (Again, as we have throughout this entire paper, we assume that after each update a snapshot is taken.)

Rather than individually sign each generation's accumulator $A(G_0, v), A(G_1, v)$ and so forth, we could instead collect these accumulators into a short hash chain $B(v) = H(A(G_0, v), H(A(G_1, v), H(A(G_2, v)\ldots)))$, and then bind the root of this hash chain, $B(v)$, to its version number and sign it. However, signing each generation individually only uses $1 + \frac{1}{\sqrt[C]{n}}$ times more signatures than using a hash chain.

On each update to the PAD, the author performs $O(C)$ amortized exponentiations, one to update the accumulator for G_0, and the remaining exponentiations account for the amortized costs of updating the accumulators for the other generations. The author then transmits the update and the new signed $B(v+1)$ to the server, who can deterministically update its copy of the PAD.

When using speculation, only G_0, containing $O(\sqrt[C]{n})$ tuples, is updated on every snapshot. The amortized cost for computing witnesses over all generations using the $O(e \log e)$ algorithm is $O((C+1)\sqrt[C]{n} * \log n)$. The server must store these witnesses at an amortized cost of $O(C\sqrt[C]{n})$ per update to the PAD.

Accumulators and tuple superseding. When we first discussed tuple superseding, in Section 4.2, it was used to reduce the signature storage on the server. This same principal may be applied to witness storage on the server for accumulators.

We alter the tuples stored in the accumulator to include the version number when they are created, e.g., $(v_q, [k_j, k_{j+1}), c_j)$. If the accumulator $A(v_{q+\delta})$ contains that tuple and is signed by the author, we consider the tuple to be valid for all versions $v \in [v_q, v_{q+\delta}]$. Thus, when a client queries for a key k in snapshot $v_{q'}$ (where $k \in [k_j, k_{j+1})$), the server may send as a proof a signed $A(v_{q+\delta})$, the tuple $T = (v_q, [k_j, k_{j+1}), c_j)$ with $k \in [k_j, k_{j+1})$ and $v_{q'} \in [v_q, v_{q+\delta}]$, and a witness proving that $T \in A(v_{q+\delta})$. The same

response can authenticate any version $v_{q'} \in [v_q, v_{q+\delta}]$. Instead of storing one witness for each snapshot, the server now can store only one witness, the one in $A(v_{q+\delta})$ that authenticates T.

As before, we assume a snapshot is taken after every update. Just as the situation described in Section 4.2, each time a snapshot occurs the server must generate a full set of witnesses. At most two of those witnesses will be for newly created tuples. The remaining witnesses are for refreshed tuples and can supersede and replace the witnesses previously stored. Computation cost is the same, but the per-update storage costs drop to $O(1)$.

Accumulators, tuple superseding, and speculation can be combined to form our final PAD design, offering constant time on the author per update, constant communication per update, constant storage per update on the server and constant lookup proof size. Computing a new set of witnesses is sublinear in the number n of keys in the pad at $O((C+1)\sqrt[C]{n})$ exponentiations per update. We individually sign each generation's accumulator in order to independently choose witnesses from different snapshots for each generation.

4.4 Implementation details

Tuple PADs offer a more complex set of design choices, including the optimizations described in Section 4.2. Apart from signing each tuple individually, tuple-superseding may be used alone, or in combination with lightweight signatures. Any of these three designs may be combined with speculation. In addition to this, we built the three RSA accumulator-based designs described in Section 4.3. Our implementations are in Python, using native code for most cryptographic primitives.

5 Implementation and methodology

Our code is a hybrid of C++ and Python, connected with SWIG-generated interface wrappers. We used OpenSSL to perform DSA signatures.

Our initial implementation of each of the 21 algorithms was in Python. Python made it much easier to design correct algorithms, debug our implementation, and cleanly modularize the code. We then progressively ported the debugged algorithms to C++, function by function and module by module while preserving the original Python implementation and applying our Python test cases against our C++ implementation. We used profile-based analysis for our porting effort, only porting modules and functions that were not bottlenecked in cryptographic or existing C++ code. To guide these choices, we separately measured the time spent in cryptographic operations, serialization, and other areas.

We first ported persistent search trees to C++, to support our tree-based PADs, yielding huge performance increases in the non-cryptographic code; authors generated updates 4 times faster, servers processed updates 7.5 times faster, servers generated lookup replies 27 times faster, and clients were able to verify the replies 15 times faster.

For tuple PADs, our current benchmarks show that crypto is expensive enough that there would be less performance benefit in rewriting the author or client code in C++. On the author, our algorithms spend at least 50% of their time in unavoidable cryptographic operations (see Table 7). Furthermore, as we will discuss in Section 5.3, we will be evaluating our algorithms based on cloud-computing monetary costs. Even without an efficient C++ implementation, the dollar-costs of bandwidth usage for tuple PADs swamp the dollar-costs of the computation when generating replies to lookup requests, so there is no particular benefit from doing the C++ port.

All of our benchmarks were run on an Intel Core 2 Duo 2.4 GHz Linux machine with 4GB of RAM running in 64-bit mode and only using one core. We used Python 2.6.4 and compiled our C++ code with gcc 4.3.4. As public-key cryptographic operations like RSA can be done with variable key lengths, trading off speed for cryptographic strength, we selected paramaters at the "112-bit security level" [28]. Keys and values are assumed to be 28-byte hashes and modular operations are done with a 2048-bit modulus. All cryptography is performed in native C or C++ code.

5.1 Serialization

For completeness, our evaluation includes the actual sizes of messages used in our PAD system. To this end, we serialized each update from the author, each request from clients, and each reply from the server. Rather than error-prone manual construction of mutually compatible serialization code in both C++ and Python, we used the Google protocol-buffer [3] library to do serialization for us. Protocol buffers support nested message types and have a very low space overhead. Protocol buffers generate very fast C++ code. The generated Python code is limited by the speed of the Python interpreter but is still reasonably fast.

5.2 RSA Accumulators

We used the GMP library for all modular operations. Our accumulators use 184-bit prime representatives[4]. The prime representative of a tuple must be found deterministically. The SHA-1 hash of a tuple is concatenated with 24 zero bits and treated as an integer. The prime representative is chosen as the numerically smallest prime number greater than that integer, found by performing 82 Miller-Rabin [33] primality tests (as advised by NIST [28]) to confirm a candidate representative. Due to the expense of finding a prime representative, the author sends the offset to the prime representative along as a hint. In our implementation, we perform all witness computation on the server.

[3] http://code.google.com/p/protobuf/

[4] Implementing the 112-bit security level would properly require 248-bit prime representatives based around SHA-224. Our current crypto library limited us to SHA-1 hashes. Our results therefore underestimate the costs of RSA accumulators.

	Amazon	Google
CPU time (cents/hour)	8.5	10
Storage (cents/GB*month)	15	15
Bandwidth (cents/GB)	10–17	10-12

Table 2: Costs charged by Amazon EC2 and Google AppEngine for cloud-computing and storage.

5.3 Cloud provider economics

Given the tradeoffs in our various implementations, with some algorithms requiring more computation and others requiring more data transmission, there is no simple way to make categorical statements as to the relative strengths of one algorithm over another. We decided to focus on the *monetary cost* of the algorithms as they might cost somebody to host a computation on third-party "cloud computing" resources, such as Amazon's EC2 and Google's AppEngine. (Monetary optimization was previously used by Gray et. al. [16] in generating their "five minute rule" for trading memory for disc accesses.) Table 2 presents current rates for these providers. Given that both providers charge very similar prices, we will use numbers from Amazon EC2 for our evaluations: $.085 per CPU hour and $.13 per gigabyte sent by the server or author.

While the absolute prices may vary in the future, what matters for our analysis is the *relative* prices of storage, bandwidth, and CPU cycles. We will assume that the author is spending the money and will attempt to minimize the total costs for themselves and the possibly outsourced server. For simplicity in our evaluation, we will assume that cloud providers charge by CPU time only while the task is executing. Or, if a cloud provider charges by wall-clock time, the CPU utilization is 100%. In Section 10, we will discuss how our performance numbers can be used to analyze scenarios with different relative costs of bandwidth and computation.

5.4 Methodology

We have too many different algorithms to compare them all directly. We reduce the complexity of our evaluation by first performing microbenchmarks to determine optimal parameters for each algorithm. We then make comparisons across algorithms with longer traces.

In Sections 7 and **??**, we evaluate the performance of tree PADs and tuple PADs with our *growing microbenchmark* where we start with an empty PAD, then insert a key, take a snapshot, perform a random query against a random snapshot, and repeat the last three steps until the dictionary size exceeds a limit. In Section 8, we present our results of running a macro-benchmark of the different PAD algorithms' performance when used to store a constantly changing set of values taken from a trace of e-commerce prices.

For each benchmark we evaluate its raw performance on the author, publisher, and client. We then evaluate the algorithms' cost effectiveness in the context of a cloud-computing environment. For each algorithm, we can evaluate the relative contribution of bandwidth or CPU time to the monetary costs of an update or a lookup. We observe

that transmitting an extra kilobyte of data costs just as much as computing for $1/190^{\text{th}}$ of a second. This defines the *provider equilibrium rate*, measured in KB/s. An algorithm need not be perfectly balanced to be optimal, of course, but this demonstrates that an optimal algorithm may well trade off somewhat more communication for a greater savings in computation or vice versa.

We define the *update bandwidth ratio* as the result of the dividing the update size (in KB) by the time to perform an update, in seconds[5]. We define the *lookup bandwidth ratio* similarity. Both are measured in KB/s. For updates, we include time spent on the author and server. For lookup proofs, we only count costs on the server.

We can compare the bandwidth ratio of an algorithm to the provider equilibrium rate to determine whether bandwidth or CPU time is responsible for the majority of the monetary costs of an algorithm. When the bandwidth ratio of an algorithm exceeds the provider equilibrium rate, the bandwidth is responsible for the majority of the costs.

Incidently, this evaluation methodology also measures the update costs, verification costs, and proof sizes of dynamic authenticated dictionaries based on these designs. Recall that the only difference between a PAD and DAD is that the server for a DAD will purge data from older versions[6].

6 Tree PAD microbenchmarks

We first consider the relative performance of treaps, red-black trees, and skiplists against microbenchmark loads. We also consider how efficiently these tree-like structures reuse state across versions, comparing path copying and three Sarnak-Tarjan variations.

6.1 Comparing tree structures

Our first evaluation considers which type of tree-like data structure runs fastest. We performed a growing microbenchmark with 100,000 keys. In general, all three tree algorithms performed similarly with 730-770 inserts per second, and 570-600 lookup proof verifications per second. All three tree algorithms spent 80%-90% of their time computing cryptographic signatures, implying that additional performance tuning on our part would yield limited gains.

We measured all three algorithms as having an update size of 150 bytes. (Section 10 considers alternative crypto parameters and the effects on these sizes.) Red-black generates the shallowest trees, causing it to have the smallest lookup proofs, the fastest performance, and the least RAM usage. These results can be seen in Table 3.

Although red-black trees are the most efficient option for authenticated dictionaries, they are also the most complex; their implementation contains code for 38 distinct

[5] Equivalently, we could multiply the size of a message by the rate (in messages/sec) at which the algorithm generates updates.

[6] While it might be tempting to remove version numbers entirely, to reduce message sizes and simplify the system, this could enable version rollback attacks, so we leave this information in the DAD.

	Proof size (KB)	Lookup rate (keys/s)	RAM used (MB)
Treap	1.96	7850	1077
Red-black	1.50	10269	841
Skiplist	2.63	4956	1587

Table 3: Performance across different tree types, inserting 100k keys, and using path-copying to implement the repository.

	Queries (per sec)	RAM used (MB)
Path Copying	10269.	841
Cache Nowhere	2.23	182
Cache Everywhere	8477.	358
Cache Median	194.	204

Table 4: Memory usage and lookup proof performance across different tree structures, storing red-black trees containing 100k keys.

cases[7]. Treaps and our skiplists are much simpler, having only 13 cases. In addition they are history independent (see Section 2.1.1), which may be required for some applications.

6.2 Comparing tree PAD repositories

Our second evaluation of tree PADs considers the different strategies for representing the repository for their efficiency at storing the forest of trees that represents the individual snapshots. In our implementation, each Sarnak-Tarjan versioned node always caches the subtree authenticator for the latest snapshot, and lookup proof generation performance on that snapshot is between 4,900-10,200 proofs per second, depending

[7]The authors wish to thank Stefan Kahrs at the University of Kent for making an open-source Haskell implementation of red-black trees that correctly handles deletion. We ported his code to Python and then C++.

	Bandwidth Ratio	
	Updates	Lookups
Cache everywhere, 10K keys	104	12265
Cache median, 10K keys	105	533
Cache everywhere, 100K keys	106	12907
Cache median, 100K keys	106	297

Table 5: Bandwidth ratios for each red-black tree PAD algorithms summarizing the relative monetary costs of bandwidth and CPU time. For ratio's over the provider equilibrium ratio (190 KB/s), proof size dominates the monetary costs. For smaller ratios, computation time dominates.

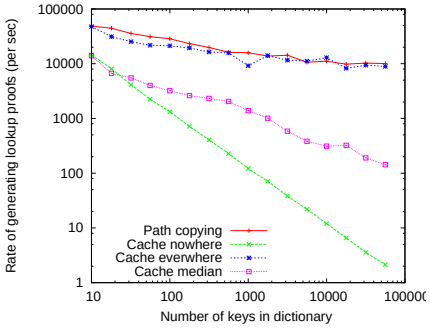

Figure 5: Steady-state lookup proof generation performance for red-black trees.

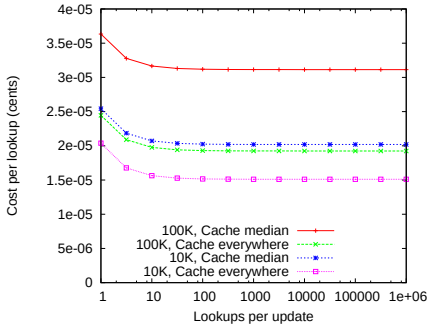

Figure 6: Amortized cost per lookup for red-black tree PADs with two different hash caching strategies.

on the tree used (see Section 6.1).

In Table 4 we present the RAM usage and the lookup rate for the four type of repositories when querying for historical snapshots. As expected, the Sarnak-Tarjan versioned trees use much less memory than path copying trees and the different caching strategies follow the expected asymptotic memory usage and performance (see Table 1). Even though Sarnak-Tarjan versioned trees that cache everywhere have the same asymptotic space and CPU costs as path copying trees, they use less memory because adding to the authenticator cache is much cheaper than cloning nodes.

To better understand the scaling behavior of tree PADs, we ran a *steady-state microbenchmark*. We filled the PAD to some capacity, and then added one key and removed one key in each snapshot. Figure 5 show how the performance of a red-black tree varies for different keycounts in the dictionary with all four of our tree repository strategies. As expected, the penalty for cache-nowhere and cache-median layer increases as the dictionary gets more keys, with cache-median degrading more slowly.

6.3 Tree PADs in a cloud-computing environment

In this section we evaluate the tradeoffs between path copying and Sarnak-Tarjan versioning with the best time/space tradeoffs (cache everywhere and cache median), in a cloud computing environment. We consider red-black trees containing 10K and 100K keys.

In Table 5 we present our results. Surprisingly, even though cache median has lookups almost 40 times *slower* than cache everywhere, both algorithms are fast enough that the bandwidth of the reply message yields the majority of the monetary cost of deployment.

The average per-lookup monetary cost of a PAD algorithm can vary depending on the ratio between lookups and updates. In Figure 6 we plot the costs per update across different lookup to update ratios for the different configurations of red-black tree PADs. Cache median is 30%-60% more expensive than cache everywhere, but required 40% less memory usage.

Base+SS	No speculation. Optimized with superseding.
Base+LW	No speculation. Optimized with lightweight signatures.
Spec+SS	Speculation with 2 generations. Optimized with superseding.
Spec+LW	Speculation with 2 generations. Optimized with lightweight signatures.
Accumulators	Speculation with 2 generations. Uses accumulators.
Chain Accum.	Speculation with 2 generations. Uses accumulators in a hash chain.

Table 6: Abbreviations used to denote the different tuple-based algorithms.

6.4 Tree conclusions

For maximum performance, tree-based PADs should use Sarnak-Tarjan versioned nodes with the cache-everywhere versioning strategy. In the case where very few queries are made for historical snapshots or where memory is low, caching on the median layer may have sufficient query throughput. We also conclude that red-black trees dominate treaps and skiplists, running faster, having smaller lookup-proof sizes, and using less storage. Treaps enable other useful semantics, but there is no reason to ever use a skiplist.

7 Tuple PAD microbenchmarks

In this section, we will evaluate the various tuple PAD designs described in Section 4, including our original designs and our new RSA accumulator extensions (see Section 4.3).

Table 6 describes the abbreviations we will use for the different algorithms. We do not report results of algorithms that do not use superseding, since they have the same performance and message sizes as the algorithms using superseding. To put the performance of tuple PADs in context, we also repeat our results for red-black trees using the cache-everywhere strategy.

7.1 Tuple PAD author costs

In Table 7, we present the performance of each tuple PAD algorithm we analyzed. Note that due to poor insert performance, we only ran Base+SS for 3851 inserts in the growing benchmark instead of 10,000. If we extrapolated its performance at 10,000 inserts, we would expect 0.05 updates per second and a 290 KB update size. Table 7 also demonstrates the effect of tuple PAD optimizations. It shows the benefits of speculation, increasing performance by a factor of 25 and reducing update sizes by a factor of 50. Lightweight signatures have a similarly strong impact on performance. Lightweight signatures are sufficiently cheap relative to full public-key signatures that crypto costs no longer completely dominant the runtime.

Our results clearly demonstrate the poor update performance of tuple PAD algorithms. Even if we could completely eliminate the non-crypto overheads on the author, the fastest tuple PAD is still six times slower than a simple red-black tree PAD. The

	Inserts		Size (KB)		Number
	(per sec)	% in crypto	Update	Proof	Inserted
Base+SS	0.35	88%	114.46	0.15	3851
Base+LW	2.67	13%	156.72	0.21	10000
Spec+SS	6.6	85%	6.44	0.30	10000
Spec+LW	63.	52%	3.76	0.42	10000
Chain Accum.	62.5	79%	0.14	1.23	10000
Accumulators	64.7	81%	0.14	1.30	10000
Red-black	753.	91%	0.15	1.20	10000

Table 7: Comparing author performance, update sizes and proof sizes across different algorithms. Crypto costs include digital signatures, finding prime representatives, lightweight signatures, and exponentiations. Except for "Base+Supersede," where 3851 keys were inserted, we ran each algorithm with 10,000 keys.

	Updates on server		Server response	Client response verification	
	(per sec)	% in crypto	generation (per sec)	(per sec)	% in crypto
Base+SS	4.2	—	5619	653	91
Base+LW	2.4	—	4781	594	91
Spec+SS	64.8	—	2896	323	91
Spec+LW	104.4	—	2497	314	90
Chain Accum.	0.90	99%	2419	290	89
Accumulators	0.93	99%	1991	205	91
Red-black	9009.	—	10221	628	88

Table 8: Comparing server and client performance. Cryptographic costs include digital signatures, finding prime representatives, lightweight signatures, and exponentiations.

network communication needed for updating the red-black tree PAD is similarly as small as the very best accumulator-enhanced tuple PAD.

We implemented the hash chain optimization described in Section 4.3 and observed the same CPU performance as signing each generation's accumulator individually because primality computation and modular exponentiation operations dominate. RSA accumulators, although being constant size, are surprisingly large and generate lookup proofs no smaller than red-black trees storing 10K keys. We will examine accumulators further in Section 7.4.

7.2 Tuple PAD server costs

In Table 8, we present the server's costs for the different PAD algorithms. On each update, most algorithms do nothing other than store tuples and signatures into the repository, taking time proportional to the update size. In this table we can see the extreme benefits of speculation, which improves performance on the server by reducing the number of tuples the server must process for each snapshot from $O(n)$ to $O(\sqrt{n})$. Even using speculation, accumulator algorithms process updates much slower than because

	Bandwidth Ratio	
	Updates	Lookups
Base+SS	37.	843
Base+LW	201.	1004
Spec+SS	39.	869
Spec+LW	149.	1046
Chain Accum.	0.124	2975
Accumulators	0.128	2975
Red-black	104.	12265

Table 9: Bandwidth ratios for each algorithm summarizing the relative monetary costs of bandwidth and CPU time. For ratio's over the provider equilibrium ratio (190 KB/s), proof size dominates the monetary costs. For smaller ratios, computation time dominates.

	Bandwidth Ratio	
	Updates	Lookups
Base+SS	43	925
Base+LW	85	1150
Spec+LW	65	1239
Red-black	1226	13892

Table 10: Bandwidth ratios for each algorithm, processing the luxury-goods macrobenchmark, summarizing the relative monetary costs of bandwidth and CPU time. For ratios over the provider equilibrium ratio (190 KB/s), proof size dominates the monetary costs. For smaller ratios, computation time dominates.

they have to compute witnesses on the server.

The time for a client to verify a lookup proof varies across the different algorithms. The cost of verifying is dominated by modular exponentiations occurring in signature verification and accumulator verification. Designs using speculation take twice as long because usually require verifying two signatures, one in each generation.

Accumulator PADs using hash chains do not have an appreciably smaller lookup proof. The size of a lookup proof is dominated by the 2048-bit accumulator value and the 2048-bit witness, required for each generation. These overheads are large compared to the 320-bit cost of an extra signature. Hash chains somewhat improve lookup proof verification performance. When a hash chain is used, only one signature need be checked. This can be seen in Table 8 in the increased performance verifying a hash chain accumulator lookup proof.

7.3 Tuple PADs in a cloud-computing environment

In this section, we evaluate the tradeoffs between the various tuple PAD designs in the context of a cloud-computing environment. In Table 9, we present the bandwidth ratio for each algorithm. Whenever the ratio exceeds 190 KB/sec, the monetary cost of transmitting the message exceeds the monetary cost of computing the message. Every

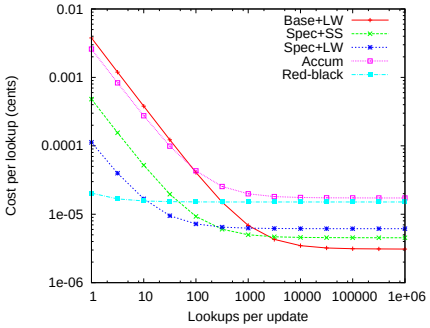

Figure 7: Amortized cost per lookup for different PAD algorithms running the growing benchmark on 10K keys.

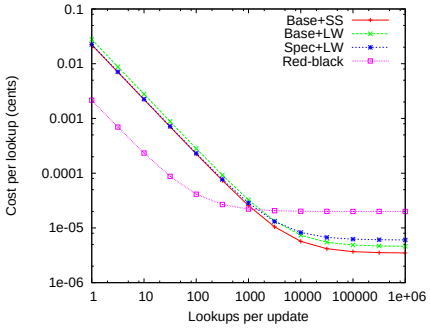

Figure 8: Amortized cost per lookup for different PAD algorithms processing the luxury-goods macrobenchmark.

implementation has a bandwidth ratio over 843 for lookups, meaning that at least 80% of the monetary costs of these algorithms will come from bandwidth of the reply, not the CPU time, disincentivizing us from porting our slower Python implementations to C++. These results also show that the majority of the monetary costs of a lookup are the bandwidth costs involved in sending the result.

The overall monetary cost of each algorithm depends on the relative ratio between updates and lookups. In Figure 7 we plot the costs per lookup across different lookup to update ratios for several algorithms. This plot illustrates the tradeoffs between the different algorithms. Except for the algorithms using accumulators, which never win, *every* other algorithm is the cheapest at some ratio of lookups to updates. In the case when there are high numbers of lookups per update, update costs becomes less important and the smaller response sizes of Base+LW and Spec+SS cause these algorithms to be preferable.

7.4 Accumulator tuple PAD costs

We now take a closer look at RSA accumulators as a stand-alone entity used to authenticate a set of elements that are stored on an untrusted server. We examine their costs on the author, server, and clients. We compare the costs with simply signing each element in the set with a DSA signature, or bundling the elements into a Merkle tree and signing the tree root. This performance evaluation assumes the "112 bit security level," requiring 224-bit hashes, 240-bit prime representatives, and 2048-bit modulus operations. Note that we are comparing the costs to store a set, not a dictionary.

In theory, the advantage of RSA accumulators compared to signing each element separately is in saving the bandwidth required for an update. However, if witnesses are computed on the author and sent, no bandwidth is saved as 2048-bit witnesses are 6 times larger than 320-bit DSA signatures *and* take twice as long to compute. If witnesses are computed on the server, then an accumulator only makes sense when the cost of the time to compute witnesses is cheaper than the cost of the time and bandwidth required to individually sign and transmit each item, as in the tuple PAD designs. With

Amazon and Google's prices for bandwidth and computation, it is 6 times cheaper to simply sign each tuple and avoid accumulators. Signing each tuple also offers lookup proof sizes 3 times smaller.

A membership proof in an RSA accumulator requires 4096 bits, 2048 bits to send the accumulator value and 2048 bits to send the witness; that is big enough to store over 18 224-bit hashes, capable of representing a path to any leaf in a Merkle tree of depth 17, which is sufficient to represent sets of up to 256k elements. RSA accumulators this big are glacially slow. Each update requires 10 minutes of CPU time for author-computed witnesses or 60 minutes for server-computed witnesses.

This analysis examined the tradeoffs of using accumulators for representing a static authenticated set. A DAD or PAD offers additional semantics, in particular, proof of non-membership and efficient updates. We can draw the same conclusions about the inefficiency of accumulators when used to implement a PAD or DAD from the results reported earlier in this section.

7.5 Tuple conclusions

We can reach several conclusions from our results. RSA accumulators are so expensive, from a CPU and bandwidth perspective, that we will never recover these costs. For PADs which are updated very frequently, red-black tree PADs clearly win. However, for more stable PADs with higher query rates, the tuple-based PAD structures, sans the RSA accumulator, become the preferable strategy. For workloads where a widely varying range of lookups per update might be expected, our full set of tuple PAD optimizations, including speculation, lightweight signatures, and superseding, seems to be an excellent strategy. For workloads where over 1000 lookups might be expected per update, the non-speculative tuple PAD, but with lightweight signatures, would seem to be the appropriate algorithm.

8 Macrobenchmark

Now that we have done many microbenchmarks of the different PAD designs, we can analzye the performance and monetary costs of the different PAD algorithms when used to store a constantly changing set of values taken from a trace of e-commerce prices.

Our data set represents the selling prices of high-end luxury goods as offered by a number of vendors on the Internet. All price observations were made between January 1, 2009 and June 30, 2009 inclusive, representing 27 distinct dates. In total, 1,272 different items were found online for the three brands, on a total of 544 different web sites. In total, there are 38,391 different observations in the data set. Our data tracked the price of each item on each web site, forming 14,374 distinct keys in the PAD.[8]

Table 11 presents the performance of the different algorithms on this benchmark. This dataset is very different than our microbenchmarks. Most notably, it has a coarse

[8]Data provided by Glenn Kramer Consulting, LLC, representing actual brands and products monitored for an anonymous client, blinded and provided with client's permission.

	Insert All Keys (sec)	Process All Updates (sec)	Size (KB) Update	Proof	Lookups (per sec)
Base+SS	383.	57.	708	0.18	5354
Base+LW	131.	44.	549	0.24	4838
Spec+LW	142.	50.	460	0.42	2952
Red-black	1.84	1.44	149	1.59	9422

Table 11: Performance of different PAD algorithms on the macrobenchmark, including the total time on the author and server to insert six months of price data, the average size of an update and lookup proof, and the lookup rate.

granularity. 38K updates are contained in only 27 snapshots, leading to large update messages. Lookup performance is as fast as we saw before.

This dataset also demonstrates that the strengths of speculation occur when there are many snapshots and relatively few keys are modified in any one snapshot. In Section 4.2 we assumed only one update per snapshot, making the ideal epoch size $O(\sqrt{n})$. Here, when there are multiple updates per snapshot the ideal epoch size is $\sqrt{n/K}$ when there are two generations and K updates per snapshot. For our macrobenchmark, this would result in an epoch length of 30 snapshots, more than the number of snapshots in our dataset and no speculation would occur at all. Rather than artificially extending our dataset, we used an epoch size of $6 \approx \sqrt{27}$ for our benchmarks, to demonstrate the advantages of speculation, which reduced the number of signature operations by 48%. Lightweight signatures were also very beneficial, reducing the number of public-key signatures by over 80%.

We also performed a cloud-computing monetary cost analysis of our algorithms over this data set. Bandwidth ratios are reported in Table 10 and the bandwidth ratios for lookup messages are within 20% of what we observed earlier in Tables 5 and 9 when running the growing benchmark. The update message bandwidth ratio for red-black trees is much larger than we saw in Table 5 because the message size has grown to include all of the updated keys, while the number of expensive digital signatures has remained the same.

In Figure 8 we plot the cost per lookup. In this dataset, the large number of changes per snapshot results in large per-update monetary costs which must be amortized over many messages before the smaller response sizes of tuple PADs reduces their overall costs.

From this, we can conclude that the red-black tree PAD (Sarnak-Tarjan, cache-everywhere) is the preferred PAD algorithm until the query rate exceeds roughly 5000 lookups per update. Only then do the tuple PAD structures become more cost effective, with the simpler "base+SS" strategy (no speculation or lightweight signatures; just superseding) ultimately winning only when the query rate exceeds 25K lookups/update.

9 Scalability

We expect that the server, in our system, may well be called upon to scale to run on large clusters and support much higher insertion and query rates. This section considers scalability issues for such environments and how our algorithms could be modified to run faster in such environments.

Faster server insertion rates. Keys exist in a large key space. We can partition that key space across a large cluster of machines, with each server responsible for only a fraction of the key space (much as is standard practice in distributed hash table implementations). Each server then maintains that fraction of the PAD. Assuming keys are uniformly distributed across the key space, each server should see a uniform fraction of the load. To guarantee this uniformity, keys could be hashed before being stored in the PAD.

For any tuple PAD implementation, without RSA accumulators, this split is quite natural. Different servers can be responsible for different key ranges, allowing for excellent scalability. For tree PADs, each server would be responsible for a different subtree, but coordination would be required for changes to the shared top levels of the tree.

Faster client query rates. Client queries require no mutation of state on the server. As such, server state may be arbitrarily replicated to support larger client query rates. This would require inbound mutation operations from authors to be distributed to each replica responsible for any given key.

Lots of snapshots. While some measure of coordination is required, as above, to handle the most current version of a PAD, older versions are static. In a large server cluster, older snapshots can be replicated onto dedicated machines. Any given range of keys from any given snapshot can be stored on multiple, different servers, allowing for excellent scalability both in terms of storage capacity and supported client query rates.

Faster authors. Presently, we assume that the "author" is running on a single computer, but we could imagine a large number of machines, sharing the author's crypto key material, concurrently authoring a PAD. Assuming the server is ready to support the higher insertion rate, as above, the challenge is to coordinate all the author nodes. For modest scalability, a single-threaded author can controls the tree or tuple layout, delegating expensive cryptographic computations to other nodes in its cluster. If DSA signatures are used, latency can be further reduced by having author nodes partially precompute signatures [25].

For broader scaling, the author nodes could follow a partitioning strategy, similar to that described for the server. Again, this partitioning is quite natural with tuple PADs and will require coordination of the higher layers in the tree for tree PADs.

10 Selecting and tuning the right algorithm

Our space and time benchmarks were collected on a specific machine with a specific set of cryptographic parameters, including 2048-bit DSA signatures, and 28-byte keys, values, and hashes. Obviously, our measurements would be different had we used

different cryptographic algorithms, chosen different cryptographic parameters, different key and value sizes, or if cryptographic operations were cheaper (e.g., through cryptographic accelerators). Without needing to re-run our benchmark runs, we can extrapolate our results to cover a variety of scenarios:

If bandwidth gets cheaper faster than CPU get cheaper, then provider equilibrium ratios will increase. The monetary costs of an algorithms whose bandwidth ratio is less than the old provider equilibrium ratio will change much less than algorithms whose bandwidth ratio is much higher. If, for example, bandwidth got ten times cheaper, then from Table 9, we could conclude that accumulators and red-black trees, which have bandwidth-intensive lookups, would be disproportionately affected. New prices for lookups and updates could be computed by adapting the numbers in Tables 7 and 8 and red-black trees would be cheaper than tuple-pads at ratios up to 200 lookup per update, rather than 15.

If CPU gets cheaper faster than bandwidth get cheaper, a similar analysis applies. In this case, the provider equilibrium ratios will decrease. Tables 9 and 5 shows which algorithms are CPU-intensive, with low bandwidth ratios, that would be disproportionately affected. If, for example, a cloud computing provider sold CPU for one tenth the cost of Amazon while charging the same for bandwidth, a tree PAD caching on the median layer would become more appealing compared the cache everywhere strategy, since the cost of recomputing the hashes becomes comparatively cheaper, allowing us to use less RAM (See Section 6.2 and Table 4). The bandwidth costs would be identical.

If cryptography was ten times cheaper, whether from hardware cryptographic accelerators, faster elliptic curve signature algorithms, or any other cause, crypto overheads would drop from their current 50%–90% fraction of CPU costs to 10%–50%, resulting in an overall increase in performance of the different algorithms by 80%–420%. The now-lower crypto overheads would offer the opportunity for additional performance benefits by rewriting Python code into C++.

If elliptic curve signatures were used, nothing would change for the server. ECDSA signatures of the same security strength as regular DSA signatures are exactly the same size, so the bandwidth costs would not change. They're faster to compute, but that expense is undertaken by the author, not the server.

If smaller keys were stored in the PAD, which could be stored directly rather than first being hashed, lookup proofs in tree-based PADs would have a comparative advantage, as their lookup proofs include a key and two hashes for each node of the $O(\log n)$ expected nodes in the path from the root. In the case of a stock ticker, where keys could easily be encoded as 8 byte strings, instead of the 28-byte hashes that were used in our benchmarking, we would expect red-black tree lookup proofs to drop from 1.2 KB to 0.95 KB, while tuple PADs would decrease by only 40 bytes if no speculation is used and $40 \cdot C$ bytes if speculating with C generations.

If more than 10K keys are stored in the PAD, tuple PAD updates would become more expensive in CPU time and bandwidth, following the $O(n)$ or $O(C\sqrt[C]{n})$ big-O expectation. Tree PAD updates would require almost the same CPU time, because the $O(1)$ signature computation dominates the $O(\log n)$ tree update operations. Red-

black tree PAD lookup proofs would grow by about 340 additional bytes each time the keycount increases by an order of magnitude, 1.53 KB for 100K keys, and 1.87 KB for 1M keys.

11 Conclusion

Our analysis considered two very different structures for implementing persistent authenticated dictionaries: those based on Merkle tree-like data structures and those based on independently signed "tuples." We implemented Merkle trees based on skiplists, treaps, and red-black trees, along with four different strategies for how to share related state across different versions of the trees. We implemented tuples, based on our prior designs, including a variety of optimizations that we proposed, and adding a new design and implementation with RSA accumulators to improve the asymptotic efficiency of our tuple design.

These algorithms make a variety of different tradeoffs between computation, bandwidth, and storage. Our strategy of converting all of these units into monetary costs, based on commodity pricing from Amazon and Google, offered useful insight into which algorithms are preferable under which conditions. Most notably, we conclude that the fixed costs of RSA accumulators dwarf their asymptotic benefits, making them unsuitable for production use. We conclude that red-black trees, implemented with Sarnak and Tarjan's versioning strategy, and caching subtree authenticators at every node for every version, is the optimal strategy for PADs experiencing frequent updates. However, when the query rate grows much larger than the update rate, our tuple PAD strategies, with our full suite of optimizations, seem to be the preferable strategy.

We have a number of directions to consider in future work. We will pursue algorithmic extensions to PADs to better support multiple, mutually-untrusting authors. We will also examine extensions to integrate privacy features together with PADs.

Given our flexible software implementation of so many different PAD variations, we intend to pursue applications of our data structures toward more concrete problems, such as building robust file storage above potentially untrusted storage like Amazon's S3 service. We also intend to release our code under a suitable open-source license.

References

[1] G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266, 1962.

[2] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In *International Conference on Information Security (ISC)*, pages 379–393, Seoul, Korea, Dec. 2001.

[3] A. Anderson and T. Ottmann. Faster uniquely represented dictionaries. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science (SFCS)*, pages 642–649, San Juan, Puerto Rico, Oct. 1991.

[4] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (SFCS)*, pages 540–545, Oct. 1989.

[5] P. Bagwell. Fast functional lists, hash-lists, deques and variable length arrays. In *In Implementation of Functional Languages, 14th International Workshop*, page 34, Madrid, Spain, Sept. 2002.

[6] N. Bari and B. Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *EuroCrypt*, pages 480–494, Konstanz, Germany, May 1997.

[7] J. Benaloh and M. de Mare. 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)*, pages 274–285, Lofthus, Norway, May 1993.

[8] G. E. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 16–26, Puerto Vallarta, Mexico, June 1998.

[9] G. S. Brodal. Partially persistent data structures of bounded degree with constant update time. *Nordic Journal of Computing*, 3(3):238–255, 1996.

[10] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO '02*, pages 61–76, Santa Barbara, CA, Aug. 2002.

[11] S. A. Crosby and D. S. Wallach. Super-efficient aggregating history-independent persistent authenticated dictionaries. In *Proceedings of ESORICS 2009*, pages 671–688, Saint Malo, France, Sept. 2009.

[12] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. *ACM Transactions on Compututer Systems*, 20(1):1–24, 2002.

[13] B. Gassend, G. Suh, D. Clarke, M. Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *The 9th International Symposium on High Performance Computer Architecture (HPCA)*, Anaheim, CA, Feb. 2003.

[14] M. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivability Conference & Exposition II (DISCEX II)*, pages 68–82, Anaheim, CA, June 2001.

[15] M. T. Goodrich, R. Tamassia, and J. Hasic. An efficient dynamic and distributed cryptographic accumulator. In *Proceedings of the 5th International Conference on Information Security (ISC)*, pages 372–388, Sao Paulo, Brazil, Sept. 2002.

[16] J. Gray and F. Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time. *SIGMOD Rec.*, 16(3):395–398, May 1987.

[17] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (SFCS)*, pages 8–21, Oct. 1978.

[18] H. Kaplan. Persistent data structures. In D. Mehta and S. Sahni, editors, *Handbook on Data Structures and Applications*. CRC Press, 2001.

[19] P. C. Kocher. On certificate revocation and validation. In *International Conference on Financial Cryptography (FC '98)*, pages 172–177, Anguilla, British West Indies, Feb. 1998.

[20] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Operating Systems Design & Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[21] R. C. Merkle. A certified digital signature. In *CRYPTO '89*, pages 218–238, Santa Barbara, CA, 1989.

[22] S. Micali. Efficient certificate revocation. Technical Report 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.

[23] D. Micciancio. Oblivious data structures: Applications to cryptography. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 456–464, El Paso, Texas, May 1997.

[24] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec. 2002.

[25] D. Naccache, D. M'Raihi, S. Vaudenay, and D. Raphaeli. Can DSA be improved? Complexity trade-offs with the digital signature standard. In *EuroCrypt*, pages 77 – 85, Perugia, Italy, May 1994.

[26] M. Naor and K. Nissim. Certificate revocation and certificate update. In *USENIX Security Symposium*, San Antonio, TX, Jan. 1998.

[27] M. Naor and V. Teague. Anti-presistence: history independent data structures. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing (STOC)*, pages 492–501, Heraklion, Crete, Greece, July 2001.

[28] NIST Special Publication 800-57. *Recommendation for Key Management — Part 1: General*. National Institute for Standards and Technology, Mar. 2007.

[29] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[30] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *ACM Conference on Computer and Communications Security (CCS '08)*, pages 437–448, Alexandria, VA, Oct. 2008.

[31] Z. N. J. Peterson, R. Burns, G. Ateniese, and S. Bono. 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.

[32] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449, 1989.

[33] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128 – 138, 1980.

[34] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 183–196, Chicago, IL, 2007.

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

[36] J. S. Shapiro and J. Vanderburgh. Access and integrity control in a public-access, high-assurance configuration management system. In *USENIX Security Symposium*, pages 109–120, San Francisco, CA, Aug. 2002.

[37] P. Williams, R. Sion, and D. Shasha. The blind stone tablet: Outsourcing durability. In *Sixteenth Annual Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, Feb. 2009.