FastAD: An Authenticated Directory for Billions of Objects

Paul T. Stanton, Benjamin McKeown, Randal Burns, Giuseppe Ateniese
Department of Computer Science, Johns Hopkins University

ABSTRACT

We develop techniques that make authenticated directories efficient and scalable toward the goal of managing tens of billions of objects in a single directory. Internet storage services have already realized this scale: Amazon's S3 contained more than 52 billion objects as of April 2009 [1]. Our contributions include defining on-disk, block-oriented data structures and algorithms for authenticated directories that exceed memory capacity and optimizations that reduce the I/O required to insert and access entries in the directory.

1. INTRODUCTION

The cloud environment provides unprecedented access to shared data storage, including key/value stores, distributed caches, and content distribution networks. Cloud storage services provide great benefits by abstracting away traditional considerations such as disk capacity, backup management, and data recovery. However, the inability of users to ensure the authenticity and integrity of shared data limits the utility of this access [4]. Users lack the confidence needed to overcome the "my sensitive corporate data will never be in the cloud" mentality [3]. Consumers who retrieve data from cloud resources must be able to verify that the data they retrieve are from an authentic source and consistent with legitimate insert, update, and delete operations. Thus, data producers require methods to attest to the freshness, authenticity, and integrity of data.

One lesson learned from the body of research on the accountability of Internet services is that authenticated directories are the preferred technique to manage data that change over time [2, 8, 12, 16]. Authenticated directories group a set of objects and their identifiers into a hierarchical structure that is uniquely defined by a small cryptographic tag. The structure's tag defines a "current version" of the set of objects, capturing the dynamic nature of data. Thus, when objects are modified or removed, their old versions will no longer be authentic.

Existing implementations of authenticated directories employ data structures that exhibit performance problems at scale. Merkle trees [15] and skiplists with pairwise cryptographic message digests [12] quickly grow beyond the bounds of memory, possibly incurring an I/O to navigate to each node along a search path. Furthermore, the fine-grained nature of these data structures dictates that they map poorly onto disk storage. Merkle trees are, in essence, binary trees and it becomes non-trivial in the face of insertions, deletions and updates to map the nodes of the binary tree to a balanced tree of large blocks of storage [20].

Our system achieves performance and scale by (1) building authenticated directories using a block-aligned deterministic skiplist [17] rather than mapping a binary tree onto block-oriented data structures and (2) providing optimizations that reduce the I/O requirements of lookups in the data structure based on replicating a small amount of metadata and by temporal organization.

Within each block of the skiplist, we employ an incremental hash [6] to reduce the number of cryptographic operations and reduce proof size. The digest values that we maintain for authentication resemble previously used digests [12, 20] in that the hashes accumulate to the root element. Our implementation combines an entire disk block worth of hash values into a single digest, rather than assembling hash values pairwise. This reduces the size of the proof—the path from root to leaf in the authenticated directory—because the higher fan out results in shallower data structures. When updating or inserting an element, we compute a block's hash incrementally based on the old value of the block's digest and the new or updated element. Incremental computation updates the digest using a constant number of operations regardless of block size.

To reduce I/Os on lookup, we replicate the most recent version of an object's authenticity proof (a path from the leaf node to root node) in object metadata. This serves as a hint and, upon performing a lookup, the client provides this path to the server. While traversing the path, if the search reaches any portion of the tree that has not changed due to updates, the search terminates successfully, avoiding the I/O associated with continued traversal. This short-circuit strategy also increases the likelihood that more evaluations occur in the higher levels of the skiplist that change most frequently and are most often in cache. Finally, we perform all inserts at the tail of the skiplist, which supports our hint optimization by making large portions of the skiplist stable and reduces the cache footprint of the insert workload.

An evaluation of our prototype system, FastAD, reveals that our algorithms improve the performance of authenticated directories asymptotically: inserts with increasing block size and verifications with the number of objects in the directory. For practical parameters, FastAD more than doubles the performance of all operations.

2. FASTAD

2.1 Model

FastAD is designed as an authenticity and integrity checking system that provides public verifiability of data stored at and retrieved from untrusted sites. When retrieving an object from an untrusted store, FastAD allows a client to verify that the untrusted store returns the correct data for the most current version of that object. The usage model includes a trusted authenticated directory server, an untrusted storage infrastructure, multiple data producers that authenticate with the directory server and store data in the storage infrastructure, and any number of ad-hoc client data consumers. To perform a write, a producer locally calculates a digest of the data and registers that digest with FastAD. FastAD recalculates its hierarchy of digests and returns to the producer a unique object identifier and metadata describing the path from root to leaf in the authenticated directory. The producer then stores the object identifier, path information, and data with an untrusted store. When reading the data, the consumer fetches the file data and path information by identifier from the untrusted store, generates a secure digest of the data, and queries FastAD to verify the authenticity and integrity of the data using the identifier, path, and hash. FastAD responds with success or failure and an accompanying proof of inclusion or exclusion [16] respectively.

In our model, the roles of producer and consumer can be assumed by the same or different users. As an example, the roles overlap for users that store their data with a storage service provider. The user writes data to an untrusted server and registers the data with FastAD. Upon retrieval, FastAD verifies that the data returned from the store match the contents of the most recent version. In other situations in which the roles apply to different users, FastAD provides public verifiability. For example, a Web or social networking application can act as a producer by placing data on publicly accessible, but untrusted storage, while maintaining a trusted authenticated directory. A large number of clients using that application can then retrieve the data and guarantee its integrity, authenticity, and freshness by querying the directory maintained by the application.

Although we designed FastAD for cloud storage, the techniques and optimizations apply to any system that uses authenticated directories, such as certificate revocation [16].

2.2 Block-oriented authenticated directories

The goals of constructing an authenticated directory using a block-oriented data structure are to minimize I/O costs associated with directory operations and avoid the complexity of mapping binary trees or binary skiplists [12] onto block-aligned on-disk data structures. We achieve these goals by combining a skiplist data structure with incremental hashing [6]. We use block-sized sets of elements to build and maintain the skiplist and as the basis for the cryptographic operations used to generate membership proofs. Inserts and verifications access $\mathcal{O}(\log_b n)$ blocks and compute $\mathcal{O}(\log_b n)$ incremental hashes for a directory of n elements in which each block holds b elements. Two incremental hashes, the MuHASH and the AdHASH, combine many elements into a single digest of 128 bytes and 200 bytes respectively, which allows FastAD to build a hierarchical data structure with arbitrary fan-out without increasing the number of cryptographic operations or the size of membership proofs. Cryptographic complexity and proof size were the key obstacles that restricted previous work to binary-tree authenticated directories [20].

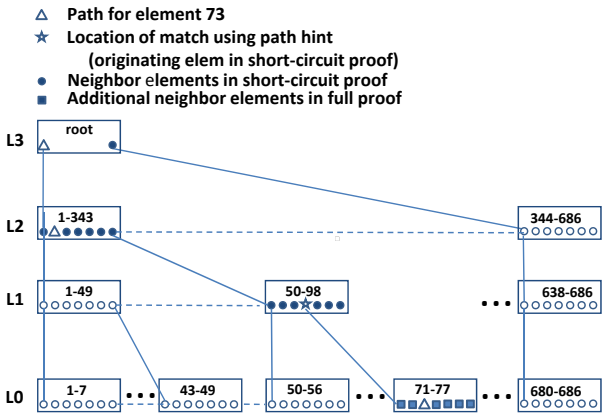

Figure 1: FastAD's skiplist associates an entire block's worth of elements using the MuHASH. The figure shows the association of elements from leaf to root.

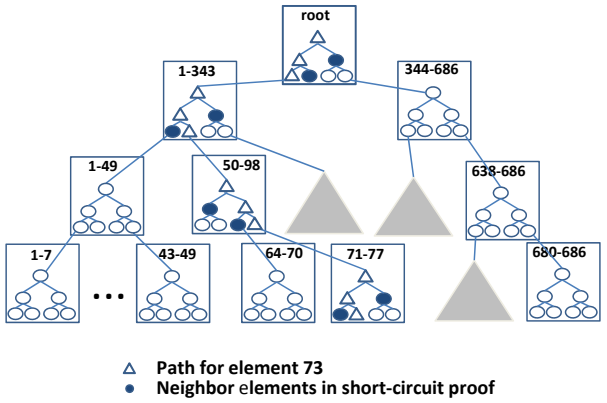

Figure 2: A block-oriented binary-tree hashes elements pairwise. The figure shows the association of elements from leaf to root.

2.3 Design of FastAD

FastAD builds an authenticated directory based on storing object identifiers and hash values of the object data in a block-oriented deterministic skiplist (Figure 1). Skiplists store n elements of a set S in a series of lists organized into hierarchical levels. Similar to search trees with a fanout of b, skiplists require $\mathcal{O}(\log_b n)$ for lookups, insertions, and deletions in which b is the *promotion factor* or, in our case, the number of elements that fit into a block. For the skiplist in Figure 1, b is equal to 7.

FastAD employs an incremental hash to accumulate all of the hashes in the directory into one unique identifier. We describe and then analyze two such hashes that present tradeoffs in security, size, and performance. A multiplicative incremental hash (MuHASH) [6] is provably secure but relatively inefficient, because the incremental computation relies on multiplication and the multiplicative inverse. The more efficient additive incremental hash (AdHASH) [6] relies on addition and subtraction for incremental computation, but must employ longer outputs (1600 bits versus 1024) to mitigate security vulnerabilities [19]. Just as with the pairwise hash-generation schemes used previously in authenticated directories, the incremen-

tal hashes stored at any node accumulates the hashes below it in a hierarchy to provide proof of object inclusion in a set or exclusion from a set [16].

MuHASH: All of the elements of each level-0 block in the skiplist are accumulated using a MuHASH that contributes to a single element in a level-1 block. For level-0 block containing objects with identifiers in the range $u, \ldots, v$, the MuHASH $Z_{0,(u,v)}$ is computed as

$$\text{MuHASH}^h_{\langle G\rangle}(Z_{0,(u,v)}) = \prod_{j=u}^{v} [h(\langle j\rangle || D_j)]$$

in which the group G is used to build the MuHASH and h is an ideal function that maps elements into G. Additionally, $\langle j\rangle$ is the binary representation of the object identifier and D_j is the digest of the data object computed by the writer. FastAD does not need to see the data; it only stores the integrity information. The higher levels in the skiplist compute the MuHASH of the accumulated hashes of its children blocks (Figure 1). In the figure, the hash of object 73 is included in a level-0 MuHASH that contains the hashes of objects 71-77. This value is combined with the those of other level-0 blocks in a level-1 MuHASH that accumulates objects 50-98. Higher levels continue accumulating lower level values in a similar manner until the root accumulates all objects in the directory.

The MuHASH construction employs a cryptographic hash function h and provides incrementality based on the commutative and invertible properties of multiplication in a selected group G. The resultant digest $\text{MuHASH}^h_{\langle G\rangle}$ has been proven to be *collision-free* when the underlying hash function is *ideal* and the product is taken in a group G in which the discrete logarithm problem is hard [6]. In our case, we select G such that $G = Z^*_p$ for some sufficiently large prime p, the product taken in group G is multiplication modulo p, and $|p| \geq 1024$[1]. For h, we use a construct first presented by Bellare et al. [7] and recently analyzed by Leurent and Nguyen [13]. We let H = SHA-384 (that is, the truncated version of SHA-512 with *strengthened* Merkle-Damgard transform) and define $h(x)$ as the truncated first 1024 bits of:

$$H(\text{c}||\langle 0\rangle||x)||H(\text{c}||\langle 1\rangle||x)||H(\text{c}||\langle 2\rangle||x)$$

in which c is a public constant unique to h (i.e., another hash function h' must use a different constant). Upon completion, we verify that $h(x)$ is in G.

FastAD benefits from the incremental computation of the MuHASH. When updating the MuHASH of a set of b objects, we compute h twice at each level of the data structure amounting to six invocations of SHA-384. A binary tree implementation requires only one hash for each level, but has $\log_2 b$ times more levels. For example, when modifying a leaf block changing the content of the object with identifier u from o_u to o'_u, the new MuHASH (Z') can be computed with the original MuHASH (Z), the original digest, and the modified digest

$$T = Z \times inv(h(\langle u\rangle||D_j)) \bmod p$$
$$Z' = T \times h(\langle u\rangle||D'_j) \bmod p$$

where $inv()$ is defined as the multiplicative inverse modulo p and D'_j is the new digest generated by the writer. The operation amounts to dividing out the original element from the cumulative hash and then multiplying in the new element. Deletions can be realized without compromising security properties that rely on the index by setting D'_j to a unique reserved string and employing the update algorithm [5].

[1]Ideally, a prime-order group should be used here to achieve a tighter security proof for MuHASH [6].

AdHASH: AdHASH uses addition and subtraction, instead of multiplication, to provide incrementality. Similar to the description of MuHASH, for level-0 block containing objects with identifiers in the range $u, \ldots, v$, the AdHASH $Z_{0,(u,v)}$ is computed as

$$\text{AdHASH}^h_M(Z_{0,(u,v)}) = \sum_{j=u}^{v} [h(\langle j\rangle||D_j) \bmod M]$$

in which h is a hash function that maps elements into Z_M and M is a publicly-disclosed random integer. The value $\langle j\rangle$ is the binary representation of the object identifier and D_j is the digest of the original data object computed by the writer. We implement h similarly to the construction used in MuHASH except we concatenate 5 outputs of SHA-384, truncate the first 1600 bits, and verify that $h(x)$ is in Z_M. If h is a random oracle, AdHASH is *collision-free* assuming that the weighted knapsack problem is hard [6]. This assumption might be significantly stronger than the one underlying the security of MuHASH and is not considered standard. In addition, to account for Wagner's analysis [19], the modulus M must be large – at least 2^{1600} to provide 80-bit security. The choice of M must also meet additional constraints in order for AdHASH to be collision-free [6], but selecting M is a one-time cost incurred during the initialization of FastAD.

Incrementality is achieved in AdHASH using subtraction. Using the same definitions for variables and functions as in the MuHASH, the AdHASH supports selective replacement of D_j with D'_j

$$T = Z - (h(\langle u\rangle||D_j)) \bmod M$$
$$Z' = T + h(\langle u\rangle||D'_j)) \bmod M$$

AdHASH provides superior performance than MuHASH owing to the performance cost of addition and subtraction operations when compared with multiplication and computation of the multiplicative inverse. We demonstrate the relative performance in our evaluation (Section 4).

We have described FastAD for deterministic skip lists, but the MuHASH construction can be used with B+-trees to provide the same performance benefits. For this paper, we consider workloads without deletions: insert, modify, and read only. This may be appropriate for versioning or archival systems. In the future, we will address deletion either by reorganization and compaction of the skiplist or by moving to a B+-tree.

3. PATH HINTS

Path hints allow FastAD to short-circuit an object's authenticity verification, eliminating unnecessary I/O. FastAD replicates an object's path through the directory from root-to-leaf on the storage server, which incurs a storage overhead of $\log_b n$ $\langle key, \text{digest}\rangle$ pairs. During verification, FastAD compares elements from the path with those at each level of the skiplist, stopping at the first level where the hint matches the directory entry. The hint avoids I/O to blocks containing elements from the lower levels of the skiplist. Figure 1 shows the path hint optimization were object 73 to be queried. The top three levels of the skiplist have been modified by updates to other objects after the path was replicated at the storage server. However, the digest of the L0 leaf node has not changed, which can

be determined at L1, and FastAD does not perform I/O to retrieve the L0 leaf block.

FastAD achieves these optimizations without jeopardizing the validity of the inclusion proof. As Merkle identified [15], a node in a cryptographic hash tree summarizes the contents of the node's entire sub-tree; MuHASHes and AdHASHes in a skiplist exhibit the same property. When successfully using a path hint, FastAD returns a short-circuit proof that contains only the neighbor elements from the root to the matching hint. The client completes the proof by constructing a verifying digest from leaf to root using the path hint to provide the lower level neighbor elements that are not in the short-circuit proof. Several optimizations exploit the temporal properties of workloads to improve cache performance and increase the effectiveness of path hints. Our design avoids I/O to a block when either (1) hints have not been invalidated by updates to objects associated with the path from leaf to the block, or (2) FastAD caches the block.

To keep hints valid, we update path hints opportunistically and insert new objects so that they minimally interfere with existing paths. The client updates the path hint on the storage server on every read query. FastAD returns the path information as part of the proof of membership in the directory. We also insert new objects into the directory sequentially (append only) by creating object identifiers in monotonically increasing order. Thus, subsequent inserts go into the same blocks and repeatedly modify the same path from leaf to root, which invalidates a minimum number of replicated path hints.

This design enhances cache effectiveness. Typically, the cache contains the top levels of the directory, which are used in most queries, and the hot portion of the skiplist where inserts occur. These are exactly the regions of the data structure that change most frequently and for which the path hints are least effective.

4. EVALUATION

We implemented a FastAD prototype to demonstrate the relative benefits of our optimizations. All experiments were performed on a Dell Precision T7400 workstation with a 2Ghz processor and 8GB of RAM. To implement the MuHASH and AdHASH algorithms, we use Libgcrypt's multi-precision integer library along with OpenSSL's implementation of SHA-384.

Block orientation: Our evaluation reveals that the use of the AdHASH reduces the performance cost of cryptographic operations by a factor of five and the total time (including I/O) by a factor of two when compared with pairwise hashing (Figure 3). The MuHASH is considerably slower due to the number of multiplication operations required to calculate the multiplicative inverse (Figure 4). We use a write microbenchmark that inserts 100,000 elements asynchronously (no forced I/O) using three systems: (1) FastAD with AdHASH (2) FastAD using MuHASH and (3) block-aligned pairwise hashing as described by Yumurefendi et al. [20]. We measure the time to perform cryptographic operations and the total runtime. In both implementations of FastAD, the cryptographic costs remain in constant proportion to the number of levels in the skiplist. FastAD with AdHASH requires ten hash operations, one addition, and one subtraction per level, whereas FastAD with MuHASH requires six hash operations, the calculation of the multiplicative inverse, and two multiplications to recalculate the digest for a block. A larger block size corresponds to fewer levels in the skiplist, explaining the significant initial decline and subsequent leveling of time to perform the inserts. The MuHASH is two orders of magnitude slower than the AdHASH (Figure 4), making the AdHASH our preferred cryptographic construct.

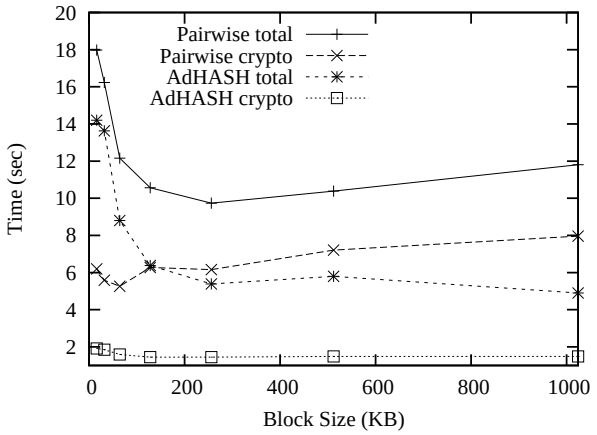

Figure 3: Insert performance (total time includes I/O).

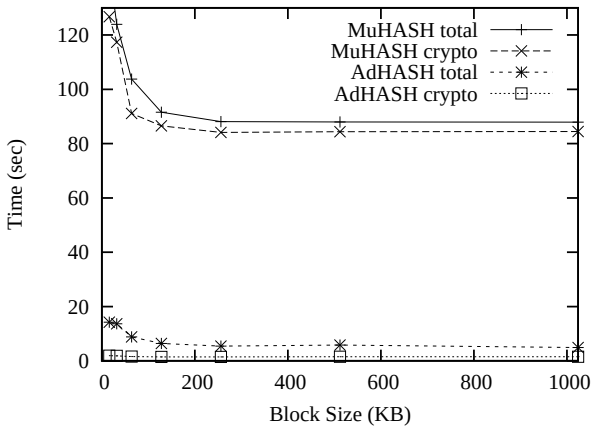

Figure 4: Insert performance (MuHash and AdHASH)

In contrast to the incremental cryptographic algorithms, the block size has no impact on the number of levels in the pairwise scheme; there are always $\log_2 n$ levels in the binary tree regardless of its organization on blocks. Pairwise cryptographic operations gain no benefit from larger block sizes. In fact, performing cryptographic operations that require accessing multiple, non-contiguous elements within a single large block results in many L1 and L2 cache misses explaining the linear increase in time for the pairwise scheme (Figure 3).

Path hinting: We compare the object verification time of lookups with and without path hints for a mixed read-write workload. Without path hints, the system must traverse the path from root to leaf for each lookup, whereas hints permit verification at higher levels of the data structure. We based our experiment on a portion of the Lair NFS trace [10]: a balanced read/write workload of 400,000 operations. In order to vary the depth of the data structure and to prevent it from fitting entirely in cache, we pre-populate a FastAD directory with a variable number of random objects before executing the operations from the trace. We distribute the objects first referenced as reads in the Lair trace randomly throughout the pre-populated directory. Prior to executing the trace, we force all blocks to disk and flush the entire directory from memory. As we execute the trace, we insert objects first referenced as writes into the directory on their first reference. For each read in the trace, we request

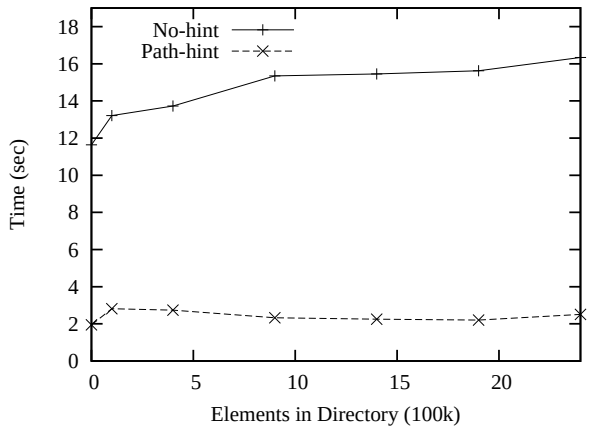

Figure 5: Impact of path hinting on lookups.

object verification and then measure the verification time. We use a 16KB block size such that 68, 230-byte objects fit in each block. We use a cache size of 10MB such that 640 blocks can reside in cache. The cache size is a thousand times smaller than a realistic server system, but our directory contains a thousand times fewer objects than our design target (millions rather than billions).

Our path hinting optimization reduced the lookup time by a factor of six (Figure 5). The initial steep increase in times corresponds to the increase in the number of levels in the skiplist. The path-hinting optimization becomes more significant as the size of the directory increases since writes to a deeper skiplist invalidate a smaller percentage of paths to blocks in the lower levels.

5. RELATED WORK

Previous skiplist implementations of authenticated directories were designed as in-memory systems using pairwise hashing [2, 8, 11, 12, 14, 18], making the cost of digest regeneration and the proof size prohibitively large as the number of elements that contribute to the hash increases. Recognizing the need for scalability, Yumerefendi et al. [20] implemented an authenticated directory that builds binary Merkle-trees within each node of a B+-tree. However, the cryptographic operations in this system are still performed pairwise and maintaining the B+-tree may incur tree rotations that recalculate hashes over substantial portions of the tree.

Zhu et al. [21] provide optimizations that minimize I/O associated with accessing an index that scales beyond memory capacity in a data deduplication system. This includes inserting data into a hot-portion of the tree to reduce the memory footprint and improve cache utilization. FastAD's append-only temporal organization was inspired by this work. They also use a Bloom filter to avoid lookup requests to objects not in the index. This technique does not apply to FastAD, because query responses from FastAD must carry a compact proof that cannot be derived from the Bloom filter.

6. CONCLUSIONS

As cloud storage becomes more popular, we anticipate that authenticated directories will emerge as the data structure of choice for tracking dynamic data. Systems will need to manage integrity, authenticity, versioning, and provenance for billions of objects. Toward these requirements, we have developed a prototype authenticated directory that maintains performance at scale by efficiently mapping the data structure to disk and by reducing I/O costs through path hinting and temporal organization. Our current prototype more than doubles overall system performance for inserts and verification performance for lookups for practical parameters when compared with existing techniques for block-aligned, out-of-core authenticated directories.

7. REFERENCES

[1] Amazon Web services Simple Storage Service. `http://aws.amazon.com/s3/`, 2009.

[2] ANAGNOSTOPOULOS, A., GOODRICH, M., AND TAMASSIA, R. Persistent authenticated dictionaries and their applications. *Information Security Conference 2200* (Oct. 2001).

[3] ARMBRUST M. ET AL. Above the clouds: A Berkeley view of cloud computing. *Technical Report No. UCB/EECS-2009-28* (Feb. 2009).

[4] BALDING, C. A question of integrity: To MD5 or not to MD5? In *cloudsecurity.org* (June 2008).

[5] BELLARE, M., GOLDREICH, O., AND GOLDWASSER, S. Incremental cryptography and application to virus protection. In *Symposium on the Theory of Computing* (1995).

[6] BELLARE, M., AND MICCIANCIO, D. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *In Eurocrypt97* (1997), Springer-Verlag, pp. 163–192.

[7] BELLARE, M., AND ROGAWAY, P. The exact security of digital signatures - how to sign with rsa and rabin. Springer-Verlag, pp. 399–416.

[8] BLIBECH, K., AND GABILLON, A. Chronos: an authenticated dictionary based on skiplists for timestamping systems. In *Workshop on Secure Web Services* (Nov. 2005).

[9] CHEN, S., GIBBONST, P. B., MOWRY, T. C., AND VALENTIN, G. Fractal prefetching b+-trees: optimizing both cache and disk performance. In *ACM SIGMOD International Conference on Management of Data* (2002), ACM Press, pp. 157–168.

[10] ELLARD, D., LEDLIE, J., MALKANI, P., AND SELTZER, M. Passive NFS tracing of email and research workloads. In *Proceedings of USENIX File and Storage Technology* (2003).

[11] GOODRICH, M., PAPAMANTHOU, C., TAMASSIA, R., AND TRIANDOPOULOS, N. Athos: Efficient authentication of outsourced file systems. In *Information Security Conference* (2008).

[12] GOODRICH, M., TAMASSIA, R., AND SCHWERIN, A. Implementation of an authenticated dictionary with skiplists and commutative hashing. In *DARPA Information Survivability Conference and Exposition* (June 2001).

[13] LEURENT, G., AND NGUYEN, P. Q. How risky is the random-oracle model? Cryptology ePrint Archive, Report 2008/441, 2008.

[14] MANIATIS, P., AND BAKER, M. Authenticated append-only skiplists. *arXiv:cs*, 0302010 (2003).

[15] MERKLE, R. A certified digital signature. *Advances in Cryptology 435* (1990), 218–238.

[16] NAOR, M., AND NISSIM, K. Certificate revocation and certificate update. *Journal on Selected Areas in Communications 18*, 4 (Apr. 1999).

[17] PUGH, W. Skiplists: A probabilistic alternative to balanced trees. *Communications of the ACM 33*, 6 (1990), 668–676.

[18] TAMASSIA, R., AND TRIANDOPOULOS, N. On the cost of authenticated data structures. In *European Symposium on Algorithms. Lecture Notes in Computer Science* (2003), vol. 2832.

[19] WAGNER, D. A generalized birthday problem. In *In CRYPTO* (2002), Springer-Verlag, pp. 288–303.

[20] YUMEREFENDI, A., AND CHASE, J. Strong accountability for network storage. *ACM Transactions on Storage 3*, 3 (Oct. 2007).

[21] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *File and Storage Technology Conference* (2008).