On-Disk Authenticated Data Structures for Verifying Data Integrity on Outsourced File Storage

Dan Rosenberg
drosenbe@cs.brown.edu

1 Introduction

Several companies now provide cost-effective, indefinitely scalable file storage in the cloud. However, the problem of verifying the integrity of data on untrusted storage sites is largely unanswered. Examining Amazon's Simple Storage Service (S3) as a case study [1], we note that there are no mechanisms incorporated into S3 that would allow a user to detect changes in stored files, due to data corruption or malicious tampering. Naive solutions, such as retaining a copy of each file for comparison, or simply maintaining checksums of each stored file, do not scale well: when dealing with very large amounts of data, even keeping a checksum for each file becomes space intensive, especially for users who seek to outsource their storage needs.

In this paper, we examine prior work in developing scalable solutions to the outsourced data integrity problem. Using insights from these works, we propose a refinement of previous techniques that would allow a completely client-side implementation that provides authenticity guarantees at minimal performance and financial cost. Notably, this solution does not require the use of an external authentication server.

The paper is structured as follows. In Section 2, we review the authenticated skip list data structure, explain how it can be used for data integrity applications, and review more recent advancements. In Section 3, we present a high-level overview of our new system. In Section 4, we describe our implementation in more detail. In Section 5, we provide initial performance measurements to support the viability of this system and assess the cost of implementation. Finally, in Section 6, we present a roadmap for future work.

2 Background

2.1 Skip Lists

The skip list is a data structure that provides an efficient way of storing a set S of objects in an ordered fashion [2]. It supports the standard data operations of GET (retrieve an element), PUT (add or modify an element), and $DELETE$ (remove an element). The skip list stores its set of elements in a series of linked lists, where the lowest list S_0 contains every element in the set ordered by key, and each element is propagated to the next list probabilistically (in our case, by simulating a coin flip). We refer to the collection of all list entries for a single key as a tower. Of these entries, we refer to the topmost entry as a plateau node, and each of the remaining entries as tower nodes. Each element contains a pointer to its neighbors to the right and in the list below. Each list level is bookended by default values known as sentinels, which we represent with the $-\infty$ and ∞ values. The top list contains only these two sentinels by definition, and the $-\infty$ sentinel in the top list is referred to as the root element.

Skip lists have the desirable property of being able to perform GET, PUT, and $DELETE$ queries in expected $O(\log n)$ time, where n is the number of elements in the set S. To search for an element s_i, we begin at the root element. Recalling that each element in the skip list has a pointer to the next element to the right and the corresponding element in the list below, we first check the element to the right of the root element. If this element's key is less than the key of s_i, then we navigate to this element. Otherwise, we descend to the next level of the skip list. By repeating this process until we reach the lowest level, we will either locate s_i or identify two consecutive elements that prove the nonexistence of s_i. The entire operation takes $O(\log n)$ time with high probability.

Insertion is performed similarly. First, the same path is traversed as in the case of a GET query to determine where in the lowest list the new element should be inserted. With each node traversed in the search, we push that node onto a stack. After identifying the appropriate insertion location, coin flips are simulated to determine how many levels up the new item should propagate, and then pointers are adjusted to insert the item, using the stack to determine which nodes require pointer correction. To perform deletion, the same search is performed, and pointers are adjusted accordingly as with insertion.

2.2 Commutative Hashing

Goodrich and Tamassia introduce the concept of commutative hashing [4]. A commutative hash function is a function that takes two arguments, maintains all properties of traditional collision-resistant hash functions, and produces the same output regardless of the ordering of its arguments. For example, given a traditional collision-resistant hash function f, a trivial commutative hash function h can be constructed such that:

$$h(x, y) = f(\min\{x, y\}, \max\{x, y\})$$

Commutative hashing conveniently simplifies schemes that use pairwise hashing to accumulate many inputs by eliminating the need for strict ordering of each pair.

2.3 Authenticated Skip Lists

Using the skip list and commutative hashing, Goodrich and Tamassia address the problem of authenticity and integrity in outsourced file storage by presenting a new data structure: the authenticated skip list [4]. An authenticated skip list functions similarly to a Merkle tree [5], where each leaf node contains the hash of a data object, and nodes are hashed together to form a directed, acyclic graph leading to a single root node that represents a digital signature on the entire data set. In our case, the lowest list of the skip list contains nodes generated using the hashes of each data object in the set S, and these hashes are propagated as depicted in Figure 2. The root hash, also known as the basis, is stored at

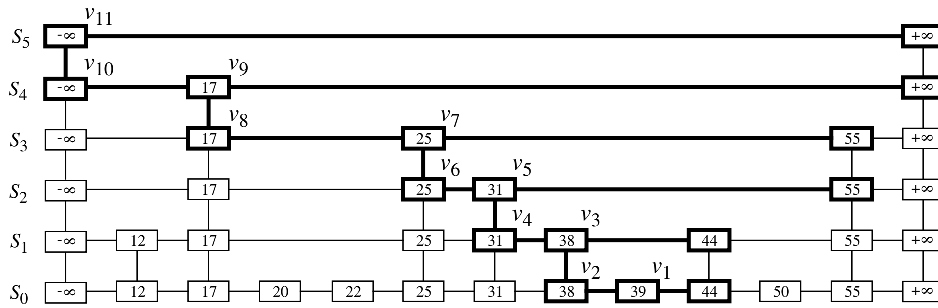

Figure 1: Search for element 39 in a skip list. The nodes visited and the links traversed are drawn with thick lines. This successful search visits the same nodes as the unsuccessful search for element 42. Skip list figures used with permission from [3]

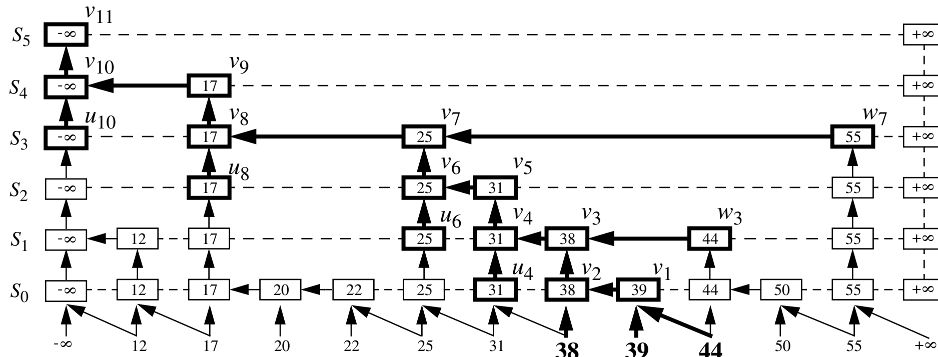

Figure 2: Hash information is propagated from the leaf nodes to a single root hash. Arrows indicate the propagation of hash information. The search path (and proof) for an authenticated *GET* query for object 39 is shown in bold.

the location of the $-\infty$ sentinel in the top list of an ordinary skip list. The basis is also stored by the client in local storage or is available as a digitally signed, trusted value.

To verify the integrity of a data element s_i, we first perform a search query for that item's key in our authenticated skip list, pushing neighboring elements along the search path onto a stack. The stack containing the neighbors of this search path constitutes a proof that enables the client to verify the integrity of that particular element. On downloading a copy of the data element, the client computes the hash of this data element, and commutatively hashes each sequential element in the proof to recalculate the basis. This newly calculated basis is compared to the expected basis stored by the client; if these bases match, the element is authenticated. This technique ensures that an adversary cannot provide a fake proof of authenticity unless a collision in the hash function has been found.

To perform an insertion or deletion, each appropriate node in the authenticated skip list must be updated to correctly propagate hash information up the structure, and the basis must be recalculated. A search for the new element is performed as described above, keeping relevant neighbors in a stack as before. After verifying that this provided search path hashes to the expected basis, the client updates each successive element to propagate hash information appropriately and recomputes its locally stored basis.

Amazon S3's *LIST* query, which returns the keys of all items matching a specified prefix, can be authenticated by returning the hashes of each element matching the specified prefix, the height of each tower associated with these elements, and proofs for *GET* queries on the items immediately before and after the matching list. With this information, the client can reconstruct the necessary portion of the skip list, verify that the returned list items commutatively hash to the expected basis, and know for certain that no elements matching the *LIST* query have been omitted.

Goodrich, Tamassia, and Schwerin present an implementation of these algorithms using a three-party model, featuring a client, an untrusted storage server, and an authentication server responsible for maintaining the authenticated skip list data structure in memory [3]. On each *GET* query, the client requests the data object from the storage server, and a corresponding proof of integrity from the authentication server. To perform a *PUT* or *DELETE* query, the client performs the desired operation on the storage server, requests information necessary to recompute its local copy of the basis, and instructs the authentication server to update its data structure.

2.4 Incremental Hashing

Stanton, McKeown, et al. present FastAD, an authenticated skip list implementation designed to minimize I/O costs to allow efficient storage of authentication data on disk [6]. They apply the technique of incremental hashing [7], which reduces the number of disk accesses required for *GET* queries by grouping authentication nodes in blocks, rather than relying on pairwise computation. An incremental hash is computed for an entire block, such that the proof size for a particular element is reduced to $O(\log_k n)$, where k is the number of elements per block. The authors evaluate performance using the MuHash and AdHash algorithms.

2.5 Path Hints

FastAD also presents a technique of using path hints to further minimize disk access. FastAD caches the most recently computed proof of authenticity for each object on the authentication server. On querying an object, this cached path hint is first retrieved, before resuming the original authentication process. If at any point in the traversal of the skip list a visited node's hash matches a hash in the path hint, the proof is "short-circuited" and no more disk I/O is incurred. Instead, the proof is concluded by using the remainder of the path hint. In order to increase the accuracy of path hints, all objects in FastAD are given incrementally increasing identifiers and inserted at the tail end of the skip list. This causes large sections of the skip list to remain stable in the face of updates, ensuring that path hints are more reliable.

3 Overview

3.1 Limitations of Previous Designs

Several limitations of previously proposed solutions motivate the design of our new scheme. Heitzmann, Palazzi, et al. present an implementation of an authenticated skip list [8] using the Jets3t Cockpit Amazon S3 client [9]. The implementation requires an additional third-party authentication server, which could be provided by the client or rented via a service such as Amazon's Elastic Compute Cloud (EC2) [10]. This additional requirement poses significant maintenance and financial overhead. In addition, it becomes impossible to accurately attribute the cause of an authentication failure to the storage server or the authentication server - failure will be detected, but it will be unclear which party is at fault. Finally, this implementation stores authentication information in memory on the external server. While this choice allows fast retrieval and updates of authentication information, the lack of persistent storage and the inability to scale to larger data sets are significant drawbacks.

FastAD addresses some of these issues, but is not without its own limitations. The FastAD system still requires an independent authentication server for verification. In order to minimize disk I/O, objects are inserted into the skip list with incremental identifiers, to ensure the stability of large portions of the data structure. This optimization allows for better path hint performance, but because objects are no longer ordered by their natural identifiers, proofs for the results of *LIST* queries are no longer computationally efficient. In addition, forcing insertion towards the tail end of the skip list makes deletion a costly operation requiring an expensive compaction algorithm. While this might be acceptable if the authentication information were stored on a server capable of computation, our model seeks to eliminate this server, requiring all computation to be performed by the client and making compaction less appealing.

3.2 Our Contributions

We make several contributions to address these limitations. Firstly, our model eliminates the need for an independent authentication server, instead requiring instrumentation solely in the client. Since our model stores everything in persistent storage, we must justify our choice of data format for storing authentication information and our preferred mechanism for retrieval of this information. We determined that on Amazon S3, using *GET* queries for files containing authentication data is significantly faster than using *LIST* queries to retrieve authentication data stored in metadata.

Next, we adopt FastAD's insight that utilizing a caching layer may significantly increase performance. We cache the most recent proof of authenticity for each object in the data set on the storage server. On querying an object, we first retrieve the path hint for the object. Then, we begin the query as usual at the root node. Whenever we encounter a node in our skip list traversal that matches a node in the path hint, we cease querying and use the remainder of the path hint as our proof. This optimization significantly reduces the number of queries we must perform on average, and in the worst case merely replaces the query for the hash of the data object itself.

FastAD orders its data elements sequentially to stabilize large sections of the skip list, making path hints more effective. Because this design decision eliminates the ability to verify *LIST* queries efficienty, we have elected to instead examine the effectiveness of caching in our setting, where data objects are ordered by key. Even without the increased stability provided by sequential insertion, caching provides worthwhile benefits in the average case.

Incremental hashing is significantly more complex when objects are sorted by key - in order to maintain a constant block size during insertion into or deletion from the middle of the skip list, a large portion of the skip list would need to be recomputed. In addition, *LIST* queries would be difficult to verify, because ranges of keys may span multiple blocks. Instead of using incremental hashing, we continue to use pairwise commutative hashing and ordering by key.

Finally, we introduce a user-defined settings that we refer to as probabilistic integrity verification, which allows the client to confirm data integrity with high probability (rather than with absolute certainty), reducing performance overhead. We note that all *PUT* and *DELETE* operations must be authenticated to maintain a consistent authenticated skip list, but *GET* and *LIST* queries may be optionally trusted without proving integrity.

4 Design Details

We have developed an implementation of our design in Java. We note that actual disk I/O operations are handled exclusively by the storage provider, and are abstracted from the client. As such, the basic cost unit we wish to minimize is the number of queries over the network, each of which is presumably associated with some reasonably executed disk operation on the storage server.

4.1 Hashing

Any candidate collision-resistant hash function may be adapted for use as a commutative hash function in our scheme. We chose to use SHA-256 for our implementation, but any secure hashing algorithm may be easily substituted. We wish to note that merely hashing the contents of a data object is not sufficient to prevent all attacks. For example, a client could request an object from a malicious storage server that returns a completely unrelated object and its proof, potentially tricking the client into accepting invalid or irrelevant data. By hashing the contents of each data object along with their specified identifiers, such attacks can be prevented.

4.2 Skip List Naming Scheme

Our implementation translates the in-memory skip list implementation directly to an on-disk data structure. Each

node of the skip list is represented by a small file on the storage server. Based on the results of preliminary experiments, we have elected to retrieve these files via *GET* queries, rather than storing all information in metadata (Section 5.1.2). The name of a node contains its corresponding key and its level in the skip list, separated by a delimiter. Each node is prefixed by a special identifier to distinguish ordinary element nodes from the left and right sentinels. The contents of each skip list file are simply the cumulative hash associated with that element, the key of that element's right neighbor, and whether or not that element is a tower or plateau node.

4.3 Path Caching

We wish to make use of a path caching scheme that is of maximum utility for queries, but requires minimal space for each cache entry. We note that a minimal proof for a given object merely consists of the neighbors along that object's search path in the skip list structure. However, a minimal proof is not sufficient for path caching, since the client will be unable to easily match this proof to the path being traversed in a search. While each neighbor element in the minimal proof could be matched individually, what we are actually interested in is whether the accumulated hash, represented by the label of a node along the search path (rather than a neighbor), matches a portion of the cached path. Therefore, our cached paths must contain not only neighbors of the search path, but the labels of the search path itself, representing this accumulation of hash information.

Each cached path is titled based on the corresponding element key, prefixed by an identifier used by all cached items to denote a path hint as opposed to a skip list node. Each path hint can be structured as follows:

$$(s_0) : (n_1), (s_1) : (n_2), (s_2) : ... : (n_k), (s_k)$$

Each s entry contains the full information for a node along the search path, and each n entry contains only the hash of a neighboring node. s_0 refers to the root of the authenticated skip list, and s_k refers to the node corresponding to the requested data object. Entries are ordered such that each entry's hash is the commutative hash of the two hashes in the next delimited group. Nodes that contain the same cumulative hash as a previous node on the search path are ommitted to save space.

After being retrieved, each path hint is parsed and kept in memory on the client machine. Comparing a traversal node to the nodes of the path hint requires $O(\log n)$ time (the size of the path hint with high probability), but this operation is performed in memory and is negligible in comparison to the network and disk I/O costs associated with fetching data from the storage server.

4.4 Exploiting Parallelism

Noting that the primary cost in authenticating outsourced files is the network I/O between the client and storage server, we can significantly reduce overhead by parallelizing as many queries as possible. For every operation (*GET*, *PUT*, *LIST*, and *DELETE*), the corresponding authentication queries are performed in a separate thread from the actual request. As a result, performance is significantly improved for operations on larger files, since authenticated these files may complete before the requested file itself is finished uploading or downloading.

Further parallelism can be used to speed up the *PUT* and *DELETE* queries. Both of these operations require nodes of the skip list along the search path to be recomputed. To optimize, we download the appropriate nodes required for authentication, recompute their hashes and neighbors locally, and upload everything simultaneously rather than sequentially, resulting in another performance boost.

4.5 Probabilistic Integrity Verification

By specifying a threshold, the user can define the proportion of *GET* and *LIST* queries that will be authenticated. Over time, integrity of all the transferred data will be confirmed with high probability. This optional threshold may be inappropriate for certain settings, such as those highly critical information, but may provide a performance boost for many typical applications relying on outsourced storage.

Assume the client sets a threshold t, between 0 and 1. Whether or not an individual query is authenticated should be decided randomly, with odds correlated to this threshold. It is important to randomize the authentication of queries, or a malicious storage server could predict which queries would be authenticated and provide incorrect responses only on unauthenticated queries.

Call c the randomly distributed proportion of responses by the storage server that contain invalid data. The probability of the client detecting that there is at least one piece of invalid data can be calculated as $P = 1 - (1 - c * t)^n$, where n is the number of queries that have been performed. Figure 3 shows the number of queries that must be performed to achieve several probabilities of failure detection with various threshold levels.

	50%	**95%**	**99%**
$t = 1/2$	139	598	919
$t = 1/5$	347	1,497	2,301
$t = 1/10$	693	2,995	4,603
$t = 1/50$	3,466	14,978	23,024
$t = 1/100$	6932	29,965	46,050
$t = 1/500$	34,658	149,786	230,257
$t = 1/1000$	69,315	299,572	460,515

Figure 3: The number of queries required to guarantee detection of corruption with specified probabilities, assuming a fixed value of $c = .01$ (1% of responses are invalid), with various threshold settings.

5 Analysis

There are two metrics by which we can evaluate the feasibility of our new authentication scheme. First, we will conduct performance evaluations supporting the viability of our contributions. Next, we will estimate the additional financial expense incurred by our design and compare to the cost of previous schemes.

5.1 Performance

5.1.1 Data Format

Each authentication node must store a key, a level in the skip list, a next pointer (the key of the element immediately to the right), a label (the hash propagated up until that point), and a state (whether or not this node is a plateau or a tower). We justify the data format for authentication data by comparing two options. The first option involves storing all authentication in the name of an empty file, and using Amazon's *LIST* query to request information as needed. The second option

instead stores only the key and level as the file name, and keeps the remaining information in the file's data.

To compare these two options, we created a simulated authenticated skip list to authenticate 10,000 uniquely keyed objects, and stored this skip list using each data format in two separate Amazon S3 buckets. Next, we manually selected 100 authenticated skip list nodes and measured the time required to retrieve all 100 nodes from S3, using *LIST* queries for the first scheme and *GET* queries for the second. We specified in our *LIST* requests that S3 should return a maximum of one result, in an attempt to eliminate unnecessary searching. We repeated this experiment 100 times, after which we computed the average retrieval time and standard deviation for each of the two data format options on our synthetic skip list. All queries were executed as REST requests via the Python S3 library provided by Amazon.

Request	Avg. Time	Std. Dev.
100 *GET* requests	8476 ms	1359 ms
100 *LIST* requests	13120 ms	3471 ms

Figure 4: Average times and standard deviations of 100 trials, each issuing 100 requests of specified type.

We found that storing only key and level information in the file name and relying on *GET* queries to retrieve the rest of the information was significantly faster on average. The two schemes require identical space requirements - Amazon S3 charges for space used, regardless of whether data is contained within a file or as metadata. Finally, *LIST* queries are more expensive per query with Amazon's pricing. Based on this evidence, storing authentication information in file data is our best option.

5.1.2 Query Performance

To assess the performance of our client-only implementation, we compare with unauthenticated operations in a realisitc setting. We created a synthetic data set of 100,000 unique items, which we then uploaded to Amazon S3 using our authentication scheme. This data set represents a baseline that allows us to simulate performance at a reasonable scale. We then measured the time required to perform 1,000 *PUT* queries for files of varying sizes: 1kb, 1mb, and 10mb. Next, we measured the time required to perform 1,000 *GET* queries on each of these newly added files. We repeated these *GET* queries twice, to demonstrate the extent to which path hints improve performance. For comparison, we repeated each of these experiments using unauthenticated operations. Our experiments were performed on an AMD Phenom II x4 955 quad-core processor running Linux 2.6.26.

From these results (Figures 5 and 6), we can determine which types of workloads would be best suited for our authentication scheme. Our experimental results confirm that there is minimal performance cost for *GET* queries that already have a corresponding path hint cached. As a result, read-intensive workloads will clearly perform best, because each *GET* request generates a fresh path hint without invalidating other paths. In fact, our path caching scheme has the desirable property that the most frequently read objects will have the most up-to-date path hints, and as such will have the best performance.

These results also confirm our intuition that workloads requiring the transfer of larger files will outperform those that transfer smaller files, in terms of additional overhead caused by authentication. Our data suggests that there is an essentially constant (relative to the size of the data set) baseline performance penalty associated with authenticating queries, and for the workloads we tested, this penalty was greater than the time required to download or upload the corresponding data object. However, for larger data objects, the authentication process will complete before finishing the transfer of the object, resulting in a much more acceptable performance penalty.

5.2 Cost

Each piece of authentication information incurs an additional storage overhead. Consider a set of data objects S with n elements. On average, the expected number of authentication nodes is approximately $2 * n$, with each node requiring approximately 48 bytes, assuming an average key length of twelve characters, use of the SHA-256 hash function, and including delimiters.

In addition to authentication data, we will assume each element contains a cached path hint. As a reasonable estimate, we assume each path hint will contain full information for $2 * \log_2 n$ nodes, which represents the expected size of a search path in a probabilistic skip list where the probability of propagating a node to the next level of the skip list is .5. As before, we will use 48 bytes as our average node size.

Amazon S3's billing charges depend on the amount of space used. At the time of this writing, Amazon's standard United States hosting charges are \$0.150 per GB for the first 50 TB, \$0.140 per GB for the next 50 TB, \$0.130 per GB for the next 400 TB, \$0.105 per GB for the next 500 TB, \$0.080 per GB for the next 4000 TB, and \$0.055 per GB for storage used over 5000 TB.

Figure 7 displays the estimated monthly storage costs incurred by our design for varying dataset sizes and average object sizes. Because the cost of our design is relative to the number of objects regardless of their size, larger average object sizes result in lower costs for our design due to better pricing rates.

In addition to storage costs, our design incurs additional charges for data transfer, which will vary with different workloads. Currently, Amazon S3 charges \$.01 per 1,000 *PUT* or *LIST* requests, \$.01 per 10,000 *GET* requests, and varying charges for the amount of data transferred in and out, depending on the total volume transferred monthly. Data transfer rates are identical to those of EC2.

Given the very small size of the average authentication node (estimated at 48 bytes), data transfer charges contribute very little to overall costs. For example, the transfer of 1 million authentication nodes via *GET* requests costs a total of \$0.08 in volume charges, with an additional \$1.00 in "per request" charges. Even with sites with very large data sets and very high workloads, we expect data transfer costs to be reasonable. For example, authenticating 1 million *GET* requests for a data set of 1 billion objects incurs an estimated additional cost of \$64.56 without the help of path hints, which may further reduce costs. Collectively, the additional expenses of storage and data transfer are negligible in comparison to the ordinary operational costs associated with maintaining actual data sets and workloads. Our design costs significantly less than previously proposed three-party solutions requiring an external authentication server.

	Authenticated PUT	Unauthenticated PUT
1 kb	8178 ms	79 ms
1 mb	9197 ms	951 ms
10 mb	10718 ms	7502 ms

Figure 5: Average time to *PUT* a single node using described method, computed from the average of 1,000 requests.

	Authenticated GET	Cached GET	Unauthenticated GET
1 kb	7326 ms	298 ms	32 ms
1 mb	7481 ms	832 ms	691 ms
10 mb	7603 ms	2870 ms	2603 ms

Figure 6: Average time to *GET* a single node using described method, computed from the average of 1,000 requests.

	1 kb	1 mb	1 gb	EC2
100 items	< $.01	< $.01	< $.01	$61.20
1,000 items	< $.01	< $.01	< $.01	$61.20
10,000 items	< $.01	< $.01	< $.01	$61.20
100,000 items	$.02	$.02	$.02	$61.20
1,000,000 items	$.28	$.28	$.26	$61.20
10,000,000 items	$3.25	$3.25	$2.28	$244.80
100,000,000 items	$36.98	$36.98	$13.56	$864.00
1,000,000,000 items	$414.37	$386.74	$151.93	N/A

Figure 7: Total additional monthly storage costs of our design, based on number of data objects and average data object size. Monthly cost of renting an EC2 server with sufficient resources to store authentication data in memory is provided for comparison.

6 Conclusion

We have presented the design details for a two-party model for authenticating data in the cloud. Our design requires no instrumentation beyond the client, and attempts to minimize both performance and financial cost. By adapting insights from previous work, we are able to optimize performance while maintaining the flexibility of being able to perform efficient *LIST* queries. We have proposed the features of probabilistic integrity verification, and caching in a sorted-key setting.

Our experimental results demonstrate that our scheme is not without a significant performance cost, but that *GET* queries on objects that have been cached perform nearly as well as unauthenticated queries. The highest priority for future work should be further improving performance. In particular, exploring alternative data structures in this setting, especially those that allow for "fat nodes" containing larger fragments of the search tree, might minimize costs by reducing the number of queries required to return authentication information. Additional optimizations might include hashing downloaded data objects in pieces as they arrive, rather than hashing the entire object in one pass. Our implementation represents a promising starting point that may lay the groundwork on which future enhancements may be incorporated.

References

[1] Amazon s3 (simple storage service), `http://aws.amazon.com/s3`.

[2] W. Pugh, Skip lists: A probabilistic alternative to balanced trees, 1990.

[3] M. T. Goodrich, R. Tamassia, and A. Schwerin, Implementation of an authenticated dictionary with skip lists and commutative hashing, in *DARPA Information Survivability Conference and Exposition*, pp. 68–82, IEEE Computer Society Press, 2001.

[4] M. T. Goodrich and R. Tamassia, Tech. Rep., Johns Hopkins Information Security Institute Report No., , 2001 (unpublished).

[5] R. C. Merkle, A certified digital signature scheme, pp. 218–238, 1990.

[6] P. T. Stanton, B. McKeown, R. Burns, and G. Ateniese, Fastad: An authenticated directory for billiions of objects, in *Proceedings of the 1st Workshop on Hot Topics in Storage and File Systems*, 2009.

[7] M. Bellare and D. Micciancio, A new paradigm for collision-free hashing: incrementality at reduced cost, in *In Eurocrypt97*, pp. 163–192, Springer-Verlag, 1997.

[8] A. Heitzmann, B. Palazzi, C. Papamanthou, and R. Tamassia, Efficient integrity checking of untrusted network storage, in *Proc. ACM CCS Int. Workshop on Storage Security and Survivability (STORAGESS)*, pp. 43–54, 2008.

[9] Jets3t, an open source java toolkit for amazon s3, `http://jets3t.s3.amazonaws.com`.

[10] Amazon ec2 (elastic compute cloud), `http://aws.amazon.com/ec2`.