On the Cost of Persistence and Authentication in Skip Lists*

Michael T. Goodrich[1], Charalampos Papamanthou[2], and Roberto Tamassia[2]

[1] Department of Computer Science, University of California, Irvine
[2] Department of Computer Science, Brown University

Abstract. We present an extensive experimental study of authenticated data structures for dictionaries and maps implemented with skip lists. We consider realizations of these data structures that allow us to study the performance overhead of authentication and persistence. We explore various design decisions and analyze the impact of garbage collection and virtual memory paging, as well. Our empirical study confirms the efficiency of authenticated skip lists and offers guidelines for incorporating them in various applications.

1 Introduction

A proven paradigm from distributed computing is that of using a large collection of widely distributed computers to respond to queries from geographically dispersed users. This approach forms the foundation, for example, of the DNS system. Of course, we can abstract the main query and update tasks of such systems as simple data structures, such as distributed versions of dictionaries and maps, and easily characterize their asymptotic performance (with most operations running in logarithmic time). There are a number of interesting implementation issues concerning practical systems that use such distributed data structures, however, including the additional features that such structures should provide. For instance, a feature that can be useful in a number of real-world applications is that distributed query responders provide *authenticated responses*, that is, answers that are provably trustworthy. An authenticated response includes both an answer (for example, a yes/no answer to the query "is item x a member of the set S?") and a proof of this answer, equivalent to a digital signature from the data source.

Given the sensitive nature of some kinds of data and the importance of trustworthy responses to some kinds of queries, there are a number of applications of authenticated data structures, including scientific data mining [3, 13, 16, 17], geographic data servers [10], third party data publication on the Internet [5], and certificate revocation in public key infrastructure [1, 4, 7, 11, 14, 20, 21].

* This work was supported in part by IAM Technology, Inc. and by the National Science Foundation under grants IIS–0324846 and CCF–0311510. The work of the first author was done primarily as a consultant to Brown University.

C. Demetrescu (Ed.): WEA 2007, LNCS 4525, pp. 94–107, 2007.

For example, in the CalSWIM project (see calswim.org), with whom we are collaborating, data providers allow users to query their water monitoring data and create publishable summaries from such queries. Because of the high stakes involved, however, in terms of both health and politics, data providers and data users want to be assured of the accuracy of the responses returned by queries. Thus, data authentication is a critical additional feature of some data querying applications. Another feature that is also of use in such applications (e.g., for non-repudiation) is the efficient implementation of persistence [6], which allows users to ask questions of a previous version of the data structure. That is, a persistent data structure can provide answers to the queries "was item x a member of the set S at time t?". For example, in the CalSWIM application, a user might wish to compare the level of a given water constituent today with its level several months ago.

In this paper, we study the features of persistence and authentication, and report on results concerning various implementation choices. In particular, we study the relative costs of adding persistence and authentication to dictionaries and maps, answering the question of how much overhead is needed in order to provide this extra functionality that is called for in some applications (like CalSWIM). Also we investigate various implementation decisions (e.g., using different primitive data structures, such as arrays and pointers, to implement more complex data structures) concerning dictionaries and maps. We focus on implementations based on the skip list data structure [26], as this simple data structure has been proven empirically to be superior in practice to binary search trees. In addition, we also show results concerning more "system-oriented" implementation issues, such as the influence of the garbage collector in our experiments, some system limitations concerning scaling of the performance in terms of memory usage and also the effect of virtual memory paging on the performance.

Concerning related previous work, as an experimental starting point, we note that Pugh [26] presents extensive empirical evidence of the performance advantages of skip lists over binary search trees. To review, the *skip list* data structure is an efficient means for storing a set S of elements from an ordered universe. It supports the operations $\mathsf{find}(x)$ (determine whether element x is in S), $\mathsf{insert}(x)$ (insert element x in S) and $\mathsf{delete}(x)$ (remove element x from S). It stores a set S of elements in a series of linked lists S_0, S_1, S_2,..., S_t. The base list, S_0, stores all the elements of S in order, as well as sentinels associated with the special elements $-\infty$ and $+\infty$. Each successive list S_i, for $i \geq 1$, stores a sample of the elements from S_{i-1}. To define the sample from one level to the next, we choose each element of S_{i-1} at random with probability $\frac{1}{2}$ to be in the list S_i. The sentinel elements $-\infty$ and $+\infty$ are always included in the next level up, and the top level, t, is maintained to be $O(\log n)$ and contains only the sentinels w.h.p. We therefore distinguish the node of the top list S_t storing $-\infty$ as the *start node* s. An element that exists in S_{i-1} but not in S_i is said to be a *plateau* element of S_{i-1}. An element that is in both S_{i-1} and S_i is said to be a *tower* element in S_{i-1}. Thus, between any two tower elements, there are some plateau elements. Pugh advocated an *array-based* implementation of skip lists [26], in C, and this

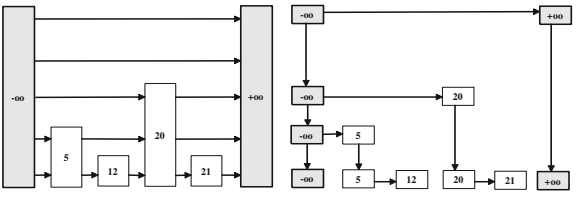


Fig. 1. Implementation of a skip list with arrays (left) and pointers (right)

is also the skip-list implementation used in the LEDA C++ library [19]. In this implementation, each element x in the set is represented with an array A_x of size equal to the height of the tower that corresponds to x. Each entry, $A_x[i]$, of the array A_x holds local data (for example in the authenticated data structures we store authentication information that correspond to $A_x[i]$) and a pointer to the array A_y, such that for all z such that $x < z < y$, the size of the array A_z is less than i. All the usual dictionary operations take time $O(\log n)$ with high probability and run quite fast in practice as well.

Using arrays for the towers is not the only way to implement skip lists, however. For example, there is an alternative pointer-based implementation of skip lists that uses nodes linked by pointers (see, e.g., [9]). We can optimize this implementation further, in fact, by taking advantage of the observation that for the operations supported by the skip list, some of the nodes and links are unnecessary and can be omitted. Then the skip list is somewhat like a binary tree (whose root is the highest-level node with value $-\infty$). In Figure 1 we illustrate two different implementations of a skip list storing the set $S = \{5, 12, 20, 21\}$, one array-based and the other pointer-based.

In addition to other studies of skip lists themselves (e.g., see [12, 15, 24, 25]), a considerable amount of prior work exists on authenticated data structures (e.g., see [2, 5, 7, 8, 10, 11, 20, 21, 27, 28]). In [8], the implementation of an authenticated dictionary with skip lists and commutative hashing is presented, again using arrays to implement the towers, but now using Java as the implementation language. In [2], a persistent version of authenticated dictionary is presented, where the user can now validate the historic membership of an element in a set, e.g., authenticated answers to the queries "was element e present in the data set S at time t?" Authenticated data structures for graph and geometric searching are presented in [10]. In [18], it is shown that almost any data structure can be converted into an authenticated one. In [22], alternate verification mechanisms are discussed. The authentication of simple database queries is studied in [23].

The main contribution of this paper is the empirical study of a number of important issues regarding the implementation of authenticated data structures using skip lists. In particular, we address the following questions: (*i*) the overheads for authentication and/or persistence; (*ii*) the relative costs of these various features and implementation decisions (e.g., which is more expensive–authentication or persistence?); (*iii*) the relative costs between updates and queries in authenticated skip lists; (*iv*) the differences between implementing

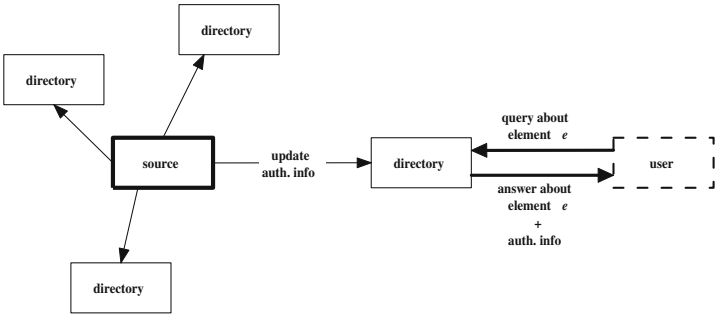


Fig. 2. Schematic illustration of an authenticated data structure

skip lists with arrays and pointers; (*v*) the impact of garbage collection on the performance (especially for fast $O(\log n)$-time operations); and (*vi*) the system limitations for implementing authenticated skip lists, including maximum memory constraints and the impacts of paging in virtual memory. We give extensive experiments that address all of the above questions. In addition, we show how to extend the notion of a persistent authenticated *dictionary* [2] to a persistent authenticated *map*, where each key is bound to a certain value and we present an efficient implementation of this new feature.

2 Implementing Authenticated Dictionaries and Maps

An authenticated data structure involves three types of parties: a trusted *source*, one or more untrusted *directories*, and one or more *users* (see Figure 2). The *source* maintains the original version of the data structure S by performing insertions and deletions over time. A *directory* maintains a copy of the data structure and periodically receives time-stamped updates from the source plus signed statements about the updated version of the data structure S. A user contacts a directory and performs membership queries on S of the type "is element e present in S?" (authenticated dictionary) or of the type "was element e present in S at time t?" (persistent authenticated dictionary) or finally of the type "what value was element e bound to at time t?" (persistent authenticated map). The contacted directory returns to the user an *authenticated response*, which consists of the answer to the query together with the *answer authentication information*, which yields a cryptographic proof of the answer assembled by combining statements signed by the source with data computed by the directory. The user then verifies the proof by relying solely on its trust in the source and the availability of public information about the source that allows the user to check the source's signature. The design of authenticated data structures should address the following goals:

– *Low computational cost:* The computations performed internally by each entity (source, directory, and user) should be simple and fast.

- *Low communication overhead:* source-to-directory communication (update authentication information) and directory-to-user communication (answer authentication information) should be kept as small as possible.
- *High security:* the authenticity of the answers given by a directory should be verifiable with a high degree of certainty.

Regarding the computational cost, an authenticated data structure should efficiently support the following operations: updates (insertions and deletions) executed by the source and directories, queries executed by directories, and verifications executed by the users. Various efficient realizations of authenticated data structures have been designed that achieve $O(\log n)$ query update, and verification time on a dictionary. In this paper, we consider authenticated data structures based on skip lists [2, 8, 28].

2.1 Authenticated Dictionary

We consider an *authenticated dictionary* for a set S with n items consists of the following components:

- A skip list data structure storing the items of S.
- A collection of values $f(v)$ that label each node v of the skip list.
- A statement signed by the source consisting of the timestamp of the most recent label $f(s)$ of the start node of the skip list. We recall that s is the left-uppermost node of the skip list.

Let h be a commutative cryptographic hash function [8]. We recall that if g is a collision resistant cryptographic hash function[1] then the respective commutative cryptographic hash function h is defined as [8]

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

We use h to compute the label $f(v)$ of each node v in the skip list, except for the nodes associated with the sentinel value $+\infty$. These labels are set to 0. The value $f(s)$ stored at the start node, s, represents a digest of the entire skip list. Intuitively, each label $f(v)$ accumulates the labels of nodes below v, possibly combined with the labels of some nodes to the right of v.

We present this computation by using the pointer-based implementation,[2] where each node v stores two pointers, $u = \mathsf{down}(v)$ and $w = \mathsf{right}(v)$. We distinguish the following cases:

- if v is on the base level, then $f(v) = h(\mathsf{elem}(v), x)$, where x is the element of the node following v on the base level (this node may or may not have a right pointer from v);

[1] This means that it is computationally infeasible to find two inputs x_1 and x_2 such that $g(x_1) = g(x_2)$ and to invert the function.

[2] Note that the implementation of an authenticated skip list mentioned in [8] is array-based.

– if v is not on the base level, then
  - if $w = \mathsf{null}$, then $f(v) = f(u)$.
  - if $w \neq \mathsf{null}$ (w is a plateau node), then $f(v) = h(f(u), f(w))$.

When we update an authenticated dictionary we need to take into account the labels of the nodes. Consider an insertion (see Figure 3(a)). We start by performing an insertion into the underlying skip list. Next, we recompute the affected labels of the nodes of the skip list as well. It turns out that all we have to do is to update the labels of the nodes belonging to the path P that we traverse when we search for the element we want to insert. The length of this path is $O(\log n)$ with high probability. The importance of the authenticated data structures comes in the verification of the result. When the user queries the directory about the membership of an element x in the dictionary, the directory returns the answer and the answer authentication information, which consists of the signed digest that was received from the source and a sequence of values stored along the search path for x (see Figure 3(a)). This sequence has length $O(\log n)$. The users verifies first the signature of the signed digest. If this verification succeeds, the user recomputes the digest, by iteratively hashing the sequence of values, and compares this computed digest with the signed digest. For more details on authenticated data structures and their implementation with skip lists, see [8, 2].

2.2 (Persistent) Authenticated Maps and Dictionaries

An *authenticated map* data structure is an authenticated data structure where each key k is bound to a certain value v. The authentication information in this case should also take into account the association between keys and values.

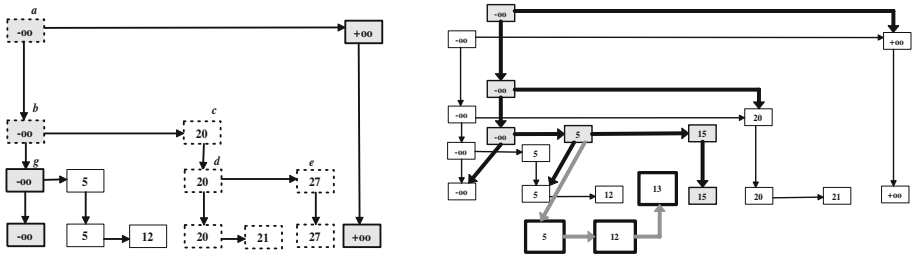


Fig. 3. (a) Insertion of element 27 in the skip list of Figure 1. The labels (hash values) of the dotted nodes change. The answer authentication information for a query on element 21 is the sequence $(27, 21, 20, f(e), f(g))$. (b) The insertion of element 15 creates a new version of the persistent authenticated dictionary. The insertion of element 13 is made next on this version, without creating a new version. Note that in this case we copy only the nodes 5, 12 (that were originally created when the very first version was instantiated) and create the large nodes 5 and 12. Note that in the final version. the node with element 5 will have only the gray down pointer.

We can build an authenticated map on top of an authenticated dictionary by storing the value together with the key only at the nodes of the base level. The authentication information for the authenticated map is computed as follows: if v is on the base level, then $f(v) = h(h(\mathsf{elem}(v), \mathsf{value}(v)), h(x, y))$, where x and y are the key and value of the node following v on the base level (this node may or may not have a right pointer from v); else the authentication information is computed exactly in the same way as in the authenticated dictionary. Regarding the query authentication information, the proof returned by the directory also contains key-value pairs. Suppose, for example, the skip list of Figure 3(a) includes the (key,value) pairs $(27, \alpha)$, $(21, \beta)$ and $(20, \gamma)$. Then the query authentication information for a query on key 21 would be the sequence $\{(27, \alpha), (21, \beta), (20, \gamma), f(e), f(g)\}$.

Additionally, a *persistent authenticated dictionary* [2] is an authenticated dictionary that supports also queries on past versions of the data structure of the type "was element e present in the dictionary at time t?". We can implement a persistent authenticated dictionary by modifying the implementation of an authenticated dictionary such that every update (insertion or deletion) creates a new version of the data structure. As shown in [2], to implement this update, we only have to copy nodes that belong to the search path of the skip list. This is because the authentication information of only these nodes differs for the two successive versions of the data structure. In [2], the insertions/deletions always create a new version of the data structure. We have implemented a more versatile form of persistence that supports also updates to the current version of the data structure, without creating a new version. In Figure 3(b), we illustrate the insertion algorithm. The insertion of element 15 creates a new version of the data structure and then the insertion of element 13 is done on the same version. Note that during the latter insertion, we only have to copy nodes that were originally created when a previous version of the data structure was instantiated.

3 Experimental Results

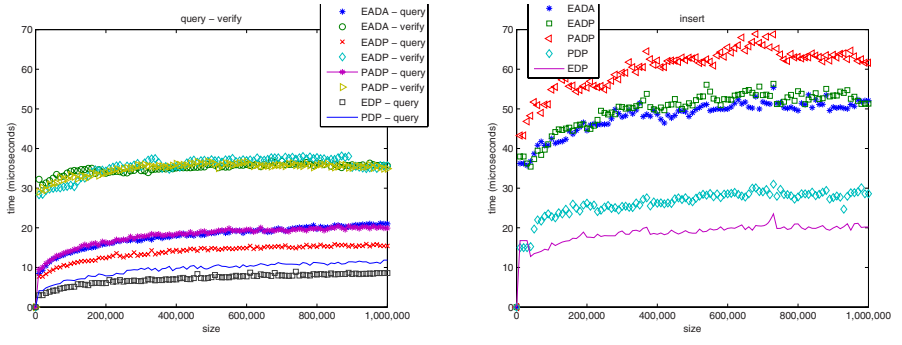
We have conducted experiments on the performance of various types of skip lists (ephemeral, persistent, and/or authenticated) containing up to 1,000,000 randomly generated 256-bit integers (for dictionaries) or pairs of two randomly generated 256-bit integers (for maps). For each operation, the average running time was computed over 10,000 executions. The experiments were conducted on a 64-bit, 2.8GHz Intel based, dual-core, dual processor machine with 8GB main memory and 2MB cache, running Debian Linux 3.1 with Linux kernel 2.6.15 and using the Sun Java JDK 1.5. The Java Virtual Machine (JVM) was most of the times launched with a 7GB maximum heap size. Cryptographic hashing was performed using the standard Java implementation of the MD5 algorithm.

We report the running times obtained in our experiments separating the time consumed by the garbage collector. Our experimental results provide a measure of the computational overhead caused by adding persistence and/or authentication to a dictionary implemented with skip lists. Also, they show some advantage

in relative performance for some operations (e.g., queries) between pointer-based and array-based implementations. The running times reported also exclude the time for signing the digest by the source and the time for verifying the signature on the digest by the user. Each of these times is about 5 milliseconds for 1,024 bit RSA signatures on MD5 (128 bit) digests using the standard java.security.∗ package. In typical applications, multiple updates are performed by the source during a time quantum. Thus, the signature creation time by the source should be amortized over all such update operations. Similarly, the user typically verifies multiple answers during a time quantum and thus the signature verification time should be amortized over all such verification operations.

3.1 Authenticated Dictionary

We first compare five different data structures based on the *dictionary* abstract data type and supporting yes/no membership queries: (a) *Ephemeral Authenticated Dictionary Array-based* (*EADA*), (b) *Ephemeral Authenticated Dictionary Pointer-based* (*EADP*), (c) *Persistent Authenticated Dictionary Pointer-based* (*PADP*), (d) *Persistent Dictionary Pointer-based* (*PDP*), (e) *Ephemeral Dictionary Pointer-based* (*EDP*). The results of the experiments are summarized in Figure 4, where we do not take into account the time taken by the garbage collector, which is discussed in Section 3.3. We discuss first the running times for pointer-based implementation and then consider array-based implementations.



(a) Query and verification times.

(b) Insertion time.

Fig. 4. Query, verification and insertion times (in microseconds) for ephemeral, persistent, and authenticated dictionaries implemented with skip lists, excluding the time for garbage collection. The times shown are the average of 10,000 executions on dictionary sizes varying from 0 to 10^6 elements.

Regarding queries (Figure 4(a)), the overhead due to persistence is less than the one due to authentication. See, e.g., the running times of PDP vs. EADP. This is explained by the fact that a query in an ephemeral authenticated dictionary has to assemble the proof by retrieving a logarithmic number of hash

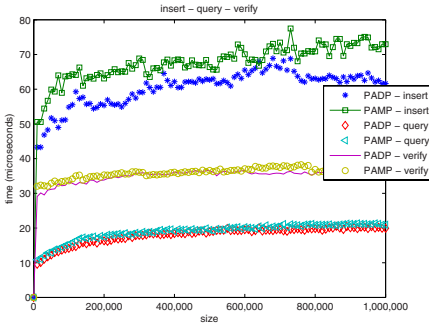
values located at nodes adjacent to the nodes encountered in the search (but does not perform any hash computation). Also, every query (whether unsuccessful or successful) must reach the bottom level of the skip list. Conversely, in a persistent (non authenticated) dictionary, a query does not have to assemble any proof and, if successful, may stop before reaching the bottom level. Adding both persistence and authentication (see the times for PADP) combines the two overheads, as expected. It should be also noted that the running time of verification operations (Figure 4(a)) does not depend on the data structure since in all cases, a verification consists of executing a sequence of cryptographic hash computations. The verification time is proportional to the size of the authentication information. Our experiments indicate that the number of values in the authentication information is about $1.5 \log n$, thus confirming the analysis in [28].

Regarding insertions (Figure 4(b)), we observe the same relative performance of the various data structures as for queries. However, the authentication overhead is now more significant since an update of an authenticated data structure requires a logarithmic number of cryptographic hash computations. Finally, we observe that for query operations array-based skip lists significantly underperform pointer-based ones. This is due to the fact that a query in the array-based implementation must traverse one by one all the levels, whereas in the pointer-based implementation, it can skip over multiple levels using pointers. In particular, the performance penalty of the array-based implementation is comparable to the authentication overhead, as can be seen in the running times for query operations in EADA vs. PADP. This can be explained by the fact that access to an array element in Java requires one level of indirection and that the JVM always checks that array indices are within bounds. For other operations, the array-based implementation has almost the same performance as the pointer-based implementation.

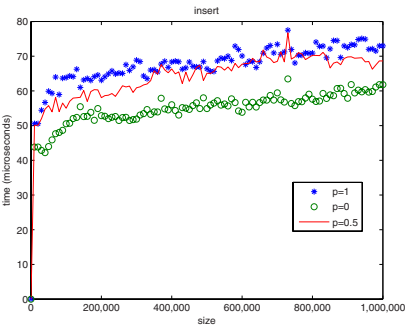
3.2 Authenticated Map

We also present experiments executed on persistent authenticated maps and dictionaries implemented with pointer-based skip lists (PAMP and PADP). The results of these experiments are summarized in Figure 5, where we do not take into account the time taken by the garbage collector, which is discussed in Section 3.3. Figure 5(a) compares the performance of persistent authenticated dictionaries with that of persistent authenticated maps. The experiments show that the additional functionality of the map abstract data type requires a small time overhead. For queries, the overhead is virtually nonexistent since searches in maps and dictionaries involve essentially the same computation. For insertion operations, the overhead is due to the need to perform some hash computations over larger data (the search key and its associated value).

In Figure 5(b) we present experiments on the running time of insertions in a variation of authenticated persistent maps such that a new version is created every $1/p$ insertions, where parameter p $(0 \leq p \leq 1)$ denotes the frequency of insertions. For example, frequency $p = 0.25$ implies that 10,000 insertions cause



(a) Query, insertion and verification times.



(b) Insertion time in a persistent authenticated map for various version creation frequencies.

Fig. 5. Query, insertion, and verification times for persistent authenticated dictionaries and maps implemented with pointer-based skip lists, excluding the time for garbage collection. The times shown are the average of 10,000 executions on dictionary sizes varying from 0 to 10^6 elements.

the creation of 2,500 versions, frequency $p = 1$ denotes the original persistent authenticated map, and frequency $p = 0$ denotes an ephemeral map. Thus, in a series of m insertions, pm insertions create a new version while the remaining $(1 - p)m$ versions are executed on the current version. This variation of a persistent map models common applications where updates are accumulated and periodically executed in batches (e.g., daily expiration/revocation of access credentials). The chart of Figure 5(b) shows how the insertion time increases with the version creation frequency p.

Figure 6 summarizes the deletion time in persistent and ephemeral authenticated dictionaries and maps implemented with pointers. We can see that the deletion times are similar to the insertion times. Indeed, both delete and insert operations on the skip list involve an initial search, followed by pointers updates and recomputation of hash values along the search path.

3.3 Garbage Collector

In the charts of Figures 4–6, we have subtracted from the running times, the time consumed by the garbage collector (GC). The garbage collector is a thread of the JVM that periodically attempts to reclaim memory used by objects that will no longer be accessed by the running program.The exact schedule of the garbage collector cannot be completely controlled. Even if we force the garbage collector to run before and after a series of 10,000 operations (by explicitly calling System.gc()), we found that it also runs during the time interval that such operations are performed. Additionally, the garbage collector cannot not be turned off in the current version (JDK 1.5) of the Sun JVM. In our attempt

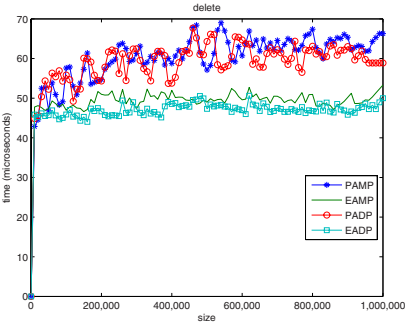
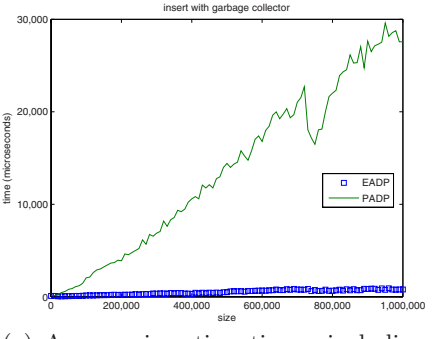
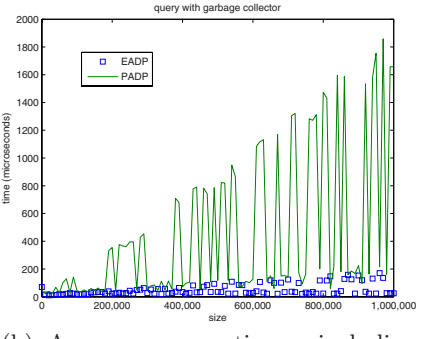


Fig. 6. Deletion times for authenticated (ephemeral and persistent) dictionaries and maps implemented with pointer-based skip lists, excluding garbage collection time



(a) Average insertion times, including garbage collection, over runs of 10,000 operations.



(b) Average query times, including garbage collection, over runs of 10,000 operations..

Fig. 7. The effect of the garbage collector on the insertion and query times on a persistent authenticated dictionary and an ephemeral authenticated dictionary

to reduce the execution of the garbage collector, we tried to tune it by using some JVM invocation options, such as

$$-\mathsf{XX} : +\mathsf{UseAdaptiveSizePolicy}, -\mathsf{XX} : \mathsf{MaxGCPauseMillis} = \mathsf{a}, -\mathsf{XX} : \mathsf{GCTimeRatio} = \mathsf{b},$$

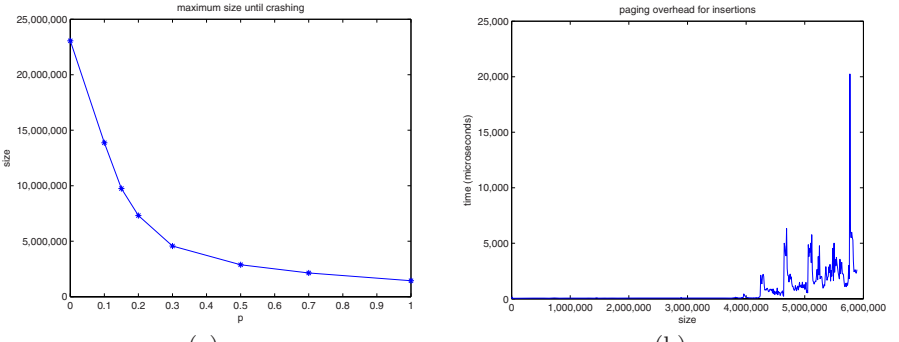
that define the ratio of the execution times of the garbage collector and application. However we did not notice any important difference when we used the option $-\mathsf{XX} : \mathsf{MaxGCPauseMillis} = \mathsf{a}$ and we noticed only minor differences for the option $-\mathsf{XX} : \mathsf{GCTimeRatio} = \mathsf{b}$, such as a slightly lower frequency of execution of the garbage collector vs. the application, combined with more time per execution. Therefore, using these options did not influence the performance of the data structure. The behavior of the garbage collector is depicted in Figure 7.

The experiments show that the work of the garbage collector affects unevenly runs of 10,000 operations (both insertions (Figure 7(a)) and queries

(Figure 7(b))). Also the effect is more obvious in the case of the persistent authenticated dictionary, compared with the ephemeral authenticated dictionary. As the size of the data structure increases, the amount of time consumed by the garbage collector also increases. Also, since the frequency of garbage collection sweeps is constant over time, we have that the garbage collector overhead is greater for insertions (longer operations) than for queries (shorter operations).

3.4 System Limitations and Virtual Memory

In an attempt to see how our implementation scales up to large data sets, we have performed a series of insertions on the data structure until we get a Java out-of-memory exception. In this experiment, the JVM was launched with 8GB maximum heap size (we use all the memory we can before the system is forced to swap to the disk). Note that the JVM could have been launched with a lot more than 8GB maximum heap size since we use a 64-bit machine. In Figure 8(a), we can see how the version creation frequency p influences the exact size that can be handled by the JVM. As expected, the maximum size of the data structure decreases as the version creation frequency p grows. Note that for



(a) (b)

Fig. 8. (a) Maximum size (until machine crashes) of the persistent authenticated map as a function of the version creation frequency p. (b) Time overhead due to paging for insertions in a persistent authenticated map with version creation frequency $p = 0.5$.

$p = 0$ (ephemeral authenticated map), we can store up to 26 million items whereas for $p = 0.5$ we can store up to about 3 million items. Finally we show the influence of virtual memory on performance. To avoid an "out of memory" exception, we launch the JVM with more than 8GB maximum heap size so that the system will eventually resort to paging to disk. We show in Figure 8(b) how paging affects the performance of our persistent authenticated map. Namely, for $p = 0.5$, paging starts to impair performance when the map size reaches 3.5 million items.

4 Conclusions

In this paper, we present extensive experiments on authenticated data structures that are implemented with a skip list. We address implementation issues concerning ephemeral and persistent authenticated dictionaries and maps and we show that authenticated skip lists are quite efficient. We show that there are low overheads for adding authentication and persistence to distributed skip lists and extending authenticated dictionaries to become authenticated maps. We finally note that the overheads involved for garbage collection and virtual memory paging are not as significant as one might at first believe they would be.

Acknowledgments

The authors are grateful to Aris Anagnostopoulos for his contributions to an initial implementation of persistent authenticated dictionaries. We would like to thank also Nikos Triandopoulos and Danfeng Yao for useful discussions.

References

[1] Aiello, W., Lodha, S., Ostrovsky, R.: Fast digital identity revocation. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, Springer, Heidelberg (1998)
[2] Anagnostopoulos, A., Goodrich, M.T., Tamassia, R.: Persistent authenticated dictionaries and their applications. In: Davida, G.I., Frankel, Y. (eds.) ISC 2001. LNCS, vol. 2200, pp. 379–393. Springer, Heidelberg (2001)
[3] Brunner, R.J., Csabai, L., Szalay, A.S., Connolly, A., Szokoly, G.P., Ramaiyer, K.: The science archive for the Sloan Digital Sky Survey. In: Proc. Astronomical Data Analysis Software and Systems Conference V (1996)
[4] Buldas, A., Laud, P., Lipmaa, H.: Eliminating counterevidence with applications to accountable certificate management. Journal of Computer Security 10(3), 273–296 (2002)
[5] Devanbu, P., Gertz, M., Martel, C., Stubblebine, S.G.: Authentic data publication over the Internet. Journal of Computer Security 11(3), 291–314 (2003)
[6] Driscoll, J.R., Sarnak, N., Sleator, S., Tarjan, R.E.: Making data structures persistent. In: Proc. ACM Sympos. on the Theory of Computing, pp. 109–121 (1986)
[7] Gassko, I., Gemmell, P.S., MacKenzie, P.: Efficient and fresh certification. In: Imai, H., Zheng, Y. (eds.) PKC 2000. LNCS, vol. 1751, pp. 342–353. Springer, Heidelberg (2000)
[8] Goodrich, M.T., Tamassia, R.: Implementation of an authenticated dictionary with skip lists and commutative hashing. In: Proc. DARPA Information Survivability Conference & Exposition II (DISCEX II), pp. 68–82. IEEE Press, New York (2001)
[9] Goodrich, M.T., Tamassia, R.: Data Structures and Algorithms in Java, 4th edn. John Wiley & Sons, New York (2006)
[10] Goodrich, M.T., Tamassia, R., Triandopoulos, N., Cohen, R.: Authenticated data structures for graph and geometric searching. In: Joye, M. (ed.) CT-RSA 2003. LNCS, vol. 2612, pp. 295–313. Springer, Heidelberg (2003)

[11] Gunter, C., Jim, T.: Generalized certificate revocation. In: Proc. 27th ACM Symp. on Principles of Programming Languages, pp. 316–329 (2000)
[12] Hanson, E.N.: The interval skip list: a data structure for finding all intervals that overlap a point. In: Dehne, F., Sack, J.-R., Santoro, N. (eds.) WADS 1991. LNCS, vol. 519, pp. 153–164. Springer, Heidelberg (1991)
[13] Karp, R.M.: Mapping the genome: Some combinatorial problems arising in molecular biology. In: Proc. ACM Symp. on the Theory of Computing, pp. 278–285 (1993)
[14] Kaufman, C., Perlman, R., Speciner, M.: Network Security: Private Communication in a Public World. Prentice-Hall, Englewood Cliffs (1995)
[15] Kirschenhofer, P., Prodinger, H.: The path length of random skip lists. Acta Informatica 31, 775–792 (1994)
[16] Lupton, R., Maley, F.M., Young, N.: Sloan digital sky survey. `http://www.sdss.org/sdss.html`
[17] Lupton, R., Maley, F.M., Young, N.: Data collection for the Sloan Digital Sky Survey—A network-flow heuristic. Journal of Algorithms 27(2), 339–356 (1998)
[18] Martel, C., Nuckolls, G., Devanbu, P., Gertz, M., Kwong, A., Stubblebine, S.G.: A general model for authenticated data structures. Algorithmica 39(1), 21–41 (2004)
[19] Mehlhorn, K., Näher, S.: LEDA: A Platform for Combinatorial and Geometric Computing. Cambridge University Press, Cambridge (2000)
[20] Micali, S.: Efficient certificate revocation. Technical Report TM-542b, MIT Laboratory for Computer Science (1996)
[21] Naor, M., Nissim, K.: Certificate revocation and certificate update. In: Proc. 7th USENIX Security Symposium, Berkeley pp. 217–228 (1998)
[22] Nuckolls, G.: Verified query results from hybrid authentication trees. In: DBSec, pp. 84–98 (2005)
[23] Pang, H., Tan, K.-L.: Authenticating query results in edge computing. In: Proc. Int. Conference on Data Engineering, pp. 560–571 (2004)
[24] Papadakis, T., Munro, J.I., Poblete, P.V.: Average search and update costs in skip lists. BIT 32, 316–332 (1992)
[25] Poblete, P.V., Munro, J.I., Papadakis, T.: The binomial transform and its application to the analysis of skip lists. In: Spirakis, P.G. (ed.) ESA 1995. LNCS, vol. 979, pp. 554–569. Springer, Heidelberg (1995)
[26] Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM 33(6), 668–676 (1990)
[27] Tamassia, R.: Authenticated data structures. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 2–5. Springer, Heidelberg (2003)
[28] Tamassia, R., Triandopoulos, N.: Computational bounds on hierarchical data processing with applications to information security. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 153–165. Springer, Heidelberg (2005)