Lock-free Oblivious Trees
A Concurrent Cryptographic Data Structure

William Strickland and Christopher Fontaine

[1]University of Central Florida, Orlando, Fl, 32817 USA

Abstract

Three important trends in computing today are the proliferation of concurrent and distributed systems, an increase in the amount and value of digital data being processed and a need for an ability to determine the authenticity of information. These three trends tend to conflict and a solution for one rarely helps another and often works to the detriment of the other goals. In this paper we examine Oblivious Trees an algorithm and data structure designed to reconcile the need to determine authenticity and the need to do so efficiently with files that are large and undergo small changes over time [1]. Apply concepts used for the design of lock-free data structures we develop a design that can provide the some advantages of a lock-free data structure to a very sequentially bound algorithm.

I. Introduction

A digital signature is a method of verifying the authenticity of some message or document. It is capable of detecting forgeries as well as determining whether or not the data had been tampered with before it reached its intended destination. Many algorithms and schemes exist for generating such signatures, and they all share one particular aspect that is of central importance to our project. If a digital signature is generated for some document or file, then any legitimate change or edit to that document or file automatically invalidates the signature. As a result, a new digital signature must be produced for every change that is made. Depending on the circumstance, this can be costly in terms of processing [2]. Over the years a handful of incremental signature schemes have been proposed, but none have the show the security

II. Incremental Digital Signatures

Incremental digital signatures operate by producing a digital signature based only on the differences between the current version of the document or message and some previous version. There are a number of advantages to such a scheme, such as increased performance in signature generation in applications where updates are frequent. In addition, the size of the signature and the computational difficulty in verifying it do not increase with the number of updates performed [2]. However, with these advantages come new security concerns that are not present in traditional non-incremental schemes. Because the signature is generated incrementally, a potential attacker is capable of gleaning the history of updates to a message or document using just the digital signature. Ideally, one would not want the revision history of the document or message to be revealed to anyone, only the final result that is actually released out into the wild [1,3]. Because non-incremental signature schemes produce a brand new signature based on the entire document or message, such concerns over privacy do not apply to them. Thus, in order for an incremental

scheme to be competitive with its non-incremental brethren, this privacy concern must be addressed.

III. Obliviousness

In 1994 an early incremental signature scheme based on 2-3 trees was developed in the paper Incremental Cryptography: The Case of Hashing and Signing by Bellare, Goldreich and Goldwasser. It used a generic non-incremental signature function as a basic building block of its tree structure. While the security of this approach was solid, it was discovered that revision information leaked out due to the nature of the structure tree that stored the incremental results [2].

In 1997, Oblivious data structures: applications to cryptography by Danielle Micciancio aimed to resolve this issue by resolving the information leakage from the tree structure. This paper defined a property a of data structures called 'obliviousness'. A data structure that possesses the property of obliviousness does not reveal information about the operations that have been performed upon it to the outside world [1]. Based on this definition, imparting this property unto the 2-3 tree used previously would resolve the information leakage issues that had been discovered. This new data structure would be called an 'oblivious tree' [1].

IV. Oblivious Tree

Being based on a 2-3 tree affords the Oblivious Tree several advantages; such as fast access to leaf nodes because they are all located at the same level, as well as high probability that the tree will be balanced once constructed. In addition, all the leaf nodes are contained in sorted order, making it possible to traverse the tree to a particular node using only the size information, which is the number of leaf nodes contained in that particular sub-tree, contained within each internal/non-leaf node [1].

The oblivious tree also supports the same modification primitives that proposed for the original incremental signature algorithm; a CREATE function which constructs the initial tree from the ground up given some list of leaf nodes, an INSERT function that takes some new leaf node and inserts into a given position, and a DELETE function that removes a leaf node from the tree [2].

All three of these functions perform their operations while ensuring that obliviousness is maintained. They accomplish this by randomizing the internal structure of the tree every time they are executed. This randomization process ensures that a potential attacker cannot extract revision information by observing how the structure of the tree changes any time operations are performed on it. Because the incremental digital signature is dependent on the structure of the tree, this also means that no revision information can leak out from the signature itself. The Create, Insert, and Delete functions operate with time O(n), O(log n) and O(log n) respectively, established through extensive mathematical proofs performed by the author of the paper [1].

V. Data Structure Concurrency

The algorithm details provided in the original paper outline a sequential version of the data structure. The randomization of the oblivious tree structure between operations would suggest that some level of concurrency should be possible as there are no hard requirements on the structure of the tree after each operation. Also, the basic operations are

limited to creation, insertion and deletion. However, even with these properties, the concurrency that can be allowed in this data structure is limited.

The cause of this limitation is primarily the need of each thread to randomize the tree structure during each insert or deletion. While necessary to provide the oblivious property that creates the privacy when using an oblivious tree, this tree randomization poses performance challenges for sequential and concurrent implementations. In sequential versions this is limited to a additional work that must be done to randomize the tree. However, for concurrent versions this leads to many situations where threads can collide mid-operation or break the tree while the randomization is taking place and cause other threads to fail when attempting to execute.

The randomization process interrupts the ability of other threads to traverse the tree. This results in the a necessary serialization of insert, delete and snapshot operations as there is no way to reliably traverse the tree while some of the nodes are unlinked to be randomized by another thread. This process also poses problems should two or more threads meet at the same parent and attempt to randomize that parent. This case is not as severe and might be overcome with some heavy handed consensus between threads during randomization, but at the cost of a great deal of performance.

While the number of nodes that will be randomized is bounded, in many cases threads would end up only randomizing separate sub-trees [1]. Unfortunately, this cannot be know before or during an insert/delete which other threads it may interact in the tree. The scope of a particular randomization is not known until it has completed. In general, It must be assumed that any thread will randomize the whole tree to the right of the leaf and to the right of each parent up to the tree to the root.

VI. Lock-free Approach

In the face of the heavily sequential nature of our problem, our general approach to prevent performance degradation in heavily concurrent applications to redesign the oblivious tree data structure to be lock-free. Here our criteria for lock-free follows from the definition put forth by Herily: That the shared object guarantees that some process will make progress in a finite number of steps [5,6]. Our goal is for at least one thread to making progress at any given point in time. Like many other lock-free data structures, we attempt to use consensus between threads to make sure that threads can execute without competing or causing contention [6, 7].

VII. Design Overview

For our concurrent implementation, given the necessary serialization of our data structure we attempt to maximize the fairness of our implementation. To do this we force all threads to push a descriptor of their operation into a queue. Using existing lock-free concurrent queue implementations we are able to efficiently order the operations and provide fairness to all threads. This queue imposes sequential ordering and execution on all threads, but this was required by the complexity of the randomization of the tree structure[6]. This queue of descriptors provides a consensus among threads and allow them to cooperate to make sure the current operation is completed in a lock-free way. In this way we are guaranteeing that exactly one thread is making progress at all times.

Each thread must keep track of the sequence of the operation that it inserted and is waiting to 'execute'. Should ever the queue become empty or the sequence of the head of the queue exceed the sequence of the operation that thread inserted, then that thread may return from the method call. Otherwise each thread must assist the previous operations in their completion and ensure that they are making progress.

The orchestration of threads is performed using queues and pointers in the descriptor to notify all threads of the current operation that is pending. Operations are very fine grained so that they may be atomic and may fail quickly after one thread has succeeded in completing that step. For tasking out randomization, each thread will attempt to create the next parent at the current level of the tree and draw from the current set of unassigned notes. The thread that succeeds will update the list of unassigned nodes and link its parent into the current position. Other threads will fail to perform this linking and instead move on to the next parent to randomize.

Similar to randomization, operations completion is handled similarly as the final randomization will cause the list of unassigned nodes to be empty. At this point each thread will attempt to pop the current descriptor from the head queue. One will succeed then all will move to the next operation after checking if the queue has become empty or the head exceeds their sequence.

The crux of this procedure is the task descriptor class that we developed, called TaskDesc. This descriptor contains not only the type of operation to be performed, but also the complete description of the current state of the tree as of the current operation. This state description is necessary to allow cooperating to pick up where the last successful left off. The possible states of the tree are encapsulated within the three operation types of “NEW”, “OPEN”, and “LINK”. A NEW operation contains a description of the tree state when a new leaf node is about to be inserted. OPEN contains a description of a tree in the midst of a randomization operation, and LINK describes the tree state when an unassigned node is being transferred between parents. The transition pattern begins with NEW, as that indicates that the node to be inserted or deleted has not been inserted yet, before transitioning to LINK, which actually proposes the new node. From that point onward, the state transitions between OPEN and LINK as randomized nodes are proposed to replace the old ones that exist in the tree before a leaf node was added or removed.

Additionally, in the design of this of our implementation we saw that an alternate but compatible method of 'traversing' the tree would be possible and potentially preferable. Given the implementation of a lock-free and efficient vector data structure that implements add_at and get_at operations, the traversal problem may be eliminated for insert and delete methods. While it is not possible to reliably traverse the tree from root to leaf while randomization is in progress, if one could use a vector to jump directly to the leaf then that problem would be eliminated. A vector structure is makes a good representation for the leaves of the oblivious tree as it they are access by order and are always at the same level of the tree. This approach also eliminates the leaf counting maintenance required to traverse the tree. This approach was adopted for the sequential implementation simply to eliminate the leaf counting overhead and simplify insertion/deletion at the small cost of some memory. Unfortunately, suitable lock-free vector methods where not available. While

many efficiently blocking vector implementations exist they would introduce opportunities for the lock-free property to be violated.

VIII. Conclusions

The Concurrent implementation could not be readied in time to experiment and provide concrete results on the performance of our design. While we are confident that in some limited cases the overhead of using descriptor data structures to orchestrate and form a consensus among all threads that may or may not be in currently operating in the shared tree data structure. The biggest problem facing this design is the shear amount of objects that must be constructed to be able to atomically change values and prevent the ABA problem from effecting our data [6]. Unique objects are constructed by many threads at once and this can cause problems for garbage collectors. Alternate garbage collection schemes may be able to mitigate this problem, but our standard scheme would likely have trouble coping.

Additionally, a great deal of parallelism exists when recalculating the signatures of each node in the tree that has been effected by randomization. While there are dependencies exist between child and parent children can be calculated in parallel. To exploit this it would almost certainly be necessary to break the lock-free properties maintained otherwise by the algorithm. However, this task represents a large portion of the work that all operations must perform and some significant speedup could be obtained to the extent that negative impacts to synchronization could be overcome and overall performance improved. While interesting, the idea of a using lock-free and parallel techniques where most advantageous was beyond what was feasible to study at this time.

IX. Appendix

Java implementation source code has been released under the GNU Lesser GPL license and can be found at the following location. http://code.google.com/p/cop6616-oblivious-tree/

X. References

[1] Daniele Micciancio. 1997. Oblivious data structures: applications to cryptography. In Proceedings of the twenty-ninth annual ACM symposium on Theory of Computing (STOC '97). ACM, New York, NY, USA, 456-464. DOI=10.1145/258533.258638 http://doi.acm.org/10.1145/258533.258638

[2] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. 1994. Incremental Cryptography: The Case of Hashing and Signing. In Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '94), Yvo Desmedt (Ed.). Springer-Verlag, London, UK, 216-233.

[3] Qingji Zheng and Shouhuai Xu. 2011. Fair and dynamic proofs of retrievability. In Proceedings of the first ACM conference on Data and application security and privacy (CODAPSY '11). ACM, New York, NY, USA, 237-248. DOI=10.1145/1943513.1943546 http://doi.acm.org/10.1145/1943513.1943546

[4] Mihir Bellare and Daniele Micciancio. 1997. A new paradigm for collision-free hashing: incrementality at reduced cost. In Proceedings of the 16th annual international conference on Theory and application of cryptographic techniques (EUROCRYPT '97),

Walter Fumy (Ed.). Springer-Verlag, Berlin, Heidelberg, 163-192.

[5] Dechev, D., Pirkelbauer, P. and Stroustrup, B., Lock-Free Dynamically Resizable Arrays, Proceedings of 10th International Conference on Principles of Distributed Systems, OPODIS 2006, Bordeaux, France, December 12-15, 2006, Lecture Notes in Computer Science, Springer, Volume 4305, pages 142-156.

[6] Herlihy, M., & Shavit, N. (2008). The art of multiprocessor programming. New York: Elsevier, Inc.

[7] Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing* (PODC '96). ACM, New York, NY, USA, 267-275. DOI=10.1145/248052.248106 http://doi.acm.org/10.1145/248052.248106

[8] Woo, M. (2002, March 06). Oblivious search trees. Retrieved from http://www.cs.cmu.edu/~maverick/Talks/2002-03-06 Oblivious Search Trees.ppt