Improved Scalable Hash Chain Traversal

Sung-Ryul Kim*

Division of Internet & Media
and Multidisciplinary Aerospace System Design Center
Konkuk University
Seoul, Korea

Abstract. Yaron Sella recently proposed a scalable version of Jakobsson's algorithm to traverse a hash chain of size n. Given the hash chain and a computation limit m ($k = m + 1$ and $b = \sqrt[k]{n}$), Sella's algorithm traverses the hash chain using a total of kb memory. We improve the memory usage to $k(b-1)$. Because efficient hash chain traversal algorithms are aimed at devices with severely restricted computation and memory requirements, a reduction by a factor of $(b-1)/b$ is considered to be important. Further, our algorithm matches the memory requirements of Jakobsson's algorithm while still remaining scalable. Sella's algorithm, when scaled to the case of Jakobsson's algorithm, has a memory requirement of about twice that of Jakobsson's.

Keywords: efficient hash chain traversal, secure hash, pebbles

1 Introduction

Many useful cryptographic protocols are designed based on the *hash chain* concept. Given a *hash function* $f()$ which is assumed to be difficult to invert, a hash chain is a sequence $< x_0, x_1, \ldots, x_n >$ of values where each value x_i is defined to be $f(x_{i-1})$. The hash chain is being used as an efficient authentication tool in applications such as the S/Key [2], in signing multicast streams [5], message authentication codes [5,6], among others.

Traditionally the hash chain has been used by one of two methods. One is to store the entire hash chain in memory. The other is to recompute the entire hash chain from x_0 as values are exposed from x_n to x_0. Both methods are not very efficient. The first one requires a memory of size $\Theta(n)$ while the second one requires $\Theta(n)$ hash function evaluations for each value that is exposed. The memory-times-storage complexity of the first method is $O(n)$ and it is $O(n^2)$ for the second method. As mobile computing becomes popular, small devices with restricted memory and computation powers are being used regularly. As these devices are being used for jobs requiring security and authentication, the memory and computation efficiency of hash chain traversal is becoming more important.

* Corresponding author. kimsr@konkuk.ac.kr

J. Zhou, M. Yung, Y. Han (Eds.): ACNS 2003, LNCS 2846, pp. 86–95, 2003.

Influenced by amortization techniques proposed by Itkis and Reyzin [3], Jakobsson [4] proposed a novel technique that dramatically reduces the memory-times-storage complexity of hash chain traversal to $O(\log^2 n)$. The algorithm by Jakobsson uses $\lceil \log n \rceil + 1$ memory and makes $\lceil \log n \rceil$ hash function evaluations for each value that is exposed. The result has been further improved by Coppersmith and Jakobsson [1], to reduce the computation to about half of the algorithm in [4]. Both results are not intended for scalability. That is, the amount of computation and memory depends only on the length of the chain n and they cannot be controlled as such needs arise. For example, some device may have abundant memory but it may be operating in a situation where the delay between exposed hash values are critical. To address the issue Sella [7] introduced a scalable technique that makes possible a trade-off between memory and computation requirements. Using the algorithm in [7], one can set the amount m of computation per hash value from 1 to $\log n - 1$. The amount of memory required is $k\sqrt[k]{n}$ where $k = m + 1$. In the same paper, an algorithm designed specifically for the case $m = 1$ is also presented. The specific algorithm reduces the memory requirement by about half of that in the scalable algorithm.

We improve the memory requirement of Sella's scalable algorithm. While Sella's algorithm has the advantage of scalability from $m = 1$ to $\log n - 1$, the memory requirement is about twice that of Jakobsson's algorithm when $m = \log n - 1$. Our algorithm reduces the memory requirement of Sella's algorithm from $k\sqrt[k]{n}$ to $k(\sqrt[k]{n} - 1)$. This might not seem to be much at first. However, if k is close to $\log n$ the reduction translates into noticeable difference. For example, if $k = \frac{1}{2}\log n$, the memory requirement is reduced by a factor of $\frac{1}{4}$. If $k = \log n$, the memory requirement is reduced by half. Of particular theoretical interest is that our algorithm exactly matches the memory and computation requirements of Jakobsson's algorithm if we set $k = \log n$. Our technique has one drawback as the scalability is reduced a little bit. It is assumed that $m \geq \sqrt[k]{n}$, that is, our algorithm does not scale for the cases where m is particularly small.

We note here that, as it is with previous works, we count only the evaluation of hash function into the computational complexity of our algorithm. This is because the hash function evaluation is usually the most expensive operation in related applications. We also note that the computation and memory bounds we achieve (and in the previous works) are worst-case bounds (i.e., they hold every time a value in the hash chain is exposed).

2 Preliminaries

2.1 Definitions and Terminology

We mostly follow the same terminology and basic definitions that appear in [7]. A hash chain is a sequence of values $< x_0, x_1, \ldots, x_n >$. The value x_0 is chosen at random and all other values are derived from x_0 in the following way: $x_i = f(x_{i-1})$, $(1 \leq i \leq n)$, where $f()$ is a hash function. A single value x_i is referred to as a *link* in the chain. The chain is generated from x_0 to x_n but is exposed from x_n to x_0. We use the following directions terminology: x_0 is placed at the

left end, and it is called the *leftmost* link. The chain is generated *rightward*, until we reach the *rightmost* link x_n. The links of the chain are exposed *leftward* from x_n to x_0. The time between consecutive links are exposed is called an iteration. The above hash chain has $n+1$ links. However we assume that at the outset, x_n is published as the public key. Hence, we refer to $X =< x_0, x_1, \ldots, x_{n-1} >$ as the full hash chain and ignore x_n.

The goal of the hash chain traversal is to expose the links of the hash chain one-by-one from right to left. The efficiency of a traversal is measured in both the amount of memory required and the number of hash function evaluations between each exposed link.

In the work by Sella the concept of a *b-partition* of the hash chain is used. It can be also be used to describe Jakobsson's algorithm. We repeat the definitions here.

Definition 1. *A* b-partition *of a hash chain section means dividing it into* b *sub-sections of equal size, and storing the leftmost link of each sub-section.*

Definition 2. *A* recursive b-partition *of a hash chain section means applying* b*-partition recursively, starting with the entire section, and continuing with the* rightmost *sub-section, until the recursion encounters a sub-section of size* 1.

The b-partition and recursive b-partition are easier to describe when we map the partition to a b-ary tree. We will use the basic terminology pertaining to trees without definition. The following is the definition of the mapping between the recursive b-partition and a b-ary tree. It is repeated also from [7].

Definition 3. *A* recursive b-tree *is a recursive mapping of a recursive* b*-partition onto a tree as follows.*

- *Each node in the tree is a section or a sub-section. In particular, the entire hash chain is represented by the root of the tree.*
- *For every section* L *that is partitioned from left to right by sub-sections* $L_1, L_2, \ldots, L_b$*, the corresponding node* N *is the parent of the corresponding nodes* $N_1, N_2, \ldots, N_b$*, and the children keeping the same left-to-right order.*

See Figure 1 for an example. It shows a recursive 4-tree with 3 levels. The root of the tree corresponds to the entire hash chain and the tree leaves correspond to single links in the hash chain. Even though each node is shown to be storing b links, only $b-1$ links will be the memory requirement for a node because the leftmost link is stored in the parent.

Sub-sections induced by recursive b-partition are *neighbors* if their corresponding nodes are at the same level and appear consecutively at the natural depiction of the recursive b-tree. Note that they may not share the same parent. Depending on its position in the recursive b-partition, a sub-section can have a *left neighbor*, a *right neighbor*, or both. A node in a recursive b-tree will sometimes be called a sub-section, meaning the corresponding sub-section in the recursive b-partition.

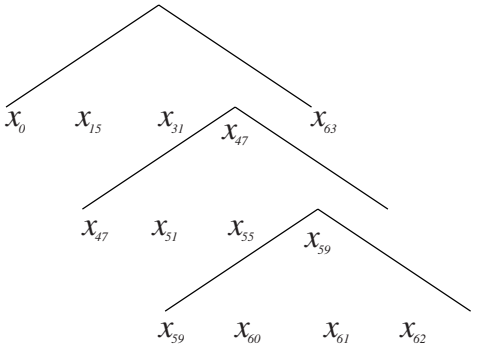

Fig. 1. Example 4-tree with 3 levels

In order to traverse the hash chain we adopt the notion of pebbles and stored links from previous works. A *pebble* is a dynamic link that moves from left-to-right in a hash chain. The mission of a pebble is to induce a b-partition of a sub-section S. That is, it starts at a stored link and traverses the chain while storing the values corresponding to the b-partition of S. The initial stored link will be received from the (sub-)section that S is a part of. For a pebble that induces a b-partition $(x_{i_1}, x_{i_2}, \ldots, x_{i_b})$, the link x_{i_b} is called the *destination* because the pebble does not have to move any further after reaching x_{i_b}.

2.2 Jakobsson's Algorithm

Jakobsson's algorithm can be considered to be using the recursive 2-tree of the entire hash chain. There are $\lceil logn \rceil + 1$ levels in the tree. One pebble is present at each level except at the root. When the algorithm starts, a pebble is placed at the mid-point of the sub-section S at each level.

When the mid-point is exposed the pebble is moved to the start of the left neighbor. From there it traverses the hash chain two links per exposed link. Because it (and all other pebbles) moves at twice the speed that the links are exposed, a pebble reaches the end of the left neighbor when all the links in S has been exposed. It can be shown that at most one of the pebbles in two adjacent levels are moving at any point. This leads to the above mentioned computation cost per iteration. We note that because the pebble starts to move only after its destination is reached and because there are only one link to store (the leftmost node is stored in the parent), the pebble never stores a link and moves to the right. Thus the memory requirement is the same as the number of pebbles (plus one at the root).

2.3 Sella's Algorithm

Sella's algorithm uses a recursive b-partition. As in the Jakobsson's algorithm, one pebble is placed at each level. However, the pebble is moved to the start of the left neighbor immediately when the rightmost link (not the rightmost stored link) in a section is exposed. The pebble moves at the same speed as the links are exposed.

The parameter b is set to be $\sqrt[k]{n}$ where $k = m+1$. That is, the tree will have k levels. Because no pebble is needed at the root, the computational requirement per iteration is m. Initially, there are $b-1$ stored links at each level. However, because pebbles start to move before any stored link is exposed, one memory cell is required for each pebble. Thus, the total memory requirement amounts to $k\sqrt[k]{n}$. If we set $k = \log n$, the memory requirement becomes $2\log n$, which is about twice that of Jakobsson's algorithm.

2.4 Intuitions for Our Algorithm

In our proposed algorithm, each pebble starts to move after the rightmost stored value is used, thus removing the memory requirement for the pebbles. Because it started late (as in Jakobsson's algorithm) it makes accelerated moves. That is, it traverses b links (called *making a move*) while one link is exposed. It cannot be moving all the time because if so, the computational requirement per iteration will be $b \times m$. It makes a move about every $b/(b-1)$ times, enough to catch up to its late start but not at a regular interval as described later.

It is quite simple in Jakobsson's algorithm because there can be no overlap in computation for pebbles residing in two adjacent levels (in the b-tree). We have to consider b adjacent levels. There will be overlaps if we allocate the moves for different levels at regular intervals. So we have to carefully allocate the iterations where each pebble makes a move. The tricky part is to show that there always exists a way to allocate iterations such that a pebble is never too late to its destination.

3 Improved Algorithm

Assume that we have a hash chain $< x_0, x_1, \ldots, x_{n-1} >$ to traverse. Let k be $m+1$ and let b be $\sqrt[k]{n}$. We assume that $k-1$ is a multiple of m and we also assume that $m \geq b$.

3.1 Algorithm

We first form a recursive b-partition (also the corresponding recursive b-tree) off-line. We put a pebble at the rightmost stored link in each sub-section. Then, the following loop is repeated for n times.

- Exposure loop
  1. Expose and discard L, the current rightmost link in the hash chain.
  2. If $(L = x_0)$ stop.
  3. For each pebble p do
     - If L is a rightmost stored link in a sub-section S' and p is on L, then move p to the left end of S where S is the left neighbor of S'.
  4. Call **Schedule Iterations** for all pebbles p starting from the bottom-most level to the top-most level.
  5. Advance all pebbles that are allocated the current iteration by b links.
  6. Repeat from Step 1.

Now we describe the procedure **Schedule Iterations**. We define *level numbers* in the recursive b-tree. The lowest level is said to be at level 1 and the level number increases as we go up the tree. Thus, the root will be at level k. Assume that p is at level t. Let S be the sub-section where p is newly assigned. Let S' be the right neighbor of S. There are b^{t+1} links in S. We give a natural correspondence between a link in S and an iteration where a link in S' is exposed. The leftmost link in S corresponds to the iteration where the rightmost link in S' is exposed. And the correspondence proceeds towards the right in S and towards the left in S'. That is, the links in S and the links in S' are matched in reverse order. When we say that we allocate a link x_i in S to p it means that p will advance b times rightward at the time when the link in S' that corresponds to x_i is exposed. Note that the iterations corresponding to the b^t leftmost links in S have already passed. Let $(L_1, L_2, \ldots, L_b)$ be the b-partition of S.

Our strategy will satisfy the following three conditions. We will shortly verify that the conditions are satisfied by our algorithm.

1. A link should not be assigned to p if it is assigned to a pebble in lower $b-1$ levels. This ensures that at any iteration, at most one pebble in consecutive b levels is making a move. There are many sub-sections in the lower $b-1$ levels but the schedule for each pebble will be the same regardless of the particular sub-section.
2. Pebble p should not move too fast. Because p has to store a link when it reaches a position of a stored link for S, one stored link in S' has to be exposed before we store one link in S.
3. The pebble q at level $t-1$ must have enough links to be allocated after all computation for p is finished. Pebble q cannot be moved until it receives the rightmost stored link in S'. Thus, the schedule for p must finish before the schedule for q starts.

Procedure **Schedule Iterations** works as follows. First, p is not scheduled if any of the pebbles in the lower $b-1$ levels are allocated the current iteration. Second, p is not scheduled if p has to store a link at the current location and there is not enough memory in the current level for the stored link. Third, if p is at the leftmost position of sub-section S and the link at the current position is not available yet. If all three conditions are false and p is yet to reach its destination, then p is scheduled the current iteration. See Fig. 2 for an example.

Fig. 2. Example run of the algorithm for $n = 27$, $k = 3$, $b = 3$.

3.2 Correctness

We show by a series of lemmas that there always exists a schedule that satisfies the conditions mentioned in the algorithm. Let S be the sub-section at level t where p is newly assigned. Also let $(L_1, L_2, \ldots, L_b)$ be the b-partition of S. The following lemma shows that we have enough unallocated links to allocate for pebble p at level t and the $b-1$ pebbles in the lower $b-1$ levels (condition 1).

Lemma 1. *There are enough links in S for the b pebbles p and in lower $b-1$ levels.*

Proof. By the time p is moved to the start of S the iterations for the links in L_1 have already passed. So there are a total of $b^t(b-1)$ links to be allocated. p has to move across $b^t(b-1) = b^{t+1} - b^t$ links to reach its destination. Thus, it has to make $b^t - b^{t-1}$ moves (same number of links are to be allocated). At level $t-1$, there are $b-1$ sub-sections to schedule and in each sub-section there are $b^{t-1} - b^{t-2}$ moves to be made by the pebble in level $t-1$. So at level $t-1$ there are $b^t - 2b^{t-1} + b^{t-1}$ moves to be made in total. From then on to the level $t-b+1$, the same number of moves are to be made.

By adding them up, we can see that we need $b^{t+1} - 2b^t + 2b^{t-1} - b^{t-2}$ links to allocate to the pebbles. Subtracting this number from the number of available links $b^{t+1} - b^t$ we have the difference $b^t - 2b^{t-1} + b^{t-2}$, which is always positive if $b \geq 2$.

The following lemma and corollary shows that the memory requirement (condition 2) is satisfied.

Lemma 2. *There are enough links in L_b for the last b^{t-1} moves of p.*

Proof. There are b^t links that can be allocated in L_b. Pebble p at level t requires b^{t-1} moves by the condition of the lemma. The pebble at level $t-1$ has to make $b^{t-1} - b^{t-2}$ total moves. From then onto the next $b-2$ levels, the same number of moves are needed. Adding them up leads to $b^t - b^{t-1} + b^{t-2}$. Substracting this number from the available number of links b^t results in

$$b^{t-1} - b^{t-2}, \tag{1}$$

which is positive if $b \geq 2$.

Corollary 1. *Pebble p can be moved slow enough so that the memory requirement for level t is $b-1$.*

Proof. The above lemma shows that the last b^t links to be traversed by p can be actually traversed after all the stored links in the right neighbor of S. Even if we assume that the first $b^t(b-2)$ moves are allocated as far to the right as possible, there are enough number of links to allocate for p, and so p will move at a regular pace at the worst case. Thus, p can be moved slow enough to satisfy the memory requirement.

Even if we have shown that there are enough number of links to allocate for every pebble, it is not enough until we can show that condition 3 is satisfied. Consider a pebble r at level 1, because no pebble needs to be scheduled after r, r can be allocated the rightmost link in the sub-section. A pebble r' at level 2 has to finish its computation before r has to start its computation. Thus, there is a leftmost link (called the *leftmost allocatable link*) for the pebble in each level where the pebble has to start its computation to reach its goal early enough for the pebble in the lower level. Conversely, we have to make sure that the leftmost link allocated to a pebble at every level is to the right enough so that the upper levels have enough room to finish its moves. The following lemma shows that there are enough links to the left of the leftmost link allocated for pebble q at level $t-1$ (condition 3). Note that the final difference in the proof of the above lemma will be used in the proof of the next lemma.

Lemma 3. *There is enough links to allocate to p and the $b-1$ pebbles in the lower levels to satisfy condition 3.*

Proof. As mentioned above, the leftmost allocatable link for level 1 is the first from the rightmost link in a sub-section. For level 2, it is the b-th link from the rightmost link. If $b = 2$, the leftmost allocated link for level 3 ls b^2-th one from the rightmost link. The lemma is easy to prove if $b = 2$. In that case, the leftmost allocatable link for level m is b^{m-1}-th link from the rightmost link in a sub-section. However, if $b > 2$, the leftmost allocated link for level 3 is $(b^2 + 2b)$-th one from the rightmost link because $2b$ links that are already allocated to the pebble at level 1 cannot be allocated. We can prove by induction that the leftmost link for level m is at most the $(b^{m-1} + 2b^{m-2} + 2^2 b^{m-3} + \cdots)$-th one from the rightmost link in a sub-section. This number sums to at most $b^m/(b-2) < b^m$.

In the proof of lemma 2, we have $b^{t-1} - b^{t-2}$ links that need not be allocated to any pebble. Because we have allocated links to pebbles in levels t down to $t-b+1$, we consider the leftmost link allocatable to the pebble at level $t-b$. If we set m in the above calculation to $t-b$, then the leftmost link allocatable to the pebble at level $t-b$ is at most the (b^{t-b})-th one from the rightmost link in a sub-section. Substracting from the available free links (1), we get $b^{t-1} - b^{t-2} - b^{t-b}$, which is nonnegative if $b \geq 3$.

Using the above lemmas, it is easy to prove the following theorem.

Theorem 1. *Given a hash chain of length n, the amount of hash function evaluation m, and $k = m + 1$, the algorithm traverses the hash chain using m hash function evaluations at each iteration and using $k(\sqrt[k]{n} - 1)$ memory cells to store the intermediate hash values.*

4 Conclusion

Given the hash chain and a computation limit m ($k = m + 1$ and $b = \sqrt[k]{n}$), we have proposed an algorithm that traverses the hash chain using a total of

$k(b-1)$ memory. This reduces the memory requirements of Sella's algorithm by a factor of $(b-1)/b$. Because efficient hash chain traversal algorithms are aimed at devices with severely restricted computation and memory requirements, this reduction is considered to be important. Further, our algorithm is matches the memory requirements of Jakobsson's algorithm while still remaining scalable. Sella's algorithm, when scaled to the case of Jakobsson's algorithm, has a memory requirement of about twice that of Jakobsson's.

Acknowledgment. The author thanks the anonymous refrees for their kind and helpful comments.

References

1. D. Coppersmith and M. Jakobsson, Almost optimal hash sequence traversal, *Proc. of the Fifth Conference on Financial Cryptography (FC) '02*, Mar, 2002.
2. H. Haller, The S/Key one-time password system, *RFC 1760*, Internet Engineering Taskforce, Feb. 1995.
3. G. Itkis and L. Reyzin, Forward-secure signature with optimal signing and verifying, *Proc. of Crypto '01*, 332–354, 2001.
4. M. Jakobsson, Fractal hash sequence representation and traversal, *IEEE International Symposium on Information Theory (ISIT) 2002*, Lausanne, Switzerland, 2002.
5. A. Perrig, R. Canetti, D.Song, and D. Tygar, Efficient authentication and signing of multicast streams over lossy channels, *Proc. of IEEE Security and Provacy Symposium*, 56–73, May 2000.
6. A. Perrig, R. Canetti, D.Song, and D. Tygar, TESLA: Multicast source authentication transform, *Proposed IRFT Draft*, http://paris.cs.berkeley.edu/ perrig/
7. Yaron Sella, On the computation-storage trade-offs of hash chain traversal, *Proc. of the Sixth Conference on Financial Cryptography (FC) '03*, Mar, 2003.