Fractal Hash Sequence Representation and Traversal

Markus Jakobsson
RSA Laboratories
mjakobsson@rsasecurity.com

Abstract

We introduce a novel amortization technique for computation of consecutive preimages of hash chains, given knowledge of the seed. While all previously known techniques have a memory-times-computational complexity of $O(n)$ per chain element, the complexity of our technique can be upper bounded at $O(log^2 n)$, making it a useful primitive for low-cost applications such as authentication, signatures and micro-payments. Our technique uses a logarithmic number of *pebbles* associated with points on the hash chain. The locations of these pebbles are modified over time. Like fractals, where images can be found within images, the pebbles move within intervals and sub-intervals according to a highly symmetric pattern.

Keywords: amortization, authentication, hash chain, smart dust.

1 Introduction

With a trend towards smaller and smaller computers, there is a need for more efficient cryptographic methods. While this is true for cellular phones, smart cards, and handheld devices, the need is perhaps most striking for *smart dust*. The term "smart dust", coined in [6], refers to very small computational devices covering a large area, often for purposes of surveillance, whether seismic or military. These dust computers, while not quite as small yet as their name suggests, still suffer severe enough computational and storage limitations to make most cryptographic methods too costly. Like most networked devices, though, dust computers need some means for authentication of information passed between them.

For devices where computational limitations rule out the use of standard digital signatures, message authentication codes (MACs) provide an alternative. However, standard use of MACs require each pair of communicating devices to share a secret key, which is impossible for applications where the number of pairs is too large given the available storage. To address this problem, Perrig et al. [3, 4, 5] proposed the use of hash chains – each device having one such chain – to compute MAC keys. According to his idea, a device would compute and broadcast a message and its MAC during a first time interval, and then release the key during a later time interval. Recipients could verify the MAC first after the key is released, and would trust the authenticity of the MAC based on the knowledge that only the sender could have known the key at the time the MAC was broadcast. (Clearly, this assumes synchronization.) The correctness of a recent key is verified by hashing and comparing the result to a less recent key associated with the same chain, and therefore with the sender. This less recent key may either be an intermediary value or the public "beginning key". Thus, Perrig's solution straddles the fence between public key and secret key cryptography, taking some of the functionality of the former while aiming for the efficiency of the latter.

However, we note that his solution fails to marry functionality with efficiency, given that it ignores the fact that computing the next hash chain value may in itself be a far costlier undertaking than computing a standard signature, or require more storage than is reasonable. We address this problem by introducing a new technique for deriving sequences of hash chain values. Thus, our method is directly applicable to authentication on computationally limited devices, and more generally, to applications involving the use of hash chains. The computation of the next hash value may either be based on time, such as in [3], or on events, such as in [2, 8].

Consider a hash chain with a starting point v_0 and an endpoint v_n, where the latter is the seed from which the chain is computed. Each element v_i is the hash image of the next value on the chain, i.e., $v_i = h(v_{i+1})$. Our aim is to compute and output the series $v_1, v_2, \ldots, v_n$ – in that order – and in a manner that requires minimal memory and computational requirements. It is clear that previously output values are not useful in computing the next value to be output, since the hash function is one-way. Instead, the output values have to be computed by iterative application of the hash one-way function to a value towards the end of the chain.

A trivial solution to the problem of generating consecutive hash chain elements requires $O(n)$ computation per value to be output, where n is the length of the chain; such a solution would plainly recompute the chain from the end value to the wanted value for each element. Another trivial solution would store not only the end value, but *all* values of the chain, and plainly perform a lookup for each value to be output. Such a solution would have a memory complexity of $O(n)$. One could easily trade memory and storage against each other in these trivial solutions by storing some fraction of the values, and computing the chain from such a stored point for each element to be output. It can be seen that such variations of these trivial approaches all will have a memory-times-computational complexity of $O(n)$.

We introduce a technique whose memory and computational complexities both are $O(log\, n)$. More specifically, the algorithm performs $\lceil log_2 n \rceil$ hash function applications per output element, and uses $\lceil log_2 n \rceil$ memory cells, where each cell stores one hash chain value along with some short state information. For example, if the hash chain has length 2^{32}, and SHA [7] is the hash function of choice, each one of the 32 storage cells has size $3\lceil n \rceil + 160 = 256$ bits, totalling 1024 bytes of storage for all pebbles. If we output one hash value per second, such a chain would last for more than 68 years.

Our techniques are intellectually related to the pebbling techniques used for key update in a recent paper by Itkis and Reyzin [1]. They store several pebbles, each one of which correspond to a node in a tree, and the values associated with this node. Then, for each step, and according to a particular pattern, pebbles move downwards in the tree, allowing the computation of leaf values in an amortized manner. While this basic idea is the same as in our technique, the movement patterns and the resulting schemes are different, as are the two applications.

2 Algorithm

Goal. We say that a hash chain $(v_0, v_2, \ldots, v_n)$ has a *starting point* v_0 and an *endpoint* v_n. Here, v_n is a random value selected at the initialization of the chain, and each other value v_i is the hash image of the next hash chain value v_{i+1}. Thus, $v_0 = h(v_1) = h(h(v_2))$, etc. We wish to output the hash chain values in order of consecutive preimages, starting with v_0 and ending with v_n. For simplicity, we assume that the length of the hash chain is a power of two, namely $n = 2^\sigma$.

Intuition. At any time, the current pointer (which coresponds to what element gets output) is in one interval of size 4; one of size 8; one of size 16; and so on, up to the length of the hash chain. A *pebble* is associated with each such interval, and "strives towards" the midpoint of the interval. Given the way the intervals are arranged, such midpoints constitute endpoints for smaller intervals. When the current pointer reaches a pebble, a new interval of the same size is created, adjacent to the old interval, and the pebble is started off at its end. Since this is a midpoint for a larger interval, another pebble will be found there. The newly moved pebble then moves to the middle of its own interval, taking a few steps for every output value we generate. With each pebble we associate a value, corresponding to the hash chain value at the location of the pebble. Thus, when a pebble is reassigned to a new interval, it obtains the value of the pebble in its aqcuired position; for each step it moves, it applies the hash function to its value. Given that the current pointer is always in intervals populated by pebbles, we can bound the computational effort to derive the output value from the pebble values.

Setup. During the setup phase, an endpoint v_n is chosen and the resulting sequence of hash images (corresponding to the desired hash chain) is computed. This may be performed on a more powerful device than that which will later compute and output consecutive preimages.

In order to allow an efficient computation of consecutive preimages of the hash chain, we store and maintain a particular representation of the chain. This representation consists of a list of pebbles, each one of which is associated with a *position* and a hash sequence *value*, among other things. Since the pebbles are continously relocated, each of them also stores a *destination* (the position to which they are going) and two values that determine where a pebble is placed after having been reached, and how its destination is computed. These values are referred to as *StartIncr* and *DestIncr*. For pebble p_j, $1 \le j \le \sigma$, the following assignment is made during setup:

$$\begin{cases} p_j.StartIncr \leftarrow 3 \times 2^j \\ p_j.DestIncr \leftarrow 2^{j+1} \\ p_j.position \leftarrow 2^j \\ p_j.destination \leftarrow 2^j \\ p_j.value \leftarrow v_{2^j} \end{cases}$$

We refer to the pebble initialized as p_j as a "type-2^j pebble". Thus, the pebble starting in position 2 is called a "type-2 pebble", while the pebble starting in position 4 is called a "type-4 pebble." Over time, the positions, destinations and values of the pebbles will be modified. The list of pebbles will be kept sorted with respect to their destination values. Here, we will call the first element of the sorted list p_1, and the last one p_σ. (Therefore, it is *not* the case that the first pebble, p_1, will remain a type-2 pebble through the execution of the protocol.) We note that in order to save space, we can store the type instead of $StartIncr$ $(= 3 \times type)$ and $DestIncr$ $(= 2 \times type)$.

In addition to the above assignments, we compute the following:

$$\begin{cases} current.position \leftarrow 0 \\ current.value \leftarrow v_0 \end{cases}$$

In some applications, the starting point $current.value$ (i.e., v_0) of the chain may be made public. The remaining values (as described above) are stored on the device that will compute and output the sequence of preimages of the hash chain.

Computing the next value. In order to compute the next hash value on the chain (starting with v_1), the following computation is performed:

1. If $current.position = n$ then	(Arrived at the end?)
halt	(Quit)
else	
increase $current.position$ by one	(Move pointer forward)
2. For $1 \leq j \leq \sigma$ do	(Consider all pebbles)
If $p_j.position \neq p_j.destination$ then	(If not arrived ...)
$p_j.position \leftarrow p_j.position - 2$	(... then move it ...)
$p_j.value \leftarrow h(h(p_j.value))$	(... and update value)
3. If $current.position$ is odd then	
output $h(p_1.value)$	(Hash and output)
else	
output $p_1.value$	(Output stored value)
$p_1.position \leftarrow p_1.position + p_1.StartIncr$	(Reached: reassign)
$p_1.destination \leftarrow p_1.position + p1.DestIncr$	(and reassign)
If $p_1.destination > n$ then	(Pebble redundant?)
$p_1.destination \leftarrow \perp$	(Then retire pebble!)
$p_1.position \leftarrow \perp$	
else	
$p_1.value \leftarrow FindValue$	(Call to set value)
Sort pebbles	(Sort w.r.t. destination)

The sorting of the pebbles at the end of the third step is done with respect to the *destination* of the pebbles, where $\perp$ is considered infinity. It suffices to sort in the newly modified pebble p_1 into the otherwise sorted list. We note that this will cause the previous pebble p_2 to be renamed p_1, along with other such renamings, as the naming of the pebbles relate to their position in the sorted list.

The function $FindValue$ goes through the list of pebbles, returning the value $p_i.value$ for the pebble p_i, $2 \leq i \leq \sigma$ for which $p_i.position = p_1.position$.

In the following section we will prove that the above protocol correctly outputs the sequence of conscutive hash preimages. We also upper-bound the computational requirements of the protocol.

3 Claims and Proofs

Lemma 1: (Successful reassignment.) Let p_1 be a pebble whose *position* and *destination* have just been updated. The assignment of p_1's *value* will always be successful, i.e., if $p_1.position \leq n$, then there exists a pebble p_i, $2 \leq i \leq \sigma$, such that $p_i.position = p_1.position$.

We say that a pebble whose position equals its destination is *inactive*. Consider an inactive pebble p of type $x \neq 2$ during the time it spends at a certain position. We have that one or more pebbles will be reassigned to p's destination during this time period, and furthermore, that the first pebble to do this will be of type $x/2$. (This can easily be seen given how intervals are divided and subdivided.)

Let us therefore consider where a pebble p is located when the first pebble is moved to p's destination. We have that the distance between the pebble p of type x and the pebble p' of type $x/2$ is exactly $x/2$ at the time when they are both inactive. Therefore, it will take exactly $x/2$ protocol executions between the time when p is reassigned and when p' is reassigned. Since each active peg moves two steps per protocol execution, and the distance between its reassigned starting positon and destination equals its type, we have that p will become inactive during the same execution as p' is reassigned. Given that during one protocol execution, any reassignment is performed *after* all pebbles are moved, we have that all reassignments will be successful, and the lemma holds.

Lemma 2: (Correct output.) The correct sequence of values is output by the protocol.

If instead of outputting $p_1.value$ resp. $h(p_1.value)$ we were to output $p_1.position$ resp. $p_1.position + 1$, then n consecutive protocol executions would generate the the output sequence $1 \ldots n$. This holds since all reassignments will be successful (lemma 1), and the distance to the next pebble (at the beginning of the step 1) is always 1 resp. 2. The latter holds since each even position $1 \leq i \leq n$ can be written as a number $i = x(1 + 2j)$ for some positive j and some x that is a power of two. This is the jth reassignment destination of the pebble of type x.

Moreover, when a pebble's position is reassigned, its value is set to the value of the pebble with the same position. For each execution of the protocol, pebbles whose positions are decreased (in step 2) have their values modified accordingly. Therefore, the value associated with each pebble will be the hash chain value corresponding to the pebble's position. Given that the output equals $p_1.value$ when $p_1.position = current.position$, and that the output equals $h(p_1.value)$ when $p_1.value = current.position + 1$, the lemma holds.

Lemma 3: (Computational cost.) For each output element, at most σ hash function evaluations have to be performed.

It can be seen that if pebble p_j is active (has a position different from its destination) then p_{j+1} is not active (i.e., it has arrived at its destination). Moreover, $p_{\sigma-1}$ and p_σ can be seen never to be active. Thus, a maximum of $\lceil(\sigma-2)/2\rceil$ pebbles are moved for each execution, and each is moved a maximum of two steps. The computation of the output given p_1 requires at most one hash function application. Therefore, the computational cost per output element is upper bounded by σ hash function evaluations.

Full proofs will be supplied in the extended version of the paper.

Acknowledgements

Many thanks to Ari Juels for help simplifying the algorithm, and to Adrian Perrig and Leo Reyzin for helpful suggestions.

References

[1] G. Itkis and L. Reyzin, "Forward-Secure Signatures with Optimal Signing and Verifying," Crypto '01, pp. 332–354.

[2] S. Micali, "Efficient Certificate Revocation," Proceedings of RSA '97, and U.S. Patent No. 5,666,416.

[3] A. Perrig, R. Canetti, D. Song, and D. Tygar, "Efficient and Secure Source Authentication for Multicast," Proceedings of Network and Distributed System Security Symposium NDSS 2001, February 2001.

[4] A. Perrig, R. Canetti, D. Song, and D. Tygar, "Efficient Authentication and Signing of Multicast Streams over Lossy Channels," Proc. of IEEE Security and Privacy Symposium S & P 2000, May 2000.

[5] A. Perrig, R. Canetti, D. Song, and D. Tygar, "TESLA: Multicast Source Authentication Transform", Proposed IRTF draft, `http://paris.cs.berkeley.edu/˜perrig/`

[6] K. S. J. Pister, J. M. Kahn and B. E. Boser, "Smart Dust: Wireless Networks of Millimeter-Scale Sensor Nodes. Highlight Article in 1999 Electronics Research Laboratory Research Summary.", 1999. See `http://robotics.eecs.berkeley.edu/˜pister/SmartDust/`

[7] FIPS PUB 180-1, "Secure Hash Standard, SHA-1," `www.itl.nist.gov/fipspubs/fip180-1.htm`

[8] S. Stubblebine and P. Syverson, "Fair On-line Auctions Without Special Trusted Parties," Financial Cryptography '01.