Almost Optimal Hash Sequence Traversal

Don Coppersmith* Markus Jakobsson†

Abstract

We introduce a novel technique for computation of consecutive preimages of hash chains. Whereas traditional techniques have a memory-times-computation complexity of $O(n)$ per output generated, the complexity of our technique is only $O(log^2 n)$, where n is the length of the chain.

Our solution is based on the same principal amortization principle as [1], and has the same asymptotic behavior as this solution. However, our solution decreases the real complexity by approximately a factor of two. Thus, the computational costs of our solution are approximately $\frac{1}{2}log_2 n$ hash function applications, using only a little more than $log_2 n$ storage cells.

A result of independent interest is the lower bounds we provide for the optimal (but to us unknown) solution to the problem we study. The bounds show that our proposed solution is very close to optimal. In particular, we show that there exists no improvement on our scheme that reduces the complexity by more than an approximate factor of two.

Keywords: amortization, hash chain, pebbles, upper and lower bounds.

1 Introduction

Hash chains have been proposed as a tool for improving the efficiency of a variety of practical and valuable cryptographic applications, e.g., [5, 6, 7, 8, 11]. However, the cost for computing the next value on a hash chain is a topic that has largely been ignored. For long chains, this task may account for a quite noticeable – if not overwhelming – portion of the total computational effort of the protocol.

The technique most oftenly employed (whether implicitly or explicitly stated) is to compute each preimage by iterative hash function application to a common seed. However, such a method – which clearly has a computational complexity of $O(n)$ for a chain of length n – is highly wasteful in that the same sequence of values is repetitively computed. Another possible method, in which all values are precomputed and stored, substantially reduces the on-line computational cost, but has a staggering storage complexity of $O(n)$. All straightforward combinations of these two techniques can be shown to have a memory-times-computation complexity of $O(n)$, which is often beyond the reasonable for desirable parameter choices. As an example, one can see that such a high complexity would be punitive in protocols like the broadcast authentication protocols by Perrig *et al.* [6, 7, 8]. There, the delay between

*IBM T.J. Watson Research Center, Yorktown Heights, NY, 10598.

†RSA Laboratories, Bedford, MA 01730. mjakobsson@rsasecurity.com

the transmission of a packet and the receiver's authenticity check of the same is determined by the amount of time between the release of consecutive hash preimages. This makes short time periods (i.e., rapid hash chain traversal) desirable. At the same time, since their method is intended for very inexpensive devices, the permissible "allowances" for computation and storage are strict. Together, these restrictions drastically limit the possible lifetime of such broadcast devices.

Influenced by amortization techniques proposed by Itkis and Reyzin [2], Jakobsson [1] showed how to reduce the above mentioned memory-times-storage complexity to $O(log^2 n)$. This was accomplished by the introduction of a technique in which multiple intermediary values are kept, but their values (and positions in the chain) constantly modified. While the protocol of Jakobsson stressed simplicity over efficiency, we take the opposite approach in this paper. By introducing a set of tricks and new techniques, we are able to improve the efficiency at the expense of simplicity – the latter both in terms of the protocol and its associated proofs. Thus, our improved protocol allows for a reduction of the computational requirements to slightly less than half of [1], while slightly reducing the storage demands for most practical parameter choices.

Using a numeric example to illustrate the differences, we have that consecutive preimages of a chain of length 2^{31} can be generated using 31 hash function applications and 868 bytes of storage in [1], while our algorithm only needs 15 hash function applications and 720 bytes of storage – both assuming the use of SHA [10]. More generally, the computational requirements of our protocol are $\lfloor log_2 \sqrt{n} \rfloor$ hash function evaluations per output element. The storage requirements, in turn, are $\lceil log_2 n \rceil + \lceil log_2(log_2 n + 1) \rceil$ memory cells, where each cell stores one hash chain value and some short state information. Note that these are *not* average costs, but upper bounds on the cost per element that is output, given a hash chain of length n.

In order to allow for these savings, it is necessary to shift from a strategy with a very simple movement pattern to a more complicated strategy where the per-element budget always is exhausted. (Another way to think about it is that the reduction of the budget demands a wiser spending strategy under which no resources are wasted.) Consequently, we develop a rationale for how to assign a now reduced budget among a set of pebbles whose demands and priorities are modified over time. Furthermore, we propose a protocol based on these design criteria and show it to be correct.

Finally, we show that our strategy, and the related protocol we propose, are near optimal. We do this by providing upper and lower bounds for the most efficient algorithm possible, and compare the resulting complexity to that of our protocol.

2 Intuition

The aim of our protocol is to compute and output a series of consecutive values on a hash chain with minimal memory and computational requirements. We call the two ends of our chain "the beginning" and "the end" of the chain, functionally corresponding to the public vs. the secret keys of the scheme. In a setup phase, the chain is constructed by choosing the end value at random and iteratively applying the one-way function to get the value at the beginning of the chain. We want to output all the values of the chain – starting with the value at the beginning of the chain, and ending (not surprisingly perhaps) with the value at the end of the chain. Each value in the chain (except the end value) is the hash one-way function of the adjacent value in the direction of the end of the chain. In other words, each output value is the hash preimage of the previously output value. Therefore, previously output values are not useful in computing the next value to be output, which instead has to be computed by iterative application of the hash one-way function to a value towards the end of the chain.

Our solution employs novel amortization techniques to reduce the worst case computational cost per element to the average cost. Simply put, we use the known principle of conserving resources by not letting anything go to waste. Technically speaking, this is done by assigning a computational budget per step in the chain, and applying any "leftover computation" towards the computation of future elements. The contribution of this paper is a technique for computing the desired sequence without ever exceeding the per-step budget, along with the establishment of the required computational budget and memory demands.

In order to reach our goals, the leftover computation we apply towards future elements to be computed must be sufficient to compute these. More importantly, it must be sufficient to compute them *on time*. Thus, at each point in the chain, the cumulative required expenditures must not exceed the cumulative computational budget.

Assume that we want to compute a value close to the beginning of the chain. If no values along the chain are stored, then we have to perform an amount of work proportional to the length of the chain, which we denote n. Let us now introduce one "helper value" at some distance d from the current chain element. Then, the cost of computing the current value is that of $d-1$ hash function evaluations. The cost for the next value to be computed, in turn, will be $d-2$ such evaluations. However, once the helper value is reached, the cost of the next value will be that of reaching the endpoint of the chain – assuming we only employ one helper value. One can see that the total cost is minimized if $d = n/2$, i.e., the helper value is located on the mid-point of the chain.

If we can use *two* helper points (which corresponds to storing two extra elements instead of one) then one could let the entire interval be split into three equally long intervals, in which case the cost of computing the next element would be upper bounded by $n/3$ hash function evaluations. On the other hand, if we first split the entire interval in two equally long intervals, and then split the first of these two into two sub-intervals, then we have upper bounded the *initial* computational cost at $n/4$ hash function evaluations. This lower cost applies to the first half of the entire interval, after which the distance to the next element (which is the endpoint) would be $n/2$. However, if we – once we reach the first helper point – relocate this to the midpoint between the "global midpoint" and the endpoint, we will maintain an upper bound of $n/4$. (See figure 1 for an illustration of how the relocation occurs.)

This assumes that we have enough remaining computation for this relocation. Note now that if we have *three* helper points, we have more values to relocate, but we also reduce the computational upper bound for each element (since the intervals decrease in length with an increasing number.) We can see that using approximately $log\, n$ helper values, we will maximize the benefits of the helper points, as we then, for every element to be computed, will have a helper point at a maximum distance of two steps away.

If we use this approximate number of helper values, the cost of computing the next value to be output is (on average) that of half a hash function evaluation, making the budget for relocation the dominating portion of the required total budget. When we relocate a helper point, the cost for computing its value at the wanted location is proportional to the distance (in the direction of the endpoint) to the next known value, whether this is a helper value or the endpoint itself. It is clear that the more helper points we employ, the lower this cost will

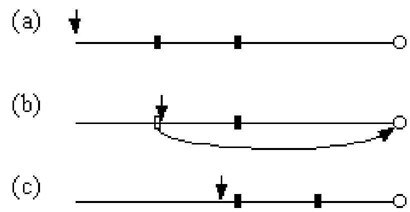

Figure 1: The figure shows the movement of helper values (squares) as the current position (small arrow) changes. In (a) the positions right after setup are shown; (b) shows the relocation of the first helper value as it has been reached. In (c), its relocation is completed.

be. However, the cost metric we are interested in minimizing is not computation alone, but the product of computational complexity and memory complexity.

For each element to be computed and output we assign a budget, corresponding to the computational upper bound per element. The computation of the next element has the highest priority to access this budget, since for each step, one has to compute and output the appropriate element. Any leftovers are assigned to the computation of helper values. These are partitioned into *high priority* helper values and *low priority* helper values. High priority helper values are relocated into already rather small intervals, located close to the current element (i.e., the element to be output in the current round). Low priority helper values, in turn, traverse larger distances, and further from the current element. The low priority helper values are only assigned those portions of the budget that remain after the current element has been computed, and the high priority helper values have exhausted their needs (i.e., arrived at their respective destinations.) Given that low priority helper values are traversing distances large enough to make it impossible for them to both get started and arrive in one and the same time interval, we can with only two such helper values guarantee that there always will be one left to "soak up" any computational leftovers.

During the setup phase, the endpoint of the chain is randomly selected, and the start point obtained by iterated hash function evaluation. This may be done by a device with less computational limitations than the device that later will compute and output the consecutive elements. During setup, the helper values will also be initialized. The first helper value will be placed at the mid-point between the endpoint and the starting point, thereby splitting the entire interval in two. The ith helper value will be placed on the midpoint between the position of the $i-1$st helper value and the starting point. Thus, each helper value will have a position and a value associated with them, where the value is obtained by iterated hash function application of the endpoint value or a previously placed helper value.

We show that our protocol is *almost optimal*. We do this by providing upper and lower bounds for the optimal solution to the problem. We wish to point out that while we do not know what the optimal solution is, we know that our solution (which is not the optimal) is "as good as one can get" – for all practical purposes. This does not count potential improvements leading to a simpler or shorter algorithm, but only refers to its memory and computational complexity.

Outline: We begin by introducing our method for computing the sequence of hash preimages, first laying out the preliminaries (section 3) and then elaborating on the protocol (section 4). In section 5, we present and prove our claims relating to the completeness and correctness of the protocol, and relating to the upper and lower bounds of the optimal solution to the problem we study.

3 Preliminaries

Definitions

We use the term *hash chain* H to mean a sequence of values $< v_0, v_1, \ldots, v_i, \ldots v_n >$, where v_n is a value chosen uniformly at random from $\{0,1\}^l$, and $v_i = h(v_{i+1})$, where $h : \{0,1\}^* \rightarrow \{0,1\}^l$ is a hash function or another publicly computable one-way function. We refer to v_0 as the *starting point*, and to v_n as the *endpoint*.

We define the *span* n to be the length of the hash chain, i.e., the number of elements in the sequence to be generated. We assume that $n = 2^\sigma$ for some integer $\sigma > 2$.

We define the *budget* b as the number of computational units allowed per element of the sequence that is output. Here, we only count hash function evaluations and not other computational steps associated with the protocol execution. This is reasonable given the fact that the computational effort of performing one hash function evaluation far exceeds the remaining work per step.

We refer to each helper value, and to the endpoint, as a *pebble*. Each pebble p_j has a position in the chain and a value associated with itself. The position of the start value is zero, and the position of the endpoint equals the span n. If *position* is the position of a pebble, then its value is $v_{position}$. Additionally, each pebble is associated with a *destination* (the position to which it is going); a *priority* (high or low); and an activity *status* (free, ready, active, arrived.) Here, pebbles that are not in use are referred to as *free*; these may be allocated to a particular task once the need arises. A pebble that is *ready* has been assigned a task, but has not yet started to move, while an *active* pebble in in motion. We use the ready state for a so-called "backup" pebble. This is a pebble that will become a low-priority pebble as soon as the active low-priority pebble reaches its destination. Finally, a pebble is assigned status *arrived* if it located at its destination point, and is still needed there (i.e., has not yet been reached by the "current" pointer, which corresponds to the position of the current output value.) We let k denote the number of pebbles used; the amount of storage needed is k times the amount needed per value, plus the amount (registers, etc.) needed for the execution of the protocol.

Goal

After performing a setup phase, we wish to generate the sequence H, element by element (and starting from v_1), using a minimal budget and a minimal number of pebbles. We will demonstrate a method with required budget $b = \lfloor \sigma/2 \rfloor$ and using $k = \sigma + \lceil log_2(\sigma + 1) \rceil$ pebbles, where $n = 2^\sigma$ is the number of elements of H.

Design Guidelines

If a pebble is located at the position corresponding to the current output, we say that this pebble has been *reached*, at which time it receives "free" status. All pebbles with status free are assigned a new position, destination, state and priority, according to guidelines that are set up to guarantee protocol completeness. To explain the proposed protocol, we present these guidelines along with their technical motivations.

"First things first". At each step, we first compute and output the appropriate hash chain value (which we call the *current* value); then, any remaining budget is assigned to active high-priority pebbles, starting with the pebble with the lowest position value (i.e., closest to the position associated with the current output value.) First then, any still remaining budget is assigned to active low-priority pebbles. This is to ensure that computational results that soon will be needed are ready on time.

Controlling high-priority pebbles. The high-priority pebbles are started at the "far end" of the first interval after the current value that does not already contain an active pebble, counting only intervals of size greater than two. In other words, if there is a pebble in *free* state, this will obtain the position and value of the first interval of size four or greater in which there is no active pebble, and will be given state *active*. Here, pebbles that just have been assigned to a position are considered to be in the interval in question. When the pebble reaches its destination (at the mid-point of the interval), it is given state *arrived*. Thus, if the resulting intervals (half the size of the original interval) are at least of size four, a new pebble may immediately be assigned a starting position equalling the position of the pebble that just reached its destination.

High-priority pebbles are only allowed to be active in positions lower than the active low-priority pebble, and are otherwise kept in *free* state. This is to make sure that high-priority pebbles do not take too much of the available budget: it slows them down to the benefit of low-priority pebbles when they complete their imminent tasks.

Controlling low-priority pebbles. We want there always to be a low-priority pebble that can "soak up" any remaining computational budget. We can achieve this by (1) having one "backup" pebble that is not assigned to any task, but which is ready to become the active low-priority pebble; and by (2) making each low-priority pebble traverse a distance that is sufficiently long that it and its backup cannot both complete before a new pebble becomes available. (When a new pebble does become available, it will be assigned to become a backup pebble if there is none, otherwise a high-priority pebble.)

According to our protocol, pebbles close to the current pointer have destinations set two steps apart. Therefore, assuming they will arrive in a timely fashion, they will be "virtually spaced" two steps from each other. Thus, a pebble will be reached by the current pointer every two moves (where a move is the computation performed between two consecutive outputs). If the distance low-priority pebbles need to travel from their inception until they reach their destination is at least twice the budget per move, then a new pebble will always be reached and relocated before the low-distance pebble and its backup reach their goals. Therefore, if the backup low-priority pebble is converted to an active low-priority pebble, a new backup pebble will be created before the converted pebble reaches its goal. Thus, our requirement will be satisfied.

By taking this approach, we can guarantee that the entire budget of each step will always be consumed, since there will always be an active low-priority pebble. According to our suggested approach, we only need *one* active low-priority pebble at the time, and one "backup" low-priority pebble.

4 Protocol

Setup. The endpoint v_n is chosen uniformly at random from $\{0,1\}^l$, where we may set $l = 160$. The sequence $H =< v_0, v_1, \ldots, v_i, \ldots, v_n >$ is computed by iterated application of the hash function h, where $v_i = h(v_{i+1})$, $0 \leq i < n-1$. Pebble p_j, $1 \leq j \leq \sigma$, for $\sigma = log_2\, n$, is initialized as follows:

$$\begin{cases} position \leftarrow 2^j \\ destination \leftarrow 2^j \\ value \leftarrow v_{2^j} \\ status \leftarrow \text{arrived.} \end{cases}$$

The remaining pebbles, p_j, $\sigma < j \leq k$, have their status set to free. All the pebble information is stored on the device we wish to later generate the hash sequence; this device also stores counters $current \leftarrow 0$ and $backup \leftarrow 0$, along with the span n. The pair $(startpoint, current) = (v_0, 0)$ is output. The starting point v_0 corresponds functionally to the public key of the chain.

Maintenance. In the following, we assume that the pebbles p_j, $1 \leq j \leq k$, are kept sorted with respect to their destination, with the lowest destination value first; and that pebbles that do not have a destination assigned appear last in the ordered list. Consequently, the next pebble to be reached (from the current position) will always be p_1. When the status of the pebbles is changed at any point, the altered item is inserted at the appropriate place in this sorted list. We let LP (short for low priority) be an alias of the active low-priority pebble. Thus, $LP.position$ is the current position of the active low-priority pebble, independently of what pebble number this corresponds to. Similarly, BU refers to the backup low-priority pebble.

Generation. The following protocol is performed in order to generate the hash sequence; each iteration of the protocol causes the next hash sequence value to be generated and output. The protocol makes use of two routines, $placeHP$ and $placeLP$; these assign values to high priority resp. low priority pebbles according to the previously given intuition. The corresponding algorithms will be described after the main routine is presented:

1. Set $available \leftarrow b$. *(Set the remaining budget.)*
2. Increase $current$ by 1.
3. If $current$ is odd then *(No pebble at this position.)*
   output $h(p_1.value)$, *(Compute and output.)*
   decrease $available$ by 1,
   else *(A pebble at this position.)*
   output $p_1.value$, *(Output value, set pebble free.)*
   set $p_1.status \leftarrow$ free,
   if $current = n$, then halt. *(Last value in sequence.)*
4. For all free pebbles p_j do *(Reassign free pebbles.)*
   if $backup = 0$ then *(Backup low-priority needed.)*
   $p_j.priority \leftarrow$ low,
   $p_j.status \leftarrow$ ready,
   $BU \leftarrow p_j$,
   $backup \leftarrow 1$,
   else
   Call `placeHP` (p_j) *(Make it high priority.)*
5. Sort pebbles.
6. Set $j \leftarrow 1$. *(First pebble first.)*
7. While $available > 0$ do
   if $p_j.status = active$ then *(Only move active pebbles.)*
   decrease $p_j.position$ by 1, *(Update its position...)*
   $p_j.value \leftarrow h(p_j.value)$, *(... and value...)*
   decrease $available$ by 1, *(... and do the accounting.)*
   if $p_j.position = p_j.destination$ then *(Pebble arrived!)*
   $p_j.status \leftarrow arrived$,
   if $p_j.priority = low$ then *(A low-priority pebble arrived.)*
   $LP \leftarrow BU$, *(Backup becomes low priority.)*
   $backup \leftarrow 0$,
   Call `placeLP`, *(Activate new low priority pebble!)*
   Sort pebbles.
   increase j by 1. *(Next pebble!)*
8. Sort pebbles.
9. Go to 1. *(Next element now.)*

Routine PlaceLP. We begin by describing how one could compute the sequence of values assigned to variables during calls to PlaceLP. (We later describe how just *one* such assignment can be computed, as opposed to an entire sequence. We also elaborate on a method that is less wasteful of stack space.) The wanted functionality of the routine is to compute the next starting point for a low-priority pebble, along with the associated destination.

In the following, we use "normalized" positions for ease of reading and uniformity over different spans. To get a real position from a normalized position, one multiples the latter by a value λ, where λ is the smallest power of two not smaller than $2b$, and where b is the budget. In other words, $\lambda = 2^{\lceil log_2 b \rceil + 1}$. Thus, the series of normalized starting points, starting with $(4, 8, 6, 8, 16, 12, 10, 12, 16, 14, 16)$, corresponds to a series $(32, 64, 48, 64, 128, 96, 80, 96, 128, 112, 128)$

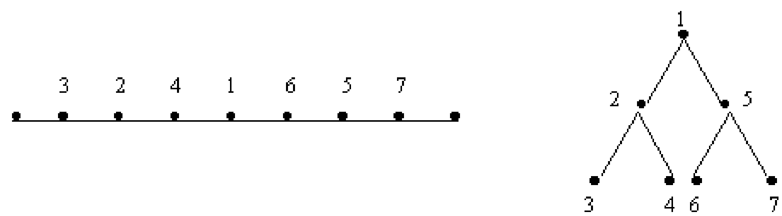

Figure 2: The left part of the figure shows the desired destinations of pebbles, where the number corresponds to the relative order of assignment. The relative order therefore corresponds to the depth-first traversal order of the corresponding tree, shown to the right. Note that the hash chain nodes correspond to a vertical projection of the tree nodes.

for $b = 4$, $\lambda = 8$. Similarly, the destination points and the distances between the starting points for the pebbles and their destinations are described in normalized terms.

When a free pebble is activized, it is placed on top of another pebble (located at its starting point) and given a destination. The destination is at the mid-point of the interval to the next pebble (in the direction of the current pointer). Thus, the intervals are split in two; as the pebble arrives, a new pebble is placed on top of it, with a destination of the next lower midpoint. If an interval is too small to be split, the pebble is instead placed at the end of the *next* interval.

We associate the first split of an interval with the root of a tree. The children of the root correspond to the splits of the resulting two intervals, and *their* children by their respective splits. The leaves correspond to the smallest splits of the intervals. Figure 2 shows the nodes of a small tree, and their order of traversal, and how these corresponds to the order of pebble placement in the hash chain. Figure 3 shows the starting points and destinations for one example tree, according to the assignment strategy described below.

We say that the height of a tree is the number of layers of nodes it has. (Thus, a tree consisting on only one node has height 1.) With each node of the tree, we associate a starting point; a distance; and a destination, where the destination is the difference between the starting point and the distance.

The normalized *starting point* for the root of a tree of height j is $start = 2^{j+1}$. The normalized *distance* of a node at height i in the tree is $dist = 2^{i-1}$; thus the distance of the root is 2^{j-1}, and leaves of the tree all have normalized distance $dist = 1$. The normalized *destination* $dest$ of any node (and the root in particular) is the difference between its starting point and distance, i.e., $dest = start - dist$. Finally, the starting point for a left child is its parent's destination value, $parent.dest$, while it is the parent's starting value $parent.start$ for a right child.

Consider the sequence of assignments of $start$ and $dest$ that one obtains from performing a depth first search of a given tree (with the left child always traversed before the right child). That is the sequence of assignments corresponding to the associated initial interval (i.e., the interval before splitting), as illustrated in figure 3. Consider further the sequence of such assignments one gets from traversing a forest of such trees, where the first tree has height one, and each tree is one level higher than its predecessor. That is the sequence of normalized

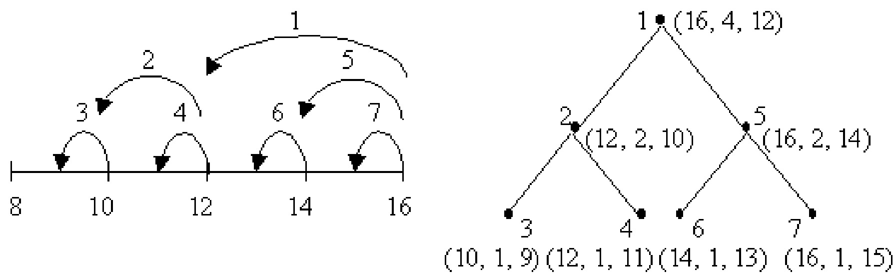

Figure 3: The left part of the figure shows the desired movement pattern of low priority pebbles in the interval between normalized positions 8 and 16, where the numbers on the arrows correspond to the relative order of the movements. The right part of the figure shows the tree structure from which these movements are obtained. The node number corresponds to the relative order of the movement, and is obtained by depth-first traversal of the tree. The triple associated with a node is the (starting point, distance, destination) of the corresponding pebble. A tree of height 2 is used for the normalized interval 4-8, and one of height 1 for the interval 2-4. Similarly, larger trees are used for later intervals.

assignments we need in our protocol.

Each call to PlaceLP first computes such a pair of normalized values, all from the above described sequence; these are then multiplied by λ and the product returned as the result of the function. Thus, it sets

$$\begin{cases} LP.priority \leftarrow low \\ LP.status \leftarrow active \\ LP.position \leftarrow \lambda\, start \\ LP.destination \leftarrow \lambda\, dest \end{cases}$$

As soon as $\lambda\, start > n$, no assignment is performed, since there is no need for low priority pebbles any longer. Any calls to `PlaceLP` after that return without any assignment.

Claim: The routine `PlaceLP` generates a sequence of elements, where the ith element of the sequence corresponds to the pair of starting position and destination of the ith low-priority pebble to be activated. The starting point corresponds to that of a pebble already placed on the hash chain, and the destination corresponds to the middle point between this same pebble and the closest pebble in the direction of the current pointer. The distance between starting point and destination is at least twice the budget per step that is available for moving pebbles, which guarantees that a recently placed low priority pebble does not reach its destination before another pebble has been activated. Once such a pebble does reach its destination, a "backup low-priority pebble" is activated by being turned into a standard (active) low-priority pebble. Once a new pebble is reached by the current pointer (at which time it is redundant in its current position, and needs to be relocated), such a pebble is made into a backup low-priority pebble, if the old such pebble has been activated.

Routine PlaceHP. The routine for computing the next location for a high-priority pebble is similar to the above, its main differences being that (1) the real and normalized values coincide, and (2) after the trees in the forest have reached height $log_2 \lambda$, they stop growing. Here, as before, $\lambda = 2^{\lceil log_2 b \rceil + 1}$, where b is the budget per output element produced.

The starting position of the jth tree is $start = 2^{j+1}$ for $j \leq log_2 \lambda$, and $start = \lambda(j - log_2 \lambda)$ for $j > log_2 \lambda$. As before, the starting point for a left child is its parent's destination value, *parent.dest*, while it is the parent's starting value *parent.start* for a right child. The *distance* of a node at height i in the tree is $dist = 2^{i-1}$. The *destination*, as before, is the difference between its starting point and destination, i.e., $dest = start - dist$.

Before any assignment is made, it is verified that $start \leq LP.position$, i.e., that the assignment in question is to a position before the active low priority pebble. If this is not the case, no assignment is made at this point, and the comparison re-performed at the next call to the routine. Otherwise, the following assignment is made to parameter p_j to the routine:

$$\begin{cases} p_j.priority \leftarrow \text{high} \\ p_j.status \leftarrow \text{active} \\ p_j.position \leftarrow start \\ p_j.destination \leftarrow dest \end{cases}$$

Claim: The routine `PlaceHP` generates a sequence of elements, where the ith element of the sequence corresponds to the pair of starting position and destination of the ith high-priority pebble to be activated. The starting point corresponds to that of a pebble already placed on the hash chain, and the destination corresponds to the middle point between this same pebble and the closest pebble in the direction of the current pointer. The starting point is chosen as a point between the current pointer and the active low-priority pebble, as close to the current pointer as possible, such that the distance between the starting point and the destination is at least two.

Memory Complexity. In order to conserve working space, one can opt for a solution that does not use a stack, but which recomputes the state from scratch when needed. One variable would store the height of the tree; one would store the number of steps (using depth-first search) from the root. Additionally, we need variables for *start*, *dist* and *dest*. To compute these values, we would (at each time) begin with their starting assignments, and then modify them according to the tree traversals leading to the wanted number of steps from the root. This is done in accordance with the respective assignments, as above.

The maximum tree height for `PlaceLP` is $\sigma - log_2 \lambda - 1$, since this corresponds to a normalized starting point of $2^{\sigma-\lambda}$ and a starting point of 2^σ. Thus, this variable needs $\lfloor log_2 \sigma \rfloor$ bits. For `PlaceHP`, the maximum height is $log_2 \lambda$, requiring $log_2 \lceil log_2 \lceil \lambda \rceil \rceil$ bits of storage. A tree of the maximum height has $2^{\sigma - log_2 \lambda - 1} - 1$ nodes for `PlaceLP`, and $2^\lambda - 1$ nodes for `PlaceHP`. Thus, the distances from the root can be represented with $\sigma - log_2 \lambda - 1$ respective λ bits. Finally, the maximum value of the last three variables is σ bits for each one of them, since the maximum value they can be assigned is 2^σ. (These only keep state within a computation, and so, we do not need one set for each algorithm.)

Therefore, the memory requirements of `PlaceLP` and `PlaceHP` are less than $4\sigma + log_2 \sigma + \lambda - 1$ bits. This is dwarfed by the memory requirements for the pebbles, requiring 160 bits each, resulting in a total of $160(\sigma + \lceil log_2(\sigma + 1) \rceil)$ bits.

5 Claims

Consider a span $n = 2^\sigma$, a budget $b = \lfloor \sigma/2 \rfloor$ and k pebbles, for $k = \sigma + \lceil log_2(\sigma+1) \rceil$.

We refer to the sum of the budgets from the setup stage until a particular step as the *cumulative budget* at the step in question. We say that the protocol with a budget restriction of b and a storage restriction of k *succeeds* at step j if and only if it outputs the jth value v_j of the hash sequence during this step, and that the protocol succeeded at step $j-1$. The protocol is said to succeed (by definition, and due to the setup procedure) at step 0 – this corresponds to the setup phase, on which we do not place any strict restrictions in terms of computation and storage.

Theorem 1: *(Completeness.)* The protocol succeeds at step j, $1 \leq j \leq n$, for a span $n = 2^\sigma$, budget $b = \lfloor \sigma/2 \rfloor$ and $k = \sigma + \lceil log_2(\sigma+1) \rceil$ pebbles.

The proof of the theorem will be based on the following lemmata, all of which relate to the above assignments for n, b and k. Recall that λ is defined to be the smallest power of two not smaller than $2b$.

Lemma 1: *(Bootstrapping.)* The protocol succeeds at step j, $1 \leq j \leq \lambda$.

Proof of Lemma 1:
We will consider two cases: $\lambda < 8$, and $\lambda \geq 8$. Recall that the cost of moving one step ahead equals the distance from the position we move to the next (forward) pebble. Recall also that λ is defined to be the smallest power of two equal to or larger than $2b$, and thus, $\lambda < 4b$.

The first case corresponds to the two possibilities $\lambda = 2$ or $\lambda = 4$. For $\lambda = 2$, we have $b = 1$. We need budget 1 to compute v_1 from v_2 (where there is a pebble), and zero budget to obtain v_2. For $\lambda = 4$, we know that $b = 2$. Since there are pebbles at positions 2 and 4, the budgets to reach these are zero, and the budget to reach step 1 and 3 from the step before is one. Therefore, the lemma holds for the first case.

For the second case, i.e., $\lambda \geq 8$, we know that there is one pebble at λ, one at $\lambda/2$, and one at $\lambda/4$, since we start with pebbles at each position that is a power of two, and which is in the interval between 0 and n. Consider the first two quarters of the interval between 0 and λ. Notice first that the budget will be sufficient for each step of these two quarters, since the pebbles are spaced $\lambda/4$ apart, and the budget is $b > \lambda/4$ (given how λ is defined). The traversal cost for both of the quarters is upper bounded by $(\lambda/4-1)+(\lambda/4-2)+\ldots+2+1+0 = (\lambda/4-1)\lambda/8$, giving a total of $(\lambda/4-1)\lambda/4$. The total budget assignment for this first half is $(\lambda/2)b$. Since $b > \lambda/4$, we have that the cumulative budget is $(\lambda/2)\lambda/4$. Thus, the budget "surplus" for the first half is at least $(\lambda/2)\lambda/4 - (\lambda/4-1)\lambda/4 = (\lambda/4+1)\lambda/4 > \lambda/4$.

This surplus will be applied to moving pebbles *outside* the first half of the interval, since any pebble movement *inside* the first half could only lower the expenditures for this portion, which would cause an even larger surplus. The surplus, in turn, is always spent on moving pebbles, with the closest high-priority pebbles receiving priority. The first pebble outside the first half of the interval would start at λ, and have destination $3\lambda/4$. Given that the surplus of the first half of the interval is at least $\lambda/4$, the pebble would reach its destination by the time the current position is $\lambda/2$.

Consider now the third and fourth quarter of the interval between 0 and λ. We have concluded that when the current position is $\lambda/2$, there are pebbles at both position $3\lambda/4$ and position λ (where the latter pebble has been there since the setup phase.) We know that

$b > \lambda/4$. Thus, the budget will be sufficient for each step of the two last quarters of the interval, and we see that the protocol will succeed at step λ. This concludes the proof.

Lemma 2: *(Discrete points.)* The protocol succeeds at step $j = 2^i\lambda + 1$, $0 \leq i \leq \sigma - log_2 \lambda$, if it succeeds at step $2^i\lambda$.

Proof of Lemma 2:
Assume that we have arrived at position $j = 2^i\lambda$ for some $0 \leq i \leq \sigma - log_2 \lambda - 1$. (This means that the protocol succeeded up until this point.) We wish to prove that the protocol will succeed for the next step, too. In order for this to occur, the next pebble must be at most $b+1$ steps away from step j, or the budget will not suffice. We will show that there will be a pebble at $j+2$, which would make the lemma hold, since $b \geq 1$.

At the time when we are at step j, the total incurred costs for pebbles equal the cost for filling the space between 0 and j with pebbles, plus that for populating (with exponentially increasing intervals) the interval between j and $2j$. We will argue that this sum is equal to the cost of populating the interval between 0 and j only, *were this interval empty.* This will result in a simpler way of determining the total required pebble expenditure up until step j.

Consider the location of all pebbles in the interval between 0 and j *at the start time.* These pebbles are located at positions $2, 4, 8, \ldots, j/2$. We want pebbles at positions $j+2, j+4, j+8, \ldots, j+j/2$ when we are at position j. Due to the setup, there will already be a pebble at $2j = 2^{i+1}\lambda$. The cost of placing the pebbles at $(j+2, j+4, j+8, \ldots, j+j/2)$, given the pebble at $2j$, is identical to the cost we would have incurred if we wanted to place pebbles at $2, 4, 8, \ldots, j/2$, given the pebble at j, since the relative distances from j resp. from $2j$ are identical for the two sequences. We can therefore substitute the cost for the real sequence between j and $2j$ by the hypothetical cost for the interval 0 to j.

Therefore, the total pebble expenditures at step j will equal the total pebble expenditures we were to incur if we were to fill an empty interval between 0 and j. The cost of filling an empty interval between 0 and j involves moving one pebble a distance $j/2$; two pebbles a distance $j/4$ each, four pebbles a distance of $j/8$ each, etc. Filling the space means that (over time) every even position in the interval has a pebble. Thus, the total pebble expenditure is $\sum_{\kappa=1}^{log_2 j-1} 2^{\kappa-1} j 2^{-\kappa}$, where $2^{\kappa-1}$ is the cardinality and $j2^{-\kappa}$ is the associated cost. This total expenditure can be seen to equal $j/2(log_2 j - 1)$.

If at any time we are either one or two steps from a pebble (where our position corresponds to the value being output), then the total expenditures for moving a step ahead is either zero (if we are at a position with an odd number, meaning next position has a pebble) or one (if we are at an evenly numbered position.) Thus, the total "stepping" expenditures up until step j are $j/2$.

We see that the total expenditures up until step $j = 2^i\lambda$ would be $j/2(log_2 j - 1) + j/2$ in order for there to be pebbles at positions $(j+2, j+4, \ldots, j+j/2)$ at the time the current position reaches step j. This equals $\frac{1}{2} j log_2 j$. Given that for each of the j steps, we are assigned a budget b, the total budget up until that point will be jb. We will be successful if $b \geq \frac{1}{2} log_2 j$. This quantity will be the largest for the end of the interval that the proof is valid for, i.e., $i = \sigma - log_2 \lambda - 1$. Plugging in i in the formula for j gives us $j = 2^{\sigma - log_2 \lambda - 1}\lambda = 2^{\sigma-1}$. Thus, if $b \geq \frac{1}{2}(\sigma - 1)$, then the lemma holds. According to the specifications, we have $b = \lfloor \sigma/2 \rfloor$. Therefore, if the protocol succeeds at $j = 2^i\lambda$, then it succeeds at $j+1 = 2^i\lambda + 1$, for $0 \leq j \leq n/2$, which concludes the proof.

Lemma 3: *(Intervals.)* The protocol succeeds at step $j = 2^{i+1}\lambda$, if it succeeds at step $2^i\lambda+1$.

Proof of Lemma 3:
Assume that the protocol succeeds at step $j = 2^i\lambda + 1$. We wish to prove that then, the protocol also succeeds at step $2j-2 = 2^{i+1}\lambda$. This will be shown using a symmetry argument. Consider intervals of size $2^i\lambda$. Consider first such an interval starting at position 1 and ending at $2^i\lambda$, and then one starting at position $2^i\lambda + 1$ and ending at $2^{i+1}\lambda$.

Assume that the current position is at the beginning of the second interval. Assume further that the relative positions of the pebbles in the second interval (in relation to the current position) are identical to the relative positions of the pebbles in the *first* interval (in relation to the current pointer when located in the beginning of the first interval). Then, the required expenditures to reach the end of the second interval from its beginning must equal the required expenditures to reach the end of the first. This is so since the resource allocation strategy is the same for both intervals, namely that pebbles close to the current position are given priority over pebbles further away. Therefore, if the budget is sufficient to reach the end-point of the first interval, then it is also sufficient to reach the end-point of the second.

We know that the first interval will have pebbles at positions $2, 4, 8, \ldots, j$ when we are at position 1. As was shown in Lemma 2, the cumulative budget available to pebbles at step j is sufficient for the placement of pebbles at positions $j + 2, j + 4, j + 8, \ldots j + j/2$, and we know that there already is one (due to the setup) at position $2j$. Therefore, the required expenditures in the interval between $2^i\lambda + 1$ and $2^{i+1}\lambda$ are the same as those for the interval between 1 and $2^i\lambda$. This concludes the proof.

Proof of Theorem 1: The theorem follows from the above lemmata, all of which are proven in the appendix. Lemma 1 establishes that the protocol succeeds for j, $1 \leq j \leq \lambda$. Then, Lemma 2 shows that it succeeds for $\lambda + 1$ (i.e., setting $i = 0$), and lemma 3 shows that it succeeds for all values up until 2λ (again using $i = 0$). Then, using $i = 1$, lemmata 2 and 3 establish that the protocol succeeds up to position 4λ. We apply lemmata 2 and 3 iteratively, and for increasing values of i, ending with $i = \sigma - 1 - log_2 \lambda$, finally establishing that the protocol succeeds for $j = n$, which completes the proof.

In the following, we show that our solution is *almost optimal*. More particularly, we consider the product of the number of hash function evaluations needed, and the number of storage cells required, where each storage cell holds one hash chain value and some short state information. Then, the complexity of the optimal solution is $\frac{1}{4}log^2 n$ per output element, while the complexity of our solution is *approximately* $\frac{1}{2}log^2 n$ – more precisely, it is $\frac{1}{2}\lfloor log\, n\rfloor(\lceil log_2 n\rceil + \lceil log_2(log_2 n + 1)\rceil)$. Thus, we are (practically speaking) no more than a factor of two away from the optimal solution in terms of computation-times-storage complexity.

Theorem 2: *(Lower bound.)* The optimal solution to the problem has a memory-times-computational complexity of at least $\frac{1}{4k}\lg^2 n$, where n is the length of the hash chain and k is the number of pebbles.

Proof of Theorem 2: We wish to show that the cumulative budget – the total cost of processing a string of length n using k pebbles – is at least $\frac{n}{4k}\lg^2 n$, which implies that the amortized cost per evaluation is at least $\frac{1}{4k}\lg^2 n$. This, in turn, will imply that the budget (worst-case cost per evaluation) is also at least $\frac{1}{4k}\lg^2 n$. The optimal case is $k = \frac{1}{2}\lg n$, where the amortized cost per evaluation is also $\frac{1}{2}\lg n$.

Let $g(n,k)$ be the required cumulative budget for covering a string of length n with k pebbles, excluding the initial cost (n) of setting up the pebbles.

Suppose that the furthest pebble is at position $n-T$, at the last time that it is used (cloned or used directly). Then we must cover an interval of length $n-T$ with $k-1$ pebbles (not using the kth pebble, which is stuck at position n), expend energy T to lay down the pebbles in the remaining interval, and cover that last interval with k pebbles. Optimizing over choice of T, we would have

$$g(n,k) = \min_T [g(n-T, k-1) + T + g(T,k)].$$

We want to show, by induction, that $g(n,k) \geq \frac{n \lg^2 n}{4k}$.

The inductive step will go through if we can show that, for all n, T, k, we have $h(k,T) \geq 0$, where for fixed n we define

$$h(k,T) = -\frac{n \lg^2 n}{4k} + \frac{(n-T)\lg^2(n-T)}{4(k-1)} + T + \frac{T \lg^2 T}{4k}.$$

For convenience we set $L = \lg n$. We evaluate h and its derivatives around the point $(k_0 = L/2, T_0 = n/2)$, which is near its global minimum. We have:

$$\begin{aligned}
h(L/2,n/2) &= -\tfrac{nL^2}{4(\frac{L}{2})} + \tfrac{\frac{n}{2}(L-1)^2}{4(\frac{L}{2}-1)} + \tfrac{n}{2} + \tfrac{\frac{n}{2}(L-1)^2}{4(\frac{L}{2})} \\
&= \tfrac{n(L-1)}{2L(L-2)} \approx \tfrac{n}{2\lg n}
\end{aligned}$$

$$\begin{aligned}
\tfrac{\partial h}{\partial k}(L/2,n/2) &= +\tfrac{nL^2}{4(\frac{L}{2})^2} - \tfrac{\frac{n}{2}(L-1)^2}{4(\frac{L}{2}-1)^2} - \tfrac{\frac{n}{2}(L-1)^2}{4(\frac{L}{2})^2} \\
&= \tfrac{n(-3L^2+6L-2)}{L^2(L-2)^2} \approx \tfrac{-3n}{\lg^2 n}
\end{aligned}$$

$$\begin{aligned}
\tfrac{\partial h}{\partial T}(L/2,n/2) &= -\tfrac{\lg^2(n/2)+2(\lg e)\lg(n/2)}{4(\frac{L}{2}-1)} + 1 + \tfrac{\lg^2(n/2)+2(\lg e)\lg(n/2)}{4(\frac{L}{2})} \\
&= -\tfrac{(L-1)^2+2(L-1)\lg e}{4(\frac{L}{2}-1)(\frac{L}{2})} + 1 \\
&= \tfrac{-(L-1)^2+2(L-1)\lg e+L(L-2)}{L(L-2)} \\
&= \tfrac{2(L-1)\lg e-1}{L(L-2)} \approx \tfrac{2\lg e}{\lg n}
\end{aligned}$$

$$\begin{aligned}
\tfrac{\partial^2 h}{\partial k^2}(L/2,n/2) &= -\tfrac{n\lg^2 n}{2(\frac{L}{2})^3} + \tfrac{\frac{n}{2}\lg^2\frac{n}{2}}{2(\frac{L}{2}-1)^3} + \tfrac{\frac{n}{2}\lg^2\frac{n}{2}}{2(\frac{L}{2})^3} \\
&= \tfrac{-(\frac{L}{2}-1)^3 nL^2 + ((\frac{L}{2})^3+(\frac{L}{2}-1)^3)\frac{n}{2}(L-1)^2}{2(\frac{L}{2})^3(\frac{L}{2}-1)^3} \\
&= \tfrac{n(4L^4+4L^3-44L^2+56L-16)}{L^3(L-2)^3} \approx \tfrac{4n}{\lg^2 n}
\end{aligned}$$

$$\begin{aligned}
\tfrac{\partial^2 h}{\partial k \partial T}(L/2,n/2) &= \tfrac{\lg^2(n/2)+2(\lg e)\lg(n/2)}{4(\frac{L}{2}-1)^2} - \tfrac{\lg^2(n/2)+2(\lg e)\lg(n/2)}{4(\frac{L}{2})^2} \\
&= \tfrac{(4L-4)((L-1)^2+2(\lg e)(L-1))}{L^2(L-2)^2} \approx \tfrac{4}{\lg n}
\end{aligned}$$

$$\begin{aligned}
\tfrac{\partial^2 h}{\partial T^2}(L/2,n/2) &= \tfrac{2(\lg e)(\lg e+\lg(n/2))/(n/2)}{4(\frac{L}{2}-1)} + \tfrac{2(\lg e)(\lg e+\lg(n/2))/(n/2)}{4(\frac{L}{2})} \\
&= \tfrac{(4L-4)(\lg e)(\lg e+(L-1))}{nL(L-2)} \approx \tfrac{4\lg e}{n}
\end{aligned}$$

Using the first-order approximations of the first and second derivatives, we calculate that the function $h(k,T)$ will reach its global minimum at about

$$\begin{array}{rcccl} k & = & k_0 + \frac{5\lg e}{4\lg e - 4} & \approx & \frac{\lg n}{2} + 4.07 \\ T & = & T_0 + (\frac{-2\lg e - 3}{4\lg e - 4})\frac{n}{L} & \approx & \frac{n}{2} - 3.32\frac{n}{\lg n} \end{array}$$

and its value there will be

$$\frac{n}{2\lg n} - O(\frac{n}{\lg^2 n}),$$

which is positive for n sufficiently large. This concludes the proof.

References

[1] M. Jakobsson, “Fractal Hash Sequence Representation and Traversal,” To appear in ISIT ’02; available at http://eprint.iacr.org/2002/001 and www.markus-jakobsson.com.

[2] G. Itkis and L. Reyzin, ”Forward-Secure Signatures with Optimal Signing and Verifying,” Crypto ’01, pp. 332–354.

[3] L. Lamport, ”Constructing Digital Signatures from a One Way Function,” SRI International Technical Report CSL-98 (October 1979).

[4] R. Merkle, ”A digital signature based on a conventional encryption function,” Proceedings of Crypto ’87.

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

[6] 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.

[7] 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.

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

[9] 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/

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

[11] S. Stubblebine and P. Syverson, ”Fair On-line Auctions Without Special Trusted Parties,” Financial Cryptography ’01.