Explicit Optimal Binary Pebbling for One-Way Hash Chain Reversal

Berry Schoenmakers

Dept of Mathematics & Computer Science
TU Eindhoven, The Netherlands
berry@win.tue.nl

Abstract. We present explicit optimal binary pebbling algorithms for reversing one-way hash chains. For a hash chain of length 2^k, the number of hashes performed per output round is at most $\lceil \frac{k}{2} \rceil$, whereas the number of hash values stored throughout is at most k. This is optimal for binary pebbling algorithms characterized by the property that the midpoint of the hash chain is computed just once and stored until it is output, and that this property applies recursively to both halves of the hash chain.
We develop a framework for easy comparison of explicit binary pebbling algorithms, including simple speed-1 binary pebbles, Jakobsson's binary speed-2 pebbles, and our optimal binary pebbles. Explicit schedules describe for each pebble exactly how many hashes need to be performed in each round. The optimal schedule exhibits a nice recursive structure, which allows fully optimized implementations that can readily be deployed. In particular, we develop in-place implementations with minimal storage overhead (essentially, storing only hash values), and fast implementations with minimal computational overhead.

1 Introduction

Originally introduced by Lamport to construct an identification scheme resisting eavesdropping attacks [Lam81], one-way hash chains have become a truly fundamental primitive in cryptography. The idea of Lamport's identification scheme is to generate a hash chain as a sequence of successive iterates of a one-way hash function applied to a random seed value, revealing only the last element of the hash chain upon registration. Later, during successive rounds of identification, the remaining elements of the hash chain are output in reverse order, one element at a time.

Due to the one-way property of the hash function, efficient reversal of a hash chain is non-trivial for long chains. In 2002, Jakobsson introduced a simple and efficient pebbling algorithm for reversal of one-way hash chains [Jak02], building on the pebbling algorithm of [IR01] for efficient key updates in a forward-secure digital signature scheme. Pebbling algorithms for one-way hash chain reversal strike a balance between storage requirements (measured as the number of hash values stored) and computational requirements (measured as the number of hashes performed). The performance constraint is that each next element of the reversed hash chain should be produced within a limited amount of time after producing the preceding element. For a hash chain of length $n = 2^k$, Jakobsson's algorithm stores $O(\log n)$ hash values only and the number of hashes performed in each round is $O(\log n)$ as well.

The problem of efficient hash chain reversal was extensively studied by Coppersmith and Jakobsson [CJ02]. They proved nearly optimal complexity for a binary pebbling algorithm storing at most $k + \lceil \log_2(k+1) \rceil$ hash values and performing at most $\lfloor \frac{k}{2} \rfloor$ hashes per round. Later, it was observed by Yum et al. that a greedy implementation of Jakobsson's original algorithm actually stores no more than k hash values, requiring no more than $\lceil \frac{k}{2} \rceil$ hashes per round [YSEL09].

In this paper we consider the class of *binary* pebbling algorithms, covering the best algorithms of [Jak02,CJ02,YSEL09] among others. A binary pebbling algorithm is characterized by the property that the *midpoint* of the hash chain is computed just once and stored until it is output; moreover, this property applies recursively to both halves of the hash chain. In particular, this means that after producing the last element of the hash chain as the first output, a binary pebbling algorithm stores (at least) the k elements at distances $2^i - 1$, for $1 \le i \le k$, from the end of the hash chain.

We introduce a simple yet general framework for efficient binary pebbling algorithms for hash chain reversal, and we completely resolve the case of binary pebbling by constructing an *explicit* optimal algorithm. The storage required by our optimal algorithm does not exceed the storage of k hash values and the number of hashes performed in any output round does not exceed $\lceil \frac{k}{2} \rceil$. This matches the performance of the greedy algorithm of [YSEL09], which is an optimal binary pebbling algorithm as well. However, we give an exact schedule for all hashes performed by the algorithm (rather than performing these hashes in a greedy fashion).

Our optimal schedule is defined explicitly, both as a recursive definition and as a closed formula, specifying exactly how many hashes should be performed in a given round by each pebble. Apart from the mathematically appealing structure thus uncovered, the explicit optimal schedule enables the development of fully optimized solutions for one-way hash chain reversal. We show the first in-place (or, *in situ*) hash chain reversal algorithms which require essentially no storage beyond the hash values stored for the pebbles; at the same time, the computational overhead for each round is limited to a few basic operations only beyond the evaluation of the hash function. Finally, as another extreme type of solution, we show how to minimize the computational overhead to an almost negligible amount of work, at the expense of increased storage requirements.

Concretely, for hash chains of length 2^{32} using a 128-bit one-way hash, our in-place algorithm only stores 516 bytes (32 hash values and one 32-bit counter) and performs, at most 16 hashes per round. We note that our results may be especially interesting in the context of post-quantum cryptography, as the security of one-way hash chains is not affected dramatically by the potential of quantum computers.

2 One-Way Hash Chains

Throughout, we use the following notation for finite sequences. We write $A = \{a_i\}_{i=1}^n = \{a_1, \ldots, a_n\}$ for a sequence A of length n, $n \geq 0$, with $\{\}$ denoting the empty sequence. We use $|A| = n$ to denote the length of A, and $\#A = \sum_{i=1}^n a_i$ to denote the weight of A. We write $A \parallel B$ for the concatenation of sequences A and B, and $A + B$ for element-wise addition of sequences A and B of equal length, where $+$ takes precedence over $\parallel$. Constant sequences are denoted by $c = c^{*n} = \{c\}_{i=1}^n$, suppressing the length n when it is understood from context; e.g., $A + c$ denotes the sequence obtained by adding c to all elements of A.

Let f be a cryptographic hash function. The length-2^k (one-way) hash chain $f_k^*(x)$ for seed value x is defined as the following sequence:

$$f_k^*(x) = \{f^i(x)\}_{i=0}^{2^k-1}.$$

For authentication mechanisms based on hash chains, we need an efficient algorithm for producing the sequence $f_k^*(x)$ *in reverse*. The problem arises from the fact that computation of f in the forward direction is easy, while it is *intractable* in the reverse direction. So, given x it is easy to compute $y = f(x)$, but given y it is very hard to compute any x at all such that $y = f(x)$. For long hash chains the straightforward solutions of either (i) storing $f_k^*(x)$ and reading it out in reverse or (ii) computing each element of $f_k^*(x)$ from scratch starting from x are clearly too inefficient.

3 Binary Pebbling

We introduce a framework that captures the essence of binary pebbling algorithms as follows. We define a pebble as an algorithm proceeding in a certain number of rounds, where the initial rounds are used to compute the hash chain in the forward direction given the seed value x, and the hash chain is output in reverse in the remaining rounds, one element at a time.

For $k \geq 0$, we define pebble $P_k(x)$ below as an algorithm that runs for $2^{k+1}-1$ rounds in total, and outputs $f_k^*(x)$ in reverse in its last 2^k rounds. It is essential that we include the initial $2^k - 1$ rounds in which no outputs are produced as an integral part of pebble $P_k(x)$, as this allows us to define and analyze binary pebbling in a fully recursive manner. In fact, in terms of a given **schedule** $T_k = \{t_r\}_{r=1}^{2^k-1}$ with $\#T = 2^k - 1$, a binary pebble $P_k(x)$ is completely specified by the following recursive definition, see also Figure 1:

Rounds $[1, 2^k)$: Compute $\{y_i = f^{2^k-2^i}(x)\}_{i=0}^k$ using t_r hashes in round r.
Round 2^k: Output y_0.
Rounds $(2^k, 2^{k+1})$: Run $P_{i-1}(y_i)$ in parallel for $i = 1, \ldots, k$.

We will refer to rounds $[1, 2^k)$ as the **initial stage** of pebble P_k and to rounds $[2^k, 2^{k+1})$ as its **output stage**. Running pebbles in parallel means that pebbles take turns to execute for one round each, where the order in which this happens within a round is irrelevant.

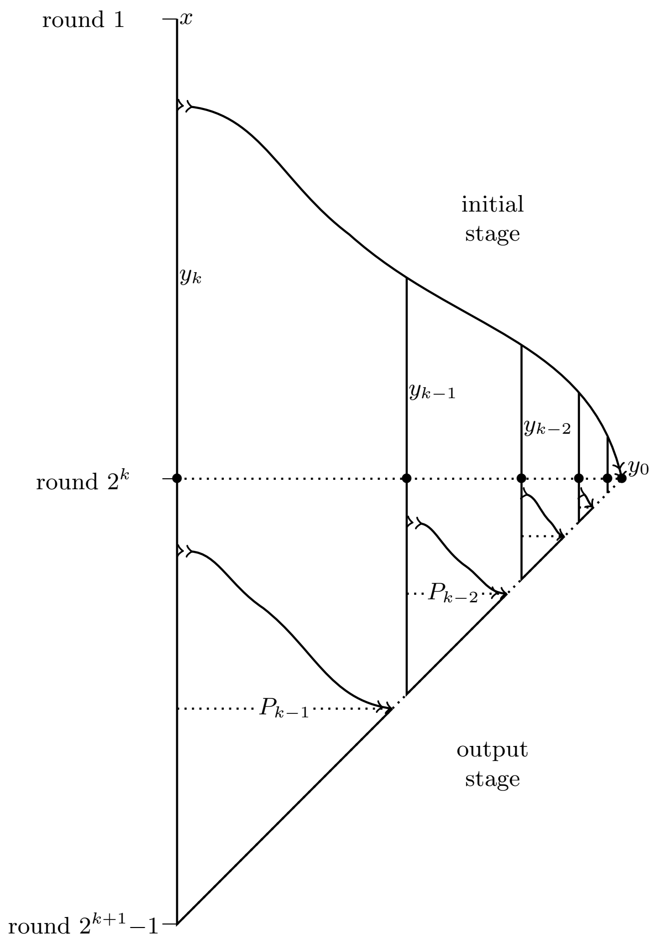

Fig. 1. Binary pebble $P_k(x)$, where $y_i = f^{2^k-2^i}(x)$ for $i = k, \ldots, 0$.

The behavior of pebbles P_0 and P_1 is fixed since $T_0 = \{\}$ and $T_1 = \{1\}$, respectively. Pebble $P_0(x)$ runs for one round only, in which $y_0 = x$ is output, using no hashes at all. Similarly, $P_1(x)$ runs for three rounds, performing one hash in its first round to compute $y_1 = x$ and $y_0 = f(x)$, outputting $f(x)$ in its second round, and then running $P_0(y_1)$ in the third round, which will output x. More generally, the following theorem shows that correct behavior follows for any pebble P_k independent of the particular schedule T_k, and furthermore that the total number of hashes performed by P_k is fixed as well.

Theorem 1. *Pebble $P_k(x)$ produces $f_k^*(x)$ in reverse in its output stage, performing $k2^{k-1}$ hashes in total.*

Proof The proof is by induction on k. For $k = 0$, we have that $P_0(x)$ outputs $f_0^*(x) = x$ in its one and only round, using 0 hashes.

For $k \geq 1$, we see that $P_k(x)$ first outputs $y_0 = f^{2^k-1}(x)$ in round 2^k, which is the last element of $f^*_k(x)$. Next, pebbles $P_{i-1}(y_i)$ run in parallel for $i = 1, \ldots, k$. The induction hypothesis yields that each $P_{i-1}(y_i)$ produces $f^*_{i-1}(f^{2^k-2^i}(x))$ in reverse in its last 2^{i-1} out of $2^i - 1$ rounds. Hence, in round $2^k + 1$, $P_0(y_1)$ outputs $y_1 = f^*_0(f^{2^k-2}(x))$. In the next two rounds, $P_1(y_2)$ outputs $f^*_1(f^{2^k-4}(x))$ in reverse. And so on until finally $P_{k-1}(y_k)$ outputs $f^*_{k-1}(f^{2^{k-1}}(x))$ in reverse in the last 2^{k-1} rounds of $P_k(x)$. The total number of hashes performed by P_k is $2^k - 1 + \sum_{i=1}^{k}(i-1)2^{i-2} = k2^{k-1}$, using that P_{i-1} performs $(i-1)2^{i-2}$ hashes per the induction hypothesis. □

Schedule T_k specifies the number of hashes for the initial stage of P_k. To analyze the work done by P_k in the output stage, we introduce sequence W_k of length $2^k - 1$ to denote the number of hashes performed by P_k in each of its last $2^k - 1$ rounds—noting that by definition no hashes are performed by P_k in round 2^k. The following recurrence relation for W_k will be used throughout our analysis.

Lemma 1. $W_0 = \{\}, \quad W_k = T_{k-1} + W_{k-1} \parallel 0 \parallel W_{k-1}$.

Proof Pebble P_0 runs for 1 round only, so $W_0 = \{\}$. For $k \geq 1$, we see that in the last $2^k - 1$ rounds of pebble P_k, a pebble P_{k-1} runs in parallel to pebbles P_i for $i = 0, \ldots, k-2$. In these rounds, pebble P_{k-1} performs $T_{k-1} \parallel 0 \parallel W_{k-1}$ hashes, whereas pebbles P_i for $i = 0, \ldots, k-2$ perform $W_{k-1} \parallel 0^{*2^{k-1}}$ hashes in total, as this matches the number of hashes for a pebble P_{k-1} in its last $2^{k-1} - 1$ rounds. Hence, in total $W_k = T_{k-1} + W_{k-1} \parallel 0 \parallel W_{k-1}$ hashes. □

We have the following lower bound for $\max(W_k)$, the maximal number of hashes performed by P_k in any round of the output stage. Interestingly, this lower bound holds for *any* schedule T_k. In Section 5 we will present an optimal schedule achieving the lower bound.

Theorem 2. $\max(W_k) \geq \lceil k/2 \rceil$, *for* $k \geq 2$.

Proof Let $k \geq 2$ and consider the average number of hashes per round during the first half of the output stage. From Theorem 1, Lemma 1, and $|T_{k-1}| = |W_{k-1}| = 2^{k-1} - 1$, we have

$$\max(W_k) \geq \frac{\#T_{k-1} + \#W_{k-1}}{|T_{k-1} + W_{k-1}|} = \frac{(k-1)2^{k-2}}{2^{k-1} - 1} > \frac{k-1}{2}.$$

Hence, $\max(W_k) \geq \lceil k/2 \rceil$. □

To analyze the storage needed by P_k we will count the number of hash values stored by P_k at the start of each round. We introduce sequence $S_k = \{s_r\}_{r=1}^{2^{k+1}-1}$ to denote the total storage used by P_k in each round. For instance, $s_1 = 1$ as P_k only stores x at the start, and $s_{2^k} = k + 1$ as P_k stores $y_0, \ldots, y_k$ at the start of round 2^k independent of schedule T_k.

4 Analysis of Speed-1 and Speed-2 Pebbles

In this section we analyze the performance of speed-1 pebbles and speed-2 pebbles. We use speed-1 pebbles to demonstrate our framework, whereas the analysis of speed-2 pebbles, which correspond to Jakobsson's original algorithm, will be used in the analysis of our optimal pebbles in the next section.

Speed-1 pebbles are defined by setting $T_k = 1^{*2^k-1}$, hence one hash evaluation in each initial round of P_k. To define speed-2 pebbles we set $T_0 = \{\}$, and $T_k = 0^{*2^{k-1}-1} \parallel 2^{*2^{k-1}-1} \parallel 1$ for $k \geq 1$, hence a speed-2 pebble is idle in the first part of the initial stage and then hashes twice in each round until the end of the initial stage. As can be seen from Theorem 4 below, the storage requirements are reduced by a factor of 2 for speed-2 pebbles over speed-1 pebbles.

Theorem 3. *Both speed-1 and speed-2 pebbles P_k use up to* $\max(W_k) = k-1$ *hashes in any output round, for $k \geq 1$.*

Proof For a speed-1 pebble, Lemma 1 implies $\max(W_k) = k-1$ for $k \geq 1$, as all elements of T_{k-1} are equal to 1.

For a speed-2 pebble we prove by induction on k that $\max(W_k) = k-1$. This clearly holds for $k = 1, 2$. For $k \geq 3$, we have, using Lemma 1,

$$\begin{aligned} T_{k-1} &= \quad 0^{*2^{k-2}-1} \quad \parallel 2^{*2^{k-2}-1} \parallel 1 \\ W_{k-1} &= 0 \parallel T_{k-2} + W_{k-2} \parallel \; 0 \parallel W_{k-2}. \end{aligned}$$

Therefore,

$$\max(W_k) = \max(T_{k-1} + W_{k-1}) = \max(W_{k-1}, 2 + W_{k-2}),$$

noting that the last element of $W_{k-2} = 0$. Applying the induction hypothesis twice, we conclude $\max(W_k) = \max(k-2, k-1) = k-1$. □

Lemma 2.

$$\begin{aligned} &S_0 = \{1\}, \\ &S_k = (1^{*2^k} \parallel S_{k-1}) + (0 \parallel 1^{*2^{k-1}-1} \parallel S_{k-1} \parallel 0^{*2^{k-1}}), \text{ for a speed-1 } P_k, \\ &S_k = (1^{*2^k} \parallel S_{k-1}) + (0^{*2^{k-1}} \parallel S_{k-1} \parallel 0^{*2^{k-1}}), \qquad \text{for a speed-2 } P_k. \end{aligned}$$

Proof Pebble $P_0(x)$ only needs to store x during its one and only round, therefore $S_0 = \{1\}$. For $k \geq 1$, any pebble $P_k(x)$ also needs to store x throughout all of its rounds, where pebble $P_{k-1}(y_k) = P_{k-1}(x)$ takes over the storage of x during the output stage. This accounts for the term $1^{*2^k} \parallel S_{k-1}$.

In addition, a speed-1 pebble needs to store a hash value from round 2 until it reaches y_{k-1} in round 2^{k-1}. From thereon, the total additional storage corresponds to running a speed-1 $P_{k-1}(y_{k-1})$ pebble. This accounts for the term $0 \parallel 1^{*2^{k-1}-1} \parallel S_{k-1} \parallel 0^{*2^{k-1}}$.

A speed-2 pebble needs no additional storage during its first 2^{k-1} rounds. Then it needs to store an additional hash value from round $2^{k-1}+1$ on. By taking $0^{*2^{k-1}} \parallel S_{k-1} \parallel 0^{*2^{k-1}}$ as additional term, we account for both the additional hash value stored by a speed-2 pebble during rounds $(2^{k-1}, 2^{k-1}+2^{k-2}]$ and the storage corresponding to a speed-2 $P_{k-1}(y_{k-1})$ pebble, running from round $2^{k-1}+1$. □

Theorem 4. *A speed-1 pebble P_k uses up to $\max(S_k) = \max(k+1, 2k-2)$ storage, and a speed-2 pebble P_k uses up to $\max(S_k) = k+1$ storage.*

Proof Using that $s_{2^k} = k+1$, we write $S_k = A_k \parallel k{+}1 \parallel B_k$, where $|A_k| = |B_k| = 2^k - 1$.

For a speed-1 pebble P_k, it can easily be checked that $\max(S_k) = \max(k+1, 2k-2)$ holds for $k = 0, 1$. To prove this for $k \geq 2$, we note that it suffices to show $\max(A_k, B_k) = 2k-2$, as $\max(S_k) = \max(A_k, k+1, B_k)$. Lemma 2 implies

$$\begin{aligned} A_k &= 1 \parallel 2^{*2^{k-1}-1} \parallel 1 + A_{k-1} \\ B_k &= A_{k-1} + B_{k-1} \parallel k \parallel B_{k-1}, \end{aligned}$$

so we have that $\max(A_k, B_k) = \max(A_{k-1}+B_{k-1}, k) = \max(2k-2, k) = 2k-2$ follows if we can show $\max(A_k + B_k) = 2k$, for $k \geq 1$. We prove the latter by induction on k. For $k = 1$, clearly true as $A_1 = B_1 = \{1\}$. For $k \geq 2$, we see that $\max(A_k+B_k) = \max(2+A_{k-1}+B_{k-1}, k+2) = \max(2k, k+2) = 2k$ follows from the induction hypothesis, also using that the first element of $A_{k-1} + B_{k-1} = k$.

For a speed-2 pebble P_k, we note that $\max(S_k) = k+1$ follows from the fact that $A_k + B_k = k+1$, which we show by induction on k. For $k = 0$, $A_k + B_k = k+1$ is vacuously true, as A_0, B_0 are empty sequences. For $k \geq 1$, we see from Lemma 2 that

$$\begin{aligned} A_k &= 1^{*2^{k-1}} \parallel 1 + A_{k-1} \\ B_k &= A_{k-1} + B_{k-1} \parallel k \parallel B_{k-1}. \end{aligned}$$

Thus, from the induction hypothesis we have $A_{k-1}+B_{k-1} = k$, hence $A_k+B_k = k+1$. □

5 Optimal Binary Pebbles

In this section, we will reduce the maximal number of hashes per round from $k-1$ for a speed-2 pebble P_k to $\lceil k/2 \rceil$ for an optimal pebble P_k, without increasing the storage requirements. We do so by letting our optimal pebbles P_k be idle for the first $2^{k-1}-1$ rounds, just as speed-2 pebbles do. During rounds $[2^{k-1}, 2^k)$, an optimal pebble will work at varying speeds, roughly as follows: the average speeds in each quarter are 2, 1, 2, and 3 hashes per round, respectively. To streamline the presentation, we will at first allow "$\frac{1}{2}$ hashes" in the definition of our optimal schedule. At the end of this section, we will show how to round the schedule to integer values without affecting optimality.

We define optimal schedule T_k as follows:

$$T_0 = \{\}, \quad T_k = 0^{*2^{k-1}-1} \parallel U_k \parallel V_k,$$

where

$$U_1 = \{1\}, \quad U_k = \tfrac{1}{2} + U_{k-1} \parallel 1^{*\lfloor 2^{k-3} \rfloor},$$
$$V_1 = \{\}, \quad\ \ V_k = \tfrac{1}{2} + U_{k-1} \parallel \tfrac{1}{2} + V_{k-1}.$$

For example, $T_1 = \{1\}$, $T_2 = \{0, \frac{3}{2}, \frac{3}{2}\}$, and $T_3 = \{0, 0, 0, 2, 1, 2, 2\}$.

Optimality is proved in the next two theorems.

Theorem 5. *An optimal pebble P_k uses up to $\max(W_k) = k/2$ hashes in any output round, for $k \geq 2$.*

Proof We use Lemma 1 without explicitly referring to it.

Since $\max(W_k) = \max(T_{k-1} + W_{k-1})$, we obtain $\max(W_k) = k/2$, if we prove by induction on k that

$$T_k + W_k = T_{k-1} + W_{k-1} \parallel \tfrac{k+1}{2}^{*2^{k-1}}.$$

This property clearly holds for $k = 1, 2$. For $k \geq 3$, the definition of T_k implies that the property is in fact equivalent to

$$(U_k \parallel V_k) + (0 \parallel W_{k-1}) = \tfrac{k+1}{2}^{*2^{k-1}}.$$

From the definition of U_k, V_k and the induction hypothesis for $T_{k-2} + W_{k-2}$ we obtain

$$\begin{array}{rcl} U_k \parallel V_k & = & \frac{1}{2} + U_{k-1} \parallel 1^{*2^{k-3}} \parallel \frac{1}{2} + (U_{k-1} \parallel V_{k-1}), \\ 0 \parallel W_{k-1} & = & 0 \parallel T_{k-3} + W_{k-3} \parallel \frac{k-1}{2}^{*2^{k-3}} \parallel 0 \parallel W_{k-2}. \end{array}$$

Since $0 \parallel T_{k-3} + W_{k-3}$ is equal to the first half of $0 \parallel W_{k-2}$, we get from the induction hypothesis that indeed all elements of $(U_k \parallel V_k) + (0 \parallel W_{k-1})$ are equal to $\frac{k+1}{2}$. □

Let $\text{len}(x) = \lceil \log_2(x+1) \rceil$ denote the bit length of nonnegative integer x. The next two lemmas give closed formulas for the optimal schedule T_k and its partial sums. Lemma 4 will be used to prove Theorem 6, but these formulas also provide the basis for our efficient in-place implementation of optimal binary pebbling.

Lemma 3. *For optimal schedule $T_k = \{t_r\}_{r=1}^{2^k-1}$, we have for $2^{k-1} \leq r < 2^k$:*

$$t_r = \tfrac{1}{2}\left(k + 1 - \text{len}\left((2r) \bmod 2^{\text{len}(2^k - r)}\right)\right).$$

Proof The proof is by induction on k. For $0 \leq k \leq 2$, the formula is easily checked. For $k \geq 3$, we distinguish two cases.

Case $2^{k-1} \leq r < 2^{k-1} + 2^{k-2}$. We first note that $(2r) \bmod 2^{\text{len}(2^k-r)} = 2r - 2^k$. If $r \geq 2^{k-1} + 2^{k-3}$, we have $t_r = 1$ by definition and we see the formula for t_r yields 1 as well as $\text{len}(2r - 2^k) = k - 1$. Otherwise $r < 2^{k-1} + 2^{k-3}$, hence we have $t_r = t_{r+2^{k-2}}$. So, this case reduces to the case below by noting that also $(2(r + 2^{k-2})) \bmod 2^{\text{len}(2^k - (r + 2^{k-2}))} = 2r - 2^k$.

Case $2^{k-1}+2^{k-2} \le r < 2^k$. From the definition of the optimal schedule we see that in this case $t_r = \frac{1}{2} + t'_{r-2^{k-1}}$, where $T_{k-1} = \{t'_z\}_{z=1}^{2^{k-1}-1}$. From the induction hypothesis we get:

$$t'_{r-2^{k-1}} = \tfrac{1}{2}\left(k - \mathrm{len}((2(r-2^{k-1})) \bmod 2^{\mathrm{len}(2^{k-1}-(r-2^{k-1}))})\right).$$

Rewriting this formula for $t'_{r-2^{k-1}}$ we obtain

$$t_r = \tfrac{1}{2} + \tfrac{1}{2}\left(k - \mathrm{len}((2r-2^k) \bmod 2^{\mathrm{len}(2^k-r)})\right).$$

Noting that $\mathrm{len}(2^k - r) \le k$, we see that the formula holds for t_r as well. □

Lemma 4. *For optimal schedule $T_k = \{t_r\}_{r=1}^{2^k-1}$, we have for $0 \le j \le 2^{k-1}$:*

$$\sum_{r=2^k-j}^{2^k-1} t_r = \tfrac{1}{2}\left(j(k-m) + (m+3-l)2^l - 2^m\right) - 1,$$

where $l = \mathrm{len}(j)$ and $m = \mathrm{len}(2^l - j)$.

Proof The proof is by induction on j. For $j = 0$, both sides are equal to 0.

For $1 \le j \le 2^{k-1}$, Lemma 3 implies that

$$t_{2^k-j} = \tfrac{1}{2}\left(k + 1 - \mathrm{len}((-2j) \bmod 2^l)\right).$$

Combined with the induction hypothesis for $j-1$ we obtain

$$\sum_{r=2^k-j}^{2^k-1} t_r = \tfrac{1}{2}\left(j(k-m') + m'+1 - \mathrm{len}((-2j) \bmod 2^l) + (m'+3-l')2^{l'} - 2^{m'}\right) - 1,$$

where $l' = \mathrm{len}(j-1)$ and $m' = \mathrm{len}(2^{l'} - j + 1)$. We distinguish two cases.
Case $l' = l - 1$. This means that $j = 2^{l-1}$, and hence $m = l$ and $m' = 1$. We are done as both sides are equal to $\frac{1}{2}j(k+4-l) - 1$.
Case $l' = l$. This means that $2^{l-1} < j < 2^l$, hence $0 < 2^{l+1} - 2j < 2^l$. This implies $\mathrm{len}((-2j) \bmod 2^l) = m+1$, so we see that both sides are equal if $m = m'$. If $m' = m + 1$, we see that $2^l - j = 2^m - 1$ and that therefore both sides are equal as well. □

Theorem 6. *An optimal pebble P_k uses up to $\max(S_k) = k + 1$ storage.*

Proof We prove that the storage requirements of an optimal pebble do not exceed the storage requirements of a speed-2 pebble, hence that $\max(S_k) = k+1$ for an optimal pebble as well.

Consider the rounds in which a speed-2 pebble and an optimal pebble store the values $y_i = f^{2^k-2^i}(x)$ for $i = k, \ldots, 1$. We claim that an optimal pebble will never store y_i before a speed-2 pebble does. Clearly, a speed-2 pebble stores y_i in round $2^k - 2^{i-1}$ for $i = k, \ldots, 1$. However, in round $2^k - 2^{i-1}$ an optimal pebble still has to compute at least as many hashes as a speed-2 pebble needs to reach y_0:

$$\sum_{r=2^k-2^{i-1}}^{2^k-1} t_r = 2^{i-2}(k+4-i) - 1 \geq 2^i - 1,$$

using Lemma 4 for $j = 2^{i-1}$. □

As a final step we will round the optimal schedule T_k to integer values, without affecting optimality. For example, we round $T_2 = \{0, \frac{3}{2}, \frac{3}{2}\}$ to $\{0, 1, 2\}$ or to $\{0, 2, 1\}$. In general, we make sure that if an element is rounded up then its neighbors are rounded down, and vice versa. The rounding also depends on the parity of k to alternate between rounding up and rounding down. Hence, we define the rounded optimal schedule by:

$$t_r = \left\lfloor \tfrac{1}{2} \left((k+r) \bmod 2 + k + 1 - \operatorname{len}((2r) \bmod 2^{\operatorname{len}(2^k - r)}) \right) \right\rfloor, \qquad (1)$$

for $2^{k-1} \leq r < 2^k$. Accordingly, we see that optimal pebble P_k will use up to $\max(W_k) = \lceil k/2 \rceil$ hashes in any output round, matching the lower bound of Theorem 2.

6 Optimized Implementations

A hash chain is deployed as follows as part of an authentication mechanism like Lamport's identification scheme. Given a random seed value x, the initial stage of any type of binary pebble $P_k(x)$ is simply performed by iterating the hash function f and storing the values $y_i = f^{2^k - 2^i}(x)$ for $i = k, \ldots, 0$. The value of y_0 is then output, e.g., as part of the registration protocol for Lamport's scheme. The other hash values $y_1, \ldots, y_k$ are stored for the remainder of the output stage, e.g., for use in later runs of Lamport's identification protocol.

The initial stage is preferably executed inside the secure device that will later use the hash chain for identification. However, for lightweight devices such as smart cards, RFID tags, sensors, etc., the initial stage will typically be run on a more powerful device, after which hash values $y_1, \ldots, y_k$ will be inserted in the lightweight device and hash value y_0 can be used for registration.

To implement the output stage of a pebble P_k one needs to handle potentially many pebbles all running in parallel. The pseudocode in [Jak02,CJ02,YSEL09] suggests rather elaborate techniques for keeping track of the (state of) pebbles. On the contrary, we will show how to minimize storage and computational overhead by exploiting specific properties of Jakobsson's speed-2 pebbles and our optimal pebbles. In particular, we present in-place hash chain reversal algorithms, where the entire state of these algorithms (apart from the hash values) is represented between rounds by a single k-bit counter only.

We introduce the following terminology to describe the state of a pebble P_k. This terminology applies to both speed-2 pebbles and optimal pebbles. Pebble P_k is said to be **idle** if it is in rounds $[1, 2^{k-1})$, **hashing** if it is in rounds $[2^{k-1}, 2^k]$, and **redundant** if it is in rounds $(2^k, 2^{k+1})$. An idle pebble performs no hashes at all, while a hashing pebble will perform at least one hash per round, except for round 2^k in which P_k outputs its y_0 value. The work for a redundant pebble P_k is taken over by its child pebbles $P_0, \ldots, P_{k-1}$ during its last $2^k - 1$ output rounds.

The following theorem provides the basis for our in-place algorithms by showing precisely how the state of all pebbles running in parallel during the output stage of a pebble P_k can be determined from the round number. Let $x_i \in \{0,1\}$ denote the i-th bit of nonnegative integer x, $0 \le i < \mathrm{len}(x)$.

Theorem 7. *For a speed-2 pebble or optimal pebble P_k in output round $2^{k+1} - c$, $1 \le c \le 2^k$, we have for every i, $0 \le i \le k$, exactly one non-redundant pebble P_i present if and only if bit $c_i = 1$, and if present, P_i is in round $2^i - (c \bmod 2^i)$.*

Proof The proof is by induction on c. For $c = 2^k$, only $c_k = 1$, which corresponds to pebble P_k being the only pebble around. Also, P_k is in its 2^kth round.

For $1 \le c < 2^k$, write $c' = c + 1$ and let $k' \ge 0$ be maximal such that $c' \bmod 2^{k'} = 0$. Hence $c'_{k'} = 1$. By the induction hypothesis for c', pebble $P_{k'}$ is in its first output round $2^{k'}$. So, in the next round $P_{k'}$ becomes redundant, and is replaced by child pebbles $P_{k-1}, \ldots, P_0$ who will all be in their first round. As $c = c' - 1$, this corresponds to the fact that $c_{k'} = 0$ and $c_{k'-1} = \cdots = c_0 = 1$, also noting that $2^i - (c \bmod 2^i) = 1$ for $i = k' - 1, \ldots, 0$.

For $i > k'$, we have $c_i = c'_i$. All non-redundant pebbles in round $2^{k+1} - c'$ remain so in round $2^{k+1} - c$, and for these pebbles the round number becomes $2^i - (c' \bmod 2^i) + 1 = 2^i - (c \bmod 2^i)$, as required. □

As a corollary, we see that a non-redundant pebble P_i is hashing precisely when $c_{i-1} = 0$, and P_i is idle otherwise, since for $c_i = 1$ we have that $c_{i-1} = 0$ if and only if $2^i - (c \bmod 2^i) \ge 2^{i-1}$. This holds also for $i = 0$ if we put $c_{-1} = 0$.

6.1 In-Place Speed-2 Pebbles

We present an in-place implementation of a speed-2 pebble P_k for which the overall storage is limited to the space for k hash values and one k-bit counter c. As explained above, we will assume that hash values $y_1, \ldots, y_k$ are given as input and that y_0 has been output already. Thus, P_k has exactly $2^k - 1$ output rounds remaining. We use c to count down the output rounds.

The basis for our in-place algorithm is given by the next theorem.

Theorem 8. *For a speed-2 pebble P_k in output round $2^{k+1} - c$, $1 \le c \le 2^k$, each non-redundant pebble P_i present stores $t+1$ hash values, where t is maximal such that $c_{i-1} = \ldots = c_{i-t} = 0$ with $0 \le t \le i$.*

Proof From Theorem 7 it follows that non-redundant pebble P_i is in round $r = 2^i - (c \bmod 2^i)$. Since $0 \le t \le i$ is maximal such that $c_{i-1} = \ldots = c_{i-t} = 0$, we have that $2^i - 2^{i-t} < r \le 2^i - 2^{i-t-1}$. This implies that P_i stores $t+1$ hash values in round r, as Lemma 2 says that for a speed-2 pebble P_i the storage requirements throughout its first 2^i rounds are given by sequence D_i, where $D_0 = \{1\}$ and $D_i = 1^{*2^{i-1}} \parallel 1 + D_{i-1}$. □

Algorithm 1 In-place speed-2 pebble P_k, round $r = 2^{k+1} - c$, $1 \le c < 2^k$.

1: output $z[0]$
2: $k \leftarrow \mathrm{pop}_0(c)$
3: **for** $i \leftarrow 1$ **to** k **do** $z[i-1] \leftarrow z[i]$
4: $k \leftarrow k+1$; $c \leftarrow \lfloor c/2 \rfloor$
5: $p \leftarrow k-1$
6: **while** $c \neq 0$ **do**
7:     $z[p] \leftarrow f(z[k])$
8:     **if** $p \neq 0$ **then** $z[p] \leftarrow f(z[p])$
9:     $k \leftarrow k + \mathrm{pop}_0(c) + \mathrm{pop}_1(c)$
10:     $p \leftarrow k$

Theorem 8 suggests a simple approach to store the hash values for a speed-2 pebble P_k throughout its output stage. We use a single array z of length k to store all hash values as follows. Initially, $z[0] = y_1, \ldots, z[k-1] = y_k$, and counter $c = 2^k - 1$. This corresponds to pebble P_k being at the start of its output stage, where it starts to run pebbles $P_i(y_{i+1})$ in parallel, for $i = 0, \ldots, k-1$, each of these (non-redundant) pebbles P_i storing exactly one hash value in array z. In general, in output round $2^{k+1} - c$ of P_k, we let each non-redundant pebble P_i store its hash values in array z in segment $z[i..i-t]$ (corresponding to $c_i = 1$ and $c_{i-1} = 0, \ldots, c_{i-t} = 0$). As a result, the non-redundant pebbles jointly occupy consecutive segments of array z, storing exactly $\mathrm{len}(c)$ hash values in total.

Algorithm 1 describes precisely what pebble P_k does in round r, $2^k < r < 2^{k+1}$. Note that we let $c = 2^{k+1} - r$ at the start of round r. Based on Theorem 7, we process the bits of c as follows, using operations $\mathrm{pop}_0(c)$ and $\mathrm{pop}_1(c)$ to count and remove all trailing 0s and 1s from c, respectively.

Let $k' \ge 0$ be maximal such that $c \bmod 2^{k'} = 0$. Hence $c_{k'} = 1$. From Theorem 7, we see that pebble $P_{k'}$ is in its first output round $2^{k'}$, hence $P_{k'}$ becomes redundant in the next round, and each of its children will take over one hash value. The hash values $y_0, \ldots, y_{k'}$ computed by $P_{k'}$ in its initial stage are stored in $z[0], \ldots, z[k']$. So, we output $z[0] = y_0$ for $P_{k'}$ and move $y_1, \ldots, y_{k'}$ to entries $z[0], \ldots, z[k'-1]$. This makes entry $z[k']$ available. We distinguish two cases.

Case $\mathrm{len}(c-1) = \mathrm{len}(c) - 1$. In this case no new hash values need to be stored, and $z[k']$ will be unused from this round on.

Case $\mathrm{len}(c-1) = \mathrm{len}(c)$. Let $k'' \ge k'+1$ be maximal such that $c \bmod 2^{k''} = 2^{k'}$. Hence $c_{k''} = 1$. We claim that speed-2 pebble $P_{k''}$ is the unique pebble that needs to store an additional hash value. Pebble $P_{k''}$ is in round $2^{k''} - (c \bmod 2^{k''}) = 2^{k''} - 2^{k'}$, so it is $2^{k'}$ rounds from the end of its initial stage. We store its additional hash value in $z[k']$.

This explains Algorithm 1. In the first iteration of the loop in lines 6–10, we have that $p = k'$ holds at line 7. Each hashing pebble performs two hashes, except when a pebble is at the end of its initial stage (corresponding to $p = 0$). Essentially no processing is done for idle pebbles, due to the use of operation $\mathrm{pop}_1(c)$ in line 9.

Algorithm 2 In-place optimal pebble P_k, round $r = 2^{k+1} - c$, $1 \le c < 2^k$.

1: output $z[0]$
2: $k \leftarrow \mathrm{pop}_0(c)$
3: **for** $i \leftarrow 1$ **to** k **do** $z[i-1] \leftarrow z[i]$
4: $k \leftarrow k + 1$; $c \leftarrow \lfloor c/2 \rfloor$
5: **while** $c \neq 0$ **do**
6: $\quad p \leftarrow k$
7: $\quad k \leftarrow k + \mathrm{pop}_0(c)$
8: $\quad j \leftarrow (-r) \bmod 2^k$
9: $\quad t \leftarrow (k + j) \bmod 2$
10: $\quad l \leftarrow \mathrm{len}(j)$
11: $\quad m \leftarrow \mathrm{len}(2^l - j)$
12: $\quad s \leftarrow (m+1) \bmod (l+1)$
13: $\quad h \leftarrow \lfloor (t + j(k-m) + (m+3-l)2^l - 2^m)/2 \rfloor$
14: $\quad g \leftarrow \mathrm{len}(h) - 1$
15: $\quad p \leftarrow p + g - l$,
16: $\quad$ **for** $d \leftarrow 1$ **to** $\lfloor (t + k + 1 - s)/2 \rfloor$ **do**
17: $\quad\quad x \leftarrow z[p]$
18: $\quad\quad$ **if** $h = 2^g$ **then** $g \leftarrow g - 1$; $p \leftarrow p - 1$
19: $\quad\quad z[p] \leftarrow f(x)$
20: $\quad\quad h \leftarrow h - 1$
21: $\quad k \leftarrow k + \mathrm{pop}_1(c)$

6.2 In-Place Optimal Pebbles

In this section we turn the algorithm for speed-2 pebbles into one for optimal pebbles by making three major changes. See Algorithm 2.

First, we make sure that the number of hashes performed by each hashing pebble P_k is in accordance with Eq. (1). The actual hashing by P_k is done in the loop in lines 16–20. To apply Eq. (1), the actual round number $j = (2^k - r) \bmod 2^k$ for P_k is determined in line 8. Writing $l = \mathrm{len}(j)$ and $m = \mathrm{len}(2^l - j)$, we have that the number of hashes as specified by Eq. (1) can be computed as $\lfloor (t+k+1-s)/2 \rfloor$, where $t = (k+j) \bmod 2$ and $s = (m+1) \bmod (l+1)$, actually using that $\mathrm{len}((2^l - 2j) \bmod 2^l) = \mathrm{len}(2^{l+1} - 2j) \bmod (l+1)$ holds for $j \ge 1$.

Second, we make sure that each hashing pebble P_k will store the correct hash values for $y_k, \ldots, y_0$. To this end, note that Lemma 4 tells precisely how many hashes pebble P_k still needs to compute at the start of round j. Thus we set h to this value (plus one) in line 13, and test in line 18 if the current hash value must be stored (that is, whether h is an integral power of 2).

Finally, we make sure that hashing pebble P_k will store its hash values in the right entries of array z. In line 6, we let variable p point to the first entry of the segment of array z used by pebble P_k. Then in line 15, the value of p is adjusted by adding the difference between the number of hash values that a speed-2 pebble P_k would still need to store at the start of round j and the number of hash values that an optimal pebble P_k still needs to store at the start of round j.

Algorithm 3 Fast optimal pebble P_k, round $r = 2^{k+1} - c$, $1 \le c < 2^k$.

1: output $z[0]$
2: $k \leftarrow \text{pop}_0(c)$
3: **for** $i \leftarrow 1$ **to** k **do** $z[i-1] \leftarrow z[i]$
4: $k \leftarrow k + 1$; $c \leftarrow \lfloor c/2 \rfloor$
5: **if** c odd **then** $a[v] \leftarrow (1, k, k, k, 2^k, 2^k)$; $v \leftarrow v + 1$
6: $u \leftarrow v$
7: **while** $c \neq 0$ **do**
8:     $k \leftarrow k + \text{pop}_0(c)$
9:     $u \leftarrow u - 1$; $(e, l, s, p, g, h) \leftarrow a[u]$
10:     $e \leftarrow e - 1$
11:     **if** $e \le 0$ **then**
12:         **if** $s = 0$ **then**
13:             $l \leftarrow l - 1$; **if** $l \neq 1$ **then** $s \leftarrow 2$
14:         **else**
15:             **if** $s = l$ **then** $s \leftarrow 0$ **else** $e \leftarrow 2^{s-1}$; $s \leftarrow s + 1$
16:     **for** $d \leftarrow 1$ **to** $\lfloor ((k + e) \bmod 2 + k + 1 - s)/2 \rfloor$ **do**
17:         $x \leftarrow z[p]$
18:         **if** $h = g$ **then** $g \leftarrow \lfloor g/2 \rfloor$; $p \leftarrow p - 1$
19:         $z[p] \leftarrow f(x)$
20:         $h \leftarrow h - 1$
21:     **if** $h \neq 1$ **then** $a[u] \leftarrow (e, l, s, p, g, h)$ **else** $v \leftarrow v - 1$
22:     $k \leftarrow k + \text{pop}_1(c)$

This explains the design of Algorithm 2. Note that in total only three bit length computations are used per hashing pebble (cf. lines 10, 11, and 14).

6.3 Optimal Pebbles with Minimal Computational Overhead

Even though the computational overhead for our in-place implementation is small, it may still be relatively large if hash evaluations themselves take very little time. For instance, if the hash function is (partly) implemented in hardware. Using Intel's AES-NI instruction set one can implement a 128-bit hash function that takes a few cycles only (e.g., see [BÖS11], noting that for one-way hash chains no collision-resistance is needed such that one can use Matyas-Meyer-Oseas for which the key is fixed). Therefore, we also provide an implementation minimizing the computational overhead at the expense of some additional storage.

We will keep some state for each pebble, or rather for each *hashing* pebble only. Although an optimal pebble P_k will store up to k hash values at any time, we observe that no more than $\lceil k/2 \rceil$ hashing pebbles will be present at any time. As in our in-situ algorithms we will thus avoid any storage (and processing) for idle pebbles, as can be seen from Algorithm 3.

A segment $a[0..v-1]$ of an array a of length $\lfloor k/2 \rfloor$ suffices to store the relevant hashing pebbles, where initially $v = 0$. In each round, at most one idle pebble P_k will become hashing, and if this happens pebble P_k is added to array a, cf. line 5.

Later, once pebble P_k is done hashing, it will be removed again from array a, cf. line 21.

For each hashing pebble we store six values called e, l, s, p, g, h, respectively. Except for counter e, these values correspond to the variables in Algorithm 2. Using j to refer to the round pebble P_k is at, as in Algorithm 2, we have $l = \mathrm{len}(j)$ and also $s = (m+1) \bmod (l+1)$, where $m = \mathrm{len}(2^l - j)$. We use e to count down to zero starting from the appropriate powers of 2, cf. line 15.

As a result, Algorithm 3 limits the computations for each hashing pebble to a few simple operations only.

7 Concluding Remarks

We have completely resolved the case of binary pebbling of hash chains by constructing an explicit optimal schedule. A main advantage of our optimal schedule is that it allows for very efficient *in-place* pebbling algorithms. This compares favorably with the greedy pebbling algorithms of [YSEL09], which require a substantial amount of storage beyond the hash values themselves. The pseudocode of Algorithms 1–3 is readily translated into efficient program code, applying further optimizations depending on the target platform.[1]

The security of one-way hash chains for use in authentication mechanisms such as Lamport's identification scheme does not depend on the collision resistance of the hash function. Therefore, it suffices to use 128-bit hash values—rather than 256-bit hash values, say. Using, for instance, the above mentioned Matyas-Meyer-Oseas construction one obtains a fast and simple one-way function $f : \{0,1\}^{128} \to \{0,1\}^{128}$ defined as $f(x) = \mathrm{AES}_{\mathrm{IV}}(x) \oplus x$, where IV is a 128-bit string used as fixed "key" for the AES block cipher. Consequently, even for *very long* hash chains of length 2^{32}, our in-place optimal pebbling algorithm will just store 516 bytes (32 hash values and one 32-bit counter) and perform at most 16 hashes per identification round. Similarly, *long* hash chains of length 2^{16} would allow devices capable only of lightweight cryptography to run 65535 rounds of identification (e.g., more than twice per hour over a period of three years), requiring only 258 bytes of storage and using at most 8 hashes per round.

We leave as an open problem whether binary pebbling yields the lowest space-time product. Reversal of a length-n hash chain using optimal binary pebbling requires $\log_2 n$ storage and $\frac{1}{2}\log_2 n$ time per round, yielding $\frac{1}{2}\log_2^2 n$ as space-time product. Coppersmith and Jakobsson [CJ02] derived a lower bound of approx. $\frac{1}{4}\log_2^2 n$ for the space-time product. Whether this lower bound is achievable is doubtful, because the lower bound is derived *without* taking into account that the maximum number of hashes during any round needs to be minimized. As a natural alternative, we have done a preliminary study of "Fibonacci" pebbling, considering hash chains of length $n = F_k$, the kth Fibonacci number. Initial results, however, suggest that the space-time product is not lower than for binary pebbling.

[1] Sample code (in C, Java, Python) available at `www.win.tue.nl/~berry/pebbling/`.

As another direction for further research we suggest to revisit the problem of efficient Merkle tree traversal studied in [Szy04], which plays a central role in hash-based signature schemes [Mer87,Mer89]; in particular, it would be interesting to see whether algorithms for generating successive authentication paths can be done in-place. More generally, research into optimal (in-place) algorithms for hash-based signatures is of major interest both in the context of lightweight cryptography (e.g., see [PCTS02,MSS13]; more references in [YSEL09]) and in the context of post-quantum cryptography (e.g., see [BDE+13]).

References

BDE+13. J. Buchmann, E. Dahmen, S. Ereth, A. Hülsing, and M. Rückert. On the security of the Winternitz one-time signature scheme. *International Journal of Applied Cryptography*, 3(1):84–96, 2013.

BÖS11. J.W. Bos, O. Özen, and M. Stam. Efficient hashing using the AES instruction set. In *Cryptographic Hardware and Embedded Systems (CHES 2011)*, volume 6917 of *Lecture Notes in Computer Science*, pages 507–522, Berlin, 2011. Springer-Verlag.

CJ02. D. Coppersmith and M. Jakobsson. Almost optimal hash sequence traversal. In *Financial Cryptography 2002*, volume 2357 of *Lecture Notes in Computer Science*, pages 102–119, Berlin, 2002. Springer-Verlag.

IR01. G. Itkis and L. Reyzin. Forward-secure signatures with optimal signing and verifying. In *Advances in Cryptology—CRYPTO '01*, volume 2139 of *Lecture Notes in Computer Science*, pages 332–354, Berlin, 2001. Springer-Verlag.

Jak02. M. Jakobsson. Fractal hash sequence representation and traversal. In *Proc. IEEE International Symposium on Information Theory (ISIT '02)*, page 437. IEEE Press, 2002. Full version `eprint.iacr.org/2002/001`.

Lam81. L. Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772, 1981.

Mer87. R. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology—CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378, Berlin, 1987. Springer-Verlag.

Mer89. R. Merkle. A certified digital signature. In *Advances in Cryptology—CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238, Berlin, 1989. Springer-Verlag.

MSS13. N. Mourier, R. Stampp, and F. Strenzke. An implementation of the hash-chain signature scheme for wireless sensor networks. In *Lightweight Cryptography for Security and Privacy (LightSec 2013)*, volume 8162 of *Lecture Notes in Computer Science*, pages 68–80, Berlin, 2013. Springer-Verlag.

PCTS02. A. Perrig, R. Canetti, J.D. Tygar, and D. Song. The TESLA broadcast authentication protocol. *RSA CryptoBytes*, 5(2):2–13, 2002.

Szy04. M. Szydlo. Merkle tree traversal in log space and time. In *Advances in Cryptology—EUROCRYPT '04*, volume 3027 of *Lecture Notes in Computer Science*, pages 541–554, Berlin, 2004. Springer-Verlag.

YSEL09. Dae Hyun Yum, Jae Woo Seo, Sungwook Eom, and Pil Joong Lee. Single-layer fractal hash chain traversal with almost optimal complexity. In *Topics in Cryptology – CT-RSA 2009*, volume 5476 of *Lecture Notes in Computer Science*, pages 325–339, Berlin, 2009. Springer-Verlag.