Submitted for publication. Author Copy - do not redistribute.

Efficient Targeted Key Subset Retrieval in Fractal Hash Sequences

Kelsey Cairns*
Washington State University
Pullman, WA, USA
kcairns@wsu.edu

Thoshitha Gamage
Washington State University
Pullman, WA, USA
tgamage@eecs.wsu.edu

Carl Hauser
Washington State University
Pullman, WA, USA
hauser@eecs.wsu.edu

ABSTRACT

This paper presents a new hash chain traversal strategy which improves performance of hash chain based one-time authentication schemes. This work is motivated by the need for efficient message authentication in low-latency multicast systems. Proposed solutions such as TV-OTS rely on hash chain generated values for keys, achieving reliable security by using only a small subset of generated values from each chain. However, protocols using hash chains are limited by the rate at which a hash chain traversal is able to supply keys. The new algorithm uses the same structure as Fractal Hash Traversal, but eliminates redundant operations incurred when used with applications such as TV-OTS. Performance is measured in terms of savings and is proportional to the chain-distance between consecutively retrieved values. For a distance of δ, we achieve $\Theta(\delta \log_2(\delta))$ savings, which is shown analytically and supported by empirical tests.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—computations on discrete structures

*This research was funded in part by Department of Energy Award Number DE-OE0000097 (TCIPG).

Disclaimer: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS'13, November 4–8, 2013, Berlin, Germany.
Copyright 2013 ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516739 .

Keywords

Key Management; Hash Chain; Traversal; Key Retrieval; Data Authentication; Time Validation; One-Time Signature

1. INTRODUCTION

A class of emerging applications — including those designed to control critical infrastructure systems — requires reliable message authentication in multicast environments. For instance, applications related to the power grid rely on several types of authenticated messages, some of which initiate control actions. These control actions are responsible for regulating physical characteristics that influence the power flows in the grid. A falsified signal is potentially very dangerous as it could be part of an attack intended to damage costly equipment or cause failure within the grid.

A general solution to the multicast authentication problem, one suitable for practically any multicast application, has yet to be found. Existing protocols tend to support specific classes of applications, with characteristics that may not be tolerable to others [3]. For low latency applications such as power grid control, the questions about appropriate multicast authentication protocols remain unsettled.

The long standing solution for multicast authentication, RSA [19], has been shown unsuitable for low latency applications [6]. RSA relies on modular exponentiation, an expensive operation which must be performed on a per-message basis. Status messages in the power grid can be generated as frequently as every 16 milliseconds, a rate that may quicken in the future [1]. If signed using RSA, the signing rate may actually lag behind the intended sending rate depending on the capability of the signing devices. This disparity eliminates RSA as a possible protocol for use in many high-rate systems, leaving a niche to be filled.

One technique facilitating lower latency authentication is one-time signatures. The most basic one-time signatures are used as credentials which allow senders to authenticate their identity to receivers. These schemes demonstrate the general principle used in even sophisticated one-time signature schemes: a public key is pre-shared between sender and receivers, and the sender authenticates itself by publishing a secret which receivers associate with the public key [5, 12]. Usually, the receiver forms this association with a one-way function of some kind, where the received secret acts as input that reproduces the public key. If deployed properly, only the correct sender could know and publish the correct secret.

This principle can be extended to verify not only the sender, but also the message contents by having the sender

calculate the signature in such a way that receivers must operate on both the received secret and message contents to recreate the public key. This way, any modification to the message would cause the signature validation to fail. True to their name, basic one-time signature schemes can only authenticate one message for each public key. However, more modern schemes expand on the one-time principle, allowing multiple messages to be signed per public key [14, 15, 16, 17, 18]. The present challenge is to create a low-latency protocol which increases this number until key distribution is infrequent enough to make the overhead of repeated distribution tolerable.

The Time-Valid One-Time-Signature (TV-OTS) family of protocols builds extended-lifetime protocols from one-time signature schemes with limited lifetimes [21]. TV-OTS applies a novel key refreshing technique which extends the overall lifetime of a wide range of one-time signature protocols. Over the running lifetime of the chosen protocol, private keys are periodically refreshed in such a way that no new public keys need to be distributed. If the key refresh operations can be performed in a short, constant-bounded time, TV-OTS with Hash of Random Subsets (HORS) [18] reliably outperforms RSA in terms of signing and verification latency [7]. Beyond having the ability to generate and verify signatures more quickly than RSA, TV-OTS incurs no additional message delays and imposes no constraints on the underlying network beyond loose time synchronization. These properties indicate TV-OTS may be a promising protocol for use in low-latency systems.

Unfortunately, the only known time-efficient way to manage keys for TV-OTS is to store all possible keys in memory. However, storing all the keys necessary for TV-OTS would be impractical for many devices. Spatially efficient schemes, such as Fractal Hash Sequence Representation and Traversal (FHT) which is suggested for use with TV-OTS, do not allow the constantly bounded retrieval time necessary to ensure low latency signing. The standard implementation of TV-OTS uses HORS [18] as its chosen one-time signature. This combination is made secure by retrieving keys at certain probabilities, but unfortunately, this scenario becomes increasingly time-inefficient as the security of TV-OTS is increased.

The work presented in this paper aims to fill the gap with a key management scheme that is efficient for all parameters of TV-OTS. The purpose of this paper is to:

1. *Present a new key management scheme* — The new key management scheme eliminates certain useless operations that are performed by other schemes when non-consecutive values are retrieved. The new scheme uses the same structure as FHT, but the algorithm used to output individual keys is new. The FHT algorithm can only retrieve values consecutively. If non-consecutive values are retrieved with FHT, the intermediate values must be retrieved and discarded. This process performs extra hash operations that do not contribute to finding the desired value. The Targeting Traversal method introduces *targeted retrievals*, which retrieve non-consecutive values without performing unnecessary work. Targeted transitions optimize retrieval time for TV-OTS. Other schemes may benefit as well, for example TSV signing [14], which uses hash chains similarly to TV-OTS.

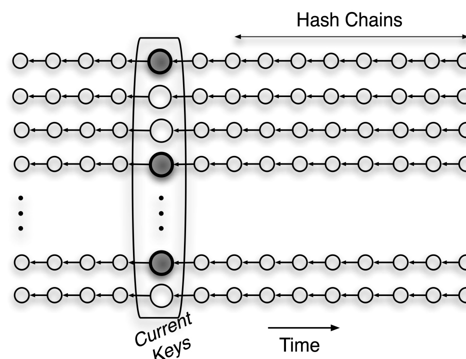

Figure 1: TV-OTS's multiple hash chains are shown horizontally. A vertical slice corresponding to the current time defines the current key pool. Darkened circles represent the small subset of keys used in the creation of a single signature.

2. *Verify correctness* — The correctness of targeted transitions is verified in relation to FHT. A formal proof is given that each targeted retrieval yields a state equivalent to multiple iterative retrievals performed by FHT. Inductively, FHT and targeting are shown to yield the same sequence of values for the same set of retrieval requests on the same chain.

3. *Demonstrate performance improvements* — Theoretical analysis reveals a $\Theta(\delta \log_2(\delta))$ bound on computational savings. Here, δ is the expected distance between successively retrieved values in the chain, and is independent of the length of the chain. Experimental results support this hypothesis, showing improved performance for TV-OTS. Furthermore, the savings improve in conjunction with the parameters that make TV-OTS more secure.

The remainder of this paper is organized as follows: Section 2 offers extended background, introducing the problems associated with TV-OTS and key management in detail. Section 3 describes the new algorithms that lead to more efficient key management when used with TV-OTS. A proof of correctness of the proposed algorithms is presented in 4 followed by experimental results in Section 5. Sections 6 and 7 offer future work and conclusions.

2. RELATED WORK

Two mechanisms are employed by TV-OTS which specifically contribute to the low latency generation of reliable signatures. First, the keys are generated by hash chains, a mechanism first proposed by Lamport for secure password authentication [13]. When used for message authentication, hash chains allow private keys to be refreshed without the need to distribute new public keys. Secondly, signatures are generated by HORS, using the keys retrieved from the hash chains. This sections covers both signature generation and hash chain management in detail.

2.1 Time-Valid One-Time-Signature

TV-OTS is able to run beyond the lifetime of its chosen one-time signature scheme by periodically refreshing the private keys used to generate signatures. New private keys are

supplied by hash chains which allows each used private key to be used as a public key at a later time. This eliminates the need to redistribute public keys with each private key update. Individual messages are signed and verified with the HORS protocol [18]. At signature generation, HORS chooses a very small subset from an available pool of hash chain generated keys, where each key is associated with an individual hash chain[1]. After t key refreshes, the currently available key pool contains the tth value from each chain in a set of hash chains, illustrated in Figure 1. To choose the keys for each signature, the message contents are hashed and split into multiple short bit strings by a publicly agreed upon function. These bit strings are then reinterpreted as integers within a given range. The allowable range of integers corresponds to the number of available keys in each pool and consequently the number of chains. These integers are used as indices to select keys which are appended to the message as part of the signature. The TV-OTS signature must also contain a time stamp, which allows receivers to determine message freshness and which key pool was used for message signing.

Verification of TV-OTS signatures is the simple process of verifying message freshness and the individual keys contained in the signature. Receivers first use the time stamp of the received signature to determine that the message was signed recently enough to be valid. After this, the receiver follows the same steps as the sender to determine the index for each key. Once these are computed, each received key is checked to see if it belongs to the chain with the corresponding index. This requires hashing the received key and ensuring that a known value from the correct chain can be recovered by the correct number of hashes. Incoming messages are considered valid only if this test passes for every index computed by the receiver.

Signatures maintain their reliability by using, and thus publicly exposing, only a relatively small portion of each key pool. This minimizes the chance of an adversary intercepting enough keys to forge the signature of a meaningful message. While the threat grows with each sent message, the key refresh interval is set to maintain a negligible probability of successful forgery [21]. One side effect is that the number of refresh intervals that pass between two signatures requiring keys from the same chain is unpredictable. The keys in between are not retrieved from the hash chains and moreover, this distance between retrieved keys is unknown ahead of time. Unfortunately, known hash chain management schemes assume that every key will be retrieved. When this is not the case, such schemes become inefficient and waste some calculation during each retrieval.

2.2 Hash Chains for One-Time Signatures

The hash chains structure works very well with the principles behind one-time signatures, but hash chains can be difficult to manage in practice. The values contained in a hash chain form an ordered list. Each value is calculated by hashing the next value in the list with a chosen one-way hash function. The entire chain can be derived from a distinct seed value by repeatedly applying the chosen hash function. Counter-intuitively, the last value generated is referred to as the first value of the chain. The seed of a chain with n values would be the nth value. This terminology reflects the use of values in reverse order of their generation.

Hash chains are especially applicable to message authentication due to their one-way properties. Asymmetric protocols that use hash chain generated keys reveal the first chain value (the last to be generated) as part of the pre-shared public key. As new keys are used and sent through the protocol, they must be verified by the receivers. Receivers verify new keys by creating a match with a known value, either the one distributed in the public key or one more recently received. Newly received keys, when hashed (possibly multiple times), should generate one of these known values. Invalid keys can not achieve this match. Only the sender knows the seed value necessary to calculate new valid keys. Assuming the unused values are kept secret, the identity of the sender is verified with each new value revealed.

The term *traversal* refers to the sequential output of hash chain values starting with the first value and working toward the last. Traversals present a challenge because the output order is opposite the order of generation. For long chains, storing all values may be impractical, but otherwise, needed values must be recomputed. If only the seed is available as a starting point, calculating values near the beginning of the chain requires heavily repetitious hashing. Wiser strategies look for ways to balance the cost of storage and computation so that neither becomes too costly.

2.3 Fractal Hash Sequencing and Traversal

In the search for a traversal strategy balancing storage and computation costs, FHT [10] has emerged as a practical and elegant solution to achieving $O(\log_2(n))$ bounds on both measures. While FHT can't maintain these same bounds unless values are retrieved consecutively, the bounds for retrieving a sequence of values are still sufficiently low to make it a traversal candidate for TV-OTS. The FHT structure serves as a foundation for the algorithms presented in Section 3. The explanation given here places special emphasis on the details from which the new algorithms are built.

FHT stores only $\log_2(n)$ chain values chosen in such a way that retrieving new values requires little computation. The arrangement of these stored values is dynamic and continually changes to accommodate future requests more easily. When one of the stored values is retrieved, it is no longer useful and is abandoned in favor of storing a later value from the chain. The stored values are kept grouped closely towards the next values that will be retrieved, limiting the amount of work performed by any single retrieval operation.

To facilitate the dynamic arrangement of stored values, a small data structure called a *pebble* is used to associate additional information with each stored value. Each pebble stores one chain value at a time, along with the value's position in the chain. Pebbles are distinguished by a unique identifier, ID, which also governs the process followed to update the values in individual pebbles. Sometimes one pebble will be referred to as *larger* than another, meaning the value of its ID is greater than the ID of the *smaller* pebble. In a chain of n values, the $\log_2(n)$ pebbles are stored in a list sorted by chain position.

Note that two lists are now present: the conceptual list of values comprising the chain and the stored list of pebbles. This creates possible confusion over the meaning of the term *position* of a value or pebble. For consistency, different terms will used in reference to these two structures. The word

[1]Note that TV-OTS actually relies on salted hash chains [21]. However, the ideas discussed here apply equally to salted and unsalted hash chains.

position is used exclusively in reference to the entire hash chain. The words *above*, *below*, *higher* and *lower* are often used to describe relative *positions*, with the words *higher* and *above* pertaining to pebbles closer to the seed of the chain. Naturally, relative positions in the chain are preserved in the pebble list, since the pebble list is kept sorted by position.

Updating the values stored in the pebbles is analogous to moving pebbles within the chain. Conceptually, when a pebble acquires a new value, it moves to the position in the chain associated with the new value. In fact, FHT works by moving pebbles through an interconnected sequence of strategic arrangements.

The positions of pebbles in each possible arrangement allow easy computation of the next output values and future arrangements. At initialization, the position of each pebble matches its ID value. More formally, there is a pebble at every position 2^i where $1 \leq i \leq \log_2(n)$. The gaps between the pebbles form intervals, with smaller intervals near the beginning of the chain. When a pebble moves, it always divides an interval evenly into two new equally sized intervals. Like pebble IDs, interval sizes are powers of two which facilitates easy splitting. The sorted order of the intervals is preserved since when a pebble moves, the new intervals created are at least as large as intervals at lower positions. This pattern of intervals is used to ensure that both retrieved values and future arrangements can be calculated efficiently.

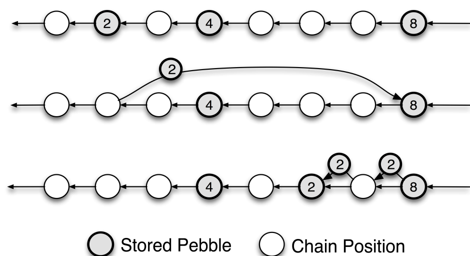

Figure 2: When a pebble moves, it cannot move directly to its destination. It must calculate the desired value by first moving upward past its destination. Once it copies the value from a stored pebble, the desired value is calculated by additional hash operations as the pebble moves downwards to its destination.

The method for calculating chain values limits the ways in which pebbles can move. When a pebble moves to a new position, it can not acquire the new value directly. Recall that pebble values must be computed from one another. The one-way properties restrict each value to being computed from values at *higher* positions. Moving a pebble requires finding and copying the value stored in some higher pebble and calculating the desired value from there. In essence, pebbles move in two phases, as illustrated in Figure 2. In the first phase, the pebble moves upwards to the same location as a higher pebble. In the second phase, the pebble's new value is hashed repeatedly, effectively moving the pebble downward in the chain to its destination.

The pattern maintained by the arrangements of pebbles is governed by the pebble IDs and provides the ability to retrieve keys within a logarithmic time bound. This arrangement pattern is formed by the second stage of pebble movement. In this second stage, pebbles step downward from their new position acquired in the first stage. The number of downward steps a pebble takes equals the value of its ID. Eventually, this creates an interval between each newly moved pebble and the pebble it copied from with the new interval size equal to the moved pebble's ID. The pebble at the upper edge of the interval, whose value was copied, was chosen because the interval it bounded before the move was twice the size of the ID of the moving pebble. (The existence of such an interval is guaranteed [10].) Thus, on each move, a pebble splits an interval into two new equally sized intervals. Furthermore, pebbles always move into the nearest interval large enough to evenly divide. Naturally, the pebbles with smaller IDs move shorter distances. Since the size of newly created intervals is at least as large as the intervals at lower positions, the intervals remain sorted by size. The pebble with the ID value of 2 never moves beyond the lowest four values in the unused chain, and ensures these lowest values are part of intervals of size two. Consequently, any value retrieved from this section of the chain will never require more than a single hash operation to compute. The remaining hash operations are performed in stepping the other pebbles towards their destinations.

To provide the amortized upper bound on retrieval time, moving pebbles rarely perform their downward movement phase all at once. Instead, moving pebbles distribute these downward steps over several retrieval operations, taking only enough steps to ensure they reach their destination by the time the value at that destination is needed. Notice that pebbles with larger IDs move further and are therefore not needed at their new positions as quickly as smaller pebbles. The number of retrievals that occur before a pebble must reach its destination is directly related to how far the pebble must travel. In fact, only the pebble with ID value 2 must reach its destination by the time the next pebble is moved. This happens exactly two retrievals after this pebble was moved. For all other pebbles, these intermediate retrievals are used to distribute hashing costs over time, avoiding a situation where some retrievals are inexpensive and others are costly. The total number of hash operations per retrieval is limited to two per actively moving pebble, plus at most one to calculate the retrieved value. The maximum hash operations per retrieval is thus $2 \times \log_2(n) + 1$.

2.4 Related Traversals

The idea of efficient hash chain traversal introduced by Itkis and Reyzin [9] started a wave of traversals with varying efficiencies and trade-offs.

The traversal suggested by Itkis and Reyzin was soon followed by Jakobsson's FHT [10], described above, from which all the others drew either direct or indirect inspiration. However, the dependency on consecutive value retrieval is common in all of such traversals. Traversals presented by Coppersmith and Jakobsson [4], and Yum et al. [22] build directly on FHT by modifying the pebble movement pattern. Coppersmith and Jakobsson achieve near maximum theoretical efficiency for consecutive retrievals by allocating a hash budget on each round, and using a sophisticated movement pattern that distributes this budget between two sets of pebbles. A set of greedy pebbles consumes as much of the hash budget as possible, and any extra is alloted to the remaining pebbles. In this way, the variance in hash operations between iterations is eliminated, lowering the worst-case number to $\frac{1}{2}\log_2(n)$ hashes per round, though at a

storage cost of slightly greater than $\log_2(n)$ pebbles. This technique is further improved upon by Yum et al. who use the same strategy to balance the distribution of hash operations over the rounds, but with a less complex movement pattern. The resulting algorithm is simpler than the Coppersmith and Jakobsson algorithm, and achieves the same lowered time bound without requiring additional storage.

In response to the traversals where computational bounds scale with chain length n, Sella proposed a traversal where computation time could be fixed at the expense of storage space [20]. Using a slightly different chain partitioning technique, the spacings between pebbles can be adjusted to accommodate the imposed computational bounds. Unfortunately, additional pebbles are needed to facilitate this spacing strategy so that for a fixed computational bound of m, the storage requirement is $k\sqrt[k]{n}$ where $k = m + 1$. For comparison, this is twice the storage required by FHT for equivalent time bounds. Later, a scheme devised by Kim was able provide the same fixed time bounds without these extra pebbles by carefully timing pebble movements to create a more synchronous system [11]. This reduced the storage by a factor of k, which matches the storage required by FHT for equivalent bounds.

One remaining strategy, capable of lowering retrieval complexity even further, stems from the introduction of multi-dimensional chains. Hu et al. present two traversals based on a modified structure of the underlying chain [8]. Sandwich chains intertwine multiple chains to form a construction whose primary purpose is efficient verification of keys. Their second construction, Comb Skip-Chains, lowers retrieval bounds by amortizing the FHT over the secondary dimension of a two-dimensional chain structure. With a total of $\log_2(n)$ secondary chains, the amortization brings retrieval time down to a constant, while storage is bounded by $O(\log_2(n))$.

3. TARGETING TRAVERSAL

Thus far the claim that FHT is inefficient in the presence of skipped values is unsubstantiated but is easy to show by example. These examples are also helpful in understanding the reasons underlying the inefficiencies of FHT. Such insights lead to areas where FHT can be improved.

3.1 Motivating Insight

The interdependency of successive FHT retrievals, though used to establish an upper bound on retrieval time, causes unnecessary operations when the interval between desired values is large. The movement of any pebble relies on the correct completion of prior pebble movements. Practically, this implies that values must be retrieved from the chain in the expected, consecutive order, discarding unneeded values. This *iterative* process is highly wasteful.

One prominent source of wasted computation is the calculation of values within ranges of skipped values. Take, for example, pebbles moved to positions below the retrieval target. Since key use is ordered (once a key is retrieved, the keys at lower positions will never be needed), the hash computations used to move pebbles into skipped ranges are completely wasted. With the understanding that a pebble is only useful once it moves above the target value, all the movements of each pebble can be predicted and grouped into a single move. This modification, illustrated in Figure 3, eliminates the hash operations performed by calculating values from the unused region.

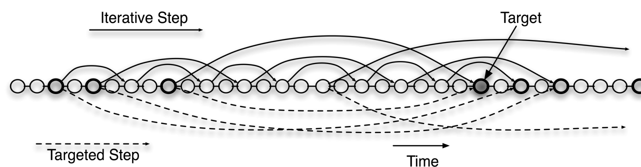

Figure 3: In this small sample chain segment, each line above the chain represents a required move in order to retrieve the darkened pebble by the iterative FHT method. Targeting eliminates this tangle by moving each pebble only once, without passing through intermediate locations, as shown by the dashed lines below the chain.

For the purpose of efficiently moving pebbles, it is useful to consider each arrangement of pebbles as a state, with retrievals triggering state transitions. Figure 3 contrasts the idea of a targeted state transition against FHT's iterative method. To simplify matters further, new states depend only on the location of the target value, and not on any previous states.

State changes are the inspiration behind the targeting algorithm. The goal of the targeting algorithm is to perform state changes meeting two criteria: first, the state changes should be functionally indistinguishable from state changes performed by iterative retrievals. Additionally, the implemented state changes should perform the minimum number of hash operations necessary to achieve the new state. That is, a hash operation should never be applied to the same value more than once during a single retrieval.

State transitions alone do not guarantee the elimination of all redundant hashes and algorithms must be crafted carefully to avoid hashing the same value more than once. The threat of duplicate hashes arises when two moving pebbles skip upward to the same position during the first movement phase. In this case, both pebbles will copy the value from the same reference pebble and hash this value in order to step downward. In essence, by hashing the same value, the two pebbles perform the same operation. These redundant operations can be avoided if one pebble waits while the other performs the operations common to both pebbles. Then instead of copying the value of the originally intended pebble the waiting pebble can copy the value from the other pebble, eliminating redundant work.

When two pebbles would ordinarily move upward to the same pebble, the question arises of which to move first and which to delay. When considering a simple case with two pebbles, two algorithms naturally emerge which are built into two transition algorithms – the *Lazy Pebble* and *Traveling Pebble* algorithms. The small example in Figure 4 illustrates the basic principles of these algorithms. The strategy taken by the Lazy Pebble algorithm is to move the pebble with the smaller ID first. Once this pebble reaches its destination, the larger pebble can copy the value of the smaller and begin hashing to step downward from there. The other algorithm, Traveling Pebble, moves the larger pebble first. With this method, the smaller pebble does not perform any hash operations at all. As the larger pebble steps past the smaller pebble's destination, the smaller pebble copies the

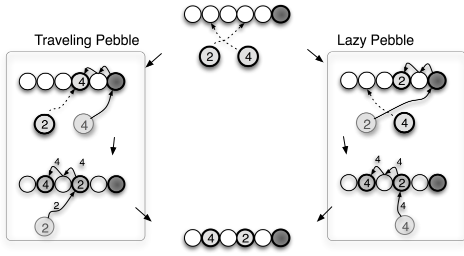

Figure 4: A simple example is pictured in which the two pebbles, with IDs 2 and 4, both intend to copy the value from the same higher pebble (shown darkened). To avoid redundant work, one pebble moves before the other. The two paths demonstrate different principles on which to base transition algorithms.

Table 1: Retrieval bounds are given for the best and worst cases of targeting when skipping δ values. $C_{it}(\delta)$ denotes the cost δ iterations of iterative FHT.

Case	Bound
Worst	$\begin{cases} C_{it}(\delta) - \frac{\delta}{2}\rho(\delta,3)) + 2^{\rho(\delta,1)} - 4 & : \lceil \log_2(\delta) \rceil > 3 \\ C_{it}(\delta) & : Otherwise \end{cases}$
Best	$C_{it}(\delta) - \frac{\delta}{2}(\rho(\delta,1)) - 2^{\rho(\delta,0)} + 2$

value from the larger pebble. The larger pebble then continues on to its final destination. The remainder of this paper will focus on the Traveling Pebble approach. This algorithm is more intuitive, and preliminary tests indicated no measurable performance difference between the two.

For the purpose of explanation, the targeting algorithm has been divided into two stages: *state calculation* and *state transition*. The state calculation determines new destinations for all pebbles that need to move. The Traveling Pebble algorithm implements the state transition and can be thought of as a sweep which hashes values across the whole range into which pebbles will move. During this sweep, pebbles move to their intended destinations. To make this process align better with the style of FHT pebble movements, the job of performing the hash sweep is given to the various moving pebbles. In the Traveling Pebble algorithm, not all pebbles are required to participate in the hash sweep, but those that do are responsible for the range of values that lie above their new destination and below the next higher pebble.

The targeting algorithm descriptions pertain to pebbles whose destinations are still within the chain. The other case, where pebbles move past the end of the chain, is ignored for simplicity. These pebbles are unnecessary for the remainder of the traversal and can be safely retired by removal from the pebble list.

3.2 Theoretical Performance

Overall performance of the targeting algorithm can be measured in terms of storage and computation, with computation cost split into hash operations and overhead incurred from rearranging the pebble list. Storage complexity is exactly the same as FHT, which is bounded by $O(\log_2(n))$ where n is the chain length. Computational costs require further analysis.

Overhead of both state calculation and state transition is bounded by $O(\log_2(\delta))$ list operations, where δ represents the distance between successive retrieval targets. The number of pebbles in a range of length δ is bounded by $\log_2(\delta) + 1$. In both Algorithms 1 or 2, no loop processes more than $\log_2(\delta) + 1$ pebbles. The first loop in both algorithms searches for a pebble above the target, with an ID larger than any pebble below the target. This pebble is first guessed by knowing the minimum number of pebbles below the target as well as the smallest possible ID for the largest of these pebbles. From this guess, at most one iteration is needed before reaching pebbles above the target. Finding a pebble with an ID larger than all the pebbles below the target is clearly bounded by $\log_2(\delta) + 1$, the number of pebbles that will move[2]. The second loop in each algorithm iterates over moving pebbles, which is already known to be $O(\log_2(\delta))$.

Performance in terms of hash operations is characterized by upper and lower bounds on the number of operations saved, $S(\delta)$, in comparison to the iterative method. For a retrieval distance δ, iterative FHT requires $O(\delta \times \log_2(n))$ hash operations. Savings gained by targeting are examined on a case by case basis to establish the lower and upper bounds.

Bounds are found by estimating the number of pebbles capable of savings and the number of operations saved by each pebble. To simplify notation, let $\rho(\delta, \alpha)$ represent the number of pebbles that cause savings:

$$\rho(\delta, \alpha) = max(\lceil \log_2(\delta) \rceil - \alpha, 0) \qquad (1)$$

An interval of size δ contains at most $\lceil \log_2(\delta) \rceil$ pebbles, however, this overestimates the number of pebbles which achieve savings by a small number α. To form best and worst case calculations, α is adjusted accordingly. In the worst case, at most three pebbles will be large enough to step above the target without achieving savings[3]. With $0 \leq \alpha \leq 3$, note that:

$$\rho(\delta, \alpha) \in \Theta(\log_2(\delta)) \qquad (2)$$

A lower savings bound can be estimated from the case where the only savings derive from pebbles that avoid extra moves below the target. This case is unlikely, but theoretically possible. A key observation about the iterative algorithm is that while each pebble hashes small ranges throughout the chain, these ranges add up to hashing about half of the entire chain. While skipping a region of length δ, each pebble can save up to $\frac{\delta}{2}$ operations, with adjustments necessary for losses at the edges of the region. For an individual pebble p_i, the savings loss at each edge is at most 2^{i+1}. Thus for each p_i, the minimum individual savings is $\frac{1}{2}(\delta - 2 \times 2^{i+1})$.

[2] This number is actually bounded by a constant, however, the explanation is lengthy and unnecessary to show the $O(\log_2(\delta))$ bounds on the targeting algorithm.

[3] Finding an example region containing three pebbles with no potential savings is straightforward. However, adding a fourth pebble requires an interval large enough to allow the smallest of the original three to achieve some savings. This logic applies inductively.

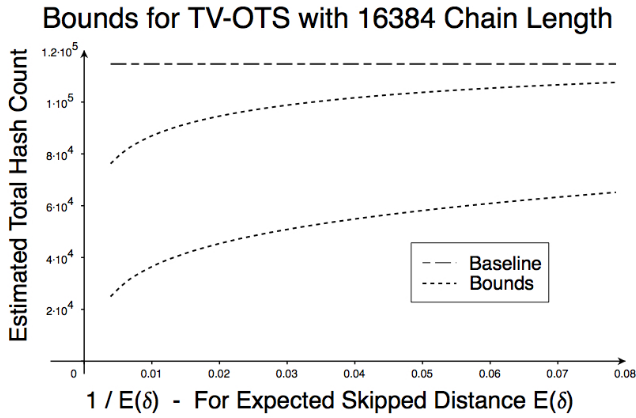

Figure 5: A bounds prediction for TV-OTS, calculated by subtracting the calculated expected savings from a baseline expectation for iterative FHT.

Summing over the $\rho(\delta, 3)$ pebbles that cause savings gives:

$$S_{min}(\delta) = \sum_{i=1}^{\rho(\delta,3)} \frac{\delta - 2^{i+2}}{2}$$
$$= \begin{cases} \frac{\delta}{2}\rho(\delta,3)) - 2^{\rho(\delta,1)} + 4 & : \lceil \log_2(\delta) \rceil > 3 \\ 0 & : Otherwise \end{cases} \quad (3)$$

An upper bound is achieved from slightly overestimating the savings below the target and adding the maximum additional savings from moving pebbles above the target. In the best case, all the moving pebbles except the largest can be moved for free. This is at most $\rho(\delta, 1) = \lceil \log_2(\delta) \rceil - 1$ pebbles. The savings below the target are also adjusted, counting $\rho(\delta, 1)$ pebbles potentially saving up to $\frac{\delta}{2}$ operations each:

$$S_{max}(\delta) = \sum_{i=1}^{\rho(\delta,1)} \frac{\delta}{2} + \sum_{i=1}^{\rho(\delta,1)} 2^i$$
$$= \frac{\delta}{2}\rho(\delta,1) + 2^{\rho(\delta,0)} - 2 \quad (4)$$

From $S_{min}(\delta) \in \Omega(\delta \log_2(\delta))$ and $S_{max}(\delta) \in O(\delta \log_2(\delta))$, and the relationship $S_{min} \leq S \leq S_{max}$, a tight bound can be placed on $S(\delta)$:

$$S(\delta) \in \Theta(\delta \log_2(\delta)) \quad (5)$$

Table 3.2 summarizes the differences between the iterative method, and targeting in the best and worst case.

3.2.1 Bounds Prediction for TV-OTS

The theoretical bounds help predict the performance of TV-OTS using targeting. Figure 5 shows expected bounds calculated per traversal, estimated by subtracting calculated savings from a baseline TV-OTS approximation. The estimated requirement of a TV-OTS traversal is $\frac{n}{2}\log_2(n)$ hash operations, recalling that over an entire traversal each pebbles hashes about half the chain, or $\frac{n}{2}$ operations[4].

Bounds predictions assume different values of δ occur over the course of a traversal. In TV-OTS, all keys have equal probability of being retrieved. Consequently, the distance δ between retrievals follows a geometric distribution. The bounds in Figure 5 were calculated by summing over the geometrically weighted savings for different δ's. Results are plotted over $E(\delta)$, the expected δ of the distribution which varies with the key retrieval probability of the protocol.

3.3 State Calculation

The correct state after any retrieval can be determined by a small set of rules. The state calculator given by Algorithm 1 uses these rules to determine final pebble positions that match those produced by an iterative FHT traversal. The primary property used in determining new positions is that each moving pebble must create an interval below a larger pebble that matches the moving pebble's ID. The larger pebbles that assist in finding new destinations for moving pebbles are referred to as *reference* pebbles. The reference pebbles limit the possible destinations of the moving pebbles, some of which will lie above the target and some below. Those below are disregarded as they lie in a region of skipped values. Of the remaining possibilities, the lowest is chosen for the moving pebble's new destination. The choice of lowest position reflects that pebbles moved iteratively will stop as soon as they move past the target. The availability of the correct reference location is ensured by deciding the destinations for larger pebbles before smaller ones. By knowing the destinations for all larger pebbles, the correct destination is certain to be found for each moving pebble.

Because state calculation always chooses the lowest possible position, only one pebble needs to be considered as a reference. Anticipating pebble movements, there are only two pebbles that serve as possible references. These are the two closest in position to the retrieval target. Of these two, the lower will be chosen assuming the moving pebble, when positioned below this lower reference, will still be above the target. If this is not the case, the other reference is chosen. In this second case, the two potential reference pebbles border the interval for the moving pebble to split. Since this division is even, placing the moving pebble at an interval *below* the *higher* of the two references is equivalent to placing the moving pebble *above* the *lower* reference. Because this process is only responsible for finding destinations and not for moving pebbles, knowing the position of only this lower reference is sufficient to find the proper destination.

With these guiding principles in place, state calculation can be described as an iterative algorithm for calculating individual pebble destinations. Setup requires finding the location, β, of a pebble to use as the initial reference. Specifically, β is the position of the pebble whose ID is the smallest from a certain set of pebbles. This set is comprised of all the pebbles whose IDs are larger than the IDs of all the pebbles below the target[5]. The number of pebbles below the target is a function of the interval length between retrieved values. Once the number of pebbles to move is known, finding a pebble to use for β requires checking pebbles above the target until one satisfies the criterion just mentioned. Once β is determined, iteration can start with the pebble of the next lower ID.

[4]The cost of FHT is reported as $2\log_2(n)+1$ operations per retrieval, but this is an upper bound and does not necessarily imply that a sequence of n retrievals requires $n \times (2\log_2(n) + 1)$ operations.

[5]Note that this set is not equivalent to all the pebbles above the target: a pebble below the target could have an ID larger than that of a pebble above the target.

Algorithm 1: State Calculation

```
Data: A list of pebbles L of length n
Input: A target position t
Result: L is modified
/* initialize iteration control variables */
idx ← ⌊ log2(t − L.getPebbleByIndex(0).pos + 1)⌋
nxtId ← 2^idx
p ← L.getPebbleByIndex(idx)
while p.dest < t or p.id ≤ nxtId do
    idx ← idx + 1
    nxtId ← 2 × nxtId
    p ← L.getPebbleByIndex(idx)
β ← p.pos
/* begin iteration */
while nxtId > 1 do
    p ← L.getPebbleByID(nxtId)
    /* Decide new position for p */
    if p.id < β − t then
        /* p fits below β and above t */
        β ← β − p.id
        p.dest ← β
    else
        p.dest ← β + p.id
    /* update nxtId */
    nxtId ← nxtId/2
```

Function movePebble(L, p, i)

```
Input: The list L of pebbles
Input: The pebble p to move
Input: An index i to which to move p
Result: p is moved in L
q ← L.getPebbleByIndex(i)
p.pos ← q.pos
p.val ← q.val
L.remove(p)
L.putPebbleAtIndex(p, i − 1)
```

The complete state calculation algorithm is given in Algorithm 1. The choice of pebble used for β likely changes as iteration progresses through the moving pebbles. At any point in time, β is intended to represent the lowest possible reference. Whenever a pebble's new destination is chosen to be below β, the current value of β is no longer valid as the lowest reference. β is then updated using the most recently found destination. Once the iteration completes, destinations have been determined for all pebbles. The lowest destination matches the position of, or one step above, the position of the target.

3.4 Traveling Pebble State Transition

With a newly calculated target state, the challenge becomes efficient movement of pebbles to their new positions. The Traveling Pebble algorithm uses the Function `movePebble` to accomplish state transition with minimal hash operations.

The general approach of the Traveling Pebble method, given in Algorithm 2, is to begin moving each value as quickly as possible, considering pebbles from largest to smallest. In this way, larger pebbles begin their moves before smaller ones. When a pebble p_t is found to move, a lookahead operation is performed to determine the destination of the next smaller pebble, p_s. The destination of p_s determines whether p_t may complete its move, or whether it must begin its move but pause when it reaches p_s's destination, allowing p_s to copy the value at p_t's position. If this is indeed the case, this procedure is followed and then repeated for a new p_s, performing lookahead operations for smaller pebbles until one is found whose destination lies below that of p_t. At this point, no more smaller pebbles will need to move above p_t and p_t completes its move.

When lookahead operations are performed, p_t's destination is already known. When moving downward, p_t will only pause for a p_s whose destination is above p_t's. To make checking p_s's destination easier, β is updated to p_t's destination as soon as this destination is known instead of waiting until it is actually occupied. This ensures that destinations of the p_s pebbles can be compared with β to correctly determine when each p_s should move.

4. CORRECTNESS

The structural model used for proving correctness can be simplified over the computational structures used in implementation. Abstractly, a chain is modeled as an ordered list of all pebbles. Pebbles are simplified to a tuple representing their ID and destination. The correspondence between a pebble's position and value is one-to-one, so that pebbles will output the correct values exactly when their positions are correct. As proved by Jakobsson, pebbles' destinations will be correct by the time their values are needed [10]. This applies to the targeting algorithms as well, since the amortization strategy used in targeting may be overzealous and move pebbles farther than actually necessary, but not less. Thus, pebbles may arrive at their destinations sooner than necessary, but never later. Due to this eager approach, pebbles' values will be correct if their destinations are chosen correctly. Proof of correct destinations follows.

Definition 1. *A pebble is represented by a tuple, $p_i = \langle i, d\rangle$, where i and d respectively represent the pebble's identifier and destination.*

Definition 2. *A chain state is represented by a set $S = \{\langle i, d\rangle \mid i = 2^j \text{ for } j \in [1, \log_2(n)]\}$. In the initial state, $d = i$ for all $\langle i, d\rangle \in S$.*

The targeting algorithm is proven correct by showing that the set of destinations found always matches the destinations found by the iterative method for matching states. A state is characterized by the position of the lowest pebble. For this proof, the target, t is assumed to be at an evenly-numbered position. This assumption is safe since the state responsible for calculating an evenly positioned value is also responsible for the lower adjacent odd position.

Definition 3. *A skip function, $\zeta(S)$, updates a pebble's final destination before moving upward during a single step in the iterative algorithm. This function is given by $\zeta(\langle i, d\rangle) = (i, d + 2i)$.*

Definition 4. *A single step in the iterative method is described by the function ψ which finds the pebble $\langle i, d\rangle$ with the lowest destination and replaces it with $\zeta(\langle i, d\rangle)$. To reach*

Algorithm 2: Traveling Pebble State Transition

```
Data: A list, L, of pebbles with destinations chosen by
      Algorithm 1
Input: A target position t
Result: Pebbles in L are moved to new destinations
        above t
/* make t even                                        */
if t is odd then t ← t + 1
/* amortized hashing of pebbles not at their
   destinations not shown                             */
/* initialize iteration control variables             */
idx ← ⌊log_2(t − L.getPebbleByIndex(0).pos + 1)⌋;
nxtId ← 2^idx
p ← L.getPebbleByIndex(idx)
while p.dest < t or p.id ≤ nxtId do
    idx ← idx + 1
    nxtId ← 2 × nxtId
    p ← L.getPebbleByIndex(idx)
β ← p.pos
/* begin iteration                                    */
while nxtId > 1 do
    p ← L.getPebbleByID(nxtId)
    /* a check could be added here to ensure
       that p actually needs to move and is not
       to be retired                                  */
    if p.dest < β then
        movePebble(L, p, idx)
        β ← p.dest
        /* set up for next pebble                     */
        idx ← idx − 1
        nxtId ← nxtId/2
        /* lookahead for smaller pebbles              */
        q ← L.getPebbleByID(nxtId)
        while nxtId > 1 and q.dest > β do
            p.dest ← q.dest
            p.hashToDestination()
            movePebble(L, q, idx)
            idx ← idx − 1
            nxtId ← nxtId/2
        p.dest ← β
        p.hashToDestination()
    else
        movePebble(L, p, idx + 1)
        p.hashToDestination()
        /* set up for next iteration                  */
        idx ← idx − 1
        nxtId ← nxtId/2
```

a new state, ψ is evaluated repeatedly until the target t corresponds to the lowest pebble.

From these definitions, each destination can be found in a unique way.

THEOREM 1. *For a state associated with the retrieval of t, there is only one legal position for each pebble.*

PROOF. Initial pebble positions are fixed, and subsequent destinations are chosen only from previous ones. Thus, the possible destinations of any pebble over the course of the traversal are also fixed. For any given pebble, these destinations are evenly spaced: the distance between possible destinations for pebble p_i is $2i$. Given that this is the distance moved by a pebble in an iterative step, only one position exists for p_i when moving from below target t to above. □

COROLLARY 1. *For a state corresponding to target t, the new destination of each pebble $\langle i, d\rangle$ satisfies $t \le d < t + 2i$.*

PROOF. The iterative method always moves the pebble with the lowest destination and continues until no more pebbles can be moved above t, so $t \le d$ holds trivially for all pebbles, Destinations are also bounded from above. Let $\langle i, d_f\rangle = \zeta(\langle i, d_i\rangle)$ represent a pebble after a move where $d_i < t \le d_f$. Using $d_f = d_i + 2i$ gives,

$$\begin{aligned} d_i &< t \\ d_i + 2i &< t + 2i \\ d_f &< t + 2i \end{aligned} \tag{6}$$

Showing that d_f must be within $2i$ positions of t. □

The equivalence of targeted states to iterative states can now be shown by showing that destinations found by targeting satisfy Theorem 1 and Corollary 1.

DEFINITION 5. *Let $\Psi(S)$ assign a new destination to a single pebble as part of a targeting algorithm state calculation. A full state calculation uses Ψ repeatedly, assigning a new destination to each pebble until all destinations are above t.*

THEOREM 2. *The set of destinations found by the iterative method for the state corresponding to t can be found without iteration by spacing new destinations relative to each other instead of relative to old pebble positions.*

The proof of this theorem is a multi-step process. First, the set A_i is defined relative to pebble p_i, which will be useful later on.

DEFINITION 6. *Let $A_i = \{\langle i_a, d_a\rangle \mid i_a > i\}$ relative to a pebble $p_i = \langle i, d\rangle$.*

LEMMA 1. *Given a pebble $p_i = \langle i, d_i\rangle$ to move, let d_f be a valid final destination. The locations $d_f - i$ and $d_f + i$ must be valid destinations for two pebbles in A_i. Moreover, $d_f + i$ and $d_f - i$ are the closest destinations to d_f which are valid for any pebble in A_i.*

PROOF. The precise set of destinations for each pebble does not overlap with any of the other pebbles, limiting the pebbles that may occupy a given location. Specifically, for a pebble $\langle i, d\rangle$, all legal values for d fit the form

$$\begin{aligned} d &= d_k \text{ where } \langle i, d_k\rangle = \zeta^k(\langle i, i\rangle) \\ &= i + 2i \times k \end{aligned} \tag{7}$$

Where $k \in \mathbb{Z}^+$ and ζ^k represents repeated application of ζ k times. Equation 7 is useful for determining relative offsets between pebbles. Let $\langle i_a, d_a\rangle, \langle i_b, d_b\rangle \in A_i$ be the pebbles above and below d_f. Because i, i_a and i_b are each a power of two, $i_a = 2^{j_a} i$ and $i_b = 2^{j_b} i$ for some $j_a, j_b \in \mathbb{Z}^+$. Using Equation 7, d_a and d_b can be rewritten to obtain expressions

in terms of d_f and i:

For d_a:

$$\begin{aligned}d_a &= i_a + 2i_a k_a \\ &= 2^{j_a} i + 2(2^{j_a} i k_a) \\ &= 2i(2^{j_a - 1} + 2^{j_a} k_a)\end{aligned}$$

$$\text{Setting } k = 2^{j_a} k_a + 2^{j_a - 1} - 1 :$$

$$\begin{aligned}d_a &= 2i(k+1) \\ &= 2i + 2ik \\ &= d_f + i \qquad (8)\end{aligned}$$

And similarly for d_b:

$$\begin{aligned}d_b &= i_b + 2i_b k_b \\ &= 2^{j_b} i + 2(2^{j_b} i k_b) \\ &= 2i(2^{j_b - 1} + 2^{j_b} k_b)\end{aligned}$$

$$\text{Setting } k = 2^{j_b} k_b + 2^{j_b - 1} :$$

$$\begin{aligned}d_b &= 2i(k) \\ &= d_f - i \qquad (9)\end{aligned}$$

The fact that there is no third pebble $\langle i_c, d_c \rangle \in A_i$ satisfying $d_b \leq d_c \leq d_a$ stems from the same positional relationships that relate i, i_a and i_b. By Equations 8 and 9, the distance between i_a and i_b is $2i$. Moreover, the distances between any two pebbles in A_i must be a multiple of $2i$. No pebble from A_i could be between d_a and d_b without violating this condition. □

Lemma 2 shifts perspective, building on Lemma 1 to show that the destinations for larger pebbles surrounding each new destination can be used to *find* this new destination.

LEMMA 2. *Let $\langle i, d \rangle$ be a pebble to be moved to an unknown destination d_f. Assume that all pebbles in A_i currently have destinations above t. In this situation, a pebble from A_i will be located at either $d_f + i$ or $d_f - i$.*

PROOF. By definition, A_i must contain a pebble, $\langle i_{2i}, d_{2i} \rangle$, such that $i_{2i} = 2i$. Corollary 1 requires that $t \leq d_{2i} < t + 2i_{2i}$. Note $2i_{2i} = 4i$, so if

$$t \leq d_{2i} < t + 3i,$$

then Lemma 2 is clearly satisfied with $d_f - i = d_{2i}$. In the remaining case,

$$t + 3i \leq d_{2i} < t + 4i.$$

Here, Equation 9 states that there must be a pebble destination at $d_{2i} - i_{2i} = d_f - i$. Furthermore, with $d_f \pm i$ above t, the pebbles corresponding to these destinations must be in A_i and therefore actively present in these positions. □

The Ψ function applies the above procedure, finding a pebble p_i with a valid A_i set and updating its destination. Induction can now be applied in a full proof of Theorem 2, showing that starting from any correct state, the new destinations found for all pebbles form a new correct state.

PROOF. *Base case:* A pebble $p_i = \langle i, d \rangle$ exists such that the corresponding A_i set contains only pebbles that are above t. This is necessarily true unless p_i will move beyond the end of the chain, in which case this proof is unnecessary. In such a state, a new destination for p_i can be correctly determined.

Inductive Step: Given a state where a new destination can be determined for $p_i = \langle i, d \rangle$, applying $\Psi(S)$ and updating this pebble's destination will result in a state where either 1) no more pebbles need to be updated, or 2) there is now a new pebble to which Ψ can be applied. The first case trivially halts the induction. In the second, using Ψ to determine a new destination for p_i automatically results in a new partitioning where all the pebbles in $A_{i/2} = A_i \cup \{\langle i, d \rangle\}$ now have destinations above t. Thus, a new pebble, $p_{1/2} = \langle i_{i/2}, d_{i/2} \rangle$, is found for which a new destination can be found. In some cases, $d_{1/2}$ will already be above t, however, this causes no change since exactly the same two cases apply as if $p_{1/2}$ had just been updated. □

Targeting transitions follow these steps of iteratively applying $\Psi(S)$ and moving each pebble, and hence the new destination of each pebble will be correct. Thus, each state will correctly match the state found by the iterative method for the same target. Since both algorithms start from identical initial states, correctness is ensured over arbitrary state changes.

5. EXPERIMENT AND ANALYSIS

To verify performance, the targeting algorithm was implemented and compared with a slightly modified version of the iterative traversal approach. The modified FHT decreases latency by neglecting hash operations needed to compute values at odd positions in skipped regions. This version was dubbed *Half Iterative* and improves efficiency by removing at most $\frac{n}{2}$ hashes per traversal. All algorithms were implemented in Java, but compiled using the GCJ front end for the GCC compiler to prevent runtime performance modifications by Java's Just-In-Time compiler. The choice of hash function was SHA-1[6]. Tests were performed on a 3GHz AMD Phenom II processor with 8GB of RAM running Ubuntu 12.04.

Tests were designed to measure performance for secure uses of TV-OTS. Only the operations used to traverse the chain are measured, with the n hash operations required to initialize the chain considered setup cost. Retrievals requested by TV-OTS are independent, with all values having equal probability, p, of being retrieved. Thus, the expected interval size between retrieved values is $n \times p$. The assumed application for all tests uses a message rate of 30Hz and a chain length of 16,384 which, for example, gives TV-OTS a total lifespan of more than 4 hours if keys are refreshed every second.

Tests were designed to show performance over a range of probabilities. Security is measured by the probability of an adversary finding a message for which a signature can be forged. Forgery probability is directly related to the percentage of keys retrieved. If each signature contains 13 keys, then out of 4096 keys, at most $v = 30\frac{messages}{second} \times 13 \frac{keys}{message} = 390 \frac{keys}{second}$ are used. For this scenario, the fraction of retrieved keys about 9%, but the probably that a message can be forged is more than .0001 [2]. However, the fraction of retrieved keys drops rapidly when increasing security

[6]The choice of hash function affects timed performance. When comparing latencies, the performance increases will appear greater with slower hash functions. For example, results presented here would be diminished by using a faster function such as MD5 and magnified by a slower function such as SHA-256.

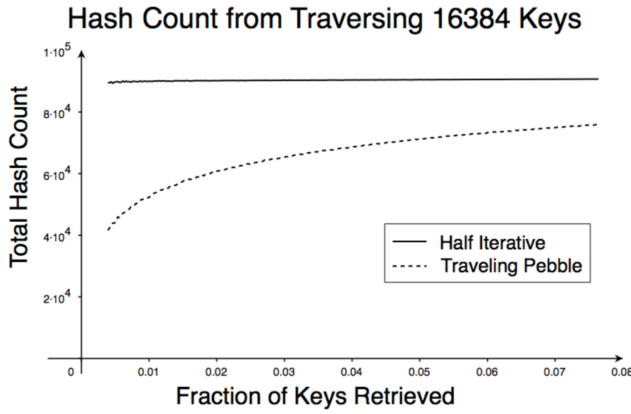

Figure 6: Comparison of the total number of hash operations performed over the course of a traversal.

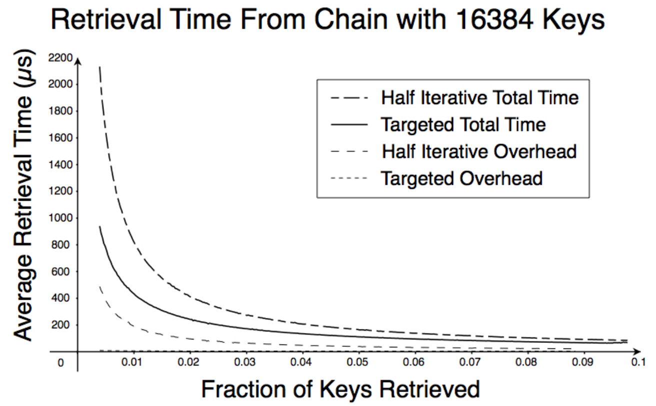

Figure 7: Average time of a single retrieval operation is plotted over the fraction of keys retrieved. Latencies were measured both for the entire retrieval and overhead only. Overhead was evaluated by running the traversals without performing hash operations. The targeting algorithm has lower latencies both when measuring overall time and overhead-only performance.

by increasing the number of chains N. It also drops when taking into account the possibility that keys may be used in more than one signature. Equation 10 calculates the expected probability of retrieval of any key by probabilistically weighting the possible numbers of retrieved keys and dividing by N. Some sample values based on Equation 10 are shown in Table 2.

$$\epsilon(N, v) = \frac{1}{N} \times \frac{\sum_{k=1}^{v} k \binom{N}{k} \binom{v-1}{k-1}}{\sum_{k=1}^{v} \binom{N}{k} \binom{v-1}{k-1}} \tag{10}$$

Table 2: Expected fraction of retrieved keys is shown for various combinations of N and v. Entries are omitted where TV-OTS signatures would be impossible when using a hash length limited to 160 bits.

Percentage Of Keys Retrieved			
N	$v = 30 \times 13$	$v = 30 \times 12$	$v = 30 \times 11$
4096	8.7%	8%	7.4%
8192	-	4.2%	3.8%
16384	-	-	1.9%

Tests were designed to highlight performance at these percentages and show the surrounding context. Tests averaged 12 trials, which roughly reflects the number of retrievals required for each signature. In each trial, the positions of the retrieved values were randomly generated, as would be expected from the execution of TV-OTS.

Performance graphs show the differences between the iterative and targeted retrievals. Figure 6 shows the difference in the numbers of hash operations performed by iterative and targeted traversals. This graph shows that the savings are approximately $\frac{\delta}{2} \log_2(\delta)$ where $\delta = \frac{1}{p}$, and fall within the bounds derived in Section 3.2. Latency measurements are shown in Figure 7 for both total retrieval time and retrieval overhead without hashes.

6. FUTURE WORK

This work could be expanded in several directions, focusing on algorithm optimization or incorporation with other signature protocols.

At the algorithm level, the number of hash operations may be optimal, but the process by which pebbles are chosen to move between states may still be optimized. With the current algorithms, an unfortunate consequence of storing pebbles ordered by position is the need to iteratively search for pebbles identified by ID. Eliminating this search would be a large step towards a more efficient targeting algorithm. This might be accomplished with different sorting strategies, or data structures that work more efficiently with multiple types of searches. Alternatively, this searching may be eliminated altogether if algorithms were found to determine pebble destinations by some means other than relative offsets. Such an algorithm would lower the traversal overhead even further beyond the improvement shown by Traveling Pebble transitions.

To achieve reliable security with TV-OTS, the number of skipped keys must be large. The base-two inspired pebble arrangements used by FHT may not be the most efficient for TV-OTS. Other arrangements might increase efficiency by grouping pebbles more densely around different intervals. A modified algorithm based on a different arrangement pattern potentially requires fewer operations to transition between states.

When TV-OTS was first published, HORS appeared to be the most appropriate one-time signature to incorporate, but this may no longer be true. FHT with targeting increases flexibility in protocol choices. The ability to skip keys more efficiently and with less scaling penalty may open up other possibilities for one-time signatures that fit in practically with the TV-OTS family.

7. CONCLUSION

Multicast data authentication is a difficult problem with no general solution suited to low latency applications. This problem is becoming increasingly important, with an emerging need for this type of protocol in critical infrastructure monitoring and control applications. TV-OTS is a protocol that shows promise for use in these systems, but too little is known about its practical use to provide an accurate assessment. To further the goal of accurately evaluating TV-OTS, the issue of key generation was addressed. The non-optimality of current methods was shown as inspiration for improvement.

The improved key management strategy builds on the FHT method by adding the ability to skip values without performing unnecessary hash operations — a necessary improvement for optimal performance of TV-OTS. This approach was motivated by the idea that FHT offers bounded retrieval time for consecutive hash chain values, but also wastes operations whenever values were skipped. With the addition of targeted state changes, the retrieval latency is decreased when skipping values. The state transition algorithms lowered not only the number of hash operations that were performed, but also the associated state transition overhead. The achieved performance gain corresponds to the percentage of keys retrieved from the chain. Savings increase with the distance δ between retrieved keys and are bounded by $\Theta(\delta \log_2(\delta))$. Security of TV-OTS increases relative to $\frac{1}{p}$, where p is the probability of key retrieval. The average distance δ grows as p is decreased, meaning TV-OTS performance savings improve as security is increased. These results indicate the targeting method to be a good choice for use with TV-OTS.

8. ACKNOWLEDGMENTS

The authors would like to thank Roberto Tamassia and the anonymous reviewers for their contributions toward improving this paper.

9. REFERENCES

[1] Bakken, D., Bose, A., Hauser, C., Whitehead, D., and Zweigle, G. Smart generation and transmission with coherent, real-time data. *Proceedings of the IEEE 99*, 6 (June 2011), 928 –951.

[2] Cairns, K., Hauser, C., and Gamage, T. Flexible data authentication evaluated for the smart grid. *2013 IEEE International Conference on Smart Grid Communications (SmartGridComm)* (2013), To appear.

[3] Challal, Y., Bettahar, H., and Bouabdallah, A. A taxonomy of multicast data origin authentication: Issues and solutions. *Communications Surveys & Tutorials, IEEE 6*, 3 (2004), 34–57.

[4] Coppersmith, D., and Jakobsson, M. Almost optimal hash sequence traversal. In *Financial Cryptography* (2003), Springer, pp. 102–119.

[5] Diffie, W., and Hellman, M. New directions in cryptography. *IEEE Transactions on Information Theory 22*, 6 (1976), 644–654.

[6] Fuloria, S., Anderson, R., McGrath, K., Hansen, K., and Alvarez, F. The protection of substation communications. In *Proceedings of SCADA Security Scientific Symposium* (2010).

[7] Hauser, C., Manivannan, T., and Bakken, D. Evaluating multicast message authentication protocols for use in wide area power grid data delivery services. In *2012 45th Hawaii International Conference on System Science (HICSS)* (2012), IEEE, pp. 2151–2158.

[8] Hu, Y., Jakobsson, M., and Perrig, A. Efficient constructions for one-way hash chains. In *Applied Cryptography and Network Security* (2005), Springer, pp. 167–190.

[9] Itkis, G., and Reyzin, L. Forward-secure signatures with optimal signing and verifying. In *Advances in Cryptology–Crypto 2001* (2001), Springer, pp. 332–354.

[10] Jakobsson, M. Fractal hash sequence representation and traversal. In *2002 IEEE International Symposium on Information Theory* (2002), IEEE, p. 437.

[11] Kim, S. Improved scalable hash chain traversal. In *Applied Cryptography and Network Security* (2003), Springer, pp. 86–95.

[12] Lamport, L. Constructing digital signatures from a one-way function. Tech. rep., Technical Report CSL-98, SRI International, 1979.

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

[14] Li, Q., and Cao, G. Multicast authentication in the smart grid with one-time signature. *Smart Grid, IEEE Transactions on 2*, 4 (2011), 686–696.

[15] Perrig, A. The BiBa one-time signature and broadcast authentication protocol. In *Proceedings of the 8th ACM conference on Computer and Communications Security* (2001), ACM, pp. 28–37.

[16] Perrig, A., Canetti, R., Song, D., and Tygar, J. Efficient and secure source authentication for multicast. In *Network and Distributed System Security Symposium (NDSS)* (2001), vol. 1, pp. 35–46.

[17] Perrig, A., Canetti, R., Tygar, J. D., and Song, D. The TESLA broadcast authentication protocol. *RSA CryptoBytes 5*, 2 (2002).

[18] Reyzin, L., and Reyzin, N. Better than BiBa: Short one-time signatures with fast signing and verifying. In *Information Security and Privacy* (2002), Springer, pp. 1–47.

[19] Rivest, R., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21*, 2 (1978), 120–126.

[20] Sella, Y. On the computation-storage trade-offs of hash chain traversal. In *Financial Cryptography*, Lecture Notes in Computer Science. Springer, 2003, pp. 270–285.

[21] Wang, Q., Khurana, H., Huang, Y., and Nahrstedt, K. Time valid one-time signature for time-critical multicast data authentication. In *INFOCOM 2009, IEEE* (2009), IEEE, pp. 1233–1241.

[22] Yum, D., Seo, J., Eom, S., and Lee, P. Single-layer fractal hash chain traversal with almost optimal complexity. *Topics in Cryptology–CT-RSA 2009* (2009), 325–339.