More Practical and Secure History-Independent Hash Tables

Michael T. Goodrich[1], Evgenios M. Kornaropoulos[2(✉)], Michael Mitzenmacher[3], and Roberto Tamassia[2]

[1] Department of Computer Science, University of California, Irvine, USA
goodrich@acm.org
[2] Department of Computer Science, Brown University, Providence, USA
{evgenios,rt}@cs.brown.edu
[3] School of Engineering and Applied Science, Harvard University, Cambridge, USA
michaelm@eecs.harvard.edu

Abstract. Direct-recording electronic (DRE) voting systems have been used in several countries including United States, India, and the Netherlands to name a few. A common flaw that was discovered by the security researchers was that the votes were stored sequentially according to the time they were cast, which allows an attacker to break the anonymity of the voters. Subsequent research pointed out the connection between vote storage and the privacy property *history-independence*. In a weakly history-independent data structure, every possible sequence of operations consistent with the current set of items is equally likely to have occurred. In a strongly history-independent data structure, items must be stored in a canonical way, i.e., for any set of items, there is only one possible memory representation. Strong history-independence implies weak history-independence but considerably constrains the design choices of the data structures. In this work, we present and analyze an efficient hash table data structure that simultaneously achieves the following properties:

- It is based on the classic *linear probing* collision-handling scheme.
- It is *weakly history-independent*.
- It is secure against *collision-timing attacks*. That is, we consider adversaries that can measure the time for an update operation, but cannot observe data values, and we show that those adversaries cannot learn information about the items in the table.
- All operations are *significantly faster in practice* (almost 2x faster for high load factors) than those of the commonly used strongly history-independent linear probing method proposed by Blelloch and Golovin (FOCS'07), which is not secure against collision-timing attacks.

To our knowledge, our hash table construction is the first data structure that combines history-independence and protection against a form of timing attacks.

Keywords: Hash table · History-independence · Timing attack · Vote storage

I. Askoxylakis et al. (Eds.): ESORICS 2016, Part II, LNCS 9879, pp. 20–38, 2016.
DOI: 10.1007/978-3-319-45741-3_2

1 Introduction

Hashing is a classic technique [21] for implementing a dynamic dictionary of key-value items supporting the following operations:

- insert(k, v): Insert item[1] (k, v).
- find(k): Return the value with key equal to k, or null if none exists.
- delete(k): Delete the item with key equal to k.

The are many applications of such a data structure and it is well-known that hashing can achieve $O(1)$ expected-time performance for each operation (e.g., see [16]). In such a scheme, the items are stored in an array, T, according to a mapping derived from a hash function, hash$(\cdot)$, such that the item (k, v) is ideally stored in the cell $T[\mathsf{hash}(k)]$. If multiple items map to the same cell, then we say that a *collision* has occurred, and we need some way of resolving the collision. One of the classic collision resolution methods is *linear probing* [21]. In this scheme, one simply incrementally searches the next cells in the array, $T[\mathsf{hash}(k) + 1]$, $T[\mathsf{hash}(k) + 2]$, and so on, modulo the size of T, until one finds an empty cell. Linear probing achieves $O(1)$ expected time performance if the ratio of the number of occupied cells to the total number of cells, which is known as the *load factor*, is a constant strictly less than 1 (e.g., see [16,21]). Although this is a classic hashing scheme, its use is nevertheless ubiquitous in computing today, including many instances where security and privacy are essential. Thus, we are interested in this paper in hash table schemes that can provide measurable protections against various security and privacy attacks.

History-Independence. The property of *history-independence* was introduced by Naor and Teague [29] by extending a related structural obliviousness property by Micciancio [25]. The goal of history-independence is to design a data structure so that an adversary who examines the computer memory will only discover the current contents of the data structure but will not learn anything about the sequence of operations that led to the current state of the data structure. History-independence comes in two flavors [29]—in *weakly history-independence* (WHI), the adversary can examine the memory only once, whereas in *strongly history-independence* (SHI), the adversary may examine the memory multiple times. Several history-independent data structures have been proposed. Blelloch and Golovin [7] present a SHI hash table based on linear probing. The work of Naor *et al.* [28] presents a SHI Cuckoo Table that performs insertions and deletions in $O(\log n)$ time, with high probability, where n is the current number of elements in the table. Buchbinder *et al.* [8] show time complexity separation between the weak and the strong notions of history-independent data structure. Our focus in this paper is on the WHI framework, as we feel it is has a more realistic risk scenario. Moreover, by a lower bound due to Hartline *et al.* [18], in order for a data structure to be SHI, it *must have* a canonical memory representation, which is a fairly restrictive requirement that seems to conflict with protections against the next type of attack that we consider in this paper.

[1] We assume that keys are unique, but all of our results extend to the setting where insertion of an already allocated key with a new value replaces the current value.

Vote Storage. The DRE AccuVote-TS voting machine was used by 10 % of the voters in the 2006 US general election. Security flaws were found both in the software [22] and the hardware [12] of the device. For example, Kohno *et al.* [22] found that each vote is written sequentially to the file that stores the votes, which can break the anonymity of the system. An attacker with some side-channel information can link the ballot to a voter based on the order that the votes are stored. The same privacy flaw was also found in the Indian Electronic Voting Machines that have been used for national elections of India since 2004. The work by Wolchok *et al.* [33] reports that votes are stored in the order cast. Later work recognized the connection of the above privacy issue with the notion of history-independence and suggested the use of strongly history-independent data structures [6,26,27]. In fact, in March 2015 the United States Election Assistance Committee approved the next generation of Voluntary Voting System Guidelines where they specify that ballot images *must be recorded in a randomized order* by the DRE for the election (Sect. 2.4.4.2 in [1]). Given that the voting machines are examined usually after the election process, i.e. post-election audits [2,31], the notion of weak history-independence seems to be more appropriate than the stronger notion of strong history-independence.

Memory Attacks. Beyond voting machines, there are other cases where an adversary can obtain access to a snapshot of the working memory and weak history-independence can be usefully applied. A direct memory access attack, or DMA attack, is an attack where the adversary with physical access to the machine bypasses all security measures of the operating system and directly accesses the memory via the high-speed expansion ports. Tools such as "Inception" [24] mount a DMA attack over PCI interfaces and acquire a complete image of the working memory. Cold boot attacks [17] allow the attacker to dump the image of the memory to an external medium, which can even identify and reconstruct the cryptographic keys from the acquired image and thus overcome disk encryption. In application scenarios such as ballot storage [26] or hospital admission management [4] such leakage might violate desired privacy.

Other History-Independent Systems. Bajaj and Sion proposed a history-independent file system named HIFS [3] that provides history-independence across both file system and disk layers of the storage stack. We note here that HIFS deploys the SHI hash table from [7], which is significantly slower than our WHI construction. A direct substitution, followed by some minor changes, would significantly speed up their design while maintaining suitable privacy guarantees for many applications. Unfortunately, the history-independence of HIFS [3] is not guaranteed for flash storage devices. The reason is that the block placement algorithms in flash storage devices are managed internally (in a non-history-independent manner) in order to maximize performance and lifetime of the storage. To remedy that, Chen and Sion proposed history-independent schemes that are tailored for flash-based block devices [10]. Another system, Ficklebase [4], suggests the use of history-independent data structures within a relational database architecture for the underlying database storage engine in order to avoid unwanted recovery of deleted information through forensic analysis.

Timing Attacks. Another type of attack that can cause a data structure to leak information is a *timing attack*. In such a side-channel attack, an adversary does not get direct access to the memory layout of a data structure or to the operands of the executed operation, but he can nevertheless precisely time the execution of data structure operations. Such attacks typically come in two forms—in the first form, an attacker passively observes the timing of data structure operations performed by others, and in the second form, he is allowed to directly interact with the data structure, e.g., to form malicious inputs that cause errors or significant time delays that can reveal information put into the data structure by others.

In the first type of attack, the eavesdropping adversary gains knowledge about the private data using the duration of an abstract-data-type operation of the data structure. An example of such a real-world attack is presented in [13], where an attacker can measure the execution time of an insertion in a B-tree in order to detect a node split. Using this split detection information, the attacker can recover values from the database table that is under attack. As a means to formally characterize such attacks, Lipton and Naughton [23] define a *clocked adversary* to be an eavesdropping attacker who can accurately time operations of a data structure and who succeeds if he can distinguish whether the system is in a state s_1 or state s_2 given just the timing information.

For the second type of timing attack, an adversary utilizes the predictable time performance of known data structure implementations to mount an active attack. For example, Crosby and Wallach [11] introduce *algorithmic complexity attacks*, where an adversary provides inputs to a data structure so as to trigger its worst case performance. In addition, Bethea and Reiter [5] introduce *timing-unpredictability*, which is used to quantify the uncertainty of an attacker about the time performance of future operations.

Our Results. In Sect. 3, we present and analyze the first efficient hash table data structure that defends against the above information leakage attacks. Our construction, denoted as WHI, achieves the following properties:

- It is *weakly history-independent*.
- It is secure against *collision-timing attacks* (see Definition 3 below).
- Operations find, insert and delete are significantly *faster in practice* than the strongly history-independent linear probing scheme of [7].

In Table 1, we qualitatively compare our WHI scheme to several previous linear-probing hashing schemes that achieve some degree of history-independence or defend against collision-timing attacks, noting that none of them achieves protection against both types of attacks. We review these other schemes in Sect. 4 and we provide the results of experimental comparisons in Sect. 5.

Table 1. Privacy properties of hashing with linear probing

	History-Independence		Secure Against Collision-Timing Attacks
	Strong HI	Weak HI	
First-Come-First-Served (FCFS)	-	-	✓
Last-Come-First-Served (LCFS) [30]	-	-	✓
Robin Hood, Tie-Break w. FCFS/LCFS [9]	-	-	✓
Robin Hood, Tie-Break w. Key Sort	✓	✓	-
Blelloch & Golovin [7]	✓	✓	-
WHI (This work)	-	✓	✓

2 Security Model

2.1 History-Independence

An *Abstract Data Type* (ADT) is a mathematical model of a data structure that describes the type of the data stored, the operations that can be performed on the data as well as the parameters of each operation. In this work, a data structure is associated with a set of items that is also called the *logical state of an ADT*, or simply *state*. An ADT *operation* deterministically transforms the state of the data structure. A *sequence of operations* S is an ordered list of ADT operations of the data structure as defined by the corresponding ADT. A *memory representation of an ADT*, or simply a *representation*, is a mapping of the state of the data structure into the memory. In general, there can be multiple memory representations for a given state. An implementation of a data structure is a function $F : M \times O \rightarrow M$, where M is the set of all possible memory representations and O is the set of all possible ADT operations.

Following the terminology of Hartline *et al.* [18], let a and b denote the memory representation of states A and B respectively. Let S be a sequence of operations then by $A \xrightarrow{S} B$ we indicate that the sequence of operations S takes state A to state B. Let also $\Pr[a \xrightarrow{S} b]$ denote the probability that starting from memory representation a of state A, the sequence of operations S run by the corresponding implementation yields memory representation b of state B. The initially empty memory representation is denoted as $\oslash$.

Definition 1 (Hartline *et al.* [18]). *A data structure is* ***weakly history-independent*** *if, for any two sequences of operations S_1 and S_2 that take the data structure from the initialization to state A, the distribution over the memory after sequence S_1 is performed is identical to the distribution after sequence S_2 is performed. That is:*

$$(\oslash \xrightarrow{S_1} A) \wedge (\oslash \xrightarrow{S_2} A) \Rightarrow \forall a \in A, \Pr[\oslash \xrightarrow{S_1} a] = \Pr[\oslash \xrightarrow{S_2} a].$$

Definition 2 (Hartline *et al.* [18]**).** *A data structure is* ***strongly history-independent*** *if, for any two (possibly empty) sequences of operations S_1 and S_2 that take a data structure from state A to state B, the distribution over the memory representations of B after sequence S_1 is performed on representation a is identical to the distribution after sequence S_2 is performed on representation a. That is:*

$$(A \xrightarrow{S_1} B) \wedge (A \xrightarrow{S_2} B) \Rightarrow \forall a \in A, \forall b \in B, \Pr[a \xrightarrow{S_1} b] = \Pr[a \xrightarrow{S_2} b].$$

2.2 Collision-Timing Attack

For an adversary with timing capabilities, the duration of an insert/delete operation in a hash table can reveal significant information. In our motivating scenario for this security notion, the attacker cannot read the data transferred in the communication channel between the user and the cloud provider (i.e., the encrypted channel) but he can accurately time the interaction between the two entities. We model the desired security property by introducing a game where the adversary picks two input items that collide in the hash table, the cloud provider inserts/deletes only one of them. The goal of the adversary is to distinguish which of the two items was processed relying solely only the execution time of the operation. We consider a hash table secure against collision-timing attacks if the adversary succeeds in the above game with negligible probability.

Our model only deals with the timing of colliding items. In order not to leak whether a collision occurs, one would have to deploy a hash table for which the execution time is *not affected by collisions*. Notice that it is particularly challenging to decouple the time performance of a hash table from the occurrence of collisions. One may think that in order not to leak whether collision occurs it is enough to have constant worst-case time performance for updates. This is not necessarily true since there can be maintenance actions in the hash table, that take constant time but reveal whether a collision took place.

Finding Colliding Items. As shown in the work of Lipton and Naughton [23], there is a straightforward process for an adversary to generate a pair of colliding items, i.e., a collision-discovery attack, even if the hash function is not known. As a first step, we describe a process from [23] that checks whether two items collide. Let t_0 be the time it takes to insert item u_0 to an empty hash table, similarly let t_1 be the time it takes to insert u_1 to an empty table. Now we insert u_1 in a hash table that already contains u_0 and check whether the time for insertion is larger[2] than t_1, if yes then the items u_0, u_1 collide. In order to find which pair of items to test for a collision, we can simply generate $\Omega(\sqrt{m})$ random items, where m is the table size. Indeed, assuming that the hash function distributes the items

[2] In the work of [23] the authors define a clocked adversary that has access to a clock that is accurate to within ϵ and can discover the difference between two measurements t_0 and t_1 with $O(\epsilon/|t_1 - t_0|)$ repetitions of the corresponding operations. For the ease of exposition we assume that our measurements are always accurate (i.e. $\epsilon = 0$) therefore no repetitions are required.

to the m bins uniformly at random, we can use the birthday-paradox and show that within this set of items there is a pair of colliding items with probability roughly $1/2$. Given that collision-discovery attacks for hash tables are difficult to avoid in practice, we focus on preventing information leakage from the eviction strategy, i.e., a collision-timing attack.

Security Definition. We indicate with λ the security parameter and with Op an update operation of the hash table, $Op \in \{\mathsf{insert}, \mathsf{delete}\}$. We indicate with $a \in A$ a memory representation of the state A of the hash table and with u_0, u_1 two items from the universe of input keys, K. With the term HT we denote an implementation of the hash table that has access to a source of randomness, e.g., a pseudorandom generator G. A memory representation is called *admissible* with respect to the implementation HT and the hash function $\mathsf{hash}()$, if it can be reached with non-zero probability. We define an *evidence of admissibility* of a, denoted as evd_a, a pair consisting of (1) a sequence of operations S_a and (2) a random tape rnd_a such that if S_a is applied to an empty hash table using rnd_a when necessary, then the hash table reaches memory representation a. We use a game-based definition to describe the security of our setup. The game is denoted with PRV-CTA$^{\mathcal{A}}_{HT}(\lambda)$ and is shown in Fig. 1, where CTA stands for collision-timing attack. The game begins with an algorithm run by adversary $\mathcal{A}$. When $\mathcal{A}$ finishes executing, the game performs further steps with $\mathcal{A}$'s output to produce the challenge for $\mathcal{A}$. The adversary processes the challenge and outputs a bit, which is returned by the game.

PRV-CTA$^{\mathcal{A}}_{HT}(\lambda)$:
1. $(a, \mathsf{hash}, evd_a, Op, u_0, u_1) \leftarrow \mathcal{A}(1^\lambda)$, where $\mathsf{hash}(u_0) = \mathsf{hash}(u_1)$
2. **if** evd_a is not an evidence of admissibility of a **return** 0
3. Initialize the implementation HT with memory representation a and $G(\lambda)$
4. Choose at random a bit $b \in \{0, 1\}$
5. Execute operation Op according to HT with input argument u_b and record the execution time in t_b
6. $b' \leftarrow \mathcal{A}(t_b)$
7. **return** $(b = b')$

Fig. 1. Indistinguishability game for the security of a hash table against collision-timing attacks.

Indistinguishability game PRV-CTA assumes a powerful adversary that is allowed to choose the memory representation of the hash table, i.e. the state, the allocation of the items to the cells of the table as well as its hash function $\mathsf{hash}()$. We denote as advantage of $\mathcal{A}$ the quantity $2 \cdot \Pr\left(\text{PRV-CTA}^{\mathcal{A}}_{HT}(\lambda) = 1\right) - 1$.

Definition 3. *Let λ be the security parameter and let HT be an implementation of a hash table. We say that implementation HT is* secure *against collision-timing attacks if for all PPT adversaries $\mathcal{A}$, the advantage $\mathcal{A}$ in game PRV-CTA$^{\mathcal{A}}_{HT}(\lambda)$ is negligible.*

3 Weakly History-Independent Linear Probing

In this section, we describe a weakly history-independent dictionary that is based on an open addressing hash table. For the proofs see the full version [15]. Let T be a hash table of size m and let K be the universe of keys. We hash the set of keys $U \subseteq K$ into T using hash function $\mathsf{hash}()$ and handle collisions with linear probing. In the following, the symbol $\perp$ indicates an empty cell and arithmetic over cell indices is modulo m. A *cluster* of T is a maximal contiguous sequence of nonempty cells of T.

Profile of a Set. Following the terminology of [20], we define the following sets and values for a cell i of T:

- H_i: set of items of U that hash to cell $T[i]$, of size $h_i = |H_i|$;
- P_i: set of items of U that probed cell $T[i]$, of size $p_i = |P_i|$.

The above quantities are a function of the set U and of $\mathsf{hash}()$, but for succinctness we do not denote that explicitly. Clearly, we have $H_i \subseteq P_i$. Also, if $p_i \geq 1$, then exactly one of the items in P_i ends up in cell $T[i]$ while the remaining $p_i - 1$ items probe the next cell $T[i+1]$. This observation yields the following recurrence relation (same relation as in [20] but different notation):

$$P_{i+1} = H_{i+1} \cup (P_i - \{v_i\}) \qquad \text{and} \qquad p_{i+1} = h_{i+1} + max(p_i - 1, 0), \tag{1}$$

where v_i indicates the item allocated in $T[i]$. Sequence $(p_0, \ldots, p_{m-1})$ is called the *profile* of set U. Note that set P_i depends on both the performed sequence of operations and on the eviction strategy between colliding items. In contrast, the profile of U does not depend on the eviction strategy [20], since it only counts the number of items that probed a cell. Using the above fact one can easily show that the profile is also independent of the order in the sequence of operations.

Intuition. In our insertion algorithm, we use a randomized eviction strategy. Suppose $p_i \geq 1$ items have probed the i-th cell so far. When a new item u probes the i-th cell, it evicts the current item with probability $1/(p_i + 1)$. Hence, each cell is a reservoir sample [32] of size 1, so that every item probing that cell has an equal likelihood of being stored there.

We show that this technique gives a weakly history-independent insertion process. The challenge is that, to delete an item u from a cell i, we must construct a memory representation that is consistent with u never having been inserted. Note that algorithm $\mathsf{find}(u)$ simply performs a linear forward scan starting from $\mathsf{hash}(u)$ until we either find u or an empty cell.

3.1 Insertion

We use an auxiliary table, $P[]$, to keep track of the profile, where $P[i] = p_i$. All entries in $P[]$ are initially set to 0. We assume that the hash table can access random values on the fly as we need them by means of method $\mathsf{getRand}(s)$, which returns a random integer in the range $\{1, \ldots, s\}$.

Analysis. Let T be a hash table with linear probing where insertions are performed with Algorithm 1. As defined before, a memory representation is called admissible if it can be reached with non-zero probability.

Algorithm 1. WHI.insert(u)

```
   Input : an item u to be inserted
1  i ← hash(u)
2  while T[i] ≠ ⊥ do
3  |  P[i] ← P[i] + 1
   |  // Item u is stored in T[i] with probability 1/p_i
4  |  r ← getRand(P[i])
5  |  if r = 1 then
6  |  |  Swap the content of T[i] and u
7  |  end
8  |  i ← i + 1
9  end
10 P[i] ← 1
11 T[i] ← u
```

Lemma 1. *Let U be a set of items and let R be the random variable over the set of admissible representations of U in table T. Let S be a sequence of* insert *operations, according to Algorithm 1, that insert the items of U. Then the probability that R takes value ρ given that we follow S is given by:*

$$\Pr(R = \rho) = \prod_{j=0}^{m-1} \frac{1}{max(p_j, 1)}.$$

The next lemma follows immediately from Lemma 1.

Lemma 2. *A hash table with linear probing where only insertions are performed, according to Algorithm 1, is weakly history-independent.*

We note here that since the notion of history-independence was originally formed under an information-theoretic security framework, for consistency with previous work, our lemmas above assume the availability of true randomness.

It is straightforward to relax both strong and weak history-independence definitions to a semantically secure framework and extend our results accordingly. In this case, denoting with λ the security parameter, getRand would be derived from the output of a cryptographic pseudorandom generator with security parameter λ [14] to which the hash table, but not the adversary, has oracle access. In practice, we implement getRand by means of the secure hardware random number generator provided by modern microprocessors.

3.2 Deletion

To delete an item u from the hash table we must change the memory representation, with the right probability, so that it is as though u was never inserted.

Algorithm. The deletion method is shown in Algorithms 2–3. Given that item u is allocated in cell $T[i]$ and that it hashes to cell $\mathsf{hash}(u)$ we have to: (1) decrease by one the values of $P[\mathsf{hash}(u)], P[\mathsf{hash}(u)+1], \ldots, P[i]$ and (2) cover the gap at $T[i]$ by picking an item uniformly at random among the items that probed $T[i]$. The above two steps are repeated, in case we create an additional gap by covering the first one.

Algorithm 2. WHI.delete(u)

```
   Input : an item u to be deleted
1  i ← hash(u)
2  while T[i] ≠ ⊥ do
       // Reverse the effect of u on table P
3      P[i] ← P[i] − 1
4      if T[i] = u then
5          T[i] ← ⊥
           // Fill the gap at cell T[i]
6          CoverGap(i)
7          return
8      end
9      i ← i + 1
10 end
```

Algorithm 3. CoverGap

```
   Input : the index i_g of the gap in T
1  if P[i_g] = 0 then
2      return
3  end
   // There are p_{i_g} items that probed T[i_g]. Cover with the
      rightmost.
4  cnt ← P[i_g]
5  i ← i_g + 1
6  while T[i] ≠ ⊥ do
7      P[i] ← P[i] − 1
8      if the item in T[i] probed cell T[i_g] then
9          cnt ← cnt − 1
10         if cnt = 0 then
               // Cover the gap at T[i_g], recurse for the new gap at
                  T[i]
11             T[i_g] ← T[i]
12             T[i] ← ⊥
13             CoverGap(i)
14             return
15         end
16     end
17     i ← i + 1
18 end
```

An interesting question is how to pick an item to cover the gap. One approach is to scan the cells from $T[i+1]$ until the end of the cluster and choose one of the

items of P_i uniformly at random. The above randomized approach is correct but requires additional randomness and can potentially lead to a significant number of moves between the allocated items. Our technique takes advantage of the fact that in an admissible memory representation of U, the relative order of the items that probed $T[i]$ is a random permutation. Therefore, by picking the item of $T[i]$ placed furthest from $T[i]$ in the cluster, the "rightmost" item or else the p_i-*th* eligible item to cover the gap, is equivalent to sampling uniformly at random among the items of set P_i. Besides maintaining the weak history-independence of our construction, the benefit of this technique is twofold in terms of performance. By recursively choosing items as far to the right as possible from the gap we *reduce the number of moves* between the allocated items, and we *do not require any randomness* for the deletion process. In Algorithm 2, we locate and delete the item u; an action that creates a gap. Let $T[i_g]$ be the cell where u was allocated before the deletion. Algorithm 3 covers the gap in $T[i_g]$ with the rightmost item that probed cell $T[i_g]$.

Line 8 of Algorithm 3 checks whether the item in cell $T[i]$ probed cell $T[i_g]$ on its way to cell $T[i]$. This can be implemented as follows, if the distance of the hashed location $\mathsf{hash}(T[i])$ from the start of the cluster is less than or equal to the distance of cell $T[i_g]$ from the start of the cluster, then the item in $T[i]$ probed cell $T[i_g]$.

Analysis. We build our WHI proof based on two lemmas. Given a sequence of insertions S, we first prove that the relative order of H_i in the resulting memory representation is a random permutation. Using this, we prove the more general statement that the relative order of P_i in the resulting memory representation is a random permutation. Thus, by picking the rightmost item of P_i in the memory representation, we cover the gap with a randomly chosen item from P_i. Finally, by recursively covering the gaps in this manner, we create the same probability distribution over the memory representation as if the deleted item was never inserted.

Lemma 3. *Let U be a set of items, let π_1, π_2 be two permutations of $H_i \subseteq U$ and let S be a sequence of insertions of U according to Algorithm 1. For a location i in the hash table, let R_i be the random variable that represents the relative order of set H_i associated with cell $T[i]$ in the resulting memory representation ρ. Then, by inserting the items of U into T according to S, we have:*

$$\Pr(R_i = \pi_1) = \Pr(R_i = \pi_2)$$

Lemma 4. *Let U be a set of items, let π_1, π_2 be two permutations of $P_i \subseteq U$ and let S be a sequence of insertions of U according to Algorithm 1. For a location i in the hash table, let R'_i be the random variable that represents the relative order of set P_i associated with cell $T[i]$ in the resulting memory representation ρ. Then, by inserting the items of U into T according to S, we have:*

$$\Pr(R'_i = \pi_1) = \Pr(R'_i = \pi_2)$$

The next theorem summarizes the history-independence of our construction.

Theorem 1. *The linear probing hash table implementation described by Algorithms 1, 2 and 3 is a weakly history-independent data structure that performs searches, insertions and deletions in $O(1)$ expected time.*

3.3 Protection Against Collision-Timing Attacks

Section 2.2 gives the definition of security against collision-timing attacks for the case of general hash table implementation. We consider now the case where the hash table follows a linear probing approach.

When using linear probing, the execution time of a find, insert, or delete operation of the hash table depends on two factors, (1) whether the input item collides with an item that is already in the hash table and (2) on the eviction strategy in case there is a collision. The work of Lipton *et al.* [23] addresses the first timing factor for hash tables with chaining as well as for hash tables with open addressing. The authors propose attacking strategies that only use the execution time to discover if two given items collide in a hash table as well as a method to generate a pair of colliding items for a given hash table. In the same spirit, the adversary of PRV-CTA chooses two items that hash to the same cell. In case the items were allowed to hash to different cells, then it would be trivial for the adversary to win game PRV-CTA. Specifically, it would be enough for the adversary to pick a memory representation in which u_0 hashes to an empty cell while u_1 hashes to the beginning of a long cluster of consecutive items. Thus we turn our attention to the second timing factor, that is on the eviction policy.

Due to the nature of linear probing, the insertion/deletion process will probe the same cells regardless of whether u_0 or u_1 is chosen in PRV-CTA. Therefore, what can potentially make a hash table insecure with respect to PRV-CTA is the eviction policy. Hash tables that are secure according to Definition 3 should follow an eviction policy that is *oblivious to the value of the items*. The SHI linear probing scheme proposed in [7] is *not* secure against collision-timing attacks since a priority function that takes the values of the two items as an input (See [29] for a thorough treatment of the subject) is necessary to decide which item to evict.

We note here that the definition of security against collision-timing attacks is formed under a semantically-secure framework, thus we use the notion of a cryptographic PRNG G with a given security parameter λ in our analysis.

Theorem 2. *Let WHI be the implementation of a hash table where the insertion and deletion methods follow Algorithms 1, 2 and 3. If G is a pseudorandom number generator then the implementation WHI is secure against collision-timing attacks according to Definition 3.*

3.4 Analysis of Individual Displacement

In a hash table with linear probing, the *individual displacement* of an item is the distance between (a) the location where the item hashes to and (b) the location where the item is placed.

In this section, we derive the asymptotic performance of the individual displacement in the case of our WHI linear probing variation. In case the individual displacement is large the algorithm find has to scan a large number of cells in order to locate the requested item which slows down the performance of the operation. We note that the techniques we use are standard but the analysis appears novel. In particular, we focus on the distribution of the individual displacement in the case $n/m \to \alpha$ for $0 < \alpha < 1$. We note that the total displacement is independent of the insertion policy (as long as the policy falls within the standard class of policies; see e.g. [20]); hence the average displacement is the same for all policies, but the distribution of displacements is not. Our starting point is the Eq. (1): $p_{i+1} = h_{i+1} + max(h_i - 1, 0)$. Given a table with load α, it is helpful to start by obtaining a distribution on the entries of the profile. That is, let η_k be the fraction of bins with a count of k in the asymptotic regime as the number of bins grows to infinity. Thus, we have $\eta_k = \#\{i : p_i = k\}/m$.

Asymptotically, we can use the standard Poissonization approach of letting the h_i be distributed as independent Poisson random variables with parameter α. In this case, the p_i form a simple Markov chain, and we can consider its stationary distribution; this gives us the asymptotic distribution for η_k. In particular, a cell i has $p_i = 0$ only if the previous cell has $p_{i-1} = 0$ or $p_{i-1} = 1$ and no items hash to i, hence, $\eta_0 = (\eta_0 + \eta_1)\Pr(h_i = 0) \Rightarrow \eta_0 = (\eta_0 + \eta_1)e^{-\alpha}$.

More generally, for $k \geq 1$, we find

$$\eta_{k+1} = \eta_k \frac{1 - e^{-\alpha}\alpha}{e^{-\alpha}} - \eta_0 \frac{\alpha^k}{k!} - \sum_{l=1}^{k-1} \eta_l \frac{\alpha^{k-l+1}}{(k-l+1)!}. \tag{2}$$

From these equations, we can numerically derive the distribution using the fact that $\sum_{k=0}^{n} \eta_k = 1$. For the individual displacement, we work with a random item u, and compute the limiting distribution of its displacement. Let X_q be a random variable that takes value 0 with probability $1/q$ and value 1 otherwise. Let also Z_k be the random variable that takes as a value the displacement of u given that $p_{\mathsf{hash}(u)} = k$. For a given $k \geq 1$ we have the following recursive function:

$$Z_k = X_k \left(1 + \sum_{j=0}^{n-k+1} Z_{k-1+j} e^{-\alpha} \frac{\alpha^j}{j!} \right). \tag{3}$$

That is, with probability $1/k$ there is no displacement because u is stored in the cell it hashed to; otherwise, 1 is added to the displacement and it moves to the next cell, which has Z_{k-1+j} items that probed it, where j is the number of items that hash to that cell.

As Z_k is conditioned on the event that for cell $\mathsf{hash}(u)$ we have that $p_{\mathsf{hash}(u)} = k$, the displacement of u is a random variable D_u given by:

$$D_u = \sum_{k=0}^{n-1} Z_{k+1}\eta_k \tag{4}$$

Using Eqs. (2) and (3) one can numerically derive the distribution of the individual displacement of u for a fixed α. In Sect. 5 we show the sample variance of the individual displacement for various load factors and compare it to the variance of other linear probing schemes.

4 Previous Linear Probing Schemes

The standard linear probing scheme is a first-come-first-served (FCFS) policy, since previously allocated items do not move during an insertion. Poblete and Munro [30] propose a last-come-first-served (LCFS) policy, where an incoming item has higher priority than those previously allocated. These two policies are easily seen not to be weakly history-independent, since the resulting memory representation clearly depends on the order of updates.

Consider, instead, an alternative scheme inspired by the well-known Fisher-Yates random shuffling algorithm. That is, in the case of collision occurring in the insertion of an item, u, place the item u in the first available cell found by linear probing under the FCFS rule. Then swap u with a uniformly randomly chosen item from range of cells $T[\mathsf{hash}(u)]$ to the last cell of the cluster (wrapping around the beginning of T if needed). Let us call this eviction technique, "Random-Swap" it is easy to see that this variation is not weakly history-independent either.

In Robin Hood hashing [9], when we probe an occupied cell $T[i]$ during the insertion of an item u, we swap u into $T[i]$ if u is further from its desired cell than the current occupant (and we then probe $T[i+1]$ for the remaining item). Ties occur when u and $T[i]$ are equidistant from their desired cell. Different tie-breaking techniques give different security properties to the resulting construction. If we break ties based on the arrival time ("Robin Hood, Tie-Break w. FCFS/LCFS" in Table 1) then the scheme is secure against collision-timing attacks since the evictions do not depend on the value but it is not history-independent. Suppose, instead, that we break ties fairly, by randomly choosing between the two items with probability 1/2, to split the "arrow" in half, using the Robin Hood metaphor. One might think that this fair-split strategy allows Robin Hood hashing to be weakly history-independent, but that is not the case.

Lemma 5. *FCFS, LCFS, Random-Swap, Robin Hood where ties break with FCFS, Robin Hood where ties break with LCFS, and Robin Hood where ties break with fair-split are (1) secure against collision-timing attacks but (2) not weakly history-independent, even in an insertion-only scenario.*

If we follow a Robin Hood hashing and break ties by choosing the larger/smaller value ("Robin Hood, Tie-Break w. Key Sort" in Table 1, also addressed as "age-rules" in [29]), then the scheme becomes strongly history-independent but it is not secure against collision-timing attacks. Note that one has to design a new appropriate deletion process that respects the SHI property of the above RH variation, in this work we only consider the insertion process of this scheme (see Sect. 5). Finally the strongly history-independent linear probing technique from [7] is also not secure against collision-timing attacks.

Lemma 6. *Robin Hood hashing where ties break with key-sort as well as the SHI scheme of [7] are (1) strongly history-independent but (2) not secure against collision-timing attacks.*

The above observations are summarized in Table 1. The proposed linear probing scheme of this work is the first that satisfies both privacy properties.

5 Evaluation

We have implemented the strongly history independent linear probing scheme of [7], denoted as SHI, and our WHI linear probing scheme in C++ and conducted experiments to compare the performance of the two history independent techniques. The values in the (key, value) pairs consist of 10-character strings. All experiments were performed in the same machine running OSX 10.10.5 with Quad Core 2.6 GHz Intel Core i7 processor and 8 GB RAM. We used Intel's on-chip hardware random generator, instruction RdRand, that is available in "Ivy Bridge" processors [19].

Method find. The first set of experiments, depicted in Fig. 2, addresses the performance of method find. We focus on the displacement, i.e., the distance of an item from its hashed location, which affects the performance of find. For completeness, we also show the results for Robin-Hood hashing, which is known to minimize the variance of the displacements among all linear probing algorithms. We use a table of size $m = 10^7$ and we initialize the data structures

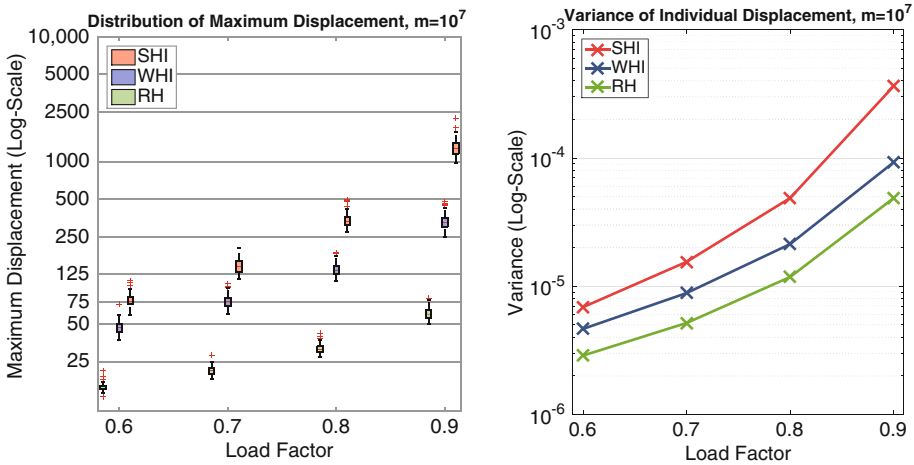

Fig. 2. Comparison of the displacement of items in Robin Hood (where ties-break with FCFS), SHI [7] and WHI (our scheme). The plot summarize results from experiments run 100 times on a table of size $m = 10^7$ with varying load factors: (a) maximum displacement; (b) variance of individual displacement.

up to load factors $\alpha = 0.6, 0.7, 0.8, 0.9$ by inserting the same set of unique randomly generated items in the same randomly chosen order to all hash tables. After the initialization, we record the displacement of each item and compute the sample variance and the maximum displacement. The above process was repeated for 100 distinct initializations. The box-plot of the maximum displacement, Fig. 2(a), shows that the average maximum displacement of WHI is much smaller than that of SHI. As another data point, in a similar experiment with a table of size $m = 10^7$, for load $\alpha = 0.9$ the maximum recorded displacement for WHI is 476 whereas for SHI it is 2228. Finally, in the plot of Fig. 2(b) shows that the variance of the individual displacement for WHI is much lower than that for SHI.

Methods insert and delete. The second set of experiments, depicted in Fig. 3, addresses the performance of methods insert and delete. In this experiment we use a table of size $m = 10^5$ and initialize the data structures up to a fixed load factor, i.e. $\alpha = 0.6 - 0.9$ by inserting the same set of unique randomly generated items in the same randomly chosen order. After the initialization we perform 10^3 find calls that take as an input a randomly chosen item among those that are already in the table in order to warm-up the cache. Then we perform an insertion (resp. deletion) of a randomly generated (resp. chosen) item and record the number of cycles executed by the method. We obtain the number of cycles as the difference between processor time stamps by means of instruction rdtsc. The above process

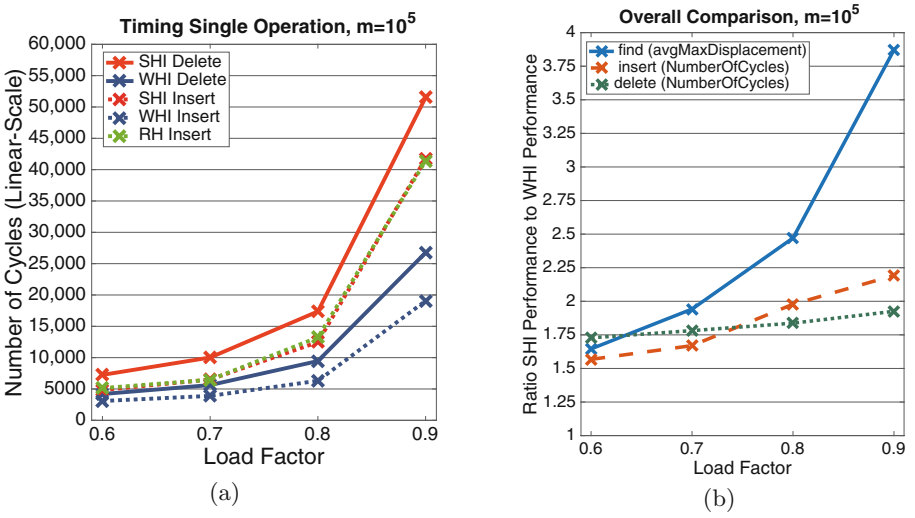

Fig. 3. Comparison of methods insert and delete in Robin Hood (where ties-break with FCFS), SHI [7] and WHI (our scheme). Plot (a) summarizes the experiments conducted 10^3 times on a table of size $m = 10^5$ with varying load factors. In each experiment, we measured the number of CPU cycles executed by a single call of method insert or delete. Plot (b) depicts the ratio between the performance of SHI to WHI.

was repeated 10^3 times for various values of α. Figure 3(a) presents the sample mean of the number of cycles, showing that our WHI scheme is significantly faster than SHI. For comparison, we also include the performance of the variation of Robin Hood linear probing where ties break in a FCFS fashion. This variation gives the fastest insertion process over all Robin Hood variations since we don't move items in case of a tie.

Overall Comparison. As an overall comparison Fig. 3(b) shows the ratio of SHI's performance to WHI's performance for each of the above operations. Each line represents an operation (i.e. find, insert, delete) and each data point of the line represents the ratio of SHI performance (in terms of average maximum displacement or number of cycles) to WHI's performance. It is clear that the average maximum displacement of WHI is significantly smaller compared to SHI, which translates to a faster worst-case time performance for find method. Specifically, the average maximum displacement of an item for the SHI eviction strategy is almost 4 times higher than the average maximum displacement of WHI, for high load factors. As for the update operations, WHI performs almost 2x faster than SHI for high load factors.

6 Conclusion and Discussion

In this paper, we have presented a linear probing hashing scheme that is weakly history-independent and secure against collision-timing attacks. According to our evaluation, all three methods of the proposed hash table (find, insert, delete) are much faster than those of the strongly history-independent analogue proposed by Blelloch and Golovin [7]. Our results suggest that weakly history-independent data structures can be more efficient than strongly history-independent ones in real-world privacy-preserving applications such as ballot storage and hospital admissions management.

Acknowledgments. This work was supported in part by the U.S. National Science Foundation under grants CCF–1535795, CCF–1320231, CNS–1228485, CNS–1228598, and CNS–1228639, and by the Kanellakis Fellowship at Brown University.

References

1. Voluntary Voting System Guidelines, Ver. 1.1, vol. 1. Technical report, United States Election Assistance Commission (2015). www.eac.gov/assets/1/Documents/VVSG.1.1.VOL.1.FINAL.pdf
2. Aslam, J.A., Popa, R.A., Rivest, R.L.: On auditing elections when precincts have different sizes. In: Proceedings of the USENIX EVT (2008)
3. Bajaj, S., Sion, R.: Ficklebase: looking into the future to erase the past. In: Proceedings of 29th IEEE ICDE, pp. 86–97 (2013)
4. Bajaj, S., Sion, R.: HIFS: history independence for file systems. In: Proceedings of 20th ACM CCS, pp. 1285–1296 (2013)

5. Bethea, D., Reiter, M.K.: Data structures with unpredictable timing. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 456–471. Springer, Heidelberg (2009)
6. Bethencourt, J., Boneh, D., Waters, B.: Cryptographic methods for storing ballots on a voting machine. In: Proceedings of 14th NDSS, pp. 209–222 (2007)
7. Blelloch, G.E., Golovin, D.: Strongly history-independent hashing with applications. In Proceedings of 48th IEEE FOCS, pp. 272–282 (2007)
8. Buchbinder, N., Petrank, E.: Lower and upper bounds on obtaining history independence. Inf. Comput. **204**(2), 291–337 (2006)
9. Celis, P., Per-Ake Larson, J., Munro, I.: Robin hood hashing. In: Proceedings of 26th IEEE FOCS, pp. 281–288 (1985)
10. Chen, B., Sion, R.: Hiflash: a history independent flash device. CoRR, abs/1511.05180 (2015)
11. Crosby, S.A., Wallach, D.S.: Denial of service via algorithmic complexity attacks. In: Proceedings of 12th USENIX Security Symposium (2003)
12. Feldman, A.J., Alex Halderman, J., Felten, E.W.: Security analysis of the Diebold AccuVote-TS voting machine. In: Proceedings of the USENIX EVT (2007)
13. Futoransky, A., Saura, D., Waissbein, A.: Timing attacks for recovering private entries from database engines. In: BlackHat USA (2007)
14. Goldreich, O.: The Foundations of Cryptography, Basic Techniques, vol. 1. Cambridge University Press, Cambridge (2001)
15. Goodrich, M.T., Kornaropoulos, E.M., Mitzenmacher, M., Tamassia, R.: More practical and secure history-independent hash tables. Cryptology ePrint Archive, Report 2016/134 (2016). http://eprint.iacr.org/2016/134
16. Goodrich, M.T., Tamassia, R.: Algorithm Design and Applications, 1st edn. Wiley (2014). ISBN:1118335910, 9781118335918
17. Alex Halderman, J., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest we remember: cold boot attacks on encryption keys. In: Proceedings of 17th USENIX Security Symposium, pp. 45–60 (2008)
18. Hartline, J.D., Hong, E.S., Mohr, A.E., Pentney, W.R., Rocke, E.: Characterizing history independent data structures. Algorithmica **42**(1), 57–74 (2005)
19. Hofemeier, G.: Intel Digital Random Number Generator (DRNG) software implementation guide. Technical report (2012)
20. Janson, S.: Individual displacements for linear probing hashing with different insertion policies. ACM Trans. Algorithms **1**, 177–213 (2005)
21. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching, vol. 3, 2nd edn. Pearson (1998)
22. Kohno, T., Stubblefield, A., Rubin, A.D., Wallach, D.S.: Analysis of an electronic voting system. In: Proceedings of 25th IEEE S&P, pp. 27–40 (2004)
23. Lipton, R.J., Naughton, J.F.: Clocked adversaries for hashing. Algorithmica **9**(3), 239–252 (1993)
24. Maartmann-Moe, C.: Inception: a physical memory manipulation and hacking tool exploiting PCI-based DMA
25. Micciancio, D.: Oblivious data structures: applications to cryptography. In: Proceedings of 29th ACM STOC, pp. 456–464 (1997)
26. Molnar, D., Kohno, T., Sastry, N., Wagner, D.: Tamper-evident, history-independent, subliminal-free data structures on PROM storage-or-how to store ballots on a voting machine. In: Proceedings of IEEE S&P, pp. 365–370 (2006)

27. Moran, T., Naor, M., Segev, G.: Deterministic history-independent strategies for storing information on write-once memories. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 303–315. Springer, Heidelberg (2007)
28. Naor, M., Segev, G., Wieder, U.: History-independent cuckoo hashing. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 631–642. Springer, Heidelberg (2008)
29. Naor, M., Teague, V.: Anti-presistence: history independent data structures. In: Proceedings of 33rd ACM STOC, pp. 492–501 (2001)
30. Poblete, P.V., Munro, J.I.: Last-come-first-served hashing. J. Algorithms **10**(2), 228–248 (1989)
31. Rivest, R.L., Shen, E.: A Bayesian method for auditing elections. In: Proceedings of USENIX EVT/WOTE (2012)
32. Vitter, J.S.: Random sampling with a reservoir. ACM Trans. Math. Softw. **11**(1), 37–57 (1985)
33. Wolchok, S., Wustrow, E., Halderman, J.A., Prasad, H.K., Kankipati, A., Sakhamuri, S.K., Yagati, V., Gonggrijp, R.: Security analysis of India's electronic voting machines. In: Proceedings of 17th ACM CCS, pp. 1–14 (2010)