Auditable Data Structures

Michael T. Goodrich
Dept. of Comp. Sci., Univ. of California, Irvine
goodrich@acm.org

Evgenios M. Kornaropoulos
Dept. of Computer Science, Brown University
evgenios@cs.brown.edu

Michael Mitzenmacher
School of Eng. and Applied Sci., Harvard University
michaelm@eecs.harvard.edu

Roberto Tamassia
Dept. of Computer Science, Brown University
rt@cs.brown.edu

***Abstract*—The classic notion of *history-independence* guarantees that if a data structure is ever observed, only its current contents are revealed, not the history of operations that built it. This powerful concept has applications, for example, to e-voting and data retention compliance, where data structure histories should be private. The concept of *weak history-independence* (WHI) assumes only a single observation will ever occur, while *strong history-independence* (SHI) allows for multiple observations at arbitrary times. WHI constructions tend to be fast, but provide no repeatability, while SHI constructions provide unlimited repeatability, but tend to be slow.**

We introduce *auditable data structures*, where an auditor can observe data structures at arbitrary times (as in SHI), but we relax the unrealistic restriction that data structures cannot react to observations, since in most applications of history-independence, data owners know when observations have occurred. We consider two audit scenarios—*secure topology*, where an auditor can observe the contents and pointers of a data structure, and *secure implementation*, where an auditor can observe the memory layout of a data structure. We present a generic template for auditable data structures and, as a foundation for any auditable data structure, an Auditable Memory Manager (AMM), which is an efficient memory manager that translates any auditable data structure with a secure topology into one with a secure implementation. We give a prototype implementation that provides empirical evidence that the worst-case time running times of our AMM are 45× to 8,300× faster than those of a well-known SHI memory manager. Thus, auditable data structures provide a practical way of achieving time efficiency, as in WHI, while allowing for multiple audits, as in SHI.

1. Introduction

An important privacy goal for e-voting is that it should be impossible for an adversary, Eve, to link voters to their ballots, even if she can inspect the voting machine used in an election. Unfortunately, several works [22], [32] have pointed out flaws in the Direct Recording Electronic (DRE) voting machines deployed for national elections where the votes were stored in the same order as they were cast. As a result, in March 2015, the U.S. Election Assistance Committee approved the next generation of Voluntary Voting System Guidelines [3], which require ballot images to be recorded in randomized order by DRE machines. This requirement is fine for a single observation at the end of an election, but in an era of electronic voting for elections that can last for days, this restriction does not go far enough.

For example, an analysis of the Estonian Internet voting system [29] shows that data center employees may perform necessary maintenance and periodic back-ups of the servers where a voting system is deployed. Thus, e-voting systems should protect the history of how votes have been cast across multiple observations of the contents of voting machines.

The sensitive nature of update histories also arises in data retention compliance. For instance, leaving traces of removed data violates retention policies such as the EU Data Protection Directive [1] and HIPAA [2]. Compliance with such retention regulations requires the elimination of any evidence establishing the past existence of removed data. Cloud providers and data centers may face liability issues if one can infer the history of previous operations. Such systems must also support periodic, scheduled data backups.

The challenge with such applications is that the internal organization of a data structure may inadvertently reveal information about its history. For example, an adversary that observes a data structure might be able to infer information about the *order of the operations* or whether data was *inserted and later removed*. Depending on the context, this leakage can be catastrophic for the privacy of the system.

This challenge has motivated the now classic notion of *history-independence* (e.g., see [4], [26]), where a data structure is designed so that if it is ever observed, only its current contents are revealed, not the sequence of operations that resulted in its current state. Researchers have considered two types of history-independence. In *strong history-independence* (SHI) the adversary gets to observe the representations of a data structure, D, as many times as she wants. An additional assumption of SHI is that D doesn't know when adversarial observations occur. Based on this restriction, Harline *et al.* [19] show that any SHI data struc-

ture must be *canonical*, meaning that each possible content set must have a unique representation, which unfortunately implies that SHI constructions tend to be slow [10], [16].

In *weak history-independence* (WHI), the privacy goal is the same as in SHI but the adversary is allowed to view the representation for D *only once*. This restriction means that WHI data structures need not be canonical, which typically allows for faster performance, but it also means that a WHI data structure loses all of its history-independence after that one observation. Thus, WHI data structures are not suitable for scenarios involving periodic observations. Therefore, SHI data structures tend to be slow, but allow for repeated observations, while WHI data structures tend to be fast, but can only be observed once.

In this paper, we introduce the notion of *auditable data structures*, where an auditor gets to observe a data structure at multiple, arbitrary times, as in the SHI framework, but we relax the SHI restriction that data structures are not allowed to react to having been observed. Formally, we assume that a data structure is notified immediately after it is observed by the auditor and it is allowed to react to this notification. Under this framework, WHI can be seen as a special case of auditable data structures, which we call "one-time auditable." The significance of our formulation, however, is that it allows for *more efficient* constructions of data structures than their SHI counterparts while providing *stronger privacy* guarantees than their WHI versions, since we allow for multiple audits. We demonstrate the efficiency of our framework by providing a generic template for auditable data structures and by designing and implementing an auditable memory manager (AMM) that outperforms a well-known SHI memory manager by several orders of magnitude.

Related Work. Work on history-independence (HI) can be traced back to Micciancio [23], who introduced an *obliviousness* property that requires that the data and link structure of a search tree yield no information regarding the sequence of operations that produced the tree. (By the way, this "obliviousness" property is completely different than the similarly-named oblivious RAM [12], [17], [18], [30] and oblivious data structures [31], which are orthogonal to the concepts of history-independence and auditable data structures.[1]) Naor and Teague [26] show that, in addition to a data structure's data and link structure, the memory allocation of a data structure can also leak information about its history. They introduced the concept of *history-independence*, including both the weak and strong variants, WHI and SHI. Subsequent to these pioneering papers, work has been done on efficient SHI results, including lower bounds for SHI queues by Buchbinder and Petrank [10], a result on SHI Cuckoo Tables by Naor *et al.* [25], and work by Blelloch and Golovin [9] on a SHI hash table with linear-probing. With respect to WHI results, Goodrich *et al.* [16] present a WHI hash table with linear probing that is faster in practice than its SHI analogue. Bender *et al.* [7] propose a WHI external-memory skip-list as well as a WHI cache-oblivious B-tree. Hartline *et al.* [19] show that satisfying the SHI definition requires a data structure to have a *canonical memory representation*, which implies practical inefficiencies. Such inefficiencies, therefore, are a necessary by-product of SHI data structures, such as those of Golovin [14], [15], who constructs complex SHI data structures, such as B-SkipLists and B-Treaps, by using a SHI memory manager.

History-independence can also play a role in the design of privacy-preserving systems. For instance, Bajaj and Sion give a history-independent file system, HIFS [6], that provides HI across file system and disk layers, and Ficklebase [5], which is a database that uses SHI data structures for the underlying database storage engine (to avoid forensic recovery of deleted information). Chen and Sion propose HI schemes tailored for flash-based devices [11]. Bethencourt *et al.* [8] propose a vote storage system that uses History Hiding Append-Only Signatures, a concept inspired by HI. Other systems [27], [28] use HI building blocks to strengthen the overall privacy of their construction.

Our Contributions.

- We introduce the notion of *auditable data structures*, where a passive auditor performs audits unexpectedly and at arbitrary times. The privacy goals are the same as in history-independence but the data structure is allowed to react after an audit has occurred. We also present a generic template for building auditable data structures and we introduce the concept of *one-time auditable data structures*, which is privacy-equivalent to weak history-independence.
- We present AMM, a memory manager that translates any auditable data structure with a secure topology into an auditable data structure with a secure implementation. The design of AMM is such that the time complexity is *decoupled* from the number of elements, n, stored in the data structure. Specifically, the time complexity is a function of the number of operations, S, that were executed since the latest audit.
- To build AMM, we propose a series of new one-time auditable building blocks which are of independent interest. In particular, we introduce a resizable memory management technique for one-time auditable data structures, called ROTA, that has update time $O(1)$ in the worst case, whereas the best proposed WHI memory management technique, presented in [26], has $\Omega(n)$ worst-case time due to resizing. We apply the core idea behind ROTA to develop a new one-time auditable dynamic hash table via chaining with complete binary trees.
- Based on our experiments, our prototype for AMM has worst-case running times that are 45× to 8,300× faster than the SHI memory manager used in other works.

1. In these other "oblivious" models [12], [17], [18], [30], [31], one is trying to hide the complete state of an out-sourced memory (e.g., through encryption and obfuscation) while fixing its size and its number of operations, whereas in HI and auditable data structures, we don't fix the memory size nor the number of operations and we are trying to hide the order of operations and the existence of operations that cancel each other (like an insert immediately followed by a delete), while still allowing the memory state to be revealed to an auditor.

2. Preliminaries

Let us begin by introducing some general terminology, as done in previous work on history independence (e.g., see [9], [19], [26]). An *Abstract Data Type* (ADT) is a model of a data structure describing the type of data stored, the operations that can be performed, and the parameters for each operation. A data structure is associated with a set of its currently stored elements, which defines the *state* of an instance of an ADT. Following the footsteps of the original work on history-independence [9] we formalize our framework on dictionaries where the state is a set of elements. Dynamic data structures typically have ADTs that include an insertion operation for adding elements and a deletion operation for removing elements.

Following the approach of Golovin [13], we assume a semantics for ADT operations that allows one to compute the state of an ADT instance from the outputs of ADT operations. That is, we assume a semantics such that each ADT operation maps an input state and an input (which may be null) to a next state and an output (which may be null).

In the following, we provide formal definitions of basic concepts such as pointer structure and memory representation of a data structure. A *record* of a data structure is a tuple consisting of (1) an element and (2) *auxiliary data* stored by the data structure for internal use, such as pointers, flags, sizes, and weights. For example, in a singly linked list, the auxiliary data of a record is the `next` pointer.

Definition 1. *The* pointer structure *of a data structure is the directed graph such that the vertices are associated with the records and there is an edge from a record r' to a record r'' when the auxiliary data of r' includes a pointer to r''.*

In case the data structure is not pointer-based (e.g., a linear-probing hash table), then the pointer structure is an graph without edges. The pointer structure captures a high-level abstraction that is indifferent to the memory addresses. Thus, the pointers at this layer of abstraction carry no additional information beyond the pairs of records linked, as illustrated in Figure 1. Using this layered approach, one can prove the security of the topology of the pointer structure before proceeding to the more challenging case where the memory addresses are visible.

Definition 2. *A* memory representation *of an ADT, or simply* representation*, is a mapping of the state to the memory used by an ADT instance and is described as a collection of pairs,*

$$(address, record). \tag{1}$$

We call *memory segment* a range of memory addresses consisting of consecutively stored records.

In a memory representation, a pointer consists of the memory address of the record it is pointing to, see the bottom part of Figure 1. From a memory representation, we can reconstruct both the state and pointer structure of a data structure. In general, there can be multiple pointer structures for a given state and multiple memory representations for a given pointer structure, as illustrated in Figure 1.

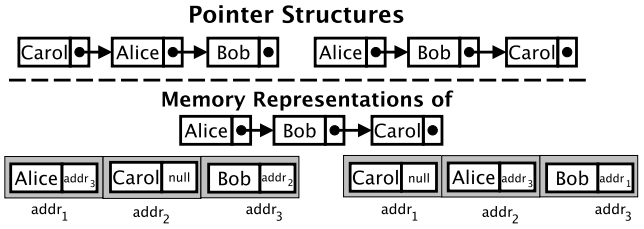


Figure 1. Top: two different pointer structures for a linked list with state {Alice, Bob, Carol}. Bottom: two different memory representations for the same state and pointer structure of a linked list.

An *implementation* of a data structure is a function, $F : R \times O \rightarrow R$, where R is the set of all possible memory representations and O is the set of all possible ADT operations. A *sequence of operations* S is an ordered list of ADT operations of the data structure as defined by the corresponding ADT.

Following terminology of Hartline *et al.* [19], let a and b denote memory representations of states A and B, respectively. Let S be a sequence of operations. The notation $A \xrightarrow{S} B$ indicates that S takes the data structure from state A to state B. Let $\Pr[a \xrightarrow{S} b]$ denote the probability that starting from representation a of state A, the sequence of operations S run by the corresponding implementation yields representation b of state B. The memory representation of an initialized but empty data structure is denoted by $\oslash$.

Definition 3. *[19] Let A be a state of the data structure. A data structure implementation 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 representations 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 4. *[19] Let A, B be two distinct states of the data structure. A data structure implementation is* strongly history-independent *if for any two (possibly empty) sequences of operations S_1 and S_2 that take the data structure from state A to state B, the distribution over the memory representations of B after S_1 is performed on the memory representation a is identical to the distribution after S_2 is performed on the memory representation a. That is:*

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

Hartline *et al.* [19] show that strong history-independence requires that there be a unique memory representation for each state, called the *canonical representation* of the state. Thus, after every operation, a SHI data structure implementation has to switch to the canonical representation for the resulting state.

3. Auditable Data Structures

In this section, we introduce the model of *auditable data structures* where the data structure is subject to observations

by an honest-but-curious auditor. The data structure is obliged to pass to the auditor the requested information but it should not reveal more than it must. The auditor acquires snapshots of the data structure and based on these observations he is attempting to infer the sequence of operations that were executed between two consecutive audits. In our model we make no assumptions about the frequency of the audits which also happen unexpectedly. As a consequence the auditee has time to react *only after* an audit took place. This setup is closer to how audits take place in the real world where the auditee shouldn't have time to discard any critical information before the audit.

Application Scenario. Our adversarial model is inspired by scenarios where an observer is allowed to take snapshots of the memory, but these snapshots are known to occur. The adversary that we consider is not an intruder that secretly obtained illegal access of the memory. Thus we do not rely on intrusion detection mechanisms to discover reads of memory. As described before, our framework can be applied to scenarios where an entity (e.g., a system administrator, auditor, another client in a collaboration) can inspect the memory of the server that runs the privacy-preserving service. A natural motivating example is an electronic voting system that must be audited.

The Model. In the auditable data structure model, the data structure executes a sequence of ADT operations *in private* until the auditor requests to perform an observation at an arbitrary time. Specifically, we consider two main scenarios: (1) the auditor can only observe the pointer structure; and (2) the auditor can observe both the pointer structure and the memory representation. The auditor in the second scenario is more powerful since the pointer structure can be inferred from the representation of the data structure in memory. After the observation, the data structure executes method Postaudit() and resumes executing ADT operations in private until the next audit.

Definition 5. *An* auditable data structure *is a data structure whose ADT is augmented with operation Postaudit(), which is executed after every observation by the auditor.*

We emphasize that when the audit is requested, no action can be taken by the data structure other than handing the relevant information (either the pointer structure or the memory representation) to the auditor. Therefore, the auditable data structure model can be viewed as a form of reactive history-independence since the data structure is *allowed to react* to the observation via operation Postaudit().

The desired privacy property for an auditable data structure (formally defined in this section) is that the pointer structure or the memory representation at the time of the audit should not reveal any information about the sequence of operations executed since the last audit.

The underlying system. There are works on history-independent data structures that assume for the sake of simplicity that the memory is a one-dimensional array that can *always* be extended at one end (Section 4 in [26]). Even though this assumption helps to seamlessly and efficiently resize the memory of the data structure it is unfortunately not a realistic one. We want to bring our auditable data structure model closer to how a real system operates.

We assume that there is a memory allocator that is handled by the underlying operating system and we differentiate between a memory allocator and a memory manager. The *memory allocator* (see, e.g., [20] for some standard approaches) assigns/releases a segment of contiguous memory to *all* applications. Notice though that distinct memory requests to the memory allocator do not necessarily return contiguous memory segments, as opposed to the previous HI model assumption. We further assume that no inferences can be made by the relative order of the address of the segments that the memory allocator returns; to prevent potential leakage from this source one has to re-design memory allocators without sacrificing their highly optimized time efficiency as their performance is crucial for modern systems.

The memory manager that we consider, indicated as $\mathcal{M}$, is an *intermediate layer* between the data structure and the memory allocator. The memory manager can only handle memory segments acquired via the memory allocator and it ensures that the allocation of the data structure in those segments does not leak the history.

To store the data structure $\mathcal{M}$ maintains the *image of* $\mathcal{M}$, denoted by I_M, which is defined as (1) the memory representation and (2) the *auxiliary space* that is used for internal structures of $\mathcal{M}$. The construction presented in Section 5 utilizes the auxiliary space by storing the component DictionaryObserved that is discussed later. For the sake of abstraction, we assume that the memory allocator implements the following two functions, which are invoked by the memory manager $\mathcal{M}$ of our model: (1) getMem(k) returns the starting address addr of a segment of k consecutive zeroed cells made available to $\mathcal{M}$; (2) freeMem(addr,k) releases k memory cells starting at address addr and makes them no longer available to $\mathcal{M}$. For the asymptotic analysis of the memory manager, we assume that these low-level system memory operations getMem and freeMem take constant time. A desirable property of a memory manager is to *efficiently resize* its image I_M, that is, the size of the image should be proportional to the space used by the data structure.

Entities. The security definition of an auditable data structure involves the following entities:

- **Data Structure** $\mathcal{D}$: accesses memory by calling memory operations (read/write, store/free) supported by the memory manager, $\mathcal{M}$. $\mathcal{D}$ may have to issue multiple memory operations in order to execute a single ADT operation. The type of data structure (e.g., list, tree) is described by a public parameter, P.
- **Memory Manager** $\mathcal{M}$: executes memory operations issued by $\mathcal{D}$ by making available to $\mathcal{D}$ a collection of memory segments. Besides reads and writes, $\mathcal{D}$ can ask $\mathcal{M}$ to allocate a new memory segment (Store) or to release an existing memory segment or portion of it (Free). $\mathcal{M}$ uses in turn low-level memory allocation primitives getMem and freeMem.
- **Auditor** $\mathcal{A}$: performs an audit at arbitrary times,

during which he observes either the pointer structure of $\mathcal{D}$ or the memory managed by $\mathcal{M}$. $\mathcal{A}$ is a passive adversary that tries to infer information about the sequence of ADT operations issued by the user on $\mathcal{D}$.

The ADT of an auditable data structure uses operation Postaudit to react to the audit, which handles, as we see later, the security of the pointer structure. To maintain the security of the memory representation of $\mathcal{D}$ we equip the memory manager $\mathcal{M}$ with an analogous Postaudit functionality.

In a memory manager without our privacy goals, the Store and Free operations could directly call the standard malloc() and free() methods of stdlib. The above standard allocation technique fails to meet our privacy goals since if a record is observed, then deleted, and then reinserted it would most likely appear in a different memory location during the next audit which leaks that the record was removed at some point. Thus, our memory manager $\mathcal{M}$ must apply more robust audit-aware strategies for handling the memory of $\mathcal{D}$.

Algorithms of $\mathcal{M}$. The memory of the system is treated as a global array for which the atomic unit of storage is a memory cell identified by a unique address. As addr we denote the starting address of a record and as record we denote a record of the data structure (to be) stored in a memory segment starting at addr. We indicate with λ the security parameter and with r an ephemeral source of randomness available to the memory manager, such as the hardware random number generators in modern microprocessors. The interface of the memory manager $\mathcal{M}$ consists of the following algorithms that may modify the image I_M of $\mathcal{M}$.

1) Initialize(1^λ): initializes the memory manager. Performed only once throughout the lifetime of $\mathcal{M}$.
2) Read(addr, r): returns the record at addr.
3) Write(addr, record, r): stores record at address addr.
4) Store(record, r): assigns a memory segment from I_M to record, stores it, and returns its starting address addr.
5) Free(addr, r): removes the record at addr from its assigned memory segment.
6) Postaudit( ): executes maintenance operations to react to an audit.

Note that there is no explicit Audit operation in the above interface, which gets implicitly notified of an audit by the invocation of Postaudit(). Executing algorithm Postaudit($\cdot$) gives the memory manager the opportunity to react to the audit. This might involve performing maintenance operations (e.g., cleaning temporary space and auxiliary data structures) but most importantly adapting the memory allocation strategy to *take into account what was observed* by the auditor. For efficiency reasons, algorithm Postaudit should run in $O(n^\epsilon)$ for $\epsilon < 1$, where n is the current size of the state of the data structure. An important feature of our model is that the memory manager $\mathcal{M}$ can not store any information outside I_M. Thus there is *no private space* from which $\mathcal{M}$ can hide information from an audit.

Categorization of Records. From the auditor's perspective, the records that are allocated in memory during the i-th audit can be categorized as either "observed records" or "new records" based on the $(i-1)$-th audit. Let $I_M^{(i)}$ be the image that the auditor acquires during the current audit i and let $I_M^{(i-1)}$ be the image acquired in the previous audit $i-1$. A *new record* with respect to $I_M^{(i)}$ is a record that is part of the state of $\mathcal{D}$ at $I_M^{(i)}$, but was not part of the state at $I_M^{(i-1)}$. An *observed record* with respect to $I_M^{(i)}$ is a record that is part of the state of $\mathcal{D}$ both at $I_M^{(i)}$ and $I_M^{(i-1)}$. We emphasize here that *only* the records that were part of the state at $I_M^{(i-1)}$ are considered observed at $I_M^{(i)}$. The records categorization is a new characteristic of the auditable data structure model that allows efficient constructions as we show in Section 5.

3.1. Security Definitions

We call *session* the timeframe between two consecutive audits. To simplify the notation, we refer to the algorithms Read, Write, Store, Free as *memory operations* and denote them as (OpM, arg) where OpM$\in$ {Read, Write, Store, Free} and arg is the sequence of the corresponding arguments. With the term n we refer to the size of the state of $\mathcal{D}$.

Formally, we define as *session S of length l for a data structure $\mathcal{D}$* the ordered list of ADT operations of $\mathcal{D}$, $S = \big((\mathsf{OpM}_0, arg_0), \ldots, (\mathsf{OpM}_{l-1}, arg_{l-1})\big)$. The pair of sessions ($S_0$, S_1) is used in our game-based definitions. The operations of $\mathcal{D}$ translate a session S_b of ADT operations into a sequence of memory operations that we denote as $\mathcal{S}_b$. We denote with $|\mathcal{S}_b|$ the length of the sequence of memory operations $\mathcal{S}_b$. For example, an InsertAfter operation from the ADT of a simple linked list translates into a sequence of Read, Write, Store memory operations. For the memory operation (OpM_u, arg_u) of $\mathcal{S}_b$ holds that OpM_u $\in$ {Store, Free, Read, Write}, for all $u \in [|\mathcal{S}_b|]$. Given a state A of a data structure we call the pair of sessions (S_0, S_1) *proper* if the state of the data structure resulting from applying session S_0 to $\mathcal{D}$ initialized to state A, is the same as the state of the data structure resulting from applying session S_1 to $\mathcal{D}$ initialized to state A.

We use a game-based definitions to describe the adaptive security of our setup, which is similar in spirit to a security against a "Chosen Plaintext Attack" in cryptography. The terms auditor and adversary are used interchangeably in the rest of the work. Essentially, the auditor shouldn't be able to distinguish which session among a proper pair (S_0, S_1) has been executed. In Figures 2 and 3, we show the two indistinguishability games played by the adversary $\mathcal{A}$ against $\mathcal{D}$ and $(\mathcal{M}, \mathcal{D})$, respectively. In game PRV-CSA-PS, the adversary observes the pointer structure whereas in game PRV-CSA-M, the adversary observes the memory representation (and hence also the pointer structure). If the game outputs True, the adversary $\mathcal{A}$ wins. In the two games,"CSA" stands for "Chosen Session Attack" referring to the adaptively chosen sessions of the adversary $\mathcal{A}$.

PRV-CSA-PS$^{\mathcal{A}}_{\mathcal{D},\,P}(\lambda, k)$

1. $\mathcal{D}$ is initialized according to 1^λ
2. $\mathcal{D}$ picks $b \xleftarrow{\$} \{0,1\}$
3. **For** $i = 1, \ldots, k$
4. $(S_0,\, S_1) \leftarrow \mathcal{A}(P)$
5. **if** $(S_0,\, S_1)$ is not proper **return** False
6. $\mathcal{D}$ executes the ADT operations of S_b
7. $\mathcal{A}$ is given the current pointer structure
8. $\mathcal{D}$.Postaudit()
9. **end**
10. $\mathcal{A}$ outputs a bit b'
11. Return $(b' = b)$

Figure 2. In game PRV-CSA-PS, the auditor observes the pointer structure.

For the first game we denote as advantage of $\mathcal{A}$ the quantity, $2 \cdot \Pr[\text{PRV-CSA-PS}^{\mathcal{A}}_{\mathcal{D},P}(\lambda, k) = 1] - 1$. Similarly, for the second game we denote as advantage of $\mathcal{A}$ the quantity, $2 \cdot \Pr[\text{PRV-CSA-M}^{\mathcal{A}}_{\mathcal{D},\,\mathcal{M},P}(\lambda, k) = 1] - 1$. In the following security definition, the PPT adversary $\mathcal{A}$ is allowed to request a polynomial number k of sessions and there is no assumption about the length of the chosen sessions.

Definition 6. *Let λ be the security parameter and let $\mathcal{D}$ be an auditable data structure whose type is indicated by parameter P. We say that the topology of $\mathcal{D}$ is* secure *if for all $k = \mathsf{poly}(\lambda)$ and for all PPT adversaries $\mathcal{A}$, the advantage of adversary (auditor) $\mathcal{A}$ in PRV-CSA-PS$^{\mathcal{A}}_{\mathcal{D},\,P}(\lambda, k)$ is negligible.*

PRV-CSA-M$^{\mathcal{A}}_{\mathcal{D},\,\mathcal{M},\,P}(\lambda, k)$

1a. $\mathcal{D}$ is initialized according to 1^λ
1.b $(I_M) \leftarrow \mathcal{M}$.Initialize$(1^\lambda)$
2. $\mathcal{M}$ picks $b \xleftarrow{\$} \{0,1\}$
3. **For** $i = 1, \ldots, k$
4. $(S_0,\, S_1) \leftarrow \mathcal{A}(P)$
5. **if** $(S_0,\, S_1)$ is not proper **return** False
6. $\mathcal{D}$ executes the ADT operations of S_b and translates them into to a sequence of memory operations, $\mathcal{S}_b$, executed by $\mathcal{M}$
7. $\mathcal{A}$ is given the current image, I_M, from which it further derives the current pointer structure
8a. $\mathcal{D}$.Postaudit()
8b. $\mathcal{M}$.Postaudit()
9. **end**
10. $\mathcal{A}$ outputs a bit b'
11. Return $(b' = b)$

Figure 3. In game PRV-CSA-M, the auditor observes the memory representation.

Definition 7. *Let λ be the security parameter and let $\mathcal{D}$ be an auditable data structure whose type is indicated by parameter P. We say that the implementation of $\mathcal{D}$ is* secure *if for all $k = \mathsf{poly}(\lambda)$ and for all PPT adversaries $\mathcal{A}$, the advantage of adversary (auditor) $\mathcal{A}$ in PRV-CSA-M$^{\mathcal{A}}_{\mathcal{D},\,\mathcal{M},\,P}(\lambda, k)$ is negligible.*

Lemma 1. *If the implementation of an auditable data structure is secure then its topology is also secure.*

The above lemma states what we noted previously: secure implementation implies secure topology. Secure topology does not imply secure implementation; given a secure topology one needs to construct a memory manager $\mathcal{M}$ that extends the security to the level of the memory representation. We propose such a memory manager in Section 5.

A weaker notion of security comes from allowing only one audit, e.g. $k = 1$. In this case, we have the following notion of *one-time secure* auditable data structure which can be shown to be equivalent to the notion of weak history-independence. We give the definition for one-time secure implementation, the definition for one-time secure topology can be formed similarly by using PRV-CSA-PS.

Definition 8. *Let λ be the security parameter and let $\mathcal{D}$ be an auditable data structure whose type is indicated by parameter P. We say that the implementation of $\mathcal{D}$ is* one-time secure *if for all PPT adversaries $\mathcal{A}$, the advantage of adversary (auditor) $\mathcal{A}$ in PRV-CSA-M$^{\mathcal{A}}_{\mathcal{D},\,\mathcal{M},\,P}(\lambda, 1)$ is negligible.*

3.2. A Template for Auditable Data Structures

In this subsection, we provide a generic template for constructing auditable data structures with secure topology, i.e., with auditable pointer-structure. As with any non-trivial auditable data structure, the template we describe here depends on the existence of an auditable memory manager, such as the one we propose in Section 5, which translates an auditable data structure with secure topology to a data structure with secure implementation. Our generic template is a general proof-of-concept theorem showing that we can convert any one-time secure data structure into an auditable data structure. This should not be viewed as necessarily being the best way to implement an auditable version for any given ADT, however, which is an interesting line of work we leave for the future.

Theorem 1. *Given a one-time secure data structure, $\mathcal{D}$, implementing a given ADT, one can construct an auditable data structure with secure topology, $\mathcal{D}'$, implementing the same ADT as $\mathcal{D}$.*

Proof. One way to achieve this conversion is by "checkpointing" the one-time secure version, $\mathcal{D}$, after every audit. If $\mathcal{D}_i$ is the current version and it gets audited, then we stop making changes to $\mathcal{D}_i$ and we begin a new one-time secure instance, $\mathcal{D}_{i+1}$. Once an instance, $\mathcal{D}_i$, is checkpointed, we never update it again, and we store it forever. So $\mathcal{D}_1$ goes up to the first audit, $\mathcal{D}_2$ goes up to the second audit, and so on. Elements can be either *live*, if they are part of the state of the current data structure, or *dead* if they are not.

To perform an operation delete(u), we first check if u is "live" in the current version, $\mathcal{D}_i$. If it is, then we simply delete u from $\mathcal{D}_i$. Otherwise, we confirm that u exists in a checkpointed version and we perform a lazy deletion by executing insert$(u,$ dead$)$ in the current version, $\mathcal{D}_i$, to mark

u as dead. To perform an insert(u) operation, we first check if u is in the current version as a dead element, in which case we delete the dead version from $\mathcal{D}_i$. If not, then we insert u in the current version by performing insert(u, live). To perform a find(u), we search the checkpointed versions backwards starting from $\mathcal{D}_i$ until we find u. If u is currently dead, we return "not found", otherwise we return u. □

With the above generic template, we get an auditable data structure with secure topology with search and update times that are $O(kT(n))$, where $T(n)$ is the running time of the one-time secure version of $\mathcal{D}$ and k is the number of audits since its construction. This is admittedly a general framework that might not result in the most efficient auditable data structure for every ADT. Thus, we leave as interesting future work possible constructions for more sophisticated dynamic checkpointing techniques, and we focus in the remainder of this work on the foundation of all auditable data structures, which is the memory manager.

3.3. Comparison with History-Independence

To the best of our knowledge, this work is the first that provides a formal security model for an auditor who can view a data structure multiple times while allowing the data structure to react to observations. The security of the implementation of an auditable data structure for $k > 1$ is weaker than the strong history-independence property but stronger than the weak history-independence property which occurs in our model when $k = 1$.

Golovin [13], [14] builds SHI data structures by using the SHI dictionary of Blelloch and Golovin [9] as a SHI memory manager where the keys correspond to unique memory locations. We refer to this memory manager as SHIMM and we compare its performance to our construction in Section 6. Note that a SHI memory manager like SHIMM can be used as the memory manager $\mathcal{M}$ of an auditable data structure, since it achieves a secure implementation according to Definition 7. Note that in this case, operation Postaudit() does not provide any functionality since a SHI memory manager is not adapting to the fact that an audit was performed. SHIMM must deploy *resizing techniques* that are also history-independent, as the one proposed in [19]. The downside is that in the worst-case the above SHI resizing techniques take *linear time* to the size of the data structure a problem that is more obvious in the performance of SHIMM that we report in Section 6.

Lemma 2. *Let $\mathcal{D}$ be an auditable data structure with secure topology. Then a SHI memory manager $\mathcal{M}$ where Postaudit takes no action yields a secure implementation of $\mathcal{D}$.*

On the flip side, if we deploy an auditable memory manager $\mathcal{M}$ for a SHI data structure $\mathcal{D}$ (e.g. [14], [23], [28]), then the result is an auditable data structure $\mathcal{D}$ with secure implementation. In other words, the auditable memory manager changes the security of $\mathcal{D}$ from SHI to auditable. Thus, previous SHI data structures $\mathcal{D}$ with canonical pointer structure can also benefit from efficient auditable memory managers.

Lemma 3. *Let $\mathcal{D}$ be a data structure with strongly history-independent pointer structure [23]. Let $\mathcal{M}$ be a memory manager that translates an auditable data structure with secure topology to a data structure with secure implementation. Then the implementation of $\mathcal{D}$ is secure according to Definition 7 if memory manager $\mathcal{M}$ is deployed.*

The next lemma points out that an adversary can break the security of Definition 7 in case we deploy the WHI management technique of [26].

Lemma 4. *Let $\mathcal{D}$ be an auditable data structure with secure topology. Let $\mathcal{M}$ the memory manager for which Store and Free follow the WHI technique (Section 4.1 in [26]) and Postaudit takes no action. Then the implementation of $\mathcal{D}$ that deploys memory manager $\mathcal{M}$ is not secure.*

Proof. (Sketch) A successful attack can be achieved by taking advantage of the fact that the memory manager of Section 4.1 in [26] does not react to an audit. The key observation is that the corresponding memory manager handles new and observed records in the same way. The attack succeeds after two audits, i.e. $i = 2$ in line 3 of PRV-CSA-M$^{\mathcal{A}}_{\mathcal{D},\mathcal{M},P}$. The adversary $\mathcal{A}$ creates session S that brings the data structure to state $A = \{u_1, \ldots, u_n\}$ and outputs (S, S) in line 4 of the game. For the second pair of sessions $\mathcal{A}$ picks a S_0 that removes and re-inserts u_1 from the image of the memory I_M, and an empty S_1. Recall that the adversary knows the address of u_1 from the first image she acquired, thus if $b = 0$ then with high probability over the number of allocated records, record u_1 will be seen at a different address after executing S_0, whereas if $b = 1$, then u_1 is going to be allocated at the same address. □

One can extend the WHI memory manager [26] to meet the security guarantees of Definition 7, albeit in an inefficient way. We call this technique Naive Auditable Memory Manager.

Naive Auditable Memory Manager (NAMM). This memory manager executes the Store and Free algorithms as the WHI construction in [26] and augments the construction with a Postaudit that reshuffles the location of records using standard linear time shuffling techniques such as Fisher-Yates shuffling [21]. In case NAMM is full it migrates its content to a different memory segment, which takes linear time. Even though this naive construction meets the privacy goals it fails to meet the efficiency goals since the time complexity of its Postaudit is proportional to the size of the data structure, n. Instead, the construction in Section 5 has time complexity proportional to the number of operations since the last audit denoted with S, which can be significantly less than n.

4. Building Blocks

In the rest of the paper we introduce and measure the performance of a new memory manager for auditable data structures, namely AMM, that is based on simpler one-time

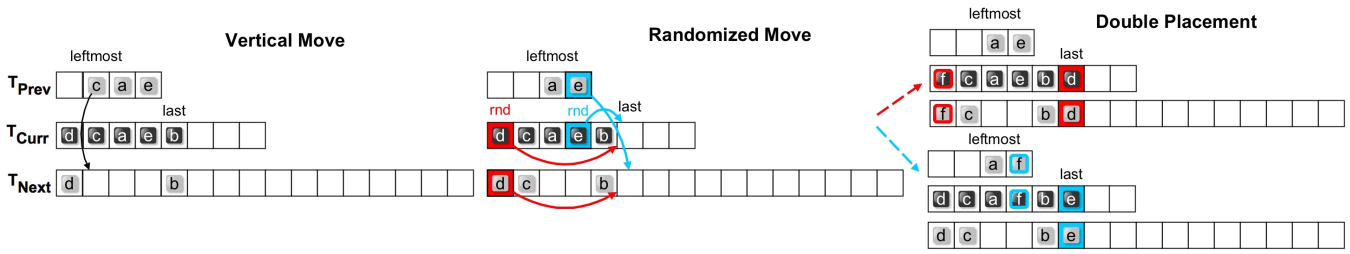


Figure 4. The three phases of Store(f) of the Resizable One-Time Auditable (ROTA) memory manager. Each record has two copies, the *active* copy is dark colored and stored in T_{Curr} and the *inactive* copy is light colored and stored either in T_{Prev} or in T_{Next}.

secure auditable data structures. The basic building block of our constructions is a new *Resizable One-Time Auditable* memory manager, ROTA for brevity, that achieves *constant worst-case time complexity per operation*. Recall that the weakly history-independent memory management technique, i.e. same privacy guarantees, proposed in [19], [26] has linear update time in the worst-case. We use the core resizing idea behind ROTA to propose a new dynamic one-time secure hash table that achieves updates in $O(1)$ in expectation.

In the following subsections we use the letter m with different interpretations, e.g. number of records or elements, depending on the context; we specify m accordingly in each subsection.

4.1. Resizable One-Time Auditable Memory Manager (ROTA)

Our ROTA construction is a dynamic memory manager that translates an auditable data structure with one-time secure topology into an audtiable data structure with one-time secure implementation. ROTA construction *efficiently resizes the allocated space* according to the memory requests to keep storage space proportional to the total size of the stored records, while maintaining *constant worst-case time* complexity per operation.

Previous HI Resizing. In comparison all the proposed strong and weak history-independent memory management techniques [19], [26] need to double their memory and migrate (resp. halve its memory and migrate) whenever the total number of records is larger than (resp. smaller than) a *threshold-value*. Thus, whenever the number of records "crosses" one of the threshold-values, e.g. threshold-values=$\{2, 2^2, 2^3, 2^4, 2^5, \ldots\}$ the corresponding HI memory manager migrates all the records to a new memory segment. Notice that the threshold-value resizing approach takes *linear time* in the worst case and is different from the standard deamortization analysis where resizing takes place when the data structure reaches a certain *threshold-load-factor*, i.e. resize when $|records| \geq 0.7 \times |segment\_size|$. As noted in [26] certain workloads, potentially adversarially chosen, can slow down the above memory managers if they frequently cross the threshold value, e.g. by alternating Free and Store around a threshold-value, and force the data structure to have linear time performance on every operation.

Intuition. ROTA handles three memory segments that are labeled as T_{Prev}, T_{Curr} and T_{Next} where $|T_{Curr}| = 2 \cdot |T_{Prev}|$ and $|T_{Next}| = 2 \cdot |T_{Curr}|$. Every record that is stored with ROTA has two distinct copies the *active* copy, which is always stored in T_{Curr}, and the *inactive* copy. Ideally when T_{Curr} is full (resp. half-full) we want to have a new memory segment ready that is twice the size (resp. half the size) of T_{Curr} and contains the same information. We achieve the above objectives by moving inactive records between the two additional segments T_{Prev} and T_{Next}. The challenge is to prepare the new segment in constant time and maintain the privacy guarantees.

At a high level, whenever a new record is stored we remove one inactive record from T_{Prev} and add two inactive records in T_{Next} thus we act as we are expanding. Whenever a record is freed we remove one inactive record from T_{Next} and add two inactive records in T_{Prev} thus we act as we are shrinking. The challenge is to perform the above tasks in a manner that does not reveal the history. An invariant that we utilize to securely perform the moves is that the relative order of the active copies in T_{Curr} is a random permutation to the eyes of the one-time auditor. Thus, storing one subsequence of this random permutation to T_{Prev} and the remaining subsequence(s) to T_{Next} is equivalent to partitioning the inactive copies to two random subsets. The relative fraction of records in T_{Prev} or T_{Next} corresponds to how close T_{Curr} is to being full enough, so we need to get more memory, or empty enough, so we need to release some memory.

The Store, Free algorithms can be described in three phases: the vertical move, the randomized move and the double placement. An example is depicted in Figure 4. We note that the inactive copy is located either in T_{Prev} or T_{Next} but at the *same index* as its active copy in T_{Curr}. We denote with *leftmost* the variable that holds the index of the leftmost occupied cell of T_{Prev} and with *last* the global variable that holds the index of the rightmost occupied cell of T_{Curr}. In the rest of the paper we assume that pointers to the moved records are adjusted appropriately without stating it explicitly. In this subsection, we denote with m the number of records stored by ROTA and we further assume that all records have constant size. We describe the moves with respect to Store.

Vertical Move. When Store is called, we act as if T_{Prev} is gradually migrating to T_{Next}. Therefore, the inactive copy from T_{Prev}[*leftmost*] is moved to the corresponding empty location of T_{Next}[*leftmost*] in order to prepare T_{Next} for

Algorithm 1: ROTA.Store(record)

if T_{Prev} *is null* **then**
  Make T_{Prev} point to T_{Curr}, T_{Curr} point to T_{Next} and T_{Next} point to a new $2|T_{Curr}|$ segment;
end
Move $T_{Prev}[leftmost]$ to $T_{Next}[leftmost]$;
Increase $leftmost$ and $last$ by one;
Choose an integer rnd at random from $[0, last]$;
if $rnd \in [leftmost, |T_{Prev}|]$ **then**
  Move $T_{Prev}[rnd]$ to $T_{Next}[last]$ and $T_{Curr}[rnd]$ to $T_{Curr}[last]$;
  Allocate $record$ to $T_{Prev}[rnd]$ and $T_{Curr}[rnd]$;
else
  Move $T_{Next}[rnd]$ to $T_{Next}[last]$ and $T_{Curr}[rnd]$ to $T_{Curr}[last]$;
  Allocate $record$ to $T_{Next}[rnd]$ and $T_{Curr}[rnd]$;
end
if T_{Curr} *is full* **then**
  Free the memory of T_{Prev};
  Set $T_{Prev} \leftarrow$ null and $leftmost \leftarrow 0$;
end

Algorithm 2: ROTA.Free(addr)

if T_{Prev} *is null* **then**
  Make T_{Prev} point to a new $|T_{Curr}|/2$ segment. Set $leftmost$ to $|T_{Prev}|$;
end
Decrease $leftmost$ and $last$ by one;
Move $T_{Next}[leftmost]$ to $T_{Prev}[leftmost]$;
Move $T_{Curr}[last + 1]$ to $T_{Curr}[addr]$;
if $addr \in [leftmost, |T_{Prev}|]$ **then**
  Move $T_{Next}[last + 1]$ to $T_{Prev}[addr]$;
else
  Move $T_{Prev}[last + 1]$ to $T_{Prev}[addr]$;
end
if T_{Next} *is empty* **then**
  Free the memory of T_{Next};
  Label segment T_{Prev} as T_{Curr} and T_{Curr} as T_{Next}.;
  Set $T_{Prev} \leftarrow$ null and $leftmost \leftarrow 0$;
end

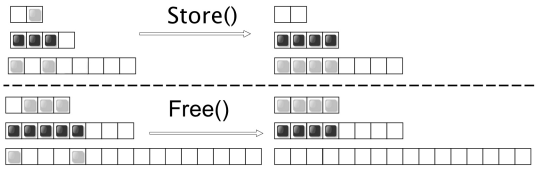


Figure 5. Corner cases for resizing with ROTA.

when T_{Curr} is full. Notice that due to the invariant of our construction the inactive record that is moved is a *random* inactive record from T_{Prev} to the eyes of the one-time auditor. The first column of Figure 4 illustrates the vertical move.

Randomized Move. In this phase we make space for the active and inactive copy of the new record. In detail, we increase *last* by one and we sample the index of an allocated active record indicated as *rnd* from the range $[0, last]$. The active copy in $T_{Curr}[rnd]$ is moved to $T_{Curr}[last]$. In case the inactive copy of $T_{Curr}[rnd]$ is in T_{Prev} we move $T_{Prev}[rnd]$ to $T_{Next}[last]$, illustrated as a blue transition in the second column of Figure 4. In case the inactive copy of $T_{Curr}[rnd]$ is in T_{Next} we move $T_{Next}[rnd]$ to $T_{Next}[last]$, illustrated as a red transition in the second column of Figure 4.

Double Placement. In this phase we allocate the new record with respect to the result of the randomized move. In case the inactive copy that was moved to $T_{Next}[last]$ was originally in T_{Next}, e.g. the red case in Figure 4, then we allocate the inactive new record in $T_{Next}[rnd]$ and the active new record in $T_{Curr}[rnd]$. In case the inactive copy that was moved to $T_{Next}[last]$ was originally in T_{Prev}, e.g. the blue case, then we allocate the inactive new record in $T_{Prev}[rnd]$ and the active new record in $T_{Curr}[rnd]$. The third column of Figure 4 illustrates the above moves.

The intuition behind Free is similar. A vertical move is performed from T_{Next} to T_{Prev}, then the locations of the inactive and active copies of the removed record are emptied and finally a double placement takes place in order to cover the gap caused by the removals. A detailed description is presented in Algorithm 2.

The construction of ROTA is such that for any given number of records, m, we know which cells from each of the three segment are occupied. If we are not careful enough, the corner case where $m = 2^j$ can reveal whether the last operation was a Store or a Free. To illustrate this point, when we execute a Store which will result in $m = 2^j$, we move the last record of T_{Prev} to T_{Next}. Thus, we have an empty T_{Prev}, a full T_{Curr} and a half-full T_{Next}, see first line of Figure 5. Whereas when we execute a Free which will result in $m = 2^j$ as well, we have a different layout since T_{Prev} is full, T_{Curr} is half-full and T_{Next} is empty, see second line of Figure 5.

We deal with this corner case with the following course of action that is explained in Algorithms 1 and 2: if in the end of the Store/Free operation we have $m = 2^j$, then we label the full segment as T_{Curr}, the half-full segment as T_{Next} and the set the label T_{Prev} point to null. Thus we have the same memory representation, regardless of whether the last operation was Store or Free. The first and last if statements of algorithms Store, Free handle this scenario.

Theorem 2. *Let $\mathcal{D}$ be an auditable data structure with one-time secure topology that deploys memory manager ROTA. Then the implementation of $\mathcal{D}$ is one-time secure. The time complexity of memory operations Store and Free of ROTA is $O(1)$ in the worst case and the space used is $O(m)$, where m is the number of allocated records.*

Proof. (Sketch) First we prove that the allocation of T_{Curr} is one-time secure and then we extend the property to the other two segments. For the case of Store operation since the relative order of the records in T_{Curr} is already a random permutation (invariant condition), the swap that is performed between $T_{Curr}[last]$ and $T_{Curr}[rnd]$ gives also a random permutation. For the case of Free operation, if the relative order of the records in T_{Curr} is already a random permutation then we maintain the randomness by covering the gap with the last record which is is a random active record to the eyes of the adversary. An adversary that can win the game of Definition 8 with a non-negligible advantage implies that he can differentiate between the random generator used for

Algorithm 3: DHT.Insert(u)

if HT_{Prev} *is null* **then**
  Make HT_{Prev} point to HT_{Curr}, HT_{Curr} point to HT_{Next} and HT_{Next} point to a new $2|\mathsf{HT}_{Next}|$ size hash table;
end
Assign X a random value from the range $[1, \#HT_{Curr}]$;
if $X \in [\#HT_{Prev} + 1, \#HT_{Curr}]$ **then**
  Move a random record from HT_{Prev} to HT_{Next};
  Insert u to HT_{Next} and HT_{Curr};
else if $X \in [1, \#HT_{Prev}]$ **then**
  Move two random records from HT_{Prev} to HT_{Next};
  Insert u to HT_{Prev} and HT_{Curr};
end
if HT_{Curr} *is full* **then**
  Free HT_{Prev} and set HT_{Prev} to null;
end

Algorithm 4: DHT.Delete(u)

if HT_{Prev} *is null* **then**
  Make HT_{Prev} point to a new hash table of size $|\mathsf{HT}_{Curr}|/2$;
end
if *secondary copy of* u *is stored in* HT_{Next} **then**
  Remove u from HT_{Next} and HT_{Curr};
  Move a random record from HT_{Next} to HT_{Prev};
else
  Remove u from HT_{Prev} and HT_{Curr};
  Move two random records from HT_{Prev} to HT_{Next};
end
if HT_{Next} *is empty* **then**
  Free HT_{Next}, make HT_{Prev} point to HT_{Curr}, and HT_{Curr} point to HT_{Next}, set HT_{Prev} to null ;
end

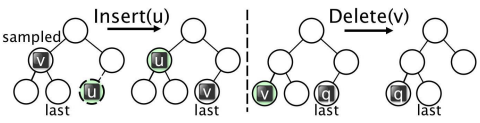


Figure 6. Insert & Delete in a One Time Auditable Complete Binary Tree.

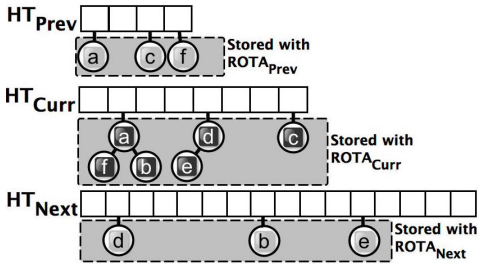


Figure 7. Algorithms and an illustration of the Dynamic One-Time Auditable Hash Table (DHT). Primary copies are dark colored while secondary copies are light colored.

rnd and true randomness. Thus, one can form an inductive argument on the number of Store/Free operations of the sequence in order to prove the security for T_{Curr}. As a next step we show that for any set of m records there is only one possible set of occupied memory cells (ignoring the content) in the image of ROTA, thus we can focus on the content of the occupied cells. Recall that for every index i that corresponds to an occupied cell of T_{Curr}, one of the following two statements hold: A) $T_{Prev}[i] = T_{Curr}[i]$ and $T_{Next}[i] = \perp$, B) $T_{Next}[i] = T_{Curr}[i]$ and $T_{Prev}[i] = \perp$. This implies that the relative order of the inactive records follow the relative order of the active records which we already proved to be a random permutation. We conclude the proof by showing that the assignment of the inactive copies to tables T_{Prev}, T_{Next} depends only on m, thus the partition of a random sequence, i.e. T_{Curr}, into subsequences of fixed size distributes the records randomly between them. □

4.2. One-Time Auditable Hash Table

We introduce a secure one-time auditable hash table via chaining with complete binary trees that we call HT. We store all the chains of the hash table using the same one-time auditable memory manager ROTA. Our design choices satisfy two desired properties, (1) fast sampling from the set of nodes from a fixed chain and (2) constant time sampling from the set of nodes from all the chains; we elaborate on the latter in the next section. As for the first property, notice that all the chains are packed in the same memory segment, i.e. T_{Curr} of ROTA, which implies that we can not sample from a fixed chain in constant time due to the fact that nodes of different chains are interleaved in this segment; thus the next available choice is to sample via traversing the corresponding chain. Our design choice of binary trees allows the insertion process to sample in $O(\log m)$ time by traversing the tree stored in ROTA, as opposed to $O(m)$ time that would take to sample from a linked list by traversing its pointers. For the hash table HT we first present the analysis assuming a constant load factor $\alpha = m/M$ and later propose an efficient resizing dynamic hash table called DHT.

4.2.1. One-Time Auditable Complete Binary Trees. We introduce the *one-time auditable complete binary tree* and show that its topology is one-time secure. As for the memory representation of the data structure, it is required to deploy a one-time auditable memory manager, for instance ROTA from Section 4.1. A binary tree with height h is called complete if we have the maximum number of nodes possible in levels $0, 1, \ldots, h-1$, while all the nodes in level $h-1$ are as far left as possible. The letter m refers to the number of elements that are stored in the tree. We emphasize that this is *not* a binary search tree, but rather a plain binary tree; the goal of the tree is to allow faster insertions, which due to the privacy-properties require sampling a random node using the pointer structure.

Intuition: Since the binary tree is complete the shape of the data structure is fixed given its size, regardless of the value of the elements. To sample a node uniformly at random we start from the root and randomly decide whether to terminate or proceed to the next level with probability dependent on the size of the associated subtrees denoted as $size(\cdot)$ in Algorithm 5. For a node u one can calculate the size of the subtree under u using a u's rank and the total number of elements.

Therefore in order to make the Insert(u) method one-time secure, we place the new node to the leftmost available place in the last level (or start a new level if the tree is full and complete), and sample a node uniformly at random from the augmented tree which takes $O(\log m)$ time in the worst case using Algorithm 5. Then we swap the inserted element

Algorithm 5: Binary Tree SampleNode(v)

Define a random variable $R \in \{Left, Right, End\}$ with probabilities $Pr(R = End) = \frac{1}{size(v)}$, $Pr(R = Left) = \frac{size(v.left)}{size(v)}$, $Pr(R = Right) = \frac{size(v.right)}{size(v)}$;
Generate a value for R according to its distribution;
switch *R* **do**
    case *R=End:* **return** *v* ;
    case *R=Left:* **return** *SampleNode(v.left)* ;
    case *R=Right:* **return** *SampleNode(v.right)* ;
endsw

with the element stored in the sampled node. The overall time complexity of Insert(u) is $O(\log m)$. An illustration of the updates is provided in Figure 6. The Find(u) process traverses the tree until it locates the element therefore the time is linear in the worst case. The Delete(u) process calls Find(u) to locate the node that stores the input element and moves the element of the rightmost node of the last level to cover the gap. We mention here that in the special case where we know the address of the deleted element u (which occurs later when we move sampled elements in DHT) there is no need to call Find(u), so the Delete(u) method takes $O(\log m)$ in the worst case.

Lemma 5. *Let $\mathcal{D}$ be a complete binary tree for which the algorithms Find, Insert, Delete follow the above description. The topology of $\mathcal{D}$ is one-time secure. Operations Delete and Find run in $O(m)$ worst-case time, operation Insert runs in $O(\log m)$ worst-case time, and the space used is $O(m)$.*

Proof. (Sketch) To prove that the topology is one-time secure we treat the complete binary tree as a sequence Q' that is a acquired via a level-order traversal. It is enough to show that Q' is a random permutation. We focus first on a tree resulted from insertions only; using the fact that Insert places the element at a random node we can build an inductive argument and show that the resulting sequence Q' is a random permutation. Then we form a security reduction over the randomness used to sample a node and show that if there is an adversary that can distinguish Q' from a random permutation then she can distinguish between the used PRNG and true randomness. For the case of deletions we build a similar inductive argument on the number of Delete operations. To argue about the base case we use the fact that we cover the "gap" in Q' with the last element; since we only had insertions in the base case the element in last is the a randomly chosen element. Following with one more security reduction on the used PRNG we show that even when there are deletions the topology is one time secure. □

Regarding the implementation of the binary tree we can deploy a ROTA without any additional time overhead as it is summarized in the next theorem.

Theorem 3. *Let $\mathcal{D}$ be a complete binary tree with one-time secure topology as described in Lemma 5 and assume that $\mathcal{D}$ uses memory manager ROTA. We have that the implementation of $\mathcal{D}$ is secure. Also, let m be the number of elements stored in the tree. Operations Delete and Find run in $O(m)$ worst-case time, operation Insert runs in $O(\log m)$ worst-case time, and the space used is $O(m)$.*

4.2.2. Analysis for the Hash Table (HT). To simplify the performance analysis we assume that the hash function is truly random, mapping each data item independently and uniformly to the range. We emphasize here that the hash function concerns the efficiency and does not affect the security of our construction.

In this first construction of the hash table we assume that the load factor α never exceeds some constant α_{max}. The expected number of elements per cell is therefore constant, and a simple application of Chernoff bounds and a union bound [24] shows that, for any desired constant c, with probability at least $1 - m^{-c}$ no bin receives more than $\Theta(\frac{\log m}{\log \log m})$ balls (See, e.g., Ch. 5.2 of [24]). Hence the expected size of a complete binary tree stored in a hash table bucket is $O(1)$ and with high probability the maximum size is $\Theta(\frac{\log m}{\log \log m})$.

Due to space limitations we do not present the algorithms Insert, Delete, Find, but having the analysis and the description of the chaining structure in place, the pseudocode is rather straightforward. For the method Insert(u) of the hash table, we use the hash function h to find the corresponding cell of the table and insert the element in the complete binary tree. Since the insertion in the one-time auditable complete tree takes logarithmic time to the size of the tree, the Insert method of the hash table takes $O(\log \log m)$ time in the worst case with high probability where m is the number of elements in the hash table. For the method Delete(u) we hash u to find the corresponding cell j and follow the deletion algorithm of the complete binary tree to remove the element from the chain. For the method Find, we hash the input element and traverse the corresponding complete binary tree.

Theorem 4. *Let m be the number of elements stored in a hash table of fixed size M. Let $\mathcal{D}$ be a hash table with chaining with one-time secure auditable complete binary trees as in Lemma 5 that are allocated with a ROTA memory manager. Then, the implementation of $\mathcal{D}$ is one-time secure. For a constant load factor, the insertion time of $\mathcal{D}$ is $O(1)$ in expectation and $O(\log \log m)$ with high probability. Searches and deletions take $O(1)$ time in expectation and $O(\log m)$ time with high probability.*

4.2.3. Dynamic Hash Table (DHT). We propose an efficient and secure resizing technique for one-time auditable hash tables so as to construct a *dynamic one-time auditable hash table*, or DHT for brevity. Our technique is inspired by the ROTA construction of Section 4.1. Let HT_X denote the hash table via chaining with trees as it is described in Section 4.2.2 and $\#\mathsf{HT}_X$ denote the number of elements stored in the hash table.

Intuition: Similarly to ROTA, every element that is stored in DHT has two distinct copies the *primary* and the *secondary*. DHT handles three fixed-size hash tables namely HT_{Prev}, HT_{Curr} and HT_{Next} where $|\mathsf{HT}_{Curr}| = 2|\mathsf{HT}_{Prev}|$

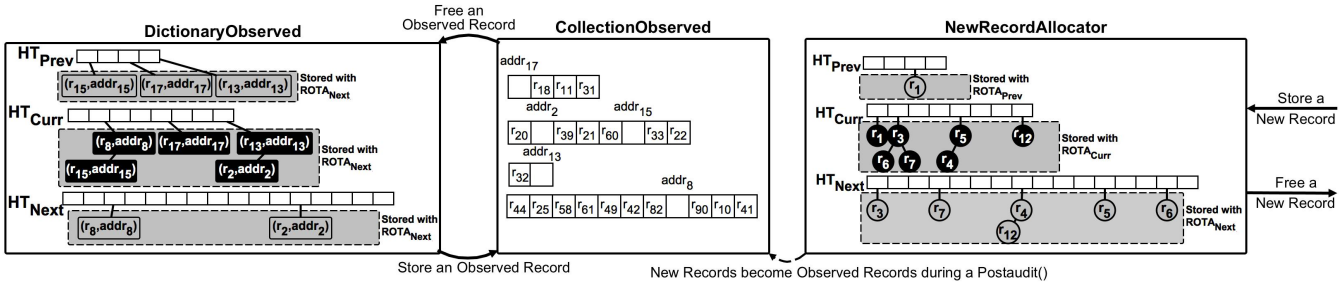


Figure 8. The algorithms and the architecture of AMM. The solid arrows show the move of records during operations Store and Free. The dotted arrow shows the move of records during a Postaudit call.

and $|\mathsf{HT}_{Next}| = 2|\mathsf{HT}_{Curr}|$. Each hash table has its own memory manager, namely ROTA_{Prev}, ROTA_{Curr} and ROTA_{Next}. The primary copy of an element, depicted with a dark color in Figure 7, is always stored in HT_{Curr} whereas the secondary copy, depicted with a light color in Figure 7, can be either in HT_{Prev} or HT_{Next}. After an insertion $\#\mathsf{HT}_{Curr}$ increases by 1, $\#\mathsf{HT}_{Next}$ increases by 2 and $\#\mathsf{HT}_{Prev}$ decreases by 1, we call this the *growing invariant*. After a deletion $\#\mathsf{HT}_{Curr}$ decreases by 1, $\#\mathsf{HT}_{Next}$ decreases by 1 and $\#\mathsf{HT}_{Prev}$ increases by 2, we call this the *shrinking invariant*. We maintain the HT_{Curr} as if we were not resizing but at the same time we move random items between HT_{Prev} and HT_{Next} so as to prepare for future growing/shrinking.

Fast sampling from an HT: We use as a running example the Insert method. The primary copy of the new element is always allocated in HT_{Curr} while we randomly decide whether the secondary copy is stored in HT_{Prev} or HT_{Next}. Specifically, with probability $\#\mathsf{HT}_{Next}/\#\mathsf{HT}_{Curr}$ (Lines 5-7 of Algorithm 3) we insert the secondary copy of the new element in HT_{Next} and with probability $\#\mathsf{HT}_{Prev}/\#\mathsf{HT}_{Curr}$ (Lines 8-10 of Algorithm 3) we insert it in HT_{Prev}. In order to satisfy the growing invariant we need to move randomly chosen elements between HT_{Prev} and HT_{Next}. Recall that in the ROTA algorithms we could conveniently sample uniformly at random from T_{Prev} by visiting the index *leftmost*. To perform a similar step for the case of DHT we need to sample uniformly at random from all the binary tree nodes of HT_{Prev}. Fast sampling from all the chains is the reason we chose to have a single ROTA for HT. By accessing the address where the *last* pointer of ROTA_{Prev} points, we sample uniformly at random a node among *all the chains* of HT_{Prev}. Notice that deleting a sampled node from the chain of HT_{Prev} takes logarithmic time to the size of the chain, as we discussed earlier, due to the fact that we know the address of the deleted node, i.e. dereference of *last*. Similarly in Algorithm 4 we satisfy the shrinking invariant by sampling and moving records from HT_{Next} to HT_{Prev}.

Theorem 5. *Let $\mathcal{D}$ be a dynamic hash table with chaining provided by auditable complete binary trees with one-time secure implementation as in Lemma 5. Then the implementation of $\mathcal{D}$ that deploys Algorithms 3 and 4 is one-time secure. Also, let m be the number of elements stored in the hash table. Then the space used is $O(m)$, the insertion time is $O(1)$ in expectation and $O(\log\log m)$ with high probability, and searches and deletions take $O(1)$ time in expectation and $O(\log m)$ time with high probability.*

Proof. (Sketch) We utilize the security properties for ROTA, complete binary trees, and fixed size one-time auditable hash table to prove the dynamic case. Thus we focus on the distribution of elements between the three hash tables. The collection of elements that are allocated in the dynamic hash table consists of two copies of the same set of elements. The first copy is entirely stored in HT_{Curr} and the second copy is partitioned in two sets where the first subset is in HT_{Prev} and the second subset in HT_{Next}. Similarly to the security proof of ROTA, the size of each subset depends only on the value of m. Since in every move between the hash tables we choose to move an element uniformly at random, the exact content of each subset looks random to the one-time auditor. The proof concludes by using the fact that each individual hash table has secure implementation. □

5. Resizable Memory Manager for Auditable Data Structures (AMM)

We describe the construction of the resizable memory manager for auditable data structures, AMM for brevity, that translates *any* auditable data structure with secure topology into an auditable data structure with secure implementation. Even though the components of AMM have only one-time secure implementations (e.g. $k = 1$ in Definition 8), the resulting construction of AMM gives a secure implementation of an auditable data structure (e.g. $k > 1$ in Definition 7). This property holds since the one-time auditable components have *a lifetime of a single session and then are discarded*. Subsequently, after every audit, AMM's Postaudit() creates fresh one-time auditable data structures.

A key insight of our construction's efficiency is that we treat differently observed records from new records (concepts introduced in Section 3) since in our model the implementation knows what the auditor observed. Thus, we decouple the time complexity of the memory operations from the total number of records. In the following, we denote the

total number of records (both observed and new) currently stored by the data structure with n and the length of the current session with S.

Recall that SHIMM is the SHI memory manager that consists of the linear probing hash table proposed in [9] with the additional functionality of resizing whenever the number of records "crosses" a predetermined threshold-value.

Algorithm 6: AMM.Free(addr)

```
1 Let record be the record allocated at addr;
  // Freeing a New Record
2 Remove record from NewRecordAllocator;
3 if the removal returns null then
      // Freeing an Observed Record
4     Insert (record,addr) to DictionaryObserved;
5     Clear but don't free the memory at addr;
6 end
```

Algorithm 7: AMM.Store(record)

```
1 Remove record from DictionaryObserved;
2 if the removal returns null then
      // Storing a New Record
3     Insert record to NewRecordAllocator;
4 else
      // Storing an Observed Record
5     Place back record at addr returned from the
      removal;
6 end
```

Components of AMM. Our construction has the following three components, also illustrated in Figure 8:

NewRecordAllocator: The auditor has no prior knowledge about records that were *not observed* in the latest audit. Thus new records can be stored/freed using any resizable one-time auditable data structure, we use DHT from Section 4. In Figure 8 the new records $\{r_1, r_3, r_4, r_5, r_6, r_7, r_{12}\}$ were stored during the running session.

CollectionObserved: After an audit the new records that are stored via the NewRecordAllocator become observed records since in the next audit the auditor expects to see them in the same memory location. We mark this transition from new to observed by considering their allocation as part of the CollectionObserved which is a set of (possibly scattered) observed records for which the canonicity of their address is maintained. This is a "conceptual" component, i.e. not a new data structure, since it contains the memory segments of audited NewRecordAllocator, specifically the segment T_{Curr} of ROTA_{Curr} of a former NewRecordAllocator. The four segments of CollectionObserved in Figure 8 contain observed records that come from previous audits. Observed records $\{r_2, r_8, r_{13}, r_{15}, r_{17}\}$ were removed since the last audit but their corresponding addresses are not freed.

DictionaryObserved: The auditor expects that in the next audit he will find an observed record at the address that was seen in the latest audit. To satisfy this property we maintain a one-time auditable dictionary where the "key" is an observed record that is *deleted* in the running session and its "value" is the observed address. We use a DHT from Section 4 for this component. For instance in Figure 8 the observed records $\{r_2, r_8, r_{13}, r_{15}, r_{17}\}$ were freed since the beginning of the running session. Essentially the canonicity of the observed records is maintained with this data structure which is part of the auxiliary space of I_M and has time complexity that is a function of S instead of n as opposed to SHIMM. When an audit occurs we don't need to maintain the canonicity of deleted observed records anymore, thus the current DictionaryObserved is discarded.

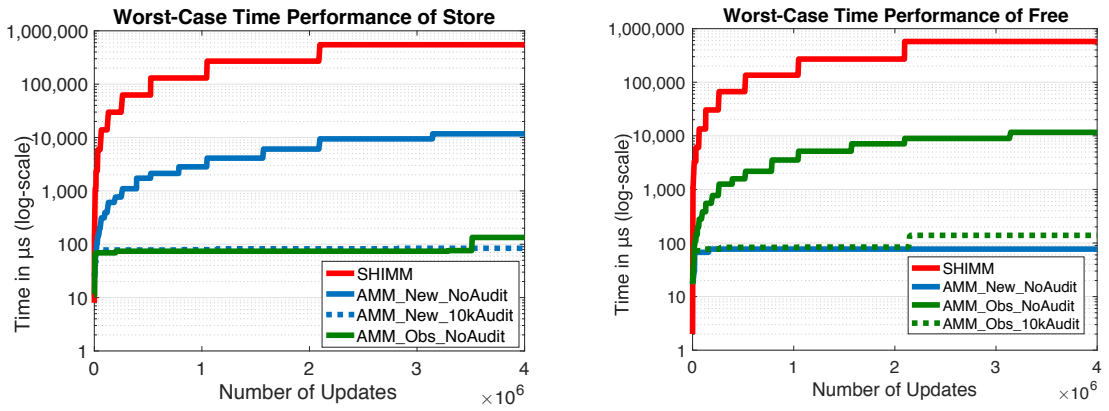
Postaudit. The algorithm Postaudit performs maintenance operations in $O(S)$ so as to prepare for the next audit. The data structure DictionaryObserved is not needed after the audit since those removed observed records were *not* part of the state when the latest audit took place. At this point, we can free the memory cell of the deleted observed records that was cleared but not freed (see Line 5 of Algorithm 6). Since NewRecordAllocator is implemented with a DHT, the primary copies are stored in its HT_{Curr}. The chains of HT_{Curr} are stored using ROTA_{Curr} thus the active copies of the chains are in T_{Curr} of ROTA_{Curr}. Therefore PostAudit passes the above T_{Curr} segment to component CollectionObserved. The rest of HT_{Curr}, as well as HT_{Prev} and HT_{Next}, is freed. The last step is the creation of DictionaryObserved and NewRecordAllocator for the next session.

Algorithm 8: AMM.Postaudit()

```
  // Freeing unnecessary memory
1 Iterate through DictionaryObserved and release all
  the memory cells where the "values" from key-value
  pairs point to;
2 Release the memory of DictionaryObserved;
3 The memory segment were the primary copies of
  NewRecordAllocator becomes part of
  CollectionObserved;
4 The rest of NewRecordAllocator's memory is
  released;
  // Create fresh copies for the next
     session
5 Create a new and empty DictionaryObserved and a
  new and empty NewRecordAllocator;
```

Security We sketch below the main arguments of the proof that memory manager AMM yields a secure implementation. The one-time security of the implementation of DHT is discussed in Section 4. Using an inductive argument on the number of sessions one can show that the overall construction yields a secure implementation. In the base case, we have the first session were the collection of observed records is empty and all inserted records are new. For the inductive step it is enough to show that the k-th session preserves the security of the construction. During a session, one can process either observed records or new records. According to Algorithms 5, 6 if an observed record is removed its observed address is stored in DictionaryObserved and its memory is cleared but *not* freed. Thus if it is re-inserted, it will be placed at the same location that was observed during the last audit. As for the new records, their memory management is handled exclusively by the NewRecordAllocator. Finally Postaudit frees the memory of DictionaryObserved, since those records are not part of the observed set any more, while "passes" the appropriate memory segment of NewRecordAllocator to the CollectionObserved.

Theorem 6. *Let $\mathcal{D}$ be an auditable data structure with secure topology that deploys memory manager AMM. We denote*



Average Time in μs	SHIMM store	SHIMM free	AMM store New No Audit	AMM store New Audit 10K	AMM store Obs. No Audit	AMM free New No Audit	AMM free Obs. No Audit	AMM free Obs. Audit 10K
	0.53	0.71	4.75	3.07	4.44	2.23	4.44	2.94
Percentage of Resizes	0.000004 %		-					

Figure 9. Comparison worst-case time performance between AMM and SHIMM under various setups. The worst-case time running times of AMM are $45\times$-$8,300\times$ faster than SHIMM.

with S the length of the current session and with n the total size of the records of $\mathcal{D}$. We have that the implementation of $\mathcal{D}$ is secure. The space used by AMM is $O(n+S)$ during a session and $O(n)$ at the end of a session, after the execution of Postaudit. Algorithms Initialize, Read, and Write take each worst-case time $O(1)$. Algorithms Store and Free take $O(1)$ in expectation and $O(\log S)$ with high probability. Algorithm Postaudit takes worst-case time $O(S)$.

6. Evaluation

We implemented both AMM and its strongly history-independent counterpart, SHIMM [9], in C++ and tested their performance with a double linked list as a pointer structure. Since AMM is a full-fledged prototype, approximately 1200 lines of code, we update the pointers of the moving records in DHTs and ROTAs. All experiments were conducted on a host running Debian GNU/Linux v3.16, equipped with a 3.5GHz Intel Quad Core i5 CPU and 16GB of RAM.

Increasing Workloads & Worst-Case Time. A setup of this experiment has three different variables: (1) the *recorded operation* which is either Store or Free; (2) the *type of records* which is either new or observed; (3) the *frequency of audits* which in our experiment occur either after 10K recorded data points or never, i.e. $S = n$. Notice that the case where $S = n$ is the least favorable scenario for AMM in terms of efficiency. On each setup we recorded $4 \cdot 10^6$ data points of the time performance of AMM under synthetic workloads that *increase the size* of the corresponding component which depending on the setup can be either NewRecordAllocator or DictionaryObserved. As an example, to observe the worst-case time performance of the setup {Store, Observed, NoAudits} we have to stress the component DictionaryObserved of AMM, which grows when an observed record is freed. Thus the sequence of operations consists of a series of two Free calls for two observed records, therefore the size of DictionaryObserved increases, and one recorded Store call for an observed record. For this workload we present the slowest individual data point among 5 runs and for the average time the average of all 5 runs. The two "Worst-case time Performance" plots of Figure 9 demonstrate the impact of resizing on the performance of SHIMM. Our construction, AMM, outperforms SHIMM by several orders of magnitude due to the efficiency of our resizable building blocks. In case we perform audits (dotted lines in Figure 9) the worst-case time running times of AMM are up to $8,300$ times faster than SHIMM due to the fact that the corresponding components of AMM are "flushed" after every audit. Frequent audits make AMM perform faster. We conducted further micro-benchmarks in order to analyze the slowest worst-case time performance of AMM, which occurs at the setups {Store, new, NoAudit} and {Free, observed, NoAudit} and we report that the large delays are due to the overhead of the OS allocator for mapping new empty memory pages.

According to the table of Figure 9 the average time performance of SHIMM is better in the increasing workload experiment which is explained by the fact that Store and Free take place in a half-empty hash table [9]. Furthermore, only 0.000004% of the total $4 \cdot 10^6$ operations of the increasing workload sequence resulted in a resize of SHIMM. Since in a typical workload we have a combination of allocations and deallocations we conducted a second experiment to better understand the impact of resizing based on a threshold-value.

Oscillating Workloads & Average-Case Time. In this experiment the sequence of operations oscillates around the resize threshold-value in a controlled manner. Recall that an operation in SHIMM takes linear time if the number of records "crosses" a fixed resize threshold-value. We pick thresholds $\theta_1 = 2^{14}$ and $\theta_2 = 2^{20}$ and we fix the length of each sequence to be 10^3 and 10^5 respectively. For each fixed

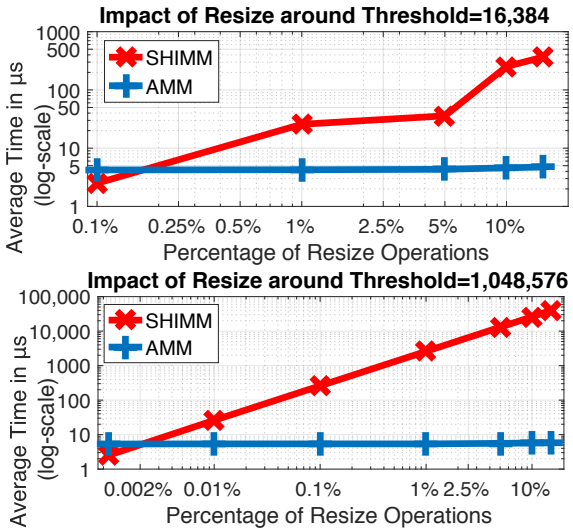


Figure 10. Comparison of average time performance between AMM and SHIMM for oscillating workloads.

length sequence we pick a combination of Store and Free so as a predetermined percentage of the operations, depicted in X-axis of Figure 10, would result in a resize.

As it shown in the corresponding plot of Figure 10 with resize threshold $\theta_1 = 2^{14}$, memory manager AMM has significantly better average case time performance than SHIMM for any percentage higher than 0.125%. Thus, if there is more than 1 resize for every 800 operations around θ_1, the performance of AMM is better than SHIMM for the average case as well. The efficiency of our technique is even more obvious as the number of resizes increases, e.g. for a sequence of 10% resizes around θ_1 the average time of AMM is around two orders of magnitude smaller. The impact of the resize operations penalize SHIMM even more dramatically for the resize threshold $\theta_2 = 2^{14}$, where AMM outperforms SHIMM for any percentage higher than 0.002%. Thus, if there are more than 1 resize for every 5000 operations around θ_2, the average time performance of AMM is better than the one of SHIMM. Also for a sequence of 10% resizes around θ_2 the average time of AMM is close to five orders of magnitude smaller. We mention here that the OS allocator overhead of AMM that was reported in the increasing workload experiment is almost unnoticeable in the experiment with oscillating workloads since the OS allocator is warmed up.

7. Conclusion and Future Work

In this paper, we introduced the notion of auditable data structures, which matches the ability of strongly history-independence to allow for multiple audits while achieving the efficiencies of weakly history-independence. In addition, we developed specific one-time secure auditable data structures and a general memory management scheme for auditable data structures. We leave for future work constructions of more complex auditable data structures, such as dynamic data structures for answering graph queries, range searching, and/or nearest-neighbor queries.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation under grants CCF–1535795, CCF–1320231, CNS–1228485, CNS–1228598, CNS–1228639, and CNS–1526631, and by the Kanellakis Fellowship at Brown University. This article reports on work supported in part by the Defense Advanced Research Projects Agency under agreement no. AFRL–FA8750–15–2–0092. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

[1] "European Union Directive L281, p. 31-50," European Parliament & Council, 1995.

[2] "The Health Insurance Portability and Accountability Act," U.S. Dept. of Health & Hum. Serv., 1996.

[3] "Voluntary Voting System Guidelines, Ver. 1.1, Vol. 1," Unit. States Election Assistance Commission, 2015.

[4] S. Bajaj, A. Chakraborti, and R. Sion, "Practical foundations of history independence," *IEEE Trans. Inform. Forens. and Sec.*, vol. 11:2, pp. 303–312, 2016.

[5] S. Bajaj and R. Sion, "Ficklebase: Looking into the future to erase the past," in *ICDE*, 2013, pp. 86–97.

[6] ——, "HIFS: history independence for file systems," in *ACM CCS*, 2013, pp. 1285–1296.

[7] M. A. Bender, J. W. Berry, R. Johnson, T. M. Kroeger, S. McCauley, C. A. Phillips, B. Simon, S. Singh, and D. Zage, "Anti-persistence on persistent storage: History-independent sparse tables and dictionaries," in *Proc. of the 35th PODS*, 2016, pp. 289–302.

[8] J. Bethencourt, D. Boneh, and B. Waters, "Cryptographic methods for storing ballots on a voting machine," in *NDSS*, 2007, pp. 209–222.

[9] G. E. Blelloch and D. Golovin, "Strongly history-independent hashing with applications," in *FOCS*, 2007, pp. 272–282.

[10] N. Buchbinder and E. Petrank, "Lower and upper bounds on obtaining history independence," in *Advances in Cryptology (CRYPTO)*, 2003.

[11] B. Chen and R. Sion, "HiFlash: A history independent flash device," *CoRR*, vol. abs/1511.05180, 2015.

[12] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," in *STOC*, 1987, pp. 182–194.

[13] D. Golovin, "Uniquely represented data structures with applications to privacy," PhD Thesis, CMU, 2008.

[14] ——, "B-treaps: A uniquely represented alternative to B-trees," in *ICALP*, 2009, pp. 487–499.

[15] ——, "The B-skip-list: A simpler uniquely represented alternative to B-trees," *Arxiv*, vol. 1005.0662, 2010.

[16] M. T. Goodrich, E. M. Kornaropoulos, M. Mitzenmacher, and R. Tamassia, "More practical and secure history-independent hash tables," *IACR ePrint 2016/134*, 2016.

[17] M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious RAM simulation," in *ICALP*, 2011.

[18] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving group data access via stateless oblivious RAM simulation," in *SODA*, 2012, pp. 157–167.

[19] J. D. Hartline, E. S. Hong, A. E. Mohr, W. R. Pentney, and E. Rocke, "Characterizing history independent data structures," *Algorithmica*, vol. 42, pp. 57–74, 2005.

[20] P.-H. Kamp, "Malloc(3) revisited," in *USENIX*, ser. ATC, 1998, pp. 36–36.

[21] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[22] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach, "Analysis of an electronic voting system," in *IEEE S&P*, 2004, pp. 27–40.

[23] D. Micciancio, "Oblivious data structures: Applications to cryptography," in *STOC*, 1997, pp. 456–464.

[24] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[25] M. Naor, G. Segev, and U. Wieder, "History-independent cuckoo hashing," in *ICALP*, 2008, pp. 631–642.

[26] M. Naor and V. Teague, "Anti-presistence: history independent data structures," in *STOC*, 2001, pp. 492–501.

[27] R. Poddar, T. Boelter, and R. A. Popa, "Arx: A strongly encrypted database system," IACR ePrint 2016/591, 2016.

[28] D. S. Roche, A. J. Aviv, and S. G. Choi, "A practical oblivious map data structure with secure deletion and history independence," in *IEEE S&P*, 2016.

[29] D. Springall, T. Finkenauer, Z. Durumeric, J. Kitcat, H. Hursti, M. MacAlpine, and J. A. Halderman, "Security analysis of the Estonian internet voting system," in *ACM CCS*, 2014, pp. 703–715.

[30] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An extremely simple oblivious RAM protocol," in *ACM CCS*, 2013, pp. 299–310.

[31] X. S. Wang, K. Nayak, C. Liu, T. H. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious data structures," in *ACM CCS*, 2014.

[32] S. Wolchok, E. Wustrow, J. A. Halderman, H. K. Prasad, A. Kankipati, S. K. Sakhamuri, V. Yagati, and R. Gonggrijp, "Security analysis of India's electronic voting machines," in *ACM CCS*, 2010, pp. 1–14.