Practical Foundations of History Independence

Sumeet Bajaj, Anrin Chakraborti, Radu Sion

Abstract—The way data structures organize data is often a function of the sequence of past operations. The organization of data is referred to as the data structure’s state, and the sequence of past operations constitutes the data structure’s history. A data structure state can therefore be used as an oracle to derive information about its history. As a result, for history-sensitive applications, such as privacy in e-voting, incremental signature schemes, and regulatory compliant data retention, it is imperative to conceal historical information contained within data structure states.

A data structure history can be hidden by making data structures history independent. In this paper, we explore how to achieve history independence.

We observe that current history independence notions are significantly limited in number and scope. There are two existing notions of history independence – weak history independence (WHI) and strong history independence (SHI). WHI does not protect against insider adversaries and SHI mandates canonical representations, resulting in inefficiency.

We postulate the need for a broad, encompassing notion of history independence, which can capture WHI, SHI, and a broad spectrum of new history independence notions. To this end, we introduce $\Delta$ history independence ($\Delta$HI), a generic game-based framework that is malleable enough to accommodate existing and new history independence notions.

As an essential step towards formalizing $\Delta$HI, we explore the concepts of abstract data types, data structures, machine models, memory representations, and history independence. Finally, to bridge the gap between theory and practice, we outline a general recipe for building end-to-end, history independent systems and demonstrate the use of the recipe in designing two history independent file systems.

Index Terms—History independence, data structures, regulatory compliance

1 INTRODUCTION

Data structures are commonly used constructs to store and retrieve data in systems. However, data structures carry more information than the raw data they organize. One aspect of this information is the history leading to the data structure’s current state [1].

Concealing historical information contained within data structure states is necessary for incremental signature schemes [2] and for privacy in voting systems [2], [3], [4], [5]. Therefore, the need arises for data structures that reveal no information about the history that led to their current state other than what is inherently visible from the data. The concept of history independence [6] has been devised to enable the design of such data structures and they are termed as “history independent data structures”.

We have identified the role of history independence in designing systems that are compliant with data retention regulations [7], [8], [9]. Retention regulations desire that once data is deleted, no evidence about the past existence of deleted data must be recoverable. Such a deletion cannot be achieved by simply overwriting data as in secure deletion [10]. This is because overwriting does not eliminate the effects that previous existence of delete data leaves on the current system state. Even after secure deletion, the current state can be used as an oracle to derive information about the past existence of deleted records. For example, the current organization of data blocks on disk is a function of the sequence of previous writes to file system or to database search indexes. The organization could be different depending on whether a particular record was deleted in the past or was never inserted in the data set. Therefore, questions about history, such as “was John’s record ever in the HIV patients’ dataset” can be answered much more accurately than guessing by simply looking at the search index organization on disk since the organization could be different depending on whether John has previously been in the data set or not. The inference of past existence of deleted data violates data retention regulations.

We posit that for regulatory compliance, truly irrevocable deletion can be achieved by utilizing history independent data structures to organize data (Section 4).

However, in order to architect systems with history independent characteristics and to prove history independence, we need a formal notion of data structures, of data structure states, and of history independence itself. In this paper we first formalize all necessary concepts and understand history independence from a theoretical perspective (Sections 3 - 7). Then, in Section 8 we use the theoretical results to architect a history independent file system (HIFS).

2 A QUICK INFORMAL LOOK AT HISTORY INDEPENDENCE

History independence is concerned with the historical information preserved within data structure states. The preserved history may be illicitly used by
adversaries to violate regulatory compliance. For example, an adversary may breach data retention laws by recovering deleted data. Therefore, to understand history independence, we need to specify what we mean by state, what we mean by history, and what an adversary can do.

What is state?  
A data structure’s state is an organization of data on a physical medium such as memory or disk.

What is history?  
History is the sequence of operations that led to the current data structure state.

What is the threat?  
For many existing data structures, the current state is a function of both data and history [1]. Hence, by analyzing the current state an adversary can derive the state’s history. Depending on the application the historical information includes the following:

- Evidence of past existence of delete data [12].
- The order in which votes were cast in a voting application [2], [3].
- The intermediate versions of a published document [2].

To illustrate, consider the sample hash table data structure of Figure 1. The sample hash table organizes the same data set differently depending on the sequence of operations used. Hence, an adversary that looks at the system memory can potentially detect which operation sequence was used to get to the current hash table state.

What is history independence?  
History independence is a characteristic of a data structure. A data structure is said to be history independent if from the adversary’s point of view, the current data structure state is a function of data only and not of history. Thus, the current state of a history independent data structure reveals no information to the adversary about its history other than what is inherently visible from the data itself.

We emphasize that history independence is concerned with historical information that is revealed from data organization and not from the data. In our hash table example of Figure 1, the fact that values \{3,6,9\} were inserted in the past is evident no matter how the data is organized. The data organization reveals the order of insertion.

Are there different kinds of history independence?  
Naor et al. [13] introduced two notions of history independence – weak history independence (WHI) and strong history independence (SHI).

WHI and SHI differ in the number of data structure states an adversary is permitted to observe. Under WHI, an adversary is permitted to observe only the current data structure state. For example, as in case of a stolen laptop. Under SHI, an adversary is permitted several observations of data structure states throughout a sequence of operations. For example, as in case of an insider adversary who can obtain a periodic memory dump. For SHI, the adversary should be unable to identify which sequence of operations was applied between any two adjacent observations.

How does history independence achieve regulatory compliance?  
The current state of a history independent data structure is a function of current data only. Data that was deleted in the past leaves no effect on the current state that an adversary can detect. History independent data structures are therefore ideal to organize data in compliance with data retention Regulations [7], [8], [9] that require truly irrecoverable data erasure.

2.1 Our Contributions

Weak history independence (WHI) and strong history independence (SHI) are the two existing notions of history independence. WHI assumes a rather weak adversary. SHI on the other hand is a very powerful notion of history independence, secure even against a computationally unbounded adversary [1].

Currently, applications are restricted to using data structures with either WHI or SHI characteristics. However, applications that do not fit into either WHI or SHI do exist. For example, a journaling system that reveals no historical information other than the last \(k\) operations\(^1\). Further, WHI does not protect against insider adversaries and SHI results in inefficiency [14]. Hence, there is a necessity for new notions of history independence targeted towards specific application scenarios.

In this paper we take the first steps towards better understanding the history independence spectrum and its applicability to systems. Following are the contributions of this paper:

- The exploration of abstract data types, data structures, machine models, and memory representations

\(^1\) We give additional examples in Section 5.1.
is a set of inputs; 

\( A \) is a set of operations; each operation maps the current state to a new state. 

\( \Gamma \) is a set of inputs; and each operation \( o \in \mathcal{O} \) is a function \( o : S \times \Gamma_o \rightarrow S \times \Psi_o \), where \( \Gamma_o \subseteq \Gamma \) and \( \Psi_o \subseteq \Psi \).

The ADT is initialized to state \( s_\phi \). When an operation \( o \in \mathcal{O} \) with input \( i \in \Gamma_o \) is applied to an ADT state \( s_1 \), the ADT outputs \( \tau \in \Psi_o \) and transitions to a state \( s_2 \). The transition from state \( s_1 \) to \( s_1 \) is denoted as \( o(s_1, i) \rightarrow (s_2, \tau) \).

### 3.1 Abstract Data Type (ADT)

The specification of data organization techniques is often done via abstract data types. The key characteristic of an ADT is that it specifies operations independently of any specific implementation. We build on the ADT concept proposed by Golovin et al. [1], wherein an ADT is considered as a set of states together with a set of operations. Each operation maps the current state to a new state.

**Definition 1. Abstract Data Type (ADT)**

An ADT \( A \) is a pentuple \( (S, s_\phi, \mathcal{O}, \Gamma, \Psi) \), where \( S \) is a set of states; \( s_\phi \in S \) is the initial state; \( \mathcal{O} \) is a set of operations; \( \Gamma \) is a set of inputs; \( \Psi \) is a set of outputs; and each operation

3.2 Models of Execution

An ADT is only a specification of operations for organizing data. For more practical use, such as for efficiency analysis, concrete implementations of the ADT operations are required. ADT implementations are provided via programs that can be executed on a given machine model. We refer to an ADT’s implementation in a given machine model as a data structure (Section 3.3).

Several machine models have been proposed [15], such as logic circuits, machines with memory, and combinatorial circuits. We focus on the RAM model of execution since we are concerned with history independent characteristics of complex software applications. Software applications such as databases, and file systems rely on data organization within the storage sub-systems.
of modern computers. The sub-systems can be accurately modelled using the RAM execution model.

3.2.1 RAM Model of Execution

The RAM model of execution models a traditional serial computer. The model consists of two components, a central processing unit (CPU) and a random access memory (RAM). Both the CPU and RAM are finite state machines (FSM) [15].

The RAM consists of $m = 2^n$ storage locations. Each location is a $b$-bit word and has a unique $\log_2 m$ bit address associated with it. Two operations are permitted on a storage location in the RAM. First, a load operation to access the $b$-bit word stored at the location. Second, a store operation that copies a given $b$-bit word to the location. Typically, the $b$-bit words are copied to or copied from CPU registers.

The CPU consists of $n$ $b$-bit registers and operates on a fetch-and-execute cycle [15]. The CPU has an associated set of instructions that it can perform. CPU instructions are specified in a programming language. A program in a RAM model is a finite sequence of programming language instructions.

A machine model can itself be considered as an ADT [1]. In this case, the set of ADT states is the set of all machine states, and the set of ADT operations is the set of all machine programs. For the RAM model, the set of ADT states, the set of inputs, and the set of outputs are all represented as bit strings.

Definition 2. Bounded RAM Machine Model

A bounded RAM machine model $\mathcal{M}$ with $m$ $b$-bit memory words and $n$ $b$-bit CPU registers is a quintuple $(S, s_0, P, \Gamma, \Psi)$, where $S = \{0, 1\}^{b(m+n)}$ is the set of machine states; $s_0 \in S$ is the initial state; $P$ is the set of all programs of $\mathcal{M}$; $\Gamma = \{0, 1\}^* \times P$ is a set of inputs; $\Psi = \{0, 1\}^* \times P$ is a set of outputs; and each program $p \in P$ is a function $p : S \times \Gamma_p \rightarrow S \times \Psi_p$, where $\Gamma_p \subseteq \Gamma$ and $\Psi_p \subseteq \Psi$.

$\mathcal{M}$ is initialized to state $s_0$. If a program $p \in P$ with input $i \in \Gamma_p$ is executed by the CPU when $\mathcal{M}$ is in state $s_1$, $\mathcal{M}$ outputs $\tau \in \Psi_p$ and transitions to a state $s_2$. The transition from state $s_1$ to $s_2$ is denoted as $p(s_1, i) \rightarrow (s_2, \tau)$.

3.3 Data Structure

In the previous section, we hinted that a data structure is an ADT’s implementation in a specific machine model. Now that we have defined both ADT and the RAM machine model we can formalize the data structure.

An implementation for an ADT in a given machine model is obtained as follows.

- A machine representation is chosen for each ADT input and output.
- For each ADT operation a machine program is selected that provides the functionality desired from the ADT operation.
- A unique machine state is selected to represent the initial ADT state.

We encapsulate the above steps in the following data structure definition.

Definition 3. Data Structure

A data structure implementation of an ADT $\mathcal{A}$ is a quadruple $(\alpha, \beta, \gamma, s_0^M)$, where $\mathcal{A} = (S, s_0, O, \Gamma, \Psi)$ as per definition 2, $\mathcal{M} = (S^M, s_0^M, \mathcal{P}, \Gamma^M, \Psi^M)$ as per definition 2, $\alpha : \mathcal{M} \rightarrow \Gamma^M$, $\beta : \mathcal{P}^M \rightarrow \Psi^M$, $\gamma : O \rightarrow \mathcal{P}^M$, $s_0^M \in S^M$, $\Gamma^M \subseteq \Gamma$ and $\Psi^M \subseteq \Psi$.

$\alpha$ is a mapping from ADT inputs to machine inputs. That is, for any ADT input $i$, $\alpha(i)$ is the machine representation of the input. Similarly, $\beta$ is the mapping from ADT outputs to machine outputs. $\gamma$ is the mapping from ADT operations to machine programs. For an ADT operation $o$, $\gamma(o)$ is the machine program implementing $o$. Finally, just as the ADT $\mathcal{A}$ is initialized to a unique state $s_0$, a unique machine state $s_0^M$ is selected to represent the initial data structure state.

3.3.1 Example: Hash table as an ADT and its data structure

To clarify the concepts of ADT and data structure, we use the example of a hash table. First, we define a hash table ADT. Then, we describe a data structure implementation of the hash table ADT.

Let $H = (S, s_0, O, \Gamma, \Psi)$ be a hash table ADT, where

- $S = 2^{b \times n}$ is the set of states\(^4\).
- $s_0 = \emptyset$ is the initial state.
- $\Gamma = \mathbb{N} \cup (\mathbb{N} \times \mathbb{N})$ is the set of inputs.
- $\Psi = \mathbb{N} \cup \{ERROR, SUCCESS\}$ is the set of outputs.
- The set of operations $O = \{\text{insert, search, delete}\}$, such that
  - insert : $S \times \mathbb{N} \times \mathbb{N} \rightarrow S \times \{ERROR, SUCCESS\}$
  - search : $S \times \mathbb{N} \rightarrow S \times (\mathbb{N} \cup \{ERROR\})$
  - delete : $S \times \mathbb{N} \rightarrow S \times \{ERROR, SUCCESS\}$

An implementation of the above hash table ADT in the RAM model, that is, a data structure $D = (\alpha, \beta, \gamma, s_0^M)$ can be obtained as follows.

- For all $n \in \mathbb{N}_b$, $\alpha(n) = \{0, 1\}^b$. Here, $\mathbb{N}_b = \{x \mid x \in \mathbb{N} \text{ and } x \leq 2^b\}$, $b$ is the machine word length, and $\alpha(n)$ is the bit string representing $n$. For all $(n_1, n_2) \in \mathbb{N}_b \times \mathbb{N}_b$, $\alpha((n_1, n_2)) = \alpha(n_1) || \alpha(n_2)$.
- For all $n \in \mathbb{N}_b$, $\beta(n) = \alpha(n)$. $SUCCESS$ is represented by $\{0\}^b$, and $ERROR$ is represented by $\{1\}^b$.
- $\gamma : O \rightarrow \mathcal{P}^M$. A machine program is provided for each hash table ADT operation. For example, implementation of the insert, search and delete algorithms of an array-based hash table using linear probing [11].
- The initial machine state $s_0^M$ corresponding to the initial ADT state $s_0$ is obtained by first loading all

---

\(^4\) This a bounded-memory RAM.

\(^5\) $2^A$ denotes the powerset of set $A$. 

machine programs implementing the ADT operations into memory and setting the memory locations reserved for the hash table to zero.

Note that the above data structure $D$ is one possible implementation of the hash table ADT. Several other implementations are possible. In general, the same ADT can have several data structure implementations.

**Abstract Data Types And Type Theory:** The above hash table ADT definition assumes a hash table over natural numbers. In general, the hash table ADT can be defined over any type, such as real numbers, bit strings, or be composed from other basic types. In type theory [16], a type is defined as a set of values that share a common logical property or attribute. It is beyond the scope of this dissertation to formalize the notion of types. Instead, we refer the reader to relevant notes on type theory [16].

### 3.3.2 Data Structure State

A data structure state is a machine state. The set of all data structure states consists of all machine states that are reachable from the initial data structure state via execution of machine programs implementing the ADT operations.

### 3.3.3 State Transition Graph For Data Structure

In Section 3.1, we introduced the state transition graph for an ADT. Similarly, we can view a data structure in terms of a state transition graph. Graph-based view of a data structure helps to identify the machine states that constitute the set of data structure states, to precisely define the relationship between ADT states and data structure states (Section 3.5), and to understand history independence (Section 3.4).

A data structure can be considered to be a directed graph $G$, where each vertex represents a data structure state and each edge is labeled with a machine program implementing an ADT operation along with a machine input and a machine output. The label for an edge between two vertices represents the program that causes the transition between the corresponding states. We call the graph $G$, the state transition graph of the data structure.

### 3.4 A Semi-Formal Look At History Independence

Equipped with the concepts of ADT (Section 3.1), RAM machine model (Section 3.2), data structure (Section 3.3), and state transition graphs, we can gain a deeper insight into history independence. Hence we pause do so here. Later, in Section 4 we formalize history independence.

#### 3.4.1 The non-isomorphism problem

In Section 2 we introduced the two existing history independence notions – weak history independence (WHI) and strong history independence (SHI)\(^6\).

---

6. Both WHI and SHI are formalized in Section 4.

---

### Table 1

| Path | From Figure |
|------|-------------|
| $p_A = s_0 \rightarrow \{1\} \rightarrow \{1,3\} \rightarrow \{1,3,6\}$ | 2(a) |
| $p'_A = s_0 \rightarrow \{1\} \rightarrow \{1,3\} \rightarrow \{1,3,6\}$ | 2(a) |
| $pp = s_0^M \rightarrow << 3,1,6 >> \rightarrow << 3,1,6 >> \rightarrow << 3,1,6 >>$ | 2(b) |
| $pp' = s_0^M \rightarrow << 3,1,6 >> \rightarrow << 6,1,3 >> \rightarrow << 6,1,3 >>$ | 2(b) |

Non-isomorphism between the state transition graph of an ADT and of its data structure implementation breaks SHI. WHI on the other hand can be achieved even when the ADT and data structure state transition graphs are non-isomorphic. First, we look at how non-isomorphism breaks SHI and then we discuss how to achieve WHI in the presence of non-isomorphism.

#### 3.4.2 Why non-isomorphism breaks SHI?

Non-isomorphism and thus the need for SHI arises when an ADT state has multiple memory representations\(^7\). We will precisely define memory representations for ADT states in Section 3.5. For now, it suffices to say the following: A memory representation for an ADT state that is reachable from the initial ADT state via a sequence of ADT operations is the machine state reachable from the initial data structure state via the corresponding program sequence. For example, in Figure 2, the data structure states $<< 3,1,6 >>$ and $<< 6,1,3 >>$ are memory representations of the ADT state $\{1,3,6\}$.

To illustrate how non-isomorphism breaks SHI, consider the example graphs from Figure 2, example paths from Table 1, and an adversary with access to the following: the initial ADT state $s_0$, the initial data structure state $s_0^M$, the current ADT state $\{1,3,6\}$, and the current data structure state which is either $<< 3,1,6 >>$ or $<< 6,1,3 >>$.

By looking at the ADT states alone, the adversary cannot determine which sequence of ADT operations was used to arrive at the current ADT state $\{1,3,6\}$. This is because there are two paths $p_A$ and $p'_A$ between the ADT states $s_0$ and $\{1,3,6\}$. Moreover, the ADT states carry no information about the exact path used to transition from $s_0$ to $\{1,3,6\}$. Hence, the data alone gives the adversary no advantage in guessing which sequence of ADT operations was applied in the past.

Now, by looking at the current data structure state, the adversary can clearly identify which sequence of machine programs was used to arrive at the current data structure state. The current data structure state is either $<< 3,1,6 >>$ or $<< 6,1,3 >>$. There is

---

7. Many existing data structures have this property and are hence, not history independent. Common examples include the linked list, hash tables and B-Trees. In these data structures different insertion order of the same set of data elements (i.e., the same ADT state) results in different memory representations.
Fig. 2. Example of non-isomorphism between ADT and data structure state transition graphs. (a) Partial state transition graph for sample hash table ADT. (b) Partial state transition graph for sample array-based hash table data structure implementation using linear probing. Number of hash table buckets is 3 and the hash function is \( h(\text{key}) = \text{key} \mod 3 \). \( \gamma(\text{insert}), \gamma(\text{search}) \) and \( \gamma(\text{delete}) \) denote the machine programs implementing the ADT operations insert, search and delete, respectively. \( o(i)/t \) denotes that ADT operation \( o \) takes input \( i \) and produces output \( t \). Similarly, \( \gamma(o)\)(\( \alpha(i) \))/\( \beta(t) \) denotes that program \( \gamma(o) \) takes input \( \alpha(i) \) and produces output \( \beta(t) \). \( \alpha(i) \) and \( \beta(t) \) are the machine representations of the ADT input \( i \) and ADT output \( t \), respectively. Note that the vertices in figure (b) represent data structure states. In the RAM model these will be bit strings. However, to convey data semantics we denote the hash table array as \( <\_a_0, a_1, a_2> \), where \( a_0, a_1, \) and \( a_2 \) are elements at buckets 0, 1 and 2, respectively. Underscore denotes an empty bucket. Highlighted paths are referenced in Table 1, and in Section 3.4.

Fig. 3. Using randomization to achieve history independence. The dotted lines indicate new transitions added to the hash table data structure state transition graph. Amongst all edges with the same starting node and the same label, the choice of edge for state transition is made at random.

a unique path from initial data structure state \( s_0^M \) to each of the states \( <3,1,6> \) and \( <6,1,3> \). Hence, by observing the current data structure state, the adversary can identify whether path \( p_D \) or path \( p_D' \) was used to transition from state \( s_0^M \) to the current data structure state. Identification of the path in the data structure state transition graph informs the adversary of the program sequence used. Knowledge of the program sequence used in-turn tells the adversary the sequence of ADT operations used. In conclusion, the data structure implementation gives the adversary an advantage in guessing the history of past execution, thereby breaking history independence.

3.4.3 How can we achieve history independence?

Currently, there are two known ways to make data structures history independent.

1) For SHI, make the ADT and the data structure state transition graphs isomorphic:

Data structures with state transition graphs isomorphic to their ADT’s state transition graph are referred to as canonically represented data structures. We discuss the necessity of canonical representations for SHI in Section 4.4. SHI implies WHI.

2) For WHI, make the data structure state transitions randomized:

Randomization here refers to the selection of the data structure state representing the corresponding ADT state. To illustrate, consider the example graphs from Figure 2. Both data structure states \( <3,1,6> \) and \( <6,1,3> \) are valid memory representations of the ADT state \( \{1,3,6\} \). For WHI, the choice between data structure states \( <3,1,6> \) and \( <6,1,3> \) to represent the ADT state \( \{1,3,6\} \) must be random.

As shown in Figure 3, randomization translates to addition of new paths in the data structure state transition graph to ensure the following: For any two ADT states \( s_0 \) and \( s_1 \), if there is a path in the ADT state transition graph between \( s_0 \) and \( s_1 \), then, there must be a path from all memory representations of ADT state \( s_0 \) to all memory representations of ADT state \( s_1 \) in the data structure’s state transition graph. The choice of path in the data structure state transition graph between representations of ADT states \( s_0 \) and \( s_1 \) is then made at random.

From the adversary’s point of view randomization
makes all memory representations of an ADT state equally likely to occur. Hence, observation of a specific representation gives the adversary no advantage in guessing the sequence of machine programs that led to the current data structure state. Since the adversary cannot identify the sequence of machine programs used, the adversary is also unable to identify the sequence of ADT operations that led to the current ADT state.

3.5 Memory Representations

The last concept that remains to be formalized before we move on to formal definitions for history independence (Section 4) is that of memory representations.

In the discussion of nonisomorphism and history independence above, we informally introduced memory representations for ADT states. We also showed that history independence comes into picture when an ADT operation sequence produced by the final operation in sequence on a machine state results in state produced by the initial data structure state; $I_1, I_2, ..., I_n$ are sequences of ADT inputs; $\delta_1, \delta_2, ..., \delta_n$ are ADT operation sequences, each of which when applied to the initial ADT state results in state $s = \chi(\delta_k)\tau_k$ denotes the program sequence corresponding to ADT operation sequence $\delta_k$; $|I_k| = |\delta_k|, 1 \leq k \leq n$.

Here $m$ is the mapping $m : S \rightarrow 2^S$, where $S$ is the set of all ADT states, $S^D$ is the set of all data structure states, and $2^S$ denotes the power set of $S^D$.

3.5.1 Dealing With Infinite ADT State Space

The set of machine states for the bounded RAM model is finite since there are finite number of available bits. Hence, a data structure implementation on a bounded RAM model can only have a finite number of data structure states. The set of ADT states on the other hand can be infinite. For an ADT with infinite states, a data structure implementation will be unable to uniquely represent all the ADT states. The case of infinite ADT states is of particular importance for canonically represented data structures that require the state transition graphs of the ADT and of the data structure to be isomorphic, that is, each ADT state has a unique memory representation.

We will look at canonical representations in detail within the context of history independence in Section 4.4. Here, we list two work-arounds to dealing with infinite ADT state space.

1) Redefine the ADT, such that the number of ADT states is less than or equal to the number of machine states.

2) Design each machine program implementing an ADT operation, such that the program produces a special output when an ADT state cannot be represented using the available machine bits. For example, an out-of-memory error.

4 History Independence

Now that we are equipped with the necessary concepts (ADT, RAM machine model, data structure, and memory representations), we proceed to formalize history independence. We give new game-based definitions for both WHI and SHI (Sections 4.1 and 4.1). The new definitions are equivalent to existing proposals [2], [6] but more appropriate for the security community since they follow the game-based construction of semantic
security. Further, our new definitions naturally extend to accommodate other notions of history independence beyond WHI and SHI.

We generalize history independence by introducing Δ history independence (ΔHI), a generic game-based definition of history independence that is malleable enough to accommodate WHI, SHI, and a broad spectrum of new history independence notions. Using ΔHI, we define new practical notions of history independence and also cover both WHI and SHI (Section 5.1). Finally, we show how ΔHI helps to reason about the history preserved or hidden by data structures including ones that were designed without history independence in mind (Sections 5.2 and 5.3).

4.1 Weak History Independence (WHI)

WHI was introduced for scenarios wherein an adversary observes only the current data structure state. For example, as in the case of a stolen laptop. The current data structure state is the memory representation of the current ADT state. WHI then requires that observation of the current data structure state reveals no additional historical information to the adversary other than what is inherently available from the current ADT state.

Informally, a data structure is said to be weakly history independent if for any two sequences of ADT operations \(\delta_1\) and \(\delta_2\), that take the ADT from initialization to a state \(s\), observation of any memory representation of state \(s\) gives the adversary no advantage in guessing whether sequence \(\delta_1\) or \(\delta_2\) was used to get to \(s\).

We define weak history independence (WHI) by the following game:

Let \(\mathcal{A} = (\mathcal{S}, s_{\phi}, \mathcal{O}, \Gamma, \Psi)\) be an ADT, \(\mathcal{M} = (\mathcal{S}^M, s_{\phi}^M, \mathcal{P}^M, \Gamma^M, \Psi^M)\) be a bounded RAM machine model, and \(\mathcal{D} = (\alpha, \beta, \gamma, s_0^M)\) be a data structure implementing \(\mathcal{A}\) in \(\mathcal{M}\) as per definitions 1, 2 and 3, respectively.

1) A probabilistic polynomial time-bounded adversary selects the following: An ADT state \(s\); two sequences of ADT operations \(\delta_0\) and \(\delta_1\); and two sequences of ADT inputs \(I_0\) and \(I_1\); such that \(\mathcal{O}(\delta_0, s_{\phi}, I_0) \rightarrow (s, \tau)\) and \(\mathcal{O}(\delta_1, s_{\phi}, I_1) \rightarrow (s, \tau)\). Both \(\delta_1\) and \(\delta_2\) take the ADT from the initial state \(s_{\phi}\) to state \(s\) producing the same output \(\tau\).

2) The adversary sends \(s\), \(\delta_0\), \(\delta_1\), \(I_0\) and \(I_1\) to the challenger.

3) The challenger flips a fair coin \(c \in \{0, 1\}\) and computes \(\mathcal{O}^M(\delta_c^M, s_0^M, I_c) \rightarrow (s^M, \tau^M)\), where \(\delta_c^M = \chi(\delta_c)\) and \(\tau^M = \beta(\tau)\). That is, the challenger applies the program sequence \(\delta_c^M\) corresponding to the ADT operation sequence \(\delta_c\) to the data structure initialization state \(s_0^M\), resulting in a memory representation \(s^M\) of ADT state \(s\) and a machine output \(\tau^M\).

4) The challenger sends the memory representation \(s^M\) to the adversary.

5) The adversary outputs \(c' \in \{0, 1\}\).

The adversary wins the game if \(c' = c\).

\(\mathcal{D}\) is said to be weakly history independent if the probability of the adversary winning the game is negligible (where “negligible” is defined over any implementation-specific security parameters of the programs in \(\mathcal{P}^M\)).

Since WHI permits the adversary to make a single observation, the adversary is allowed to choose the end state only in step 1. The starting state for the chosen ADT operation sequences is always the initial ADT state \(s_{\phi}\). Recall from the data structure definition (Section 3.3) that the initial ADT state has a fixed memory representation, which is the initial data structure state \(s_0^M\). Hence, in step 3, the challenger applies the adversary-selected sequence to the memory representation \(s_0^M\) of \(s_{\phi}\).

If the adversary is able to identify the ADT operation sequence chosen by the challenger in step 3, then the adversary wins the game. Winning the game implies the adversary was able to determine the operation sequence that led to the current ADT state by observing the state’s memory representation, thereby breaking WHI.

4.2 Strong History Independence (SHI)

Unlike WHI, SHI is applicable when an adversary can observe multiple memory representations throughout a sequence of operations. For example, as in case of an insider who can obtain a periodic memory dump. SHI requires that the adversary must not gain any additional information about the sequence of operations between any two adjacent observations than what is inherently available from the corresponding ADT states.

Informally, a data structure is said to be strongly history independent if for any two sequences of ADT operations \(\delta_1\) and \(\delta_2\), that take the ADT from a state \(s_1\) to a state \(s_2\), observations of any memory representations of states \(s_1\) and \(s_2\) give the adversary no advantage in guessing whether sequence \(\delta_1\) or \(\delta_2\) was used to go from \(s_1\) to \(s_2\).

We define strong history independence (SHI) by the following game:

Let \(\mathcal{A} = (\mathcal{S}, s_{\phi}, \mathcal{O}, \Gamma, \Psi)\) be an ADT, \(\mathcal{M} = (\mathcal{S}^M, s_{\phi}^M, \mathcal{P}^M, \Gamma^M, \Psi^M)\) be a bounded RAM machine model, and \(\mathcal{D} = (\alpha, \beta, \gamma, s_0^M)\) be a data structure implementing \(\mathcal{A}\) in \(\mathcal{M}\) as per definitions 1, 2 and 3 respectively.

11. For example PRNG seeds when using randomization, or keys when using encryption.
1) A probabilistic polynomial time-bounded adversary selects the following.
   • Two ADT states $s_1$ and $s_2$; two sequences of ADT operations $\delta_0$ and $\delta_1$; and two sequences of ADT inputs $I_0$ and $I_1$; such that $\mathcal{O}(\delta_0, s_1, I_0) \rightarrow (s_2, \tau)$ and $\mathcal{O}(\delta_1, s_1, I_1) \rightarrow (s_2, \tau)$. Both $\delta_1$ and $\delta_2$ take the ADT from state $s_1$ to state $s_2$ producing the same output $\tau$.
   • A memory representation $s_1^M$ of ADT state $s_1$.

2) The adversary sends $s_1, s_1^M, \delta_0, \delta_1, I_0$ and $I_1$ to the challenger.

3) The challenger flips a fair coin $c \in \{0, 1\}$ and computes $\mathcal{O}^M(\delta_1^M, s_1^M, I_c) \rightarrow (s_2^M, \tau^M)$, where $\delta_1^M = \chi(\delta_1)$ and $\tau^M = \beta(\tau)$. That is, the challenger applies the program sequence $\delta_1^M$ corresponding to the ADT operation sequence $\delta_1$ to the data structure state $s_1^M$, resulting in a memory representation $s_2^M$ of state $s_2$ and a machine output $\tau^M$.

4) The challenger sends the memory representation $s_2^M$ to the adversary.

5) The adversary outputs $c' \in \{0, 1\}$.

The adversary wins the game if $c' = c$.

$\mathcal{D}$ is said to be strongly history independent if the probability of the adversary winning the game is negligible (where “negligible” is defined over any implementation-specific security parameters of the programs in $\mathcal{P}^M$).

Winning the game means that the adversary was able to determine the operation sequence that took the ADT from state $s_1$ to state $s_2$, thereby breaking SHI.

SHI implies WHI. If the ADT state $s_1$ chosen by the adversary in step 1 is the initial ADT state $s_{\phi}$, then the SHI game reduces to the WHI game of Section 4.1.

4.3 Equivalence to Existing History Independence Definitions

WHI and SHI were first introduced by Naor et al. [13]. Later, Hartline et al. [6] introduced new definitions for WHI and SHI. However, Hartline et al. showed that their definitions although less complex are equivalent to the ones proposed by Naor et al. Our game-based definitions of WHI and SHI (Sections 4.1 and 4.2) differ slightly from the definitions by Hartline et al. Specifically, Hartline et al. assume a computationally unbounded adversary. We address history independence in the presence of computationally bounded adversaries to be more inline with reality. Further, new definitions were necessary to overcome impreciseness in existing definitions and to develop a framework for new history independence notions beyond WHI and SHI. We detail in the following.

Hartline et al. defined weak history independence as follows.

**Definition 5. Weak History Independence (WHI)**

A data structure implementation is weakly history independent if, for any two sequences of operations $X$ and $Y$ that take the data structure from initialization to state $A$, the distribution over memory after $X$ is performed is identical to the distribution after $Y$. That is:

$$ (\phi \xrightarrow{X} A) \land (\phi \xrightarrow{Y} A) \implies \forall a \in A, Pr[\phi \xrightarrow{X} a] = Pr[\phi \xrightarrow{Y} a] $$

In the above definition, $\phi \xrightarrow{X} B$ denotes that a operation sequence $X$ when applied to the initial state $\phi$, results in state $A$. The notation $a \in A$ means that $a$ is a memory representation of state $A$. \( Pr[\phi \xrightarrow{X} a] \) denotes the probability that a sequence $X$ when applied to initial state $\phi$, results in representation $a$.

Reconciling terminology

Hartline et al. do not formalize the concepts of data structure, data structure state and memory representations. A data structure’s state is referred to as the data structure’s content. Memory representation of a data structure state is the physical contents of memory that represent that state. We note that Naor et al. also used the same terminology in their definitions.

The WHI definition by Hartline et al. is imprecise in the following.

• Operation inputs and outputs are not considered.
• The same operation sequences are considered applicable to both data structure states and to memory representations. The mechanisms for the applicability are not specified.
• The connection between a data structure’s state and the state’s memory representations is not precisely specified.

Following Golovin et al. [1] we use the ADT concept to model logical states (or content) and define a data structure as an ADT’s implementation (Sections 3.1 - 3.3). A data structure state is therefore the memory representation of an ADT state. Separating ADT and data structure concepts enables us to precisely define memory representations (Section 3.5) for various machine models; understand history independence from the perspective of state transition graphs; and to build a framework for defining new history independence notions other than SHI and WHI (Section 5).

To summarize the differences in terminology, what Hartline et al. refer to as data structure state in definition 6 is an ADT state in our model. Further, we refer to a memory representation in definition 6 as a data structure state.

For WHI, Hartline et al. require a data structure implementation to satisfy the following:

$$ (\phi \xrightarrow{X} A) \land (\phi \xrightarrow{Y} A) \implies \forall a \in A, Pr[\phi \xrightarrow{X} a] = Pr[\phi \xrightarrow{Y} a] $$

Our game-based definition of WHI poses the following slightly relaxed requirement:
(φ \xrightarrow{X} A) \land (φ \xrightarrow{Y} A) \implies \forall \mathbf{a} \in A, |Pr[φ \xrightarrow{X} \mathbf{a}] - Pr[φ \xrightarrow{Y} \mathbf{a}]| \text{ is negligible}

We will show that the game-based WHI definition (Section 4.1) is equivalent to statement 4.3, that is, a data structure preserves WHI only if statement 4.3 is true. However, before we show the equivalence we point out the necessity for the difference between conditions 4.3 and 4.3. As discussed in Section 3.4, there are two known ways to achieve history independence. The first way is to make the ADT and the initial state graph isomorphic. The second way is to make the data structure state transition graph random. The requirement for identical memory distributions as per statement 4.3 rules out the use of randomization to achieve history independence. A randomized data structure implementation will rely on pseudo random generators. Therefore, the relaxed requirement of negligibility introduced in statement 4.3 is in fact not a limitation, but rather a reconciliation of the definition by Hartline et al. with reality where we have computationally bounded adversaries.

Although Naor et al. proposed a WHI definition that requires identical distributions, they also used randomization to design a history independent data structure.

**Equivalence of WHI definitions**

We now show that our gamed-based WHI definition (Section 4.1) is equivalent to a WHI definition based on statement 4.3.

We rewrite statement 4.3 for consistent notations as follows.

\[(s_{\phi} \xrightarrow{\delta_{0}} s) \land (s_{\phi} \xrightarrow{\delta_{1}} s) \implies \forall s^{M} \in s, |Pr[s^{M}_{\phi} \xrightarrow{\delta^{M}_{0}} s^{M}] - Pr[s^{M}_{\phi} \xrightarrow{\delta^{M}_{1}} s^{M}]| \text{ is negligible}\]

Here, \( \delta_{0} \) and \( \delta_{1} \) are two ADT operation sequences that take the ADT initial state \( s_{\phi} \) to state \( s \). \( s_{\phi} \) and \( s^{M} \) are the initial ADT and initial data structure states, respectively. \( \delta^{M}_{0} \) and \( \delta^{M}_{1} \) are the machine programs corresponding to ADT operation sequences \( \delta_{0} \) and \( \delta_{1} \), respectively.

History independence only considers cases where the condition \((s_{\phi} \xrightarrow{\delta_{0}} s) \land (s_{\phi} \xrightarrow{\delta_{1}} s)\) is true, that is, both sequences \( \delta_{0} \) and \( \delta_{1} \) take the ADT to the same end state \( s \). Otherwise, the ADT states themselves reveal history.

We therefore have two cases to consider.

**Case 1:** The distributions are computationally distinguishable, that is,

\[\exists s^{M} \in s \text{ such that } |Pr[s^{M}_{\phi} \xrightarrow{\delta^{M}_{0}} s^{M}] - Pr[s^{M}_{\phi} \xrightarrow{\delta^{M}_{1}} s^{M}]| \text{ is non-negligible.}\]

Now consider the following adversarial strategy. Given a data structure state \( s^{M} \) in step 4 of the WHI game, the adversary outputs \( c \) such that \( \delta^{M}_{c} \) has a higher probability of producing \( s^{M} \). For such an adversarial strategy \( |Pr[c' = c] - \frac{1}{2}| \) is non-negligible for some \( s^{M} \).

Therefore, the data structure implementation does not preserve WHI.

**Case 2:** The distributions are computationally indistinguishable, that is,

\[\forall s^{M} \in s, Pr[s^{M}_{\phi} \xrightarrow{\delta^{M}_{0}} s^{M}] - Pr[s^{M}_{\phi} \xrightarrow{\delta^{M}_{1}} s^{M}]\text{ is negligible}\]

In this case, from a computationally bounded adversary’s perspective, the representation \( s^{M} \) received in step 4 of the WHI game is equally likely to have been produced by either \( \delta^{M}_{0} \) or \( \delta^{M}_{1} \). Hence, observation of a data structure state gives the adversary a negligible advantage in guessing \( c \). The data structure implementation therefore preserves WHI.

**Equivalence of SHI definitions**

For strong history independence Hartline et al. proposed the following definition.

**Definition 6.** Strong History Independence (SHI)

A data structure implementation is strongly history independent if, for any two (possibly empty) sequences of operations \( X \) and \( Y \) that take a data structure in state \( A \) to state \( B \), the distribution over representations of \( B \) after \( X \) is performed on a representation \( a \) is identical to the distribution after \( Y \) is performed on \( a \). That is:

\[(A \xrightarrow{X} B) \land (A \xrightarrow{Y} B) \implies \forall a \in A, \forall b \in B, Pr[a \xrightarrow{X} b] = Pr[a \xrightarrow{Y} b]\]

In the above definition, \( A \xrightarrow{X} B \) denotes that a operation sequence \( X \) when applied to state \( A \), results in state \( B \). The notation \( a \in A \) means that \( a \) is a memory representation of state \( A \). \( Pr[a \xrightarrow{X} b] \) denotes the probability that a sequence \( X \) when applied to memory representation \( a \), results in representation \( b \).

Similar to the case for WHI, our game-based SHI definition (Section 4.2) differs from the above definition only by relaxing the requirement for identical distributions. That is, for SHI, we require the following:

\[(A \xrightarrow{X} B) \land (A \xrightarrow{Y} B) \implies \forall a \in A, \forall b \in B, |Pr[a \xrightarrow{X} b] - Pr[a \xrightarrow{Y} b]| \text{ is negligible}\]

The equivalence of SHI definitions follows similarly to the case of WHI.

**Summary of Differences**

The main differences between our definitions and the definitions by Hartline et al. are the following:

- The definitions by Hartline et al. are imprecise about the concepts of data structures, states, and memory representations. We precisely formalize all of these concepts.
Hartline et al. do not consider the case of computationally bounded adversaries. We permit computationally bounded adversaries and thus have the negligibility definition instead of equality for memory distributions.

4.4 Canonical Representations And History Independence

Canonically (or uniquely) represented data structures have the property that each ADT state has a unique memory representation. Unique representation implies that the ADT and data structure state transition graphs are isomorphic\(^\text{13}\). Canonically represented data structures give very strong guarantees for history independence and in many cases are the only way to achieve history independence.

We first define canonically represented data structures and then discuss several important results pertaining to canonical representations and history independence. We also summarize (Table 2) the scenarios where canonical representations are necessary for history independence across all combinations of types of programs, secrecy of random bits, adversarial computational ability, and the desired notion of history independence.

Definition 7. Canonically represented data structure

A data structure \(D\) implementing an ADT \(A\) on a bounded RAM machine model \(M\) is canonically represented if each ADT state has a unique memory representation, that is, the mapping \(m : S \rightarrow 2^{S^D}\) is injective and \(|m(s)| = 1\), where \(S\) is the set of all ADT states, \(S^D\) is the set of all data structure states, and \(m(s)\) denotes the set of memory representations of an ADT state \(s \in S\) as per definition 4.

4.4.1 Impossibility of canonical representations for ADTs with infinite states

In Section 3.5.1, we discussed how to handle the case when the set of ADT states is infinite. The case of infinite ADT states is of particular importance for canonically represented data structure implementations on a bounded RAM machine model. Since the bounded RAM machine model has a finite number of available bits, the machine state space is not large enough to provide a unique representation for each ADT state when the ADT state space is infinite. Impossibility of unique representations clearly suggests that canonical representations for infinite state set ADTs are not possible in practice since machines with infinite state space do not exists in reality. This straightforwardly leads to the following theorem.

Theorem 1. Canonically represented data structure implementations for ADTs with infinite states are impossible in practice.

However, prior work [1], [2], [13] has claimed designs for canonically represented data structures for the RAM model in direct contradiction to Theorem 1. The contradiction arises from the fact that prior work has implicitly considered ADTs with finite state space. Specifically, the ADTs considered have have fewer states than the total number of machine states.

4.4.2 The necessity of canonical representations for SHI

Since history independence was first proposed [13], it has been known that canonically represented data structures support SHI. An interesting question posed in this context was whether canonical representations are necessary to achieve SHI. The question about the necessity of canonical representations for SHI was answered by Hartline et al. Hartline et al. [6] showed that SHI cannot be achieved without canonical representations.

Thus, we have the following theorem

Theorem 2. A data structure is strongly history independent if and only if it is canonically represented.

Proof. The proof by Hartline et al. [6] builds on the case that if a data structure is not canonically represented, then an adversary can distinguish an empty sequence of operations from a nonempty sequence of operations. In the context of our game based definition for SHI, we provide an equivalent proof for the same.

Consider an ADT \(A\) and a data structure \(D\) implementing \(A\) on a bounded RAM machine model \(M\). Also assume that \(D\) is not canonically represented. Now, Let \(Q\) be an adversary and \(C\) be the challenger in our game. \(Q\) selects the following

- Two ADT state \(s_1\) and \(s_2\).
- Two sequence of operations \(\delta_0\) and \(\delta_1\); and two sequences of ADT inputs \(I_0\) and \(I_1\), such that \(Q(\delta_0, s_1, I_0) \rightarrow (s_2, \tau)\) and \(Q(\delta_1, s_1, I_1) \rightarrow (s_2, \tau)\).

Let \(\alpha_1\) and \(\alpha_2\) be two distinct memory representations for ADT state \(s_1\). We show that, with this setup, the adversary \(Q\) can distinguish between an empty sequence of operations and a non empty sequence of operations. Consider that \(Q\) selects \(\delta_0\) to be an empty sequence of operation and \(\delta_1\) to be a non empty sequence of operations. In step 2 of the SHI game, the adversary sends \(s_1, \delta_0, \delta_1, I_0\) and \(I_1\) to \(C\). In step 3 of the SHI game, \(C\) flips a coin and applies either \(\delta_0\) or \(\delta_1\) to \(s_1\) and returns the memory representation of the output state to \(Q\). There are two possible cases for step 3:

1) \(C\) selects \(\delta_0\) – there is no change in the ADT state and the corresponding memory representation since an empty sequence of operations does not cause state changes. Hence in step 4, \(C\) returns \(\alpha_1\) to \(Q\).

2) \(C\) selects \(\delta_1\) – the final ADT state reached after performing all the operations in \(\delta_1\) is \(s_1\) but the memory representation for \(s_1\) in this case may be either \(\alpha_1\) or \(\alpha_2\). In step 4, if \(C\) returns \(\alpha_2\) to \(Q\), then \(Q\) can correctly predict with non-negligible probability that \(C\) has applied \(\delta_1\) on \(s_1\) to reach \(\alpha_2\). This breaks strong history independence for \(D\).

\(\Box\)

13. Isomorphism is discussed in Section 3.4.
### 4.4.3 Why canonical representations are not necessary for WHI?

In the absence of canonical representations, it has been shown that an adversary can distinguish an empty sequence of operations from a nonempty sequence of operations thereby breaking SHI [6]. If operation sequences are always assumed to be nonempty, canonical representations are not necessary [6]. We will define such a slightly relaxed notion of history independence that permits only nonempty sequences in Section 5.1. Here, we show that WHI is preserved even for empty operation sequences in the absence of canonical representations.

Consider the WHI game from Section 4.1. The case in which the adversary selects two empty ADT operation sequences in step 1 is trivial since empty sequences cause no state transitions and hence there is no history to be revealed.

Now, consider the case when the adversary selects an empty sequence $\delta_0$ and a nonempty sequence $\delta_1$ of ADT operations. Both $\delta_0$ and $\delta_1$ are required to take the ADT from the initial state to the same end state. Since the empty sequence $\delta_0$ causes no state transitions, end state for both sequences $\delta_0$ and $\delta_1$ will be the initial ADT state itself.

Then, in step 3, the challenger chooses either $\delta_0$ or $\delta_1$ and sends the resulting memory representation to the adversary. Since the end state for the two operation sequences is the initial ADT state, the memory representation sent to the adversary in step 4 will be the data structure initialization state. From the data structure definition (Section 3.3), we know that the initial ADT state has a corresponding fixed unique memory representation. Hence, irrespective of the nonempty sequence that the adversary selects in step 1, the adversary receives the initial ADT state’s memory representation in step 4. Since the adversary receives the same representation each time, the adversary gains no advantage in guessing whether $\delta_0$ or $\delta_1$ was chosen by the challenger in step 3.

ADT states other than the initial ADT state can have multiple memory representations. Multiple representations for ADT states does not break WHI as long it is ensured that from the adversary’s perspective, all representations of the current ADT state are equally likely to be observed. Equal likelihood for all representations of an ADT state can be achieved using randomization (Section 4.5).

### 4.4.4 Canonical representations and adversary models

Canonically represented data structures are history independent in the strongest sense, secure even against a computationally unbounded adversary [1]. For a computationally unbounded adversary, canonical representations are also necessary for WHI.

### 4.5 Randomization and History Independence

In the previous section, we discussed the necessity of canonical representations for SHI. In this section, we discuss the use of randomization to achieve history independence. We note that using randomization only gives WHI.

In Section 3.4, we introduced the use of randomization for WHI from the point of view of state transition graphs. We showed that randomization involves ensuring that for any two ADT states $s_0$ and $s_1$, if there is a path in the ADT state transition graph between $s_0$ and $s_1$, then there must be a path from all memory representations of $s_0$ to all memory representations of $s_1$ in the data structure’s state transition graph. The choice of path in the data structure state transition graph between representations of $s_0$ and $s_1$ is then made at random.

In practice, randomization is achieved using the machine programs implementing the ADT operations. An ADT operation $o$ takes the ADT from a state $s_1$ to a state $s_2$. A machine program implementing $o$ takes the data structure from a memory representation of state $s_1$ to a memory representation of state $s_2$. Since each ADT state can have several memory representations (Section 3.4), the program has a choice amongst all representations of state $s_2$ and picks one representation as the result of a transition. Starting from a fixed memory representation of $s_1$, and a fixed input, if the program takes the data structure to a fixed resulting representation of $s_2$ on each execution, then the program is said to be deterministic. If on each execution the resulting representation is chosen uniformly at random from all possible representations of state $s_2$, then the program is said to be randomized.

To illustrate, consider an ADT operation $o$ and a machine program $p$ implementing $o$. Let $o(s_1, i) \rightarrow (s_2, \tau)$ denote the transition from ADT state $s_1$ to ADT state $s_2$ using an ADT input $i$ and producing an ADT output.
Also, let \( m(s_1) \) and \( m(s_2) \) denote the set of memory representations of states \( s_1 \) and \( s_2 \), respectively. Then, for history independence, the following must hold for program \( \psi \):

\[
Pr[p(s_1^M, \alpha(i)) \rightarrow (s_2^M, \beta(\tau))] = \frac{1}{|m(s_2)|}, \ \forall s_2^M \in m(s_1) \text{ and } \forall s_2^M \in m(s_2).
\]

Here, \( \alpha(i) \) and \( \beta(\tau) \) are the machine representations of ADT input \( i \) and ADT output \( \tau \), respectively.

Note that randomization here refers to the selection of memory representations for ADT states and not to program outputs. A program’s output is the machine representation of the corresponding ADT operation’s output.

If randomization is used for history independence, then random choices made by the machine programs must be hidden from the adversary. If the adversary has knowledge of the random bits, then from the adversary’s point of view the machine programs are deterministic. Data structures with deterministic machine programs require canonical representations.

5 Generalizing History Independence

SHI is a very strong notion of history independence requiring canonical representations [1], [6]. Canonically represented data structures are not efficient [14]. For heap and queue data structures Buchbinder et al. [14] show that certain operations that require logarithmic time under WHI take linear time under SHI. Hence, it is worth to question the need for canonical representations for history independence. Many scenarios may not require such a strong notion of history independence making the use of SHI data structures with canonical representations an inefficient solution.

Following are some scenarios that can be efficiently realized by new history independence notions weaker than SHI.

- Hiding evidence of specific operations only. For example, hiding only the fact that a specific data item has been deleted in the past. Eliminating evidence of past deletes is directly applicable for regulatory compliance. Retention Regulations [7], [8], [9] are only concerned with hiding the past existence of deleted data and not with other aspects of history, such as the insertion sequence of current data.
- A MRU caching or a journaling system by definition reveals the last \( k \) operations. Hence, journaling and caching require a new notion of history independence, wherein no history is revealed other than the last \( k \) operations [13].
- Revealing only the number of times each operation is performed [1]. For example, in a file-sharing application disclosing file-access counts may be permissible, but not the access order.

Existing work [1], [13] has already suggested that for efficiency, it is important to benefit from new relaxed notions of history independence. A proper theoretical framework is needed to precisely define new history independence notions. In the following, we take first steps towards such a framework.

A straight-forward way to define new notions of history independence is to provide a new game-based definition for each scenario. However, defining distinct scenario-specific games can quickly become a tedious process. Instead, we introduce a definitional framework that can accommodate a broad spectrum of history independence notions. We term the new framework as \( \Delta \) history independence (\( \Delta \)HI), where \( \Delta \) is the parameter determining the history independence flavor. As we shall see, \( \Delta \)HI also captures both WHI and SHI. In addition, \( \Delta \)HI helps to reason about the history revealed or concealed by existing data structures which were designed without history independence in mind.

5.1 \( \Delta \) History Independence (\( \Delta \)HI)

The WHI and SHI games (Sections 4.1 and 4.2 respectively) are defined over a subset of ADT operation sequences. For WHI, the adversary is permitted to select sequences that take the ADT from initialization to the same end state. For SHI, the permitted sequences are ones that take the ADT from the same starting state to the same ending state. The selection is made by the adversary in step 1 of both the WHI and SHI games. Hence, the initial selection permitted to the adversary determines the history that is desired to be revealed or hidden. By generalizing the selection step, we can accommodate a broad spectrum of history independence notions. We achieve the generalization in \( \Delta \)HI, which is defined by the following game:

Let \( \mathcal{A} = (\mathcal{S}, s_\phi, \mathcal{O}, \Gamma, \Psi) \) be an ADT, \( \mathcal{M} = (\mathcal{S}^M, s_\phi^M, \mathcal{P}^M, \Gamma^M, \Psi^M) \) be a bounded RAM machine model, and \( \mathcal{D} = (\alpha, \beta, \gamma, s_\phi^M) \) be a data structure implementing \( \mathcal{A} \) in \( \mathcal{M} \), as per definitions 1, 2 and 3, respectively. Also, let \( \zeta \) be the set of all ADT operation sequences, \( \Upsilon \) the set of all ADT input sequences, and \( \Delta \) be a function \( \Delta: \mathcal{S} \times \mathcal{S} \times \zeta \times \zeta \times \Upsilon \times \Upsilon \rightarrow \{0, 1\} \).

1) A probabilistic polynomial time-bounded adversary selects the following.
- Two ADT states \( s_1 \) and \( s_2 \); two sequences of ADT operations \( \delta_0 \) and \( \delta_1 \); and two sequences of ADT inputs \( I_0 \) and \( I_1 \); such that \( \Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = 1 \).
- A memory representation \( s_1^M \) of ADT state \( s_1 \).

2) The adversary sends \( s_1, s_1^M, \delta_0, \delta_1, I_0 \) and \( I_1 \) to the challenger.

3) The challenger flips a fair coin \( c \in \{0, 1\} \) and computes \( \mathcal{D}^M(\delta_c^M, s_1^M, I_c) \rightarrow (s_0^M, \tau^M) \), where \( \delta_c^M = \chi(\delta_c) \). That is, the challenger applies the program sequence \( \delta_c^M \) corresponding to
ADT operation sequence \(\delta_s\) to the data structure state \(s_1^M\), resulting in a memory representation \(s^M\), and a machine output \(\tau^M\).

4) The challenger sends the memory representation \(s^M\) and the machine output \(\tau^M\) to the adversary.

5) The adversary outputs \(c' \in \{0, 1\}\).

The adversary wins the game if \(c' = c\).

\(D\) is said to be \(\Delta\) history independent if the probability of the adversary winning the game is negligible (where “negligible” is defined over any implementation-specific security parameters of the programs in \(D^M\)).

Function \(\Delta\) determines the pairs of ADT states, ADT operation sequences, and ADT input sequences that the adversary is permitted to select in step 1 of the \(\Delta HI\) game. For the adversary-selected ADT states, operation sequences, and input sequences, the \(\Delta HI\) game can be played and the data structure implementation is required to ensure that the advantage of the adversary is negligible. Thus, for a given ADT, \(\Delta\) defines two sets,

\[
H_\Delta = \{(s_1, s_2, \delta_0, \delta_1, I_0, I_1) \mid \Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = 1\},
\]

and

\[
\Pi_\Delta = \{(s_1, s_2, \delta_0, \delta_1, I_0, I_1) \mid \Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = 0\}.
\]

For all tuples in \(H_\Delta\), history independence is preserved, that is, neither the ADT nor the data structure implementation reveals the operation sequence selected by the challenger in step 3. For all tuples in \(\Pi_\Delta\), history independence is not required to be preserved since the ADT itself reveals the sequence of operations used.

A careful choice of \(\Delta\) allows us to precisely define both SHI and WHI, and a broad spectrum of new history independence notions. In the following, we illustrate the use of \(\Delta HI\) framework to define some familiar history independence notions. In the following, we illustrate the use of \(\Delta HI\) framework to define some familiar history independence notions. In the following, we illustrate the use of \(\Delta HI\) framework to define some familiar history independence notions.

5.1.1 Strong History Independence (SHI)

We discussed SHI in Section 4.2. Here, we define the function \(\Delta\) for SHI.

\[
\Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = \begin{cases} 
1 & \text{if } \bigcirc(\delta_0, s_1, I_0) \rightarrow (s_2, \tau) \text{ and } \bigcirc(\delta_1, s_1, I_1) \rightarrow (s_2, \tau) \\
0 & \text{otherwise}
\end{cases}
\]

For SHI, the adversary’s advantage in the \(\Delta HI\) game must be negligible when in step 1, the adversary selects any two ADT operation sequences that take the ADT from a state \(s_1\) to a state \(s_2\) producing the same ADT output \(\tau\).

5.1.2 Weak History Independence (WHI)

Refer to Section 4.2 for discussion on WHI, which requires the following definition of \(\Delta\).

\[
\Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = \begin{cases} 
1 & \text{if } \bigcirc(\delta_0, s_1, I_0) \rightarrow (s_2, \tau) \text{ and } \\
0 & \text{otherwise}
\end{cases}
\]

Since WHI permits the adversary to observe a single data structure state, the adversary chooses only the end state \(s_2\) in step 1 of the \(\Delta HI\) game. The starting state on which sequences \(\delta_0\) and \(\delta_1\) are applied is the initial ADT state \(s_0\).

5.1.3 Null history independence (\(\phi HI\))

Under null history independence, a data structure conceals no history except for the trivial case when the ADT operation sequences and ADT input sequences selected by the adversary in the \(\Delta HI\) game are identical. Example of a data structure with \(\phi HI\) is an append-only log. We can reflect \(\phi HI\) using the following.

\[
\Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = \begin{cases} 
1 & \text{if } \bigcirc(\delta_0, s_1, I_0) \rightarrow (s_2, \tau) \text{ and } \bigcirc(\delta_1, s_1, I_1) \rightarrow (s_2, \tau) \\
0 & \text{otherwise}
\end{cases}
\]

5.1.4 \(SHI^*\)

The necessity of canonical representations for SHI was proven by Hartline et al. [6]. The proof by Hartline et al. [6] builds on the case that if a data structure is not canonically represented, then an adversary can distinguish an empty sequence of operations from a nonempty sequence. Hartline et al. [6] then proposed \(SHI^*\), which is defined over nonempty ADT operation sequences. \(SHI^*\) data structures were initially expected to more efficient than data structures providing SHI. However, Hartline et al. [6] found that \(SHI^*\) still poses very strict requirements on a data structure and may not differ from SHI in asymptotic complexity. Here, we give the \(\Delta\) function for \(SHI^*\).

\[
\Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = \begin{cases} 
1 & \text{if } \bigcirc(\delta_0, s_1, I_0) \rightarrow (s_2, \tau) \text{ and } \bigcirc(\delta_1, s_1, I_1) \rightarrow (s_2, \tau) \text{ and } |\delta_0| > 0 \text{ and } |\delta_1| > 0 \\
0 & \text{otherwise}
\end{cases}
\]

\(SHI^*\) closely resembles SHI except that the operations sequences \(\delta_0\) and \(\delta_1\) must be nonempty.

5.1.5 Reveal last \(k\) operations (MRU Cache, File System Journal)

System features such as caching and journaling by definition reveal the last \(k\) operations performed from the ADT state itself. Thus, for caching and journaling, we need to define a \(\Delta\) function, such that no additional historical information is leaked from the memory representations other than the last \(k\) operations. We define the new notion as follows.

Let \(\delta[i]\) denote the \(i^{th}\) operation in the sequence \(\delta\). Also, let \(\delta[i, j]\) denote a subsequence of \(\delta\), \(i \leq j\).
\[
\Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) =
\begin{cases}
1 & \text{if } \psi(\delta_0, s_1, I_0) \to (s_2, \tau) \text{ and } \\
\psi(\delta_1, s_1, I_1) \to (s_2, \tau) \text{ and } \\
|\delta_0| \geq k \text{ and } |\delta_1| \geq k \text{ and } \\
|\delta_0| - k, |\delta_0| = \delta_1[|\delta_1| - k, |\delta_1|] \\
0 & \text{otherwise}
\end{cases}
\]

Here, the adversary is permitted to choose two sequences \(\delta_0\) and \(\delta_1\), such that last \(k\) operations in \(\delta_0\) and \(\delta_1\) are the same. Other than the last \(k\) operations, sequences \(\delta_0\) and \(\delta_1\) may differ. Yet, the adversary should be unable to identify the sequence chosen by the challenger in step 3.

5.1.6 Delete Agnostic History Independence (DAHI)

Consider a secure deletion application that wishes to destroy any evidence of a delete operation performed in the past. That is, an adversary should be unable to detect whether a delete operation was performed or not other than simply guessing. We refer to this notion of history independence as delete-agnostic history independence (DAHI).

Given two sequence of ADT operations \(\delta_1 =< \ I(\cdot), \ldots, I(v), \ldots, D(v), \ldots, I(\cdot) >\) and \(\delta_2 =< \ I(\cdot), \ldots, I(\cdot) >\), a data structure is delete-agnostic history independent if it is \(\Delta\) history independent for the following \(\Delta\) function. Here \(I(x)\) inserts an element \(x\) in the ADT and \(D(x)\) deletes an element \(x\) from the ADT.

\[
\Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) =
\begin{cases}
1 & \text{if } \psi(\delta_0, s_1, I_0) \to (s_2, \tau) \text{ and } \\
\psi(\delta_1, s_1, I_1) \to (s_2, \tau) \text{ and } \\
|\delta_0| = |\delta_1| \text{ and } (I(v), D(v))_{k,j} \in \delta_1 \text{ and } \\
\delta_1[k, j - 1] = |\delta_0| - k, |\delta_0| = \delta_1[|\delta_1| - k, |\delta_1|] \\
0 & \text{otherwise}
\end{cases}
\]

The \(\Delta\) function implies that for a sequence of ADT operations with a delete operation in between, there exists a sequence of insert only operations which results in the same memory representation. The \(\Delta\) function therefore ensures that an adversary has no advantage in guessing whether a delete operation has taken place in the past.

5.2 Measuring History Independence

We have seen that new notions of history independence can be easily derived from \(\Delta\) history independence by defining the appropriate \(\Delta\) function. In this section, we present an intuitive way of comparing \(\Delta\) functions on the basis of the history they require to be concealed or preserved.

For a given \(\Delta\) function we defined the set \(H_\Delta\) (Section 5.1) that represents all combinations of ADT states, operation sequences, and ADT input sequences for which the adversary’s advantage is negligible in the \(\Delta\) history independence game. That is, for all members of \(H_\Delta\), history independence is preserved. One insight is to use the cardinality of \(H_\Delta\) as a measure of history independence.

Recall from Section 3.3 that an ADT can have several data structure implementations. Let \(D\) and \(D'\) be two implementations of an ADT \(A\), such that \(D\) is \(\Delta\) history independent and \(D'\) is \(\Delta'\) history independent for two functions \(\Delta\) and \(\Delta'\). Now, we say that \(D\) is more history independent than \(D'\) if \(H_\Delta \subset H_\Delta'\).

Note that \(|H_\Delta| > |H_\Delta'|\) alone does not imply that \(D\) is more history independent than \(D'\) since an application may be more sensitive to the history preserved by \(D'\) than the history preserved by \(D\). Only in the case where \(H_\Delta \subset H_\Delta'\) can we consider \(D\) to be a more history independent implementation than \(D'\).

5.3 Deriving History Independence

In order to provide a history independent implementation for an ADT, we first require the \(\Delta\) function to be precisely defined. Then, a history independent data structure can be designed that satisfies the \(\Delta\) function. Satisfying a \(\Delta\) function means that the adversary’s advantage is always negligible in the \(\Delta\) history independence game. In effect, so far we have approached history independence as a define-then-design process.

However, data structures have been in use for a long time and most data structures have been designed for efficiency or functionality with no history independence in mind. A natural question then arises – are there any meaningful\(^{14}\) \(\Delta\) functions satisfied by existing data structures?

A data structure can be \(\Delta\) history independent for several \(\Delta\) functions. For example, a data structure that satisfies SHI, also satisfies WHI, OAHI, and OIAHI. Hence, for a given data structure \(D\) finding a \(\Delta\) function may not be a particularly difficult task. It may be more useful instead to determine an uncontained \(\Delta\) function for \(D\). We define an uncontained \(\Delta\) function for a data structure as follows.

**Definition 8.** Uncontained \(\Delta\) function

A \(\Delta\) function for a data structure \(D\) is uncontained if \(D\) is \(\Delta\) history independent and \(\not\subseteq \Delta'\), such that \(\Delta'\) is also \(\Delta'\) history independent and \(H_\Delta \subset H_\Delta'\), where

\[
\begin{align*}
H_\Delta &= \{ (s_1, s_2, \delta_0, \delta_1, I_0, I_1) | \\
\Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = 1 \}; \\
H_\Delta' &= \{ (s'_1, s'_2, \delta'_0, \delta'_1, I'_0, I'_1) | \Delta'(s'_1, s'_2, \delta'_0, \delta'_1, I'_0, I'_1) = 1 \}; \\
s_1, s_2, s'_1, s'_2, I_0, I_1, I'_0, I'_1 \text{ are ADT states; } \delta_0, \delta_1, \delta'_0, \text{ and } \delta'_1 \text{ are ADT operation sequences; and } I_0, I_1, I'_0, I'_1 \text{ are ADT input sequences.}
\end{align*}
\]

We can determine an uncontained \(\Delta\) function for existing data structures on a case-by-case basis. An open question is whether there exists a general mechanism for deriving an uncontained \(\Delta\) function for a given data structure.

\(^{14}\) \(\Delta = 0\) is satisfied by all data structures. Hence, we need to determine \(\Delta\) functions that are more useful in practice.
6 FROM THEORY TO PRACTICE

6.1 Defining Machine States

The RAM model of execution described in Section 3.2 consists of two components, the RAM and the CPU. Hence, the machine state for the RAM model includes bits from both the RAM and the CPU. In general, the machine state for a system-wide machine model will comprise all system component states. A system-wide history independent implementation has to then consider each individual component’s characteristics along the interaction between the components. Providing system-wide history independence is therefore challenging.

However, in practice an adversary may have access to only a subset of system components. In this case, for the purpose of history independence, the machine state can be defined over the adversary-accessible components only. For example, history independent data structures proposed in existing work (Section 10) are designed with the RAM model in mind. However, the machine states considered for history independence only include bits from the RAM and exclude the CPU.

6.2 Building History Independent Systems

Various techniques for designing history independent data structures for commonly used ADTs such as queues, stacks, and hash tables have been proposed [1]. Our focus on the other hand is designing systems with end-to-end history independent characteristics. The difference between history independent implementations for simple ADTs, such as stacks and queues versus a complete system, such as a database, or a file system is a matter of often exponentially increasing complexity. Fundamentally, any system can be modeled as an ADT and an history independent implementation can be sought for the system.

We introduce a general recipe for building history independent systems as follows:

1) Model the system as an ADT. For a specific example of file system as an ADT, refer to section 8.
2) Select a machine model for implementation. While defining the machine state identify all machine components that the adversary has access to and define the machine state associated with the adversary-accessible components.
3) Depending on the application scenario, fix a desired notion of history independence and the corresponding ∆ function.
4) Based on the definition of ∆, provide an implementation over the selected machine model. For complex systems, the implementation will likely require the most effort since the machine programs implementing the ADT operations must provably ensure that the advantage of the adversary is negligible in the ∆HI game.

In section 8, we follow the above recipe to design a history independent file system.

7 ON A PHILOSOPHICAL NOTE

At a very high level, the motivation for history independence can be stated as follows.

For any logical state $S_L$, the physical state $S_P$ representing $S_L$ may reveal information about the history leading to $S_L$, that is otherwise not discernible via solely $S_L$.

So far, we have considered the logical state to be the ADT state and the physical state to be the underlying machine state representing the ADT state, that is, the physical state is the set of all bits of the machine. Our selection of logical and physical states seems rather arbitrary. We do this specific selection due to our adversary model, which assumes that the adversary can interpret information at the level of bits. An adversary, that can for example, examine the electric charge in individual capacitors used to represent the bits will require a different choice of logical and physical state descriptions. A straight-forward choice would be to consider a bit as a logical state and the precise capacitor state as the physical state.

The following interesting question arises from this discussion – is history independence only a matter of perspective?. The short answer is yes, history independence is a matter of perspective. There is no universal history independence.

To clarify, consider the universe as a whole from the viewpoint of classical physics. Under the classical viewpoint, knowledge of current state of all objects in the universe enables determination of any past or future universal state since the laws of physics work both forwards and backwards in time. Hence, the past is never hidden and history independence is impossible. For example, using the currently observed movement of galaxies, the past states of the universe can be inferred up to the very initial moments of the big bang.

Physical phenomena at the subatomic scale is explained by quantum physics. At the quantum level, the universe appears nondeterministic. Further, the uncertainty principle [17] restricts the ability to accurately measure the current state of a quantum system. Since the current state cannot be accurately known, it may seem the past states cannot be determined either and history independence can be achieved at the quantum level.

However, even at the quantum level history independence is still a matter of perspective. The perspective is governed by the interpretation of quantum physics used. Under the many-worlds interpretation, the multiverse as a whole is deterministic [18]. The probabilistic nature at the quantum level is only our perception since our observations are limited to a single universe. A hypothetical all-powerful adversary that can view the entire multiverse would have a full view of the past and the future similar to the case of classical physics making history independence in the presence of such an adversary impossible.
8 PRACTICAL SHI FOR FILE SYSTEMS

In previous sections we laid the theoretical foundations for history independence. We explored the concepts of ADTs, machine models, data structures, and memory representations. We then formalized history independence and introduced the $\Delta$ history independence framework.

We now apply our theoretical concepts and results towards practical history independent system designs. Using the recipe outlined in Section 6.2, we design, implement, and evaluate a history independent file system (HIFS).

In Sections 8.1 - 8.5, we describe HIFS, an SHI implementation for file systems. Then, in Section 9 we introduce DAFS (delete agnostic file system). DAFS is our ongoing effort, which extends HIFS beyond SHI to implement new history independence notions. DAFS aims to be more efficient for scenarios in which canonical representations can be avoided. Further, DAFS extends both functionality and resilience of the file system.

8.1 Introduction

Existing file systems, such as Ext3 [19] are not history independent because they organize data on disk as a function of both files’ data and the sequence of file operations. The exact same set of files can be organized differently on disk depending on the sequence of file system operations that created the set. As a result, observations of data organization on disk can potentially reveal file system’s history. Moreover, file system metadata also contains historical information, such as list of allocated blocks. Therefore, when observations of data organization are combined with file system meta-data, and with knowledge of application logic, significantly more historical information can be derived, for example, full recovery of deleted data. It is therefore imperative to hide file system history.

File system history can be hidden by making file system implementations history independent. A straightforward way to achieve this is to use existing history independent data structures to organize files’ data on disk. Current techniques to make history independent data structures persistent require the use of history independent hash tables [1]. The history independent hash tables [3] in turn use uniform hash functions. The use of uniform hash functions distributes files’ data on storage with no consideration to data locality. Hence, existing history independent data structures destroy data locality making their use in file system design impractical.

In HIFS, we overcome the challenge of providing history independence while preserving data locality. In the following we detail.

8.2 Model

We assume an insider adversary with full access to the system disk. By analyzing data organization on disk, the adversary aims to derive file system’s history. We assume that the adversary can make multiple observations of disk contents. Recall from Section 4.4 that thwarting such an adversary requires SHI with canonical representations. Hence, HIFS targets canonical representations for file storage.

8.3 Concepts

In Section 6.2 we outlined a general recipe for building history independent systems. In the following, we use the recipe to design HIFS. First, we define a file system ADT (Section 8.3.1). Then, we describe the machine model over which we seek an history independent implementation for the file system ADT (Section 8.3.2). In Section 6.2 we have already outlined the need of canonical representations for SHI. Hence, the $\Delta$ function for HIFS is the same as that for SHI defined in Section 5.1. Finally, we detail our HIFS implementation (Sections 8.4 - 8.5).

8.3.1 File System ADT

A file system organizes data as a set of files. We consider a file to consist of some meta-data and a bit string. That is, a file $f = \{m_f, b_f\}$, where $m_f$ is the file meta-data and $b_f \in \{0, 1\}^*$. We define a file system ADT using the file type. Refer to Section 3.3.3 for a discussion on ADTs and types.

A file system is an ADT, that is, a pentuple $(\mathcal{S}, s_0, \mathcal{O}, \Gamma, \Psi)$, where
- $\mathcal{S} = 2^\mathcal{F}$, is the set of states. Here $\mathcal{F}$ is the set of all files.
- $s_0 \in \mathcal{S}$ is the initial state.
- $\Gamma = \{0, 1\}^* \cup (\mathbb{N} \times \mathbb{N} \times \mathbb{N}) \cup (\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \{0, 1\}^*)$ is the set of inputs.
- $\Psi = \mathbb{Z} \cup \{(0, 1)^* \times \mathbb{Z}\}$ is the set of outputs.
- The set of operations $\mathcal{O} = \{\text{open, read, write, delete, close}\}$, such that
  - open : $\mathcal{S} \times \{0, 1\}^* \rightarrow \mathcal{S} \times \mathbb{Z}$.
  - read : $\mathcal{S} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathcal{S} \times \{0, 1\}^* \times \mathbb{Z}$.
  - write : $\mathcal{S} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \{0, 1\}^* \rightarrow \mathcal{S} \times \mathbb{Z}$.
  - delete : $\mathcal{S} \times \mathbb{N} \rightarrow \mathcal{S} \times \mathbb{Z}$.
  - close : $\mathcal{S} \times \mathbb{N} \rightarrow \mathcal{S}$.

File systems including HIFS support several additional operations. We have included only a small subset of the operations here for brevity.

8.3.2 RAMDisk Machine Model

In Section 3.2.1 we introduced the RAM machine model. The RAM model consists of two components, a central processing unit (CPU), and a random access memory (RAM). However, a file system is generally used to store and manage data over a secondary storage device. Hence, we define the RAMDisk model which in addition to the CPU and memory also includes the storage disk.

Definition 9. RAMDisk Machine Model

A RAMDisk machine model $\mathcal{M}_D$ with $m$ b-bit memory words,
n b-bit CPU registers, and c k-bit disk blocks is a pentuple $(S, s_0, \mathcal{P}, \Gamma, \Psi)$, where $S = \{0, 1\}^{b(m+n)+c-k}$ is a set of machine states, $s_0 \in S$ is the initial state, $\mathcal{P}$ is the set of all programs of $\mathcal{M}_D$, $\Gamma = \{0, 1\}^*$ is a set of inputs, $\Psi = \{0, 1\}^*$ is a set of outputs; each program $p \in \mathcal{P}$ is a function $p : S \times \Gamma_p \rightarrow S \times \Psi_p$, where $\Gamma_p \subseteq \Gamma$ and $\Psi_p \subseteq \Psi$.

$\mathcal{M}_D$ is initialized to state $s_0$. When a program $p \in \mathcal{P}$ with input $i \in \Gamma_p$ is executed by the CPU when $\mathcal{M}_D$ is in state $s_1$, $\mathcal{M}_D$ outputs $\tau \in \Psi_p$, and transitions to a state $s_2$. This transition is denoted as $p(s_1, i) \rightarrow (s_2, \tau)$.

According to our model (Section 8.2), the adversary has access to the storage disk. Recall from Section 6.1 that for the purpose of history independence, we need to consider the machine states associated with the adversary-accessible components only. Hence, from this point onwards we refer to the storage device state as the machine state. Since the adversary does not access CPU and RAM components we permit the CPU and RAM states to reveal history.

### 8.3.3 File System Implementation (Data Structure)

The objectives of HIFS design are three-fold.

1) For a given set of files, the organization of files’ data and files’ meta-data on disk must be the same independent of the sequence of file operations. That is, file system implementation must be canonically represented and thereby preserve SHI.

2) Despite history independent storage, data locality must be preserved.

3) The implementation must be easily customizable to suit a wide range of data locality scenarios.

HIFS is a history independent implementation of the file system ADT from Section 8.3.1. That is, HIFS is a data structure $D = (\alpha, \beta, \gamma, s_0^M)$ obtained as follows.

- For all $n \in \mathbb{N}_b$, $\alpha(n) = \{0, 1\}^b$. Here, $\mathbb{N}_b = \{x | x \in \mathbb{N} \text{ and } x \leq 2^b\}$, $b$ is the machine word length, and $\alpha(n)$ is the bit string representing $n$. For all $t_s \in \{0, 1\}^{c-k}, \alpha(t_s) = t_s$. For all $(n_1, n_2, n_3) \in \mathbb{N}_b \times \mathbb{N}_b \times \mathbb{N}_b$, $\alpha((n_1, n_2, n_3)) = \alpha(n_1)||\alpha(n_2)||\alpha(n_3)$. For all $(n_1, n_2, n_3, t_s) \in \mathbb{N}_b \times \mathbb{N}_b \times \mathbb{N}_b \times \{0, 1\}^{c-k}$, $\alpha((n_1, n_2, n_3, t_s)) = \alpha(n_1)||\alpha(n_2)||\alpha(n_3)||t_s$.

- For all $z \in \mathbb{Z}_b$, $\alpha(z) = \{0, 1\}^b$. Here, $\mathbb{Z}_b = \{x | x \in \mathbb{Z} \text{ and } x \leq 2^b\}$, $b$ is the machine word length, and $\alpha(z)$ is the bit string representing $z$.

- $\gamma : \mathcal{O} \rightarrow \mathcal{P}^M$. The programs that we provide for each file system operation are the key to achieving SHI. We discuss the HIFS programs in Section 8.4.

- The initial data structure state $s_0^M$ corresponding to the initial file system ADT state is obtained by initializing all file system meta-data.

#### 8.4 Architecture

##### 8.4.1 Overview

A file system ADT state contains two pieces of information for each file – the file meta-data and the file data. HIFS supports SHI by providing unique memory representations for each ADT state. To ensure unique representations, we first select an existing SHI data structure implementation for a hash table ADT (Section 8.4.2). Then, we re-design the hash table implementation to endow it with data locality properties (Section 8.4.3). Finally, we use two instances of re-designed hash table implementation to store data on disk, one for files’ meta-data and the other for files’ data (Section 8.4.4).

Due to space constraints we refer the reader to [20] for detailed HIFS architecture. In the following we focus only on the key features that make HIFS history independent and locality preserving.

##### 8.4.2 History Independent Hash Table [3]

The SHI data structure of choice is the history independent hash table from [3]. The hash table in [3] is based on the stable matching property of the Gale-Shapley Stable Marriage algorithm [21].

**Stable Marriage Algorithm:** Let $M$ be a set of men and $W$ be a set of women, such that $\lvert M \rvert = \lvert W \rvert = n$. Also, let each man in $M$ rank all women in $W$ as per his set of preferences. Similarly, each woman in $W$ ranks all men in $M$.

The goal of the stable marriage algorithm is to create $n$ matchings, such that no two pairs $(m_i, w_j)$ and $(m_k, w_l)$ exist, where (a) $m_i$ ranks $w_j$ higher than $w_j$ and (b) $w_l$ ranks $m_i$ higher than $m_k$. If no such pairings exist then all matchings are considered stable.

15 File system meta-data includes superblock, group descriptors, inode tables, and disk buckets map. Refer to [20] for detailed HIFS architecture. The loading of file system programs and memory management are done by the operating system.
The algorithm works as follows. In each round, a man \( m \) proposes to one woman at a time based on his ranking of \( W \). If a woman \( w \) being proposed to is un-matched then a new match \((m, w)\) is created. If the woman \( w \) is already matched to some other man \( m' \) then one of the following occurs. (a) if \( w \) ranks \( m \) higher than \( m' \) then the match \((m', w)\) is broken and a new match \((m, w)\) is created, or (b) if \( w \) ranks \( m \) lower than \( m' \), then \( m \) proposes to the next woman based on his rankings. The algorithm terminates when all men are matched.

[21] shows that if all the men propose in decreasing order of their preferences then the resulting stable matching is unique. This holds even if in each round, the selection of a man \( m \) who gets to propose is arbitrary.

**SHI Hash Table:** [3] uses the above property of the Stable Marriage algorithm to construct a canonically represented SHI hash table as follows. (1) The set of keys are considered as the set of men. (2) The set of hash table buckets are considered as the set of women. (3) Each key has an ordered preference of buckets and vice versa. (4) The preference order of each key is the order in which the buckets are probed for insertion, deletion, and search. (5) In case of a collision between two keys, the key which ranks higher on the bucket’s preference list takes the slot. The lower ranked key is relocated to the next bucket in its preference list.

(1) - (5) ensure that for a given set of keys, the hash table data structure state is the same irrespective of the sequence of key insertions and deletions, thereby making the hash table data structure canonically represented.

### 8.4.3 Key Insights

The SHI hash table of [3] can be used as is to organize file’s meta-data and files’ data on disk. This will yield a SHI file system implementation. However, doing so does not preserve data locality, which is an important goal in HIFS design. Then a key observation in this context is the following. In the Stable Marriage algorithm each man in \( M \) can rank the \( n \) women in \( W \) in \( n! \) ways and vice versa. Hence, several sets of preferences from keys to buckets and buckets to keys are possible, each resulting in a distinct hash table instance. Therefore, by changing the preference order of keys and buckets we can control the organization of keys within the hash table.

To enable the re-ordering of preferences we re-write the algorithms of [3]. The re-write categorizes hash table operations in two Procedure Sets, a generic set and a customizable set. The generic procedures implement the overall search, insert, and delete operations, and can be used unaltered for all scenarios. The customizable procedures determine the specific key and bucket preferences thereby governing data organization, including canonical representations and data locality. This new procedure classification and rewrite enables HIFS to realize different data locality scenarios for the same data set through modifications of the customizable procedure set only. The generic procedures and the overall file system operations remain unchanged. We note that this customization is achieved while preserving SHI.

Due to space constraints we refer the reader to [20] for complete listing of generic and customizable procedures for several data locality scenarios. In this paper we focus on the scenario of block group locality. Under block group locality, it is desired that blocks of the same file are located close together on disk ideally within the same block group.

#### 8.4.4 File Storage

File data is stored in blocks on disk. The blocks are grouped into fixed-size units. Each unit is termed as a disk bucket (Figure 4). Like Ext3 [19], HIFS divides the disk into block groups. Each block group contains an inode table, a disk buckets map, and a set of disk buckets. Each entry within the disk buckets map has a one-one mapping to the corresponding disk bucket within the same block group. The entry in the map contains meta-data about the corresponding disk bucket, such as whether the bucket is free or occupied.

#### 8.4.5 Achieving SHI With Data Locality

Existing file systems, such as Ext3 [19] maintain a list of allocated blocks within the file inode, which renders the disk space allocation history dependent. HIFS on the other hand does not rely on allocation lists. Instead, in HIFS, locations of data blocks are determined using only the current operation parameters and do not depend on past operations.

In HIFS, the disk bucket maps from all block groups are collectively treated as a single SHI hash table. Then, to achieve canonical representations for file system ADT states, the file system operations are translated in to SHI hash table operations as follows: (a) Keys are derived from the full file path, and from read or write offset parameters to file system operations. (b) The hash table buckets are the disk buckets map entries. (c) Key preferences are set such that each key first prefers all buckets from one specific block group in a fixed order. Then, buckets from the next adjacent block group and so on. This ensures that with high probability blocks of the same file will be located within the same block group. (d) Buckets prefer keys with higher numerical values.

The above translation realizes one data locality scenario referred to as block group locality. In [20] we demonstrate several other scenarios such as sequential file storage and locality based on external parameters.

#### 8.5 Experiments

A detailed evaluation of HIFS for different application profiles and data locality scenarios is available in [20]. Here, we only list partial results (Figure 5) to give a sense of throughputs that can be achieved under SHI.

The performance of HIFS for read operations is comparable to read throughputs of Ext3 for load factors up to

---

16. Refer to [3] for proof of canonical representation.
9.1 Journaled History Independence (JHI): Reveal Last $k$ Operations

In the event of a system failure, it is imperative that the file system state is not corrupted. To ensure this, file systems typically employ a journal. File system operations are first recorded in the journal and then applied to the file storage area. If a failure occurs while writing to the journal, the operations can be ignored on system recovery. On the other hand, if failure occurs while writing to file storage, then on recovery the operations can be re-played from the journal. Thus each write request to the file system causes two disk writes, one to the journal and one to file storage.

Recall from Section 5.1.5 that journaling by definition reveals the last $k$ operations. This is a necessary tradeoff between resilience and security.

9.1.1 DAFS Journaling

In DAFS, a separate region on disk is reserved for a journal in the form of a circular log. The journal contains information for a finite number of file system operations, say $k$ operations. Operations are recorded in the journal in the order in which they are received by the file system. To restore consistency after system failure, it is essential to maintain operation order. Hence, the sequence of $k$ operations recorded in the journal cannot be hidden. The file storage areas, such as the inode tables, disk bucket maps, and the disk buckets provide SHI just as in the case of HIFS. Hence, once a file system operation is applied to file storage and removed from the journal, its timing can no longer be identified.

In summary, DAFS journaling provides consistent failure recovery while...

9.1.2 Apparent paradox: why journaling increases efficiency

History independence relaxations that come with journaling allow significantly more efficient file system operations due to batching. This is explained in the following.

To maintain canonical representations in HIFS, data is potentially re-located on each file system write operation. The frequency of data re-location increases exponentially with the file system load factor. Hence, for higher load factors, the number of disk writes for each write request to the file system is much greater than the two disk writes required for journaling. Further, the same data blocks may be re-located several times in consecutive write operations. If write operations can be batched, then the number of times a data block is re-located can be reduced by avoiding redundant moves.

In DAFS, we choose to use the journal not only for failure recovery but also as a buffer to batch write operations. Write operations are applied to file storage areas only when the journal is full. During this process, redundant disk writes are eliminated significantly improving write throughputs.

### Table: HIFS Experimental Parameters and Throughputs

| Parameter          | Value |
|--------------------|-------|
| File system size   | 100 GB|
| Mean file size     | 1 GB  |
| No. of files       | 100   |
| Disk block size    | 4 KB  |
| No. of block groups| 8     |
| Disk blocks / bucket| 5120 |
| Inode size         | 281 bytes |
| IO Size            | 32 KB |

Fig. 5. HIFS experimental parameters and throughputs. Load factor = space utilization.
### 9.2 Delete-Agnostic History Independence (DAHI)

Regulations [7], [8], [9] that are specifically concerned with irrecoverable data erasure and not with other artifacts of history can be met by systems that support DAHI for the delete operation. As discussed in Section 5.1.6, unlike SHI, DAHI for deletes can be achieved without canonical representations. Relaxing the requirement to noncanonical representations presents significant efficiency benefits.

To make DAFS preserve DAHI only, we first transform the SHI hash table [3] into a DAHI hash table. Then, we use the DAHI hash table to organize files’ data and files’ metadata.

#### 9.2.1 DAHI hash table

The SHI hash table from [3] can be transformed into an DAHI hash table as follows. The hash table insert operation is modified to not maintain canonical representations. Instead, the insert operation uses linear probing [11] and inserts a key in the first available bucket.

The SHI hash table delete operation\(^{17}\) alone provides DAHI. Deletion of a key from the hash table leaves an empty bucket, say bucket \(b_1\). The delete operation then finds a key that prefers bucket \(b_1\) more than the bucket it is located in, say bucket \(b_2\). If such a key is found it is moved from \(b_2\) to \(b_1\) making \(b_2\) empty. The process is then repeated for bucket \(b_2\) and so on, until no key is found for relocation. The net effect of this process is that a sequence of hash table operations that contains a delete operation results in the same hash table state as an insert-only sequence hiding all evidence of the delete.

**Theorem 3.** DAHI hash table preserves delete-agnostic history independence.

\(^{17}\) For complete listing of SHI hash table operations refer to [20].

\[ C = \{y \mid \delta_1[i] = I(y), i > j\}. \]

Further, \(y \in C\), \(y\) can be mapped to position \(k\) in the hash table using linear probing.

The three sets are so designed that the elements of the set are sorted on the order in which the elements were inserted into the hash table. Consider a set \(S \subset \{A, B, C\}\) and two elements \(a, b \in S\) such that \(S_i = a\) and \(S_j = b\) where \(S_k\) is the \(k^{th}\) element of set \(S\). Also consider \(\delta_1[p] = I(a)\) and \(\delta_1[q] = I(b)\). The sorted property of the sets implies that \(i < j\) only if \(p < q\).

Deleting \(x\) from the file system does not affect the elements in \(A\) and \(B\). Due to the design of DAHI hash table, once \(x\) is deleted from the file system, the first element from \(C\) is placed at position \(k\) in the hash table. Let \(C = \{c_1, c_2, \ldots\}\). Consider an insert only sequence \(\delta_2\) with the same insert order of elements as in \(\delta_1\) excluding the element \(x\). This sequence of operations will result in a configuration of the hash table with \(c_1\) being placed at position \(k\). This must be true since if some other element \(c \in C\) were to be placed at position \(k\), then \(c\) would have to be inserted before \(c_1\) into the hash table which would contradict the fact that \(\delta_1\) and \(\delta_2\) have the same order of inserts excluding \(x\). Therefore when \(x\) is deleted from the hash table, the configuration of the hash table is restored to a state such that it appears to an adversary that the hash table was brought to this state from the initial state using \(\delta_2\) instead of \(\delta_1\) and \(x\) was never inserted and subsequently deleted. Since the delta history independence game for delete-agnostic history independence allows an adversary to select only \(\delta_1\) and \(\delta_2\) as the sequence of operations to send to the challenger in the initial step, given the current state of the DAHI hash table, the adversary cannot win the game with more than negligible probability.

#### 9.2.2 DAHI in DAFS

DAFS uses the DAHI hash table for file storage. The DAHI hash table insert operation is not required to maintain canonical representations. Since the hash table insert operation is used by file system write operation, the overhead of maintaining canonical representations on file writes is eliminated.

When a file is deleted in DAFS, for each disk bucket allocated to the file, the same effect is achieved as that for a key deleted from the DAHI hash table. As a result, no evidence of a delete remains in the file system state and DAHI is preserved.

Changing the history independence notion from SHI in HIFS to DAHI in DAFS has significant potential for efficiency. As shown in Table 3, the number of writes to disk buckets needed for DAHI is significantly lower as compared to the number of writes needed for SHI. This is because write operations are no longer required to maintain canonical representations. As a result, when disk buckets are allocated to a file, other files’ data needs no relocation. The relocation of data was precisely the reason for lower throughputs of HIFS writes.
### Parameter Value

| Parameter                  | Value  |
|----------------------------|--------|
| File system size           | 10 GB  |
| Mean database size         | 1 GB   |
| No. of databases           | $L \cdot 10$ |
| Disk block size ($d_b$)    | 4 KB   |
| No. of block groups ($g_n$) | 4      |
| Disk blocks per bucket ($d_{bn}$) | 512    |
| Inode size                 | 281 bytes |

### 9.3 Experiments

DAFS implements two new history independence notions, JHI and OAHI. Both JHI and OAHI are aimed to increase file system efficiency. In this section, we compare the performance of DAFS and HIFS.

#### 9.3.1 Setup

All experiments were conducted on servers with 2 Intel Xeon Quad-core CPUs at 3.16GHz, 8GB RAM, and kernel v3.13.0-24. The storage devices of choice are Hitachi HDS72302 SCSI drives.

#### 9.3.2 Implementation

DAFS is implemented as a C++ based user-space Fuse [22] file system. All data structures, including OAHI hash table were written from scratch. File system setup parameters are listed in Figure 6.

#### 9.3.3 Measurements

To experiment for a real-world scenario we use the TPCC [23] database benchmark. The database of choice is Sqlite. Sqlite data files are stored using HIFS, DAFS, and Ext3. The BenchmarkSQL tool is used to generate the TPCC workload.

Each test run commences with an empty file system and creates new databases on file system storage. The number of databases is increased until the file system is 90% full. The TPCC scale factor is 10 giving a size of 1GB for each database. Throughputs are measured at specific load factors ranging from 10% to 90%.

#### 9.3.4 Results

Figure 6 reports the throughputs for HIFS, DAFS, and Ext3. As per the TPCC specification, throughputs are reported as new order transactions executed per minute (tpmc). As seen, the performance of DAFS is up to 4x times better than HIFS for load factors >50%. Note that the performance of Ext3 is included as a reference. Ext3 does not provide OAHI.

For load factors ≤ 50%, HIFS and DAFS exhibit similar performance. At lower load factors fewer collisions occur as new files are added to file system storage. Fewer collisions mean that the frequency of data relocation to maintain canonical representations is low at load factors ≤ 50%. Hence, performance of DAFS and HIFS is similar at low load factors.

### 10 Related Work

Existing history independent data structures are summarized in Table 4. The data structures in Table 4 assume a re-writable storage medium. [4] designed a history independent solution for a write-once medium. The construction is based on the observation from [13] that a lexicographic ordering of elements in a list is history independent. However, write-once memories do not allow in-place sorting of elements. Instead [4] employs copy-over lists [13]. When a new element is inserted, a new list is stored while the previous list is erased. This requires $O(n^2)$ space to store $n$ keys.

[5] improves on [4] requiring only linear storage. The key idea here is to store all elements in a global hash table and for each entry of the hash table maintain a separate copy-over list containing only the colliding elements.

### 11 Conclusions

In this paper, we took a deep look into history independence from both a theoretical and a systems perspective. We explored the concepts of abstract data types, machine models, data structures and memory representations. We identified the need for history independence from the perspective of ADT and data structure state transition graphs. Then, we introduced Δ history independence, which serves as a general framework to define a broad spectrum of history independence notions including strong and weak history independence. We also outlined a general recipe for building history independent systems and illustrated its use in designing two history independent file systems.

### References

[1] D. Golovin, “Uniquely represented data structures with applications to privacy,” Ph.D. dissertation, 2008, aAI3340637.

[2] M. Naor, G. Segev, and U. Wieder, “History-independent cuckoo hashing,” in *Proceedings of international colloquium on Automata, Languages and Programming, Part II*. Springer-Verlag, 2008, pp. 631–642.
TABLE 4
Summary of history independent data structures. $\alpha \leftarrow$ load factor, $N \leftarrow$ number of keys, $B \leftarrow$ block transfer size.

Also, I : insert, L : lookup, D : delete, R : range

| Data Structure | SHI or WHI? | Year | Ops | Runtime |
|----------------|------------|------|-----|---------|
| 2-3 Tree [24]  | WHI        | 1997 | I,L,D | $O(\log N)$ |
| Hash Table [13]| SHI        | 2001 | L,L,D | $O(\log(1/(1 - \alpha)))$ |
| Hash Table [3] | SHI        | 2007 | I,L,D | $O(1/(1 - \alpha)^3)$ |
| Hash Table [2] | SHI        | 2008 | I,L,D | $LD \rightarrow O(\log N), S \rightarrow O(1)$ |
| B-Treaps [25]  | SHI        | 2009 | L,D,R | $O(\log N)$ |
| B-SkipList [26]| SHI        | 2010 | L,D,R | $O(\log N)$ |

[3] G. E. Blelloch and D. Golovin, “Strongly history-independent hashing with applications,” in Proceedings of IEEE Symposium on Foundations of Computer Science, ser. FOCS ’07, 2007, pp. 272–282.

[4] D. Molnar, T. Kohno, N. Sastry, and D. Wagner, “Tamper-evident, history-independent, subliminal-free data structures on prom storage—or-how to store ballots on a voting machine,” in Proceedings of IEEE Symposium on Security and Privacy, 2006, pp. 365–370.

[5] T. Moran, M. Naor, and G. Segev, “Deterministic history-independent strategies for storing information on write-once memories,” in Proceedings of International Colloquium on Automata, Languages and Programming. Springer, 2007, pp. 303–315.

[6] J. Hartline, E. Hong, A. Mohr, E. E. Mohr, W. Pentney, and E. Rocke, “Characterizing history independent data structures,” 2002.

[7] CFR240, “Code of Federal Regulations, Part 240.17a-4,” 2010.

[8] PIPEDA, “Personal Information Protection and Electronic Documents Act,” 2000.

[9] EU-DRD, “EU Data Retention Directive,” http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2005:0063:0063:EN:PDF, 2005.

[10] S. M. Diesburg and A.-I. A. Wang, “A survey of confidential data storage and deletion methods,” ACM Comput. Surv., vol. 43, no. 1, pp. 2:1–2:37, Dec. 2010.

[11] D. P. Mehta and S. Sahni, Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.). Chapman & Hall/CRC, 2004.

[12] S. Bajaj and R. Sion, “Ficklebase: Looking into the future to erase the past,” 2013 IEEE 29th International Conference on Data Engineering (ICDE), vol. 0, pp. 86–97, 2013.

[13] M. Naor and V. Teague, “Anti-persistence: History independent data structures,” in Proceedings of ACM symposium on Theory of computing, ACM Press, 2001, pp. 492–501.

[14] N. Buchbinder and E. Petrank, “Lower and upper bounds on obtaining history independence,” Inf. Comput., vol. 204, no. 2, pp. 291–337, Feb. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.ic.2005.11.001

[15] J. Savage, “Models of computation: Exploring the power of computing,” 1998.

[16] L. Cardelli and P. Wegner, “On understanding types, data abstraction, and polymorphism,” ACM Comput. Surv., vol. 17, no. 4, pp. 471–523, Dec. 1985. [Online]. Available: http://doi.acm.org/10.1145/641604.641608

[17] A. Rae, Quantum physics: a beginner’s guide, ser. Oneworld beginners’ guides. Oneworld, 2005.

[18] H. Everett, “The theory of the universal wavefunction,” 1956, PhD Thesis.

[19] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, “A study of linux file system evolution,” Trans. Storage, vol. 10, no. 1, pp. 3:1–3:32, Jan. 2014.

[20] S. Bajaj and R. Sion, “HIFS: History Independence for File Systems,” in Proceedings of the 20th ACM Conference on Computer and Communications Security, ser. CCS ’13, 2013.

[21] D. Gale and L. Shapley, “College admissions and the stability of marriage,” American Mathematical Monthly, vol. 69, no. 1, pp. 9–15, 1962.

[22] A. Rajgarhia and A. Gehani, “Performance and extension of user space file systems,” in Proceedings of the 2010 ACM Symposium on Applied Computing, ser. SAC ’10. New York, NY, USA: ACM, 2010, pp. 206–213. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774130

[23] T. P. F. Council, “TPC-C,” Online at http://www.tpc.org/tpcc/, 1992, database Benchmark Specification.

[24] D. Micciancio, “An oblivious data structure and its applications to cryptography,” in Proceedings of ACM Symposium on the Theory of Computing. ACM Press, 1997, pp. 456–464.

[25] D. Golovin, “B-treaps: A uniquely represented alternative to b-trees,” in Proceedings of International Colloquium on Automata, Languages and Programming: Part I. Springer-Verlag, 2009, pp. 487–499.

[26] ——, “The B-skip-list: A simpler uniquely represented alternative to B-trees,” CoRR, vol. abs/1005.0662, 2010.