# eHIFS: An Efficient History Independent File System

Biao Gao*†
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
gaobiao@iie.ac.cn

Bo Chen
Department of Computer Science, Michigan Technological University
Houghton, USA
bchen@mtu.edu

Shijie Jia†‡
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China
jiashijie@iie.ac.cn

Luning Xia*†
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
xialuning@iie.ac.cn

## ABSTRACT

Securely deleting obsolete data is of significant importance, as reserving them may not only endanger data owners' privacy, but also violate data protection regulations like GDPR, SOX and HIPPA. However, completely eliminating data is extremely challenging in modern computer systems. One reason is the past existence of the deleted data may leave artifacts in the layout of a storage system at all layers, and such structural artifacts may be utilized by the adversary to derive sensitive information about the data having been deleted. A novel security notion, history independence, can ensure that memory representation of a data structure is independent of the operation sequences leading to it. Therefore, history independence can be utilized to remove the structural artifacts created by the deleted data, making it possible to achieve secure deletion guarantee.

In this work, leveraging history independence, we build history independent systems. The existing history independent file system (HIFS) suffers from a significant degradation compared to the regular file system, rendering it impractical for real-world applications. A fundamental reason for such a degradation is, HIFS simply re-locates the entire data in a history independent manner for each single write. This is unfortunately unnecessary since a multi-snapshot adversary has observed a previous snapshot, and relocating data appearing in this old snapshot is vain and incurs unnecessary overhead. This will be exacerbated when the file system load factor is large. We thus design eHIFS, the first efficient History Independent File System, in which we smartly take advantage of knowledge on the adversary's observations and eliminate those unnecessary re-locations. Security analysis and experimental evaluation show that, compared to HIFS, eHIFS can achieve a similar history independence guarantee, with a 33X write throughput improvement when the file system load factor is 90%.

## CCS CONCEPTS

• **Software and its engineering** → **File systems management**; • **Information systems** → **Data management systems**; • **Security and privacy** → *Pseudonymity, anonymity and untraceability*; • **Social and professional topics** → *File systems management*.

## KEYWORDS

History Independence; Secure Deletion; Order Privacy; Efficiency; File System

**ACM Reference Format:**
Biao Gao, Bo Chen, Shijie Jia, and Luning Xia. 2019. eHIFS: An Efficient History Independent File System. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019, Auckland, New Zealand*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3321705.3329839

## 1 INTRODUCTION

Data protection regulations like GDPR [41], HIPAA [16], Sarbanes-Oxley Act [38] require disposing data timely once they become obsolete. However, irrecoverably deleting data in modern computer systems is a challenging task. For example, the New York Times published a CIA document that contained the original document and an overlay was used to cover up sensitive information in the document. The overlay was then removed by users from Internet and critically sensitive information about role of CIA in the 1953 overthrow of Iranian Government was revealed [24]. Secure deletion [44] was thus introduced to ensure complete elimination of sensitive data. Conventionally, overwriting [4, 19, 23, 27, 42] or encryption with ephemeral keys [20, 32, 33, 35, 39, 43] are used to ensure secure deletion. They unfortunately may not be sufficient since write history itself may implicitly leave artifacts in the layout of a storage system at all layers. These structural artifacts may be used by the adversary as an oracle to learn sensitive information

*Also with Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China.
†Also with Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing, China.
‡This author is the corresponding author.


*AsiaCCS '19, July 9–12, 2019, Auckland, New Zealand*

ACM ISBN 978-1-4503-6752-3/19/07…$15.00
https://doi.org/10.1145/3321705.3329839

about the deleted data [3, 13, 14, 25] and, in the worst case, the deleted data can be restored entirely [13].

**History independence and secure deletion**. A *history independent* data structure [31] ensures that memory representation of the data structure is independent of the operation sequences leading to it. In other words, given two operation sequences $A_1$ and $A_2$ such that: 1) $A_1$ contains a delete and its corresponding insert operation, and 2) $A_2$ does not contain the aforementioned delete and insert operation, and 3) they lead to the same abstract state but potentially different memory representations, history independence ensures that by looking at the final memory representation, the adversary will not be able to differentiate which operation sequence, $A_1$ or $A_2$, leads to it. Therefore, after organizing data items history independently, when a data item is removed, the resulting memory representation will be indistinguishable from that this data item never appeared, and thus this data item can be considered as securely deleted[1].

**Other security properties of history independence**. Besides secure deletion, another security property brought by history independence is hiding order of the operations leading to a memory representation (i.e., order privacy). If data are organized history independently, given two operation sequences with the same set of operations but different orders, the adversary is not able to identify which operation sequence leads to an observed memory representation. This can be found a lot of real-world applications, e.g., protecting privacy of voters [31], crime data sharing without disclosing order and times in which data were added [6], etc.

There are mainly two types of history independence [24, 31]: a weak history independence (WHI) and a strong history independence (SHI). If the adversary observes the data structure once (i.e., *a one-snapshot adversary*) and cannot derive historic information about pattern of access, the data structure ensures WHI. This captures real-world scenarios like a victim computer is lost [10, 11, 26]. Similarly, if the adversary can have multiple observations (i.e., *a multi-snapshot adversary*) over time and are not able derive historic access, the data structure ensures SHI. This captures real-world scenarios like a hacker breaks into a hotel room periodically and obtains a snapshot of the victim computer each time [12, 34]. This paper focuses on the SHI, which can achieve a much stronger history independence guarantee. A unique representation (i.e., *canonical* representation[2]) will be necessary for any data structure aiming at SHI [24], which is not required for WHI. Apparently, the SHI can ensure a higher level of history independence, at the cost of additional overhead to remain canonical after each observation from the attacker.

To enhance data protection, pioneering research [3, 14] tried to incorporate history independence into real-world systems. As the first history independence design for file systems, HIFS [3] uses an SHI data structure to organize file blocks such that their placements in the hard drive are always independent of the operation sequence leading to them. In other words, the layout of file blocks needs to remain canonical after each file system write, which may require a large number of relocations for each single write. For example, there are 3 file blocks $b_1$, $b_2$, $b_3$ in the disk, and their canonical representation is assumed to be: $b_2 - b_1 - b_3$, regardless of their insertion order; additionally, the canonical presentation for 4 file blocks $b_1, b_2, b_3, b_4$ is assumed to be, $b_4 - b_2 - b_1 - b_3$; therefore, after a new file block $b_4$ is written to the disk, each previous file block is relocated to its contiguous location to remain canonical, which may be prohibitively expensive, especially when the hard drive is almost filled (e.g., HIFS [3] suffers from more than 93.8% throughput degradation for random write when the file system load factor reaches 90%). Another history independent system HiFlash [14], which is built for flash memory, also suffers from a significant degradation (more than 80%) on write throughput compared to a non-history-independent system.

Clearly, a major obstacle for real-world deployments of history independent systems is their poor performance in write. A fundamental reason for such poor performance is the need of keeping canonical representation among the entire data all the time to ensure history independence when facing the multi-snapshot adversary. However, we have observed that it is unnecessary to "touch" all the data for history independence upon each single new write. Assume the adversary can have two subsequent snapshots on the storage at $t_1$ and $t_2$ respectively, relocating those data having been observed by the adversary at $t_1$ will be meaningless for ensuring history independence at time $t_2$. This is because, by comparing both snapshots, the adversary can easily identify those data at $t_1$ and find out how they were relocated between $t_1$ and $t_2$. In the aforementioned example, assume $t_1$ is the time right before $b_4$ is written, and $t_2$ is the time after $b_4$ is written; at $t_2$, the attacker can easily find out that $b_2$, $b_1$, $b_3$ have been relocated to the corresponding contiguous locations, and thus the effort of relocating them is vain and will not help history independence.

Following the aforementioned observation, we propose eHIFS, an efficient History Independent File System. eHIFS is the first history independent system which can ensure SHI without suffering from degradation of write throughput, e.g., compared to HIFS, eHIFS achieves a 33X write throughput improvement when the file system load factor reaches 90%. A key insight of eHIFS is taking advantage of the knowledge on the adversary's observations, and actively reacting to those observations to eliminate unnecessary relocations of data, such that it can ensure SHI with minimal overhead.

**Contributions**. Our major contributions are as follows:

- We propose eHIFS, an efficient history independent file system. eHIFS is the first practical history independent system which can ensure SHI and remain efficient. Conversely, all the existing history independent systems suffer from prohibitively large overhead, rendering them impractical.
- The design of eHIFS is based on the knowledge on the adversary's observations and actively reacting to those observations. To the best of our knowledge, we are the first to incorporate the adversary's observations into building history independent systems.

[1] This statement relies on an implied assumption that there are no correlations among content of the data items. The leak of deleted data from data correlations is not the focus of this paper

[2] A simple example for canonical representation is: for two data items $b_1$, $b_2$, their representation in memory is always unique, e.g., $b_2 b_1$, where $b_2$ always is stored at the first memory cell, while $b_1$ always the second, regardless which comes first. If $b_1$ comes first, it will be stored at the first memory cell; after $b_2$ comes, $b_1$ will be moved to the second cell, and $b_2$ will be stored at the first cell.

- We provide a more rigorous security proof of history independence file system than HIFS. Our proofs show that eHIFS can ensure SHI. In addition, we experimentally evaluate eHIFS, and the performance comparison with HIFS shows that eHIFS is much more efficient than HIFS in terms of write throughput, i.e., resulting in a 33X improvement when the file system load factor reaches 90%.

# 2 BACKGROUND

## 2.1 History Independence

History independent data structures are designed to avoid disclosing any information beyond what is necessarily provided by the content of the data structure [24, 31]. In other words, a data structure implementation is said to be history independent if nothing can be learned from its memory representation except for its current abstract state [24]. Therefore, by having access to the memory representation, the adversary is not able to identify the operation sequence leading to it, protecting its historical pattern of access. There are two types of history independence, namely, strong history independence (SHI) and weak history independence (WHI). We focus on the stronger history independence notion, SHI, which ensures history independence against an adversary who can have access to the data structure multiple times. The original definition of SHI was introduced by Naor and Teague [31]:

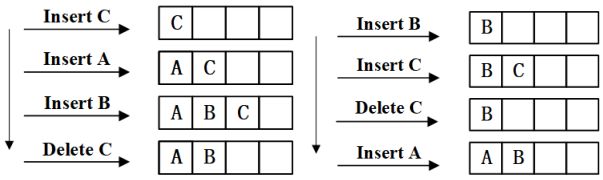
*Definition 2.1.* (**Strong History Independence** [31]).
Let $S_1$ and $S_2$ be sequences of operations having $P_1 = \{i_1^1, i_2^1, \cdots, i_l^1\}$ and $P_2 = \{i_1^2, i_2^2, \cdots, i_l^2\}$ be their checkpoints respectively, where $1 \leq i_j^b \leq |S_b|$ for all $b \in \{1, 2\}$ and $1 \leq j \leq l$. A data structure implementation is strongly history independent, if for any such sequences the distributions of the memory representation are identical at each checkpoint of $P_1$ and the corresponding checkpoint of $P_2$.

The paper [31] focuses on history independent abstract data structure (ADS), where it uses the word *content* to represent the content of an ADS. We should notice that the same content yields the same data to be operated on. Any differences resulted from data themselves are out of the scope. For example, inserting $a, c$, when compared with inserting $a, b$, contradicts the basic rule of the *same content*. The core focus is on the distribution of the memory representation. Hartline *et al.* [24] introduced an equivalent definition of SHI:

*Definition 2.2.* (**Strong History Independence** [24])
A data structure implementation is strongly history independent if, for any two (possibly empty) sequences of operations $X$ and $Y$, the distributions of memory representations of $B$ are identical after the $X$ and $Y$ are performed independently on $B$.

$$(A \xrightarrow{X} B) \wedge (A \xrightarrow{Y} B) \Rightarrow \\ \forall \boldsymbol{a} \in A, \forall \boldsymbol{b} \in B, \mathbf{Pr}[\boldsymbol{a} \xrightarrow{X} \boldsymbol{b}] = \mathbf{Pr}[\boldsymbol{a} \xrightarrow{Y} \boldsymbol{b}] \quad (1)$$

In Equation 1, $a$ and $b$ denote the representations of states of $A$ and $B$ respectively; $\mathbf{Pr}[\boldsymbol{a} \xrightarrow{X} \boldsymbol{b}]$ denotes the probability that $\boldsymbol{a}$ transforms into $\boldsymbol{b}$ after conducting $X$. Following this new definition of SHI, Hartline *et al.* [24] showed that SHI data structures have




(a) Operation sequence 1. (b) Operation sequence 2.

**Figure 1: An example for canonical representation.**

**canonical representations**. Figure 1 shows a simple example for canonical representation, in which two different but equivalent operation sequences lead to the same final representation which is canonical.

## 2.2 Gale-Shapley Stable Marriage Algorithm

**Stable Marriage Problem (SMP)**. Given two sets with equal size (assuming two sets $M = \{m_i | 1 \leq i \leq n\}$ for men and $W = \{w_i | 1 \leq i \leq n\}$ for women), with each element in one set having a preference order for all the elements in the other set, SMP aims to find out a stable matching $\{(m, w)_n\}$, such that for any two matching pairs $(m_i, w_i)$ and $(m_j, w_j)$, it is not possible that $w_i$ prefers $m_j$ and $w_j$ prefers $m_i$.

**Gale-Shapley Stable Marriage Algorithm (G-S SMA)**. Gale and Shaply [18] solved the SMP and proved its stability and correctness, where the definition of **unstable** is necessary.

*Definition 2.3.* A match of men to women will be called unstable if there are two men $m_1$ and $m_2$ who are matched to women $w_1$ and $w_2$, respectively, although $m_2$ prefers $w_1$ to $w_2$ and $w_1$ prefers $m_2$ to $m_1$.

The algorithm works in a number of iterations as follows: First, an unmatched man $m_1$ proposes to a woman he prefers most (e.g., $w_1$) based on his ranking for the set $W$, the $w_1$ will reply "yes" or "no": if $w_1$ is engaged with another man $m_2$, then $w_1$ will choose the one who ranks higher than the other (e.g., the statement that $a$ ranks higher than $b$ in the preference of $B$ means that $B$ prefers $a$ than $b$). But if $w_2$ is not engaged with any man, then a provisional match $(m_1, w_1)$ is created. Second, all the following iterations follow the basic rules above. Each unmatched man proposes to his most preferred woman on his preference list and this woman is the one he has not proposed to. The algorithm ends until all the men are well-matched and there exists no unstable match.

Gale and Shaply [18] showed that for any equal number of men and women, there is always a stable set of marriages and the algorithm always results in the same well-matched pairs. Note that the algorithm described above is suitor-optimal i.e., any man's partner under this rule will not rank lower than the one under any other rules in this man's preference list.

# 3 MOTIVATING SCENARIOS

In this section, we will use concrete scenarios to motivate why history independence is necessary for data protection. Without history independence, the adversary may derive sensitive information

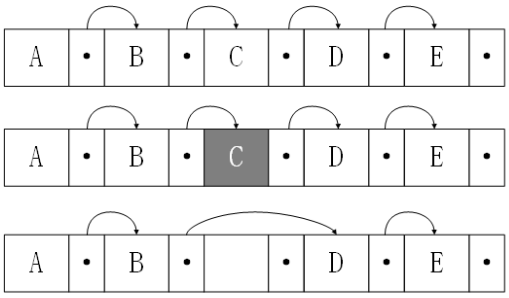
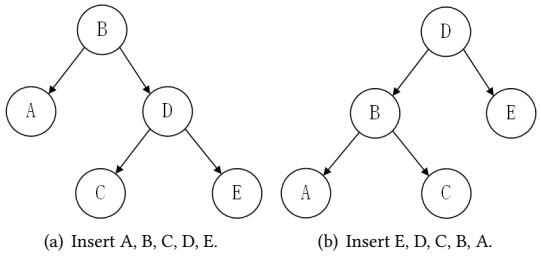



**Figure 2: Delete C after inserting A,B,C,D,E sequentially.**




(a) Insert A, B, C, D, E. (b) Insert E, D, C, B, A.

**Figure 3: The B-trees resulted from two insertion sequences in different orders.**

from the memory representation of a data structure. We consider five patients in a hospital whose private information is sensitive, including the name, the time when the patient is hospitalized, and the duration of the treatment, etc. We use A, B, C, D, E to identify these five patients and assume they can be sorted alphabetically.

**Scenario 1 (secure deletion)**. *Without history independence, the deleted data may leave traces in the representation of the data structure which may be used to derive sensitive information about it.*

The five patients A, B, C, D and E are assumed hospitalized in a sequential order, *i.e.*, they are inserted sequentially into the patient data store, which is built on top of a log-structured file system (Figure 2) and organized as a singly linked list. The data structures used are not history independent. C is then cured and left the hospital, and for security, the entire record of C is required to be securely deleted. Figure 2 shows the entire process. Step 1: each patient is inserted sequentially and each is linked to his/her neighbor. Step 2: C is deleted. Note that a regular log-structured file system may mark the old data as deleted, and reclaim the space later by garbage collection [37]. For secure deletion purpose, we require the entire record of C to be deleted immediately (e.g., by overwriting). Step 3: Patient B's next pointer will be updated with D. Right after patient C has been deleted, the attacker comes, and may find out the following sensitive private information about C: 1) C was hospitalized during the time between B and D. The exact time can be further estimated based on the specific hospitalization time of B and D. The treatment time for C can also be estimated since currently C is the only patient who has left and the time for his/her leaving may not be very far from now. 2) If the attacker has obtained other information like a list of potential candidate patients, and by linking them with the knowledge in 1), he/she may further identify who is C.

**Scenario 2 (order privacy)**. *Without history independence, different operation sequences may cause different memory representations of the data structure, and by observing the memory representation, the adversary may be able to identify the operation sequence.*

Similarly, we use the aforementioned five patients A, B, C, D, and E. A B-tree [5, 15] is used to organize them. Figure 3 shows the representations of the data structure for two different operation sequences. They have the same set of insertion operations but different orders. We observe that the resulting B-trees are different. In other words, the final memory representation of data structure is determined by the operation sequence, which is not history independent. By having access to the representation of the B-tree, the adversary may be able to derive private information about the order in which the patients were added (which may help guess the time when a patient was added). For example, if the adversary finds the representation of data structure is something like Figure 3-(b), it will know the operation sequence is more likely E, D, C, B, A rather than A, B, C, D, E.

# 4 MODEL AND ASSUMPTIONS

## 4.1 System Model

We consider storage systems using mechanical disk drives (rather than flash memory [14]), on which file systems are deployed and sensitive data are stored. We assume all data structures stored within files are history independent [3].

## 4.2 Adversarial Model

We consider a *multi-snapshot adversary*, which is computationally bounded, and aims to illegitimately derive sensitive historical information unavailable through a "legitimate" interface. The multi-snapshot adversary can observe the victim storage device multiple times [8, 12, 34]. This captures a lot of real-world scenarios. For example, an attacker periodically breaks into a hotel room, obtaining a "memory dump" of a victim's computer, aiming to derive sensitive historical information from the snapshots captured. The application scenarios are not necessarily malicious. For example, an entity (e.g., a system administrator or an auditor) is hired to periodically inspect storage of a server running privacy-preserving services like electronic voting [22], who tries to derive sensitive historical information, unbeknownst to the victim system.

## 4.3 Assumptions

Our design relies on a few assumptions as listed in the following:

First, we assume the victim system can detect or be aware of the intrusion in time. In other words, observations from the adversary are known by the victim system [22]. This implies: 1) if the adversary is an external malicious attacker, the system should incorporate intrusion detection mechanism such that intrusion from the adversary can be detected; and 2) if the adversary is an internal non-malicious attacker (e.g., a system administrator or an auditor)

who is authorized to inspect the storage of the system, their access to the system is always known.

Second, we assume the adversary can have access to original content of the data stored in the file system, but the content itself should not leak "historical information" which compromises history independence. History independence design mainly cares about information leakage from memory representation rather than content of the data structure. This assumption is applied to real-world scenarios where the file system is not an encrypted file system. This assumption is also applied to real-world scenarios where the adversary is an entity (e.g., a system administrator, auditor) who is authorized to inspect storage of a server running privacy-preserving services. In this circumstance, the adversary is able to have access to plaintext content even though an encrypted file system is deployed, but the adversary is not expected to derive any sensitive historical information except the abstract state of the storage being inspected.

# 5 AN EFFICIENT HISTORY INDEPENDENT FILE SYSTEM

## 5.1 Overview

To provide SHI for file systems, HIFS [3] uses a history independent hash table (HIHT, Sec. 5.2) for file placement, yielding a history independent layout of files on disk. Having observed that the incorporation of HIHT completely destroys locality of existing file systems, HIFS further improves the design by customizing HIHT with locality consideration. A major problem of HIFS is it causes significant degradation in file system I/O performance. Especially, compared to a regular EXT3 file system without history independence, its write throughput suffers from more than 93.8% degradation for random write when the file system load factor reaches 90% [3]. This renders HIFS impractical for real-world applications. The most difficult challenge is, SHI requires that memory representation of the data structure remains canonical all the time, and maintaining canonical seems prohibitively expensive.

It has been wrongly believed that the SHI data structure does not know when the adversary's observations occur [22] and it is not allowed to react to the adversary's observations. Auditable data structure [22] showed that such an SHI restriction can be relaxed and the data structure can react to having been observed. This, for the first time, makes it possible to re-design the history independent file system which can be both efficient and ensuring SHI. The insights of our design are captured in the following questions.

Question 1: *Why the knowledge on the adversary's observations can help*?

Without having any knowledge on the adversary's observations, to ensure SHI, the data structure needs to relocate data to remain canonical for each single insertion. This explains why HIFS suffers from significant write throughput degradation. However, with knowledge on the adversary's observations, frequently relocating file blocks after each single write turns unnecessary. We use a concrete example to justify the aforementioned statement.

Suppose there are four slots (named $a$, $b$, $c$, $d$) in the disk to be allocated to four file blocks (named $A$, $B$, $C$, $D$). The preference of each slot for the four file blocks is described in Table 1, where a smaller number indicates a higher preference, *e.g.*, the slot $c$ prefers $C$ more than $B$ according to $c$'s preference list. In addition, each file block has the same preference for all the four slots, namely, $a > b > c > d$. Using **G-S SMA**, the four file blocks "propose" to the four slots: When inserting $A$, there is no other competitors, and $A$ prefers slot $a$ the most, and therefore, $A$ will be placed to slot $a$; When inserting $B$, though $B$ prefers slot $a$ the most, slot $a$ unfortunately prefers $A$ more than $B$, and therefore, $B$ can only be placed to slot $b$; and so on. The final canonical layout is shown in Figure 4(a) without considering the adversary's observations. However, by having an observation over the file system before $C$ is written, the attacker can have access to $A$ and $B$. When $C$ and $D$ are written, relocating $B$ does not help history independence, since the adversary can simply keep track of $B$ from the first snapshot. In other words, once the adversary has observed $B$, the global relocation of $B$ seems unnecessary for history independence but incurring significant overhead and, by having knowledge on the adversary's observations, we can eliminate those unnecessary relocations, improving efficiency.

**Table 1: The preference of four slots $(a, b, c, d)$ for four file blocks $(A, B, C, D)$.**

| | $A$ | $B$ | $C$ | $D$ |
|---|---|---|---|---|
| $a$ | 1 | 2 | 3 | 4 |
| $b$ | 3 | 4 | 2 | 1 |
| $c$ | 4 | 3 | 2 | 1 |
| $d$ | 1 | 4 | 3 | 2 |

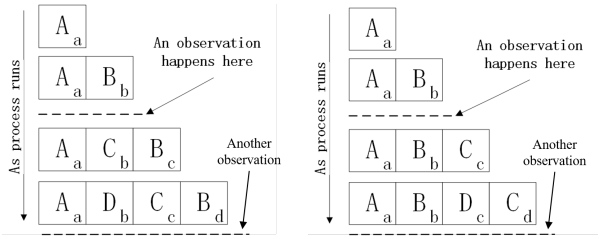
Question 2: *How does the data structure (i.e., history independent file system) react to the adversary's observations to improve efficiency*?

The idea for inserting data are as follows: Initially, inserting data can simply follow HIFS. Once an observation is performed and the system is aware of it (Sec. 4.3), all the current file blocks should be "fixed", and newly coming data will be placed to empty locations following HIFS excluding the "fixed" file blocks. Note that HIFS uses a locality-preserving history independent hash table, and the observations utilized by our approach will not affect locality. In the aforementioned example, following this new idea, the final canonical layout of file blocks is shown in Figure 4(b), in which after the first observation of the adversary, all the file blocks which have been observed by the adversary will be fixed, and only relocating those data which have not been observed by the adversary. We can see that, when C and D are written, HIFS (Figure 4(a)) incurs 3 relocations, but the new design (Figure 4(b)) only incurs 1 relocation.

The idea for deleting data is similar. Initially, deleting a file block can simply follow HIFS. Once an observation is performed, all the current file blocks should be "fixed", and deleting should follow HIFS, excluding those "fixed" file blocks. In this way, the unnecessary relocations for those "fixed" file blocks can be eliminated.

## 5.2 History Independent Hash Table

Similar to HIFS, eHIFS also needs to use a history independent hash table (HIHT) [9]. For any operation sequences, HIHT can always ensure a canonical representation for the data stored in the hash table, achieving SHI.




(a) HIFS: directly using G-S SMA. (b) Our design: taking advantage of the adversary's observations.

**Figure 4: Our design vs. HIFS.**

**The relationship between G-S SMA and SHI**. The SHI concerns about canonical representations while the G-S SMA concerns about a stable marriage matching regardless of the process as long as the basic rules (such as the role of initialing the process, etc.) are followed. Intuitively, G-S SMA somehow achieves what SHI is looking for. Therefore, a history independent hash table (HIHT) can be constructed using G-S SMA [9].

**Constructing an HIHT using G-S SMA**. In [9], keys to be inserted into a hash table correspond to men, and hash table buckets correspond to women in the G-S SMA, respectively. The keys will be derived from the hash of the file path and the offset for the corresponding file data. The preferences of keys for buckets are the sequence that is used for probing the buckets for insert, delete and locate. The preferences of buckets for keys are determined by the hash values. And both of these two kinds of preference lists are fixed after the keys and buckets are determined. The collision will be addressed as follows: If any two keys prefer the same bucket, the key which ranks higher on the bucket's preference order wins, and the other key will "propose" to its next preferred bucket. In this way, the resulting layout of the hash table will be canonical, achieving SHI.

## 5.3 The Design Details

We design eHIFS, an efficient history independent file system, and elaborate the design details of eHIFS in the following.

### 5.3.1 *Preprocessing.*

**Time-Interval**. The key insight named **Time-Interval** is inspired from the detailed description in Sec. 4.3 about why we consider the attacks (audits) to be aware of by the victims, even though the protected system does not know when the attackers want to launch attacks. The system will record when an audit action or analogous attack (hereinafter, audit(s) or observation(s)) happens, which we can take advantage of to construct eHIFS. The interval between the two adjacent audits is called Time-Interval. It reflects the frequency of the unexpected audits along with the sequences of operations that the system has conducted during such these periods between them. The **Time-Interval**s are determined by the time when the observations happen. It is worth mentioning that we can practically set them to some fixed time in real world rather than purely passively react to the observations, which we will introduce in the following.

**Algorithm 1** Locate a key $\boldsymbol{k}$ : $LOCATE(\boldsymbol{k})$

```
1:  Given the hash table HT_t[h], the key k
2:  for i=0; i<t; i++ do
3:      s ← Preferencelist_For_Buckets(k, i)
4:      for j=0; j<h; j++ do
5:          if HT_s[j] is not occupied OR PREFER_OF_BUCKET(s, j, k)
            then
6:              return key k is not found
7:          else if k == HT_s[j] then
8:              return The key k is located in HT_s[j]
9:          end if
10:     end for
11: end for
```

### 5.3.2 *Basic Operations.*

**Reaction**. We allow the system to react to audits immediately after the audits finish. The operation **Reaction** is introduced according to the **Time-Interval**, which is key to achieve a much better performance of efficiency in practice.

We have assumed that the attackers can have full privileges to inspect the data. The location and content of the observed data are completely exposed to the attackers. For simplicity, we think that there is only one attacker or a group of attackers, in which case the attackers will share their all knowledge about the system and the data they have observed at multiple checkpoints.

Once the attacker has observed the storage of the system and obtained the representation of memory, the system makes all the observed data keep matched with their current locations (*i.e.*, from the perspective of the employed hash table, the observed ($key$, $bucket$) pairs will maintain their current states until some rules allow them to make rescission of the relationship). The deletion operation will free the buckets from their keys, which we will discuss below in detail. We call such an operation as **Reaction**.

**Locate**. Each file will have a key in the hash table, with which the file system conducts **Locate** operation to obtain specific information about its location. The function $LOCATE(\boldsymbol{k})$ will be called and its corresponding algorithm is given in Algorithm 1.

When the system needs to locate a key $\boldsymbol{k}$, it will check the hash and probe the sequence of the key's preference list for the buckets to find which is its most preferred bucket in current state. The system will check all the slots whether there is such a key $\boldsymbol{k}$. As declared, a key will be matched to its tentative "mos" preferred bucket which does not violate the **G-S SMA** as the history independent process runs. The function PREFER_OF_BUCKET($s, j, \boldsymbol{k}$) means that the bucket $HT_s[j]$ prefers $\boldsymbol{k}$ than the key with which the bucket is currently matched. The key $\boldsymbol{k}$'s final tentative partner will rank not lower than the bucket $HT_s[j]$ if PREFER_OF_BUCKET($s, j, \boldsymbol{k}$) is true.

**Insertion**. The insertion operation is one of the basic operations which our system supports. Based on the history independent hash tables, we modify the ordinary insertion operation to combine with the efficiency-optimization factor, named **Time-Interval**. The insertion algorithm is provided in Algorithm 2 and 3. The Statement 4 in Algorithm 2 means the bucket with its key has not been observed

**Algorithm 2** Insert a key $\boldsymbol{k}$

```
Given the hash table HT_t[h], the key k
for i=0; i<t; i++ do
  s ← Preferencelist_For_Buckets(k, i)
  for j=0; j<h and HT_s[j] hasn't been marked fixed; j++ do
    if HT_s[j] is not occupied then
      return HT_s[j] = k
    else if PREFER_OF_BUCKET(s, j, k) then
      SWAP the k and current value of HT_s[j]
      Insert the key of replaced value
    end if
  end for
end for
```

**Algorithm 3** The entire insertion process

```
As our system runs...
Supposing that the sequence of T_0, T_1, T_2, ··· are the detected
unexpected non-uniform Time-Intervals
while the system runs do
  set T = T_i (i = 0, 1, 2, ···) as time goes by
  while t has not reached T do
    if the the system wants to insert some key k then
      call the function described in Algorithm 2
    else
      other operations
    end if
  end while
  Make all the observed buckets with their keys bound together
  until some unbinding operations happen
end while
```

by an attacker which otherwise would have been tied together until the key is deleted. In general case, the system will conduct the history independent operations of insertion similar to HIFS unless the observations happen. Then the system will call the **Reaction** to react to these observations immediately. The algorithm will be bucket-optimal according to the **G-S SMA**. Here we emphasize again that the attacker would not modify any data or their location. It is noted that the system is aware of the observations instead of knowing it in advance. We do not exclude the case of regular inspection, which will help our assumptions make more sense.

The Algorithm 2 presents a history independent version of inserting a key $\boldsymbol{k}$ in the single duration of a Time-Interval between two adjacent audits, while the Algorithm 3 presents the related whole process as the system runs, where the functions of insertion operation are called separately and can be replaced with deletion or some others without affecting the explanations of the process.

**Deletion**. Supposing that the $t_{latest}$ denotes the latest checkpoint and the next adjacent unexpected checkpoint is denoted as $t_{next}$, we want to delete a key $\boldsymbol{k}$ whose corresponding bucket is denoted as $HT_s[j]$ for some $s$ and $j$. We free the key $\boldsymbol{k}$ from $HT_s[j]$ and the bucket $HT_s[j]$ will participate in the following process of match, the same as all the remaining unmatched buckets. We

**Algorithm 4** Delete a key $\boldsymbol{k}$

```
Given the hash table HT_t[h], the key k
call the LOCATE(k) to get HT_r[i]
if above return is null then
  return NULL
else
  get HT_r[i]'s next-perferred key k' along with its bucket
  HT_s[j]
  while (s, j) is not NULL AND HT_r[i] is empty do
    if PREFER_OF_KEY(s, j, r, i) then
      HT_r[i] = k'
      r = s, i = j
      get current HT_r[i]'s next-perferred key k along with
      its bucket HT_s[j]
    end if
  end while
end if
```

conduct another operation **Rearrangement** besides the **Free** operation, where the **Free** means the termination of the match between the key and its corresponding bucket.

**Rearrangement**. The operation **Rearrangement** will actually happen among the keys which are not tied with their buckets. It is noted that such an operation will be actually conducted according to the core idea of the **G-S SMA**. For example, we let the ***process*** denote the duration between the checkpoint $t_{latest}$ and $t_{next}$ and the system is going to delete a key now.

The key will be removed, whose bucket will be freed, and such a slot will be ready for some key that is to be matched with the bucket and the bucket prefers most at the meanwhile. In **G-S SMA**, $m_2$ will tentatively replace $m_1$ if $m_2$ proposes to $w_1$ first and $w_1$ prefers $m_2$ than $m_1$. Then $m_1$ will be tentatively matched to his next-perferred woman $w_2$, who ranks lower than $w_1$ in the preference of $m_1$ for the set of women. Then we have $(m_2, w_1)$ and $(m_1, w_2)$. At this time, if the man $m_2$ quits the process, *i.e.*, deleting the man $m_2$, the $w_1$ will lose her partner and wish to find her next-preferred man. It is noted that in this example the $m_1$ will be the choice and he actually prefers $w_1$ than $w_2$. Therefore, $m_1$ should terminate the match with $w_2$ and take that slot of $w_1$ again in order to reach a final stable marriage result, leaving the $w_2$ unmatched. The $w_2$ will repeat the same process of searching for her next-preferred man.

Certainly, the **Rearrangement** operations will also work in the process of insertion as long as the collisions caused by the insertion bring about some newly-released keys. We suppose that the key $\boldsymbol{k}$ is matched with $HT_s[j]$, which has not been observed. There is a key $\boldsymbol{k'}$ to be inserted, which ranks higher than $\boldsymbol{k}$ in the preference list of $HT_s[j]$, also prefers $HT_s[j]$ most. Then the $\boldsymbol{k'}$ will replace $\boldsymbol{k}$ and $\boldsymbol{k}$ will be unmatched. We need to assign a new bucket to establish a new match with $\boldsymbol{k}$. The system searches for the $\boldsymbol{k}$'s next-perferred bucket (denoted as $HT_r[i]$) and checks whether $HT_r[i]$ is occupied with some key $\boldsymbol{k''}$. If not, the system directly assigns $HT_r[i]$ to the $\boldsymbol{k}$. If it is true, the system compares the $HT_r[i]$'s preference for $\boldsymbol{k''}$ and $\boldsymbol{k}$, and selects the key with a higher rank to be assigned to $HT_r[i]$, leaving the other unmatched. Then the system searches for another proper bucket for this unmatched key as above and conducts this

**Algorithm 5** The running process of eHIFS

As our system runs...
Supposing that the sequence of $T_0, T_1, T_2, \cdots$ are the detected unexpected non-uniform Time-Intervals
**while** the system runs **do**
  set $T = T_i$ $(i = 0, 1, 2, \cdots)$ as time goes by
  **while** $t$ has not reached $T$ **do**
    **if** locate a key $\boldsymbol{k}$ **then**
      call the algorithm 1
    **else if** insert a key $\boldsymbol{k}$ **then**
      call the algorithm 2
    **else if** delete a key $\boldsymbol{k}$ **then**
      call the algorithm 4
    **else**
      **return** ERROR
    **end if**
  **end while**
  Make all the observed buckets with their keys bound together until some unbinding operations happen
**end while**

series of operations recursively. The **Rearrangement** will finish when all the keys have been stably matched with buckets again.

The Algorithm 4 presents the process of **Deletion** including the **Rearrangement**, where the **Rearrangement** operation can be seen from Line 6 to Line 13. Analogous to the function PREFER_OF_BUCKET, the function PREFER_OF_KEY$(s, j, r, i)$ means that the key in $HT_s[j]$ prefers the bucket $HT_r[i]$ than its current bucket.

Freed from the match with its keys, each bucket will participate in the following process using **G-S SMA**. Its history key(s) will not be found, or in other words, such key(s) have never existed ever from the view of the attackers. The whole process of **Deletion** is also similar to the **Insertion**. The Algorithm 5 describes the whole process including all the aforementioned operations with the consideration of **TimeInterval**.

## 6 SECURITY ANALYSIS

This section provides security analysis showing that eHIFS can ensure SHI.

### 6.1 Preliminaries for Security Proof

The *representation* $\boldsymbol{a}$ of the *state* $A$ will follow those descriptions in **Definition** 2.2. We recall that the notation $\mathbf{Pr}[\boldsymbol{a} \xrightarrow{O} \boldsymbol{b}]$ denotes the probability that the representation $\boldsymbol{a}$ of state $A$ transforms into the representation $\boldsymbol{b}$ of state $B$ after the sequence of operations $O$ is performed on the system data. The strong history independence requires the canonical representations after the sequence of operations $O_1$ and $O_2$ independently performed for any previous state and any final state.

In our system, we allow the attacker to have audit actions and therefore we let $T = \{t_1, t_2, \cdots, t_r\}$ denote the checkpoints of the attacker's observations. The corresponding *states* at these checkpoints are denoted as a set $S = \{S_1, S_2, \cdots, S_r\}$ along with the corresponding distributions set $D = \{d_1, d_2, \cdots, d_r\}$. We also let $P = \{p_1, p_2, \cdots, p_r\}$ denote the process sets, where, for $\forall i \in 1, \cdots, r$, $p_i$ denotes the operation process from the checkpoint $t_{i-1}$ to $t_i$ and $p_1$ is the process during the period from the initialization *state* to the state at checkpoint $t_1$.

**The currently-involved buckets set**. In the process of dealing with data, our system will be faced with different sets of buckets performing different roles respectively. From the point of the **Reaction**, there are two kinds of basic states for the buckets: having been tied with their corresponding keys or having not been. Besides, many empty buckets are unmatched and wait for future potential keys. To distinguish these buckets of multiple roles for a clearly explained proof, we concern about the buckets which are assigned to the newly-matched keys, *i.e.*, taking participate in the process of match according to **G-S SMA**. We should notice that the buckets whose keys are replaced also belong to the set of buckets under such a concept, no matter whether they have been observed. We let $B_i$ denote the set of such buckets in the process $p_i$, where $1 \le i \le r$. When a single process is finished, its $B_i$ of this time-point is determined. However, $B_i$ will exclude some buckets and update itself *iff* some buckets' corresponding keys are deleted in some subsequent processes. The $B_i$ represents some related information about its process at different time. Therefore, we have a natural corollary 6.1, which will be used in the proof of eHIFS's strong history independence property.

Corollary 6.1. $\forall\ i \in \{1, \cdots, r\}$, $B_i$ *is non incremental.*

**The residue set**. Let $\mathcal{Q}$ denote the set of all buckets in our system and let the $\mathcal{R}_i$ denote the set of all the remaining, named **the residue set**, in the process $p_i$ such that we have

$$\mathcal{R}_i := \mathcal{Q} \backslash \bigcup_{j=1}^{i-1} B_j \tag{2}$$

For convenience, we will simply use $\mathcal{R}$ to replace $\mathcal{R}_i$ if we have declared the index of some specific process clearly.

**The distribution on specific buckets**. We use the symbol "$\sim$" to denote the meaning that the distribution on the discussed buckets set(s) since there so many different states of buckets. There **must** be clearly declared corresponding notations of distributions as long as we are talking about distributions on some buckets. For example, if we are talking about the distribution $d_1$ on bucket $B_1$ and distribution $d_2$ on buckets $B_1 \cup B_2$, we must use the symbol $\sim$ to bind the buckets and distributions on them (e.g., $d_2 \sim (B_1 \cup B_2)$). Because the $B_1$ in expression $d_1 \sim B_1$ may differ from the bucket $B_1$ in expression $d_2 \sim (B_1 \cup B_2)$ due to the possible update declared in the descriptions of the reasons for Corollary 6.1.

### 6.2 Security Proof

Security of eHIFS is captured in Theorem 6.2, which is proved in detail in Appendix A.

Theorem 6.2. eHIFS *can ensure strong history independence.*

## 7 SIMULATION AND EVALUATION

### 7.1 Simulation Details

We did not have eHIFS implemented/simulated either on the full-system level or on the full-file-system level. Instead, we simulated

all the core algorithms of eHIFS (e.g., Algorithm 1 to Algorithm 5) in the application layer. The simulator was written in C++ with 1,000+ lines of code. We provide a few more details about implementation of our simulator as follows:

We introduce two arrays in the memory representing keys and buckets. The key array represents all file data to be stored into the physical storage, and each key identifies a chunk of file data (e.g., 4KB). The bucket array represents the physical storage managed by the file system, organized into buckets. Each element (a *struct* data type) in the key array maintains the corresponding information about the key, including key index, its preference of buckets, index of potentially linked bucket (assigned during the process), a tag which indicates whether the key is deleted or not, etc. Each element (a *struct* data type) in the bucket array maintains the corresponding information about the bucket, including index of the bucket, address of the bucket, its preference list for keys, a tag which indicates whether the bucket is fixed (means that the data in this bucket has been observed) or not, etc.

Initially, we fix the preference lists for both keys and buckets. To determine the preference list of each bucket over keys, we compute a hash value (based on SHA-1) for each key over this bucket, using the address of this bucket and information about the key (e.g., the file path as well as the offset for the corresponding file data) as the input. For each bucket, we compare all its relevant hash values among keys, obtaining its preference list over keys, i.e., a bucket always prefers a key with a smaller hash value. We also determine the preference list of each key over buckets. The rule is, each key always prefers a bucket with a smaller index.

**Insertion simulation**. Inserting file data to a bucket is simulated by establishing a link between its key and the corresponding bucket. When the file data is removed from the bucket, the link should be removed. During each insertion, a rearrangement operation may be conducted if a collision happens, which is simulated by removing the existing link and establishing a new link between the newly-matching key and bucket. Note that rearrangement may happen recursively due to recursive collisions. After an observation from the adversary (i.e., a checkpoint), we mark all the stably-matching buckets and keys as fixed, i.e., each link remains unchanged until the corresponding key is deleted.

**Deletion simulation**. When file data is removed from a bucket, we remove the link between its key and the corresponding bucket. We mark the bucket (denoted as $b$) as unoccupied and the key as deleted using their respective tags. We then check whether or not all the unobserved buckets and their keys are still stably matched after $b$ is released. If it is true, the deletion is finished. Otherwise, there is a key $k$ (linking to the unobserved bucket), which prefers $b$ more than its current bucket $b'$, and $b$ also prefers $k$ most among all those keys (linking to the unobserved buckets) who prefer $b$. We need to remove the link between $k$ and $b'$, and establish a link between $k$ and $b$. Now $b'$ becomes unoccupied and we will repeat the previous checking process.

## 7.2 Experimental Evaluation

### 7.2.1 *Experimental Setup.*

*We ran our simulator (Sec. 7.1) on a local machine (4 Intel i7 CPUs, 2GB RAM, Ubuntu 18.04 with kernel v4.15.0-42-generic).* We used access traces extracted from running benchmark fio [1]. We also compared eHIFS and HIFS [3] in terms of write performance. For fairness, we simulated both of them on the same platform[3].

In our evaluation, we fixed the number of buckets/keys as 10,000 for simplicity, since we did not observe significant differences when varying this number. Hash table is sensitive to load factor. Therefore, similar to HIFS, we varied the file system load factor between 10% and 90% to assess performance impact under different load factors.

For eHIFS, the major impact of different types of workloads mainly comes from the different ratios of #insertion to #deletion. For example, for regular editing workloads, the number of insertions should be close to that of deletions, but for voting workloads, the number of insertions should be dominant. Therefore, to assess impact of eHIFS under different types of workloads, we chose the ratio of #insertion to #deletion as 10:1, 10:3, 10:5, 10:7. In addition, to evaluate impact of different Time-Intervals on eHIFS, we chose Time-Interval as 100, 500, 1000, respectively, each refers to the number of insert/delete operations performed between any two adjacent observations in the simulation.

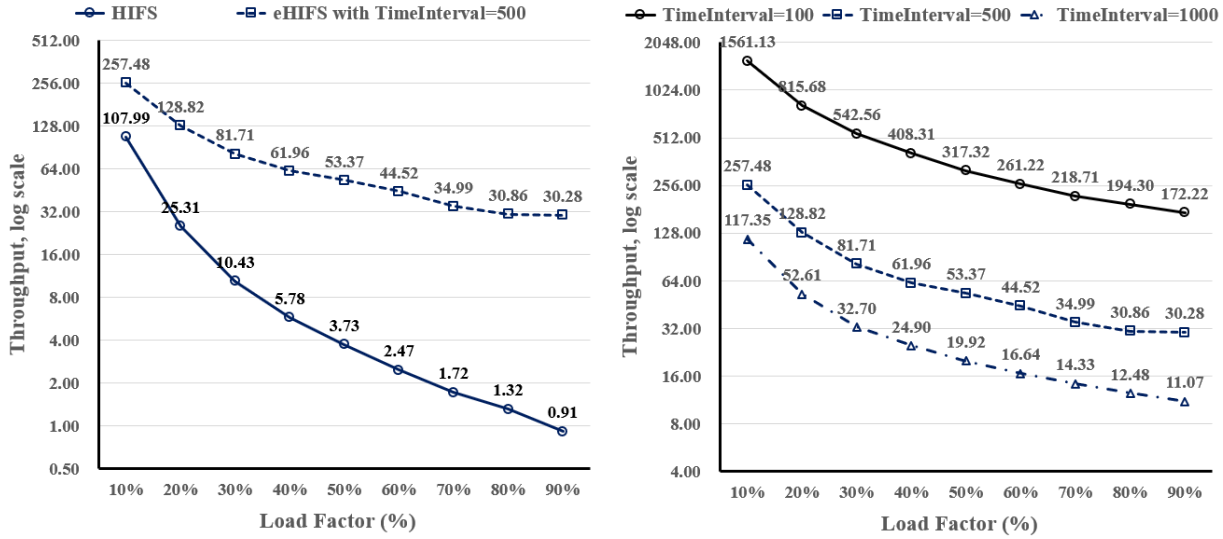
We will measure the write throughput for both eHIFS and HIFS, which is defined as the total number of keys being processed (i.e., inserted or deleted) per time unit.

Note that our experimental results are completely obtained from simulation, which is not able to capture overhead added purely by the actual system. However, we believe that the overhead added by the system is very close for both eHIFS and HIFS. Therefore, the comparion of them in the simulation should be close to that in the real-world system. In addition, how the performance of eHIFS varies with load factors and Time-Intervals in the simulation should be also close to that in the real system, since for each single write operation, the additional overhead added by the real system should be close to a constant value.

### 7.2.2 *Experimental Results.*

**Performance comparison between** eHIFS **and HIFS**. We compared write throughput between HIFS and eHIFS under different file system load factors, with Time-Interval set to a middle value, 500, for eHIFS. The experimental results are shown in Figure 5(a). We observe that the write throughput of both schemes decreases when the load factor increases. This is mainly due to handling collision of a history independent hash table, which turns more expensive when the hash table is filled with more elements. However, we observe that the write throughput of eHIFS decreases much more slowly than that of HIFS, which makes that eHIFS has a higher write throughput than HIFS under each load factor. For example, when the load factor is 50%, eHIFS is 14X faster than HIFS, and 33X faster when the load factor reaches 90%. This is because, by taking advantage of the adversary's observations, eHIFS can "fix" the data which have been observed, and the expensive rearrangement only happens among those data which have not been observed, significantly reducing unnecessary overhead.

[3]Note that eHIFS can easily support locality-preserving feature, since the introduction of the adversary's observations will not create any obstacles for preserving locality. For simplicity, we did not simulate the locality-preserving feature in both eHIFS and HIFS.




(a) A comparison between eHIFS and HIFS in write throughput.

(b) Write throughput of eHIFS under different Time-Intervals.

**Figure 5: Write throughput of eHIFS under different load factors and Time-Intervals.**

To assess impact of different types of workloads, we varied the ratio of #insertion (i.e., #INS) to #deletion (i.e., #DEL) for both eHIFS and HIFS. We also varied the file system load factor among 20%, 50%, and 80% for both schemes. For example, if the load factor is 20%, since the total number of buckets is 10,000, the total number of insertions is 2,000, which will occupy 2,000 buckets; and if the ratio of #INS to #DEL is 10:1, the total number of deletions is 200. For both schemes, we obtained: 1) the total number of collisions, and 2) the write throughput. For eHIFS, we chose Time-Interval as 500. The experimental results are shown in Table 2.

We have a few observations. First, for a fixed load factor, the number of collisions does not vary significantly with different types of workloads for both eHIFS and HIFS. However, when the load factor is increased, the number of collisions increases for both schemes. This is reasonable, since the collision rate of hash table is mainly determined by the load factor, and a large load factor usually results in a higher collision rate. Second, under a fixed load factor, for the same type of workload, the number of collisions in HIFS is larger than that of eHIFS. This is because, compared to HIFS, eHIFS fixes those data having been observed, which will not be relocated, naturally reducing the potential number of collisions. Third, for a fixed load factor, the throughput of eHIFS is always higher than HIFS for the same type of workload, due to two potential reasons: 1) eHIFS has less collisions compared to HIFS; and 2) the rearrangement due to collision is less expensive in eHIFS since the observed data are fixed and do not need to be relocated. Fourth, for a fixed load factor, the degree of performance improvement of eHIFS over HIFS decreases as the ratio of #INS to #DEL decreases. This is because, under a fixed Time-Interval, if the ratio of #INS to #DEL decreases, the total number of data being fixed will be decreased, reducing the effectiveness of eHIFS.

**Table 2: Hash table collision and throughput for different types of workloads ( Time-Interval for eHIFS is 500).**

| Load Factor | Ratio of #INS/#DEL | #Collisions | | Write Throughput | |
|---|---|---|---|---|---|
| | | eHIFS | HIFS | eHIFS | HIFS |
| 20% | 10:1 | 7892 | 9526 | 285.08 | 64.37 |
| 20% | 10:3 | 7730 | 7948 | 465.77 | 293.33 |
| 20% | 10:5 | 7654 | 7979 | 524.25 | 385.00 |
| 20% | 10:7 | 7375 | 7831 | 614.43 | 491.81 |
| 50% | 10:1 | 21515 | 27840 | 79.92 | 9.74 |
| 50% | 10:3 | 21547 | 24607 | 106.17 | 39.29 |
| 50% | 10:5 | 21459 | 24018 | 98.07 | 56.63 |
| 50% | 10:7 | 21575 | 24124 | 123.28 | 65.97 |
| 50% | 10:1 | 35876 | 47332 | 46.45 | 4.34 |
| 50% | 10:3 | 36332 | 42550 | 55.06 | 16.38 |
| 50% | 10:5 | 36316 | 41966 | 55.98 | 21.02 |
| 50% | 10:7 | 35750 | 42018 | 64.92 | 24.17 |

**Evaluating impact of Time-Interval on** eHIFS. We evaluated performance of eHIFS under different Time-Intervals and different file system load factors. The experimental results are shown in Figure 5(b). We observe that under a fixed load factor, a smaller Time-Interval has a higher write throughput. For example, when Time-Interval is chosen as 100, eHIFS is 15X more efficient than that when Time-Interval is chosen as 1000, under 70% load factor. Intuitively, a smaller Time-Interval implies that the adversary observes the disk more frequently, and eHIFS can utilize more knowledge from the adversary for optimizing performance. Technically, a small time-Interval means that there are less number of operations in this period, hence less number of new keys inserted independently

for this period; eHIFS follows HIFS except excluding those previously "fixed" data, and therefore the rearrangement among the unobserved keys (i.e., new keys) becomes less, leading to less overhead resulted from the expensive rearrangement and hence higher throughput.

### 7.3 Discussion

**Towards system-level implementation of eHIFS**. The real implementation of eHIFS is similar to that of HIFS, in which the entire disk is divided into block groups, each of which has its own inode table, map and data blocks. Data blocks are used to stored actual file data. Disk space is allocated to files in multiples of buckets, each of which is a collection of data blocks. Each block group maintains a disk buckets map, which keeps track of allocation of disk buckets to files. Each file system read/write operation will first locate a target entry in the disk buckets map and the actual read/write operation will then be performed on the corresponding bucket. The disk bucket maps from all block groups can be collectively viewed as a single history independent hash table. Each key of this hash table can be derived from file attributes (e.g., file paths, and read/write offsets). The major differences of eHIFS implementation from HIFS are: when inserting/deleting file data (identified by its corresponding key) into/from a disk bucket, the new layout of all the file data in the data blocks (as well as the new layout of the allocation information in the disk bucket maps) will be determined by combining knowledge from both the history independent hash table and the adversary's observations, as described in Sec 5.3.

In addition, various metadata are used in the file system, e.g., the inode tables each contains inode entries describing attributes of files and directories, the information recording each observation. They may be utilized by the attacker to derive sensitive historical information about the sensitive data. For example, by having access to these metadata, the adversary can identify past existence of deleted file data, since they may not have been sanitized even though the file data have been deleted. Therefore, they should be also organized in a history independent manner.

**The scalability of eHIFS**. In HIFS, inserting/deleting a file block may result in re-allocating all the $n$ blocks in the disk in the worse case, and therefore, inserting/deleting $n$ file blocks in HIFS has $O(n^2)$ complexity. eHIFS optimizes HIFS by utilizing observations from the adversary. Assume there are $m$ observations in total. Since eHIFS follows HIFS for file blocks between two subsequent observations, inserting/deleting $n$ file blocks in eHIFS should have a complexity $O(\frac{n^2}{m})$. For example, if $m = \sqrt{n}$, $O(\frac{n^2}{m})$ should be $O(n \cdot \sqrt{n})$. Compared to a regular non-history-independent file system (e.g., FAT, NTFS, EXT4) which usually has a complexity of $O(n)$, eHIFS has an additional $\sqrt{n}$ factor, rendering it a poorer scalability. To conclude, the scalability of eHIFS should be in the middle of HIFS and a regular non-history-independent file system.

**About read operations in** eHIFS. eHIFS improves efficiency of write operations by eliminating unnecessary re-locations. The strategy we use does not directly affect read operations, and the read throughput should be similar to that of HIFS [3]. Therefore, we do not provide experimental results for read, but readers are encouraged to refer to HIFS for read throughput.

**About encryption storage**. Our current work assumes that the adversary can have access to the original content of the storage (Sec. 4.3). If the storage is encrypted (e.g., encrypted file system or full disk encryption is used), we assume the adversary can have access to the decryption key. We currently do not consider the scenario that the adversary is not able to have access to the decryption key and hence not able to have access to the plaintext data. For such a scenario, the content is encrypted, and the adversary is not able to identify the abstract state of the data structure, and it is unclear to us how the history independence guarantee can be affected. Our intuition is, if storage is not history independent, by comparing multiple snapshots of the encrypted storage, the adversary is still able to derive the operation history by tracking relocations of encrypted data, compromising history independence. We will investigate the impact of encryption on history independence in our future study.

## 8 RELATED WORK

**History independence theory**. The origin of history independence could be traced back to oblivious data structure introduced by Micciancio [28], which yields no knowledge about sequence of operations applied to it other than final result of the operations. Naor et al. [31] initiated the concept of strong/weak history independence. Following efforts enriched history independence theory by designing various history independent data structures [9, 17, 21, 30, 40]. The connection between history independence and issues faced by voting system were studied in [7, 29].

**History independence systems**. Bajaj et al. [3] introduced HIFS, a history independent file system which ensures history independence when allocating disk blocks to file data, such that by having access to the disk layout, the adversary is not able to identify sequence of operations leading to it. A significant limitation of HIFS is that its write throughput decreases significantly when load factor is large. eHIFS eliminates this limitation by utilizing knowledge on the adversary's observations, eliminating unnecessary relocation of data for history independence. Bajaj et al. [2] also applied history independence to relational databases to achieve un-traceable deletion on data records. This design is specific for a certain type of data which stays on top of a file system. Another history independent system HiFlash [14] incorporated history independence into flash translation layer. Such a low-level design is specifically for flash storage media. eHIFS, however, is designed for file systems which stay in the upper layer and is a necessary component for all types of storage media.

**Auditable data structures**. There are many applications where observations on sensitive data are required over time, and privacy of history becomes more necessary than before. For example, after an election process, there are usually examinations on the voting machines [36]. The observations known by the system have been utilized to design an auditable data structure [22]. Unlike history independent data structures which cannot react to observations, an auditable data structure relaxes such an unrealistic restriction and is allowed to react to the observations, resulting in a more efficient construction with a similar privacy guarantee than history independence.

## 9 CONCLUSION

In this work, we propose eHIFS, the first efficient history independent file system. The key idea of eHIFS is to take advantage of the knowledge on the adversary, and actively react to each observation of the adversary. This can avoid unnecessary relocations of data while being able to remain history independent. Security analysis and experimental evaluation confirm that, eHIFS is much more efficient than the traditional history independent file system design and, meanwhile, achieve a similar SHI guarantee.

## ACKNOWLEDGMENTS

We would like to thank our shepherd, Aravind Prakash, and anonymous reviewers for their insightful suggestions for improving the paper. This work was partially supported by the National Natural Science Foundation of China (No. 61802395, No. 61602475, No.61602476 and No.61872357), the National S&T Major Project of China (No.2018ZX09201011), and National Cryptographic Foundation of China (Grant No. MMJJ20170212).

## REFERENCES

[1] Jens Axboe. [n. d.]. Flexible I/O tester. https://linux.die.net/man/1/fio.
[2] Sumeet Bajaj and Radu Sion. 2013. Ficklebase: Looking into the future to erase the past. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 86–97.
[3] Sumeet Bajaj and Radu Sion. 2013. HIFS: History independence for file systems. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 1285–1296.
[4] Steven Bauer and Nissanka B Priyantha. 2001. Secure data deletion for Linux file systems. In *Proceedings of the 10th conference on USENIX Security Symposium-Volume 10*. USENIX Association, 12.
[5] Rudolf Bayer. 1972. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta informatica* 1, 4 (1972), 290–306.
[6] Michael Bender, Jonathan Berry, Rob Johnson, Thomas Kroeger, Samuel McCauley, Cynthia Phillips, Bertrand Simon, Shikha Singh, and David Zage. 2016. Anti-Persistence on Persistent Storage: History-Independent Sparse Tables and Dictionaries. In *Principle of Database Systems (PODS 2016)*.
[7] John Bethencourt, Dan Boneh, and Brent Waters. 2007. Cryptographic Methods for Storing Ballots on a Voting Machine.. In *NDSS*. 209–222.
[8] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. 2014. Toward robust hidden volumes using write-only oblivious RAM. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 203–214.
[9] Guy E Blelloch and Daniel Golovin. 2007. Strongly history-independent hashing with applications. In *Foundations of Computer Science, 2007. FOCS'07. 48th Annual IEEE Symposium on*. IEEE, 272–282.
[10] Bing Chang, Yao Cheng, Bo Chen, Fengwei Zhang, Wen-Tao Zhu, Yingjiu Li, and Zhan Wang. 2017. User-friendly deniable storage for mobile devices. *Computers & Security* 72 (2017), 163–174.
[11] Bing Chang, Zhan Wang, Bo Chen, and Fengwei Zhang. 2015. MobiPluto: File System Friendly Deniable Storage for Mobile Devices. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC '15)*. ACM, 381–390.
[12] Bing Chang, Fengwei Zhang, Bo Chen, Yingjiu Li, Wen-Tao Zhu, Yangguang Tian, Zhan Wang, and Albert Ching. 2018. Mobiceal: Towards secure and practical plausibly deniable encryption on mobile devices. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 454–465.
[13] Bo Chen, Shijie Jia, Luning Xia, and Peng Liu. 2016. Sanitizing data is not enough!: towards sanitizing structural artifacts in flash media. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 496–507.
[14] Bo Chen and Radu Sion. 2015. HiFlash: A history independent flash device. *arXiv preprint arXiv:1511.05180* (2015).
[15] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.
[16] United States Congress. 1996. Health Insurance Portability and Accountability Act. http://www.hhs.gov/ocr/privacy/index.html
[17] Scott A Crosby and Dan S Wallach. 2009. Super-efficient aggregating history-independent persistent authenticated dictionaries. In *European Symposium on Research in Computer Security*. Springer, 671–688.
[18] David Gale and Lloyd S Shapley. 1962. College admissions and the stability of marriage. *The American Mathematical Monthly* 69, 1 (1962), 9–15.
[19] Simson L Garfinkel and Abhi Shelat. 2003. Remembrance of data passed: A study of disk sanitization practices. *IEEE Security & Privacy* 99, 1 (2003), 17–27.
[20] Roxana Geambasu, Tadayoshi Kohno, Amit A Levy, and Henry M Levy. 2009. Vanish: Increasing Data Privacy with Self-Destructing Data.. In *USENIX Security Symposium*. 299–316.
[21] Daniel Golovin. 2009. B-treaps: A uniquely represented alternative to B-trees. In *International Colloquium on Automata, Languages, and Programming*. Springer, 487–499.
[22] Michael T Goodrich, Evgenios M Kornaropoulos, Michael Mitzenmacher, and Roberto Tamassia. 2017. Auditable data structures. In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*. IEEE, 285–300.
[23] Peter Gutmann. 1996. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the Sixth USENIX Security Symposium, San Jose, CA*, Vol. 14. 77–89.
[24] Jason D Hartline, Edwin S Hong, Alexander E Mohr, William R Pentney, and Emily C Rocke. 2005. Characterizing history independent data structures. *Algorithmica* 42, 1 (2005), 57–74.
[25] Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. 2016. Nfps: Adding undetectable secure deletion to flash translation layer. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 305–315.
[26] Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. 2017. DEFTL: Implementing Plausibly Deniable Encryption in Flash Translation Layer. In *Proceedings of the 24th ACM conference on Computer and communications security*. ACM.
[27] Nikolai Joukov and Erez Zadok. 2005. Adding secure deletion to your favorite file system. In *Third IEEE International Security in Storage Workshop (SISW'05)*. IEEE, 8–pp.
[28] Daniele Micciancio. 1997. Oblivious data structures: applications to cryptography. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 456–464.
[29] Tal Moran, Moni Naor, and Gil Segev. 2007. Deterministic history-independent strategies for storing information on write-once memories. In *International Colloquium on Automata, Languages, and Programming*. Springer, 303–315.
[30] Moni Naor, Gil Segev, and Udi Wieder. 2008. History-independent cuckoo hashing. In *International Colloquium on Automata, Languages, and Programming*. Springer, 631–642.
[31] Moni Naor and Vanessa Teague. 2001. Anti-persistence: History independent data structures. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*. ACM, 492–501.
[32] Radia Perlman. 2005. *The Ephemerizer: Making Data Disappear*. Technical Report. Mountain View, CA, USA.
[33] Radia Perlman. 2005. File system design with assured delete. In *Third IEEE International Security in Storage Workshop (SISW'05)*. IEEE, 6–pp.
[34] Timothy M Peters, Mark A Gondree, and Zachary NJ Peterson. 2015. DEFY: A Deniable, Encrypted File System for Log-Structured Storage. In *22th Annual Network and Distributed System Security Symposium, NDSS*.
[35] Joel Reardon, Srdjan Capkun, and David Basin. 2012. Data node encrypted file system: Efficient secure deletion for flash memory. In *Proceedings of the 21st USENIX conference on Security symposium*. USENIX Association, 17–17.
[36] Ronald L Rivest and Emily Shen. 2012. A Bayesian Method for Auditing Elections.. In *EVT/WOTE*.
[37] Mendel Rosenblum and John K Ousterhout. 1991. The design and implementation of a log-structured file system. In *ACM SIGOPS Operating Systems Review*, Vol. 25. ACM, 1–15.
[38] United States. 2002. Senator Paul Sarbanes and U.S. Representative Michael G. Oxley. Sarbanes-Oxley Act. http://www.sec.gov/about/laws.shtml#sox2002.
[39] Yang Tang, Patrick PC Lee, John Lui, and Radia Perlman. 2012. Secure overlay cloud storage with access control and assured deletion. *Dependable and Secure Computing, IEEE Transactions on* 9, 6 (2012), 903–916.
[40] Theodoros Tzouramanis. 2012. History-independence: a fresh look at the case of R-trees. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM, 7–12.
[41] European Union. [n. d.]. REGULATION (EU) 2016/679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL. http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=uriserv:OJ.L_.2016.119.01.0001.01.ENG&toc=OJ:L:2016:119:TOC
[42] Michael Yung Chung Wei, Laura M Grupp, Frederick E Spada, and Steven Swanson. 2011. Reliably Erasing Data from Flash-Based Solid State Drives.. In *FAST*, Vol. 11. 8–8.
[43] Apostolis Zarras, Katharina Kohls, Markus Dürmuth, and Christina Pöpper. 2016. Neuralyzer: Flexible Expiration Times for the Revocation of Online Data. In *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy*. ACM, 14–25.
[44] Qionglu Zhang, Shijie Jia, Bing Chang, and Bo Chen. 2018. Ensuring data confidentiality via plausibly deniable encryption and secure deletion–a survey. *Cybersecurity* 1, 1 (2018), 1.

# A PROOF OF THEOREM 6.2

Proof. We let $S_k$ and $S_l$ denote any two *states* in $S$, where $k, l \in \{1, \cdots, r\}$ and $k \leq l$. We suppose that there are two sequences of operations that both transform the system from *state* $S_k$ to *state* $S_l$.

In eHIFS, the system will conduct **Reaction** operations to react to the attacker's observations, *i.e.*, eHIFS will be aware of these audit actions and the time-points of the **Reaction** are set according to these observations. All the observations are regarded as known during the period from $S_k$ to $S_l$. There are two conditions:

Condition 1: if there is no observation between $S_k$ and $S_l$ (excluding $S_k$ and $S_l$), *i.e.*, $l = k + 1$, the process between $S_k$ and $S_l$ is $p_l$. Then $p_l$ is run according to the basic **G-S SMA** and achieves strong history independence apparently [18].

Condition 2: if there is more than one observation between $S_k$ and $S_l$ (excluding $S_k$ and $S_l$), we let the set $\{S_{j_1}, \cdots, S_{j_i}\}$ denote the states corresponding to these observations with their corresponding distributions denoted as the set $\{d_{j_1}, \cdots, d_{j_i}\}$, where $k < j_1 \leq \cdots \leq j_i < l$. According to Sec. 6.1, the notation $d_k \overset{O_1}{\rightarrow} d_l$ denotes the transformation from $d_k$ to $d_l$ under the operation sequence $O_1$. Although the specific sequence of operations in the process from $d_{j_1}$ to $d_{j_2}$ is the subset of that from $d_k$ to $d_l$, we still use $d_{j_1} \overset{O_1}{\rightarrow} d_{j_2}$ for convenience to denote such a sub-process instead of introducing new notations. Here, the $O_1$ in $d_{j_1} \overset{O_1}{\rightarrow} d_{j_2}$ means the whole operation sequence is $O_1$. Then combining the probability theory, we have the Equation 3 for $O_1$ (replacing $O_1$ with $O_2$, we can have a corresponding equation for $O_2$, which we do not list here)

$$\begin{aligned}\mathbf{Pr}[d_k \overset{O_1}{\rightarrow} d_l] &= \mathbf{Pr}[(d_k \overset{O_1}{\rightarrow} d_{j_1}) \wedge (d_{j_1} \overset{O_1}{\rightarrow} d_{j_2}) \wedge \cdots \wedge (d_{j_i} \overset{O_1}{\rightarrow} d_l)] \\ &= \mathbf{Pr}[d_k \overset{O_1}{\rightarrow} d_{j_1}] \times \mathbf{Pr}[d_{j_1} \overset{O_1}{\rightarrow} d_{j_2} | d_k \overset{O_1}{\rightarrow} d_{j_1}] \times \cdots \\ &\quad \times \mathbf{Pr}[d_{j_i} \overset{O_1}{\rightarrow} d_l | (d_k \overset{O_1}{\rightarrow} d_{j_1}) \wedge (d_{j_1} \overset{O_1}{\rightarrow} d_{j_2}) \\ &\quad \wedge \cdots \wedge (d_{j_i - 1} \overset{O_1}{\rightarrow} d_{j_i})]\end{aligned} \tag{3}$$

Without loss of generality, we now take $\mathbf{Pr}[d_{j_1} \overset{O_1}{\rightarrow} d_{j_2} | d_k \overset{O_1}{\rightarrow} d_{j_1}]$ in the sequence of expressions for consideration.

$$\begin{aligned}&\mathbf{Pr}[d_{j_1} \overset{O_1}{\rightarrow} d_{j_2} | d_k \overset{O_1}{\rightarrow} d_{j_1}] \\ &= \mathbf{Pr}\left[(d_{j_1} \sim \bigcup_{z=1}^{j_1} B_z) \overset{O_1}{\rightarrow} (d_{j_2} \sim (\bigcup_{z=1}^{j_2} B_z)) \middle| d_k \overset{O_1}{\rightarrow} d_{j_1}\right] \\ &= \frac{\mathbf{Pr}\left[\left((d_{j_1} \sim \bigcup_{z=1}^{j_1} B_z) \overset{O_1}{\rightarrow} (d_{j_2} \sim (\bigcup_{z=1}^{j_2} B_z))\right) \wedge \left(d_k \overset{O_1}{\rightarrow} d_{j_1}\right)\right]}{\mathbf{Pr}[d_k \overset{O_1}{\rightarrow} d_{j_1}]}\end{aligned} \tag{4}$$

For the same reason that a single process between two adjacent checkpoints is strongly history independent, we have

$$\mathbf{Pr}[d_k \overset{O_1}{\rightarrow} d_{j_1}] = \mathbf{Pr}[d_k \overset{O_2}{\rightarrow} d_{j_1}]$$

according to **Definition** 2.2. Therefore we now consider the numerator. We rewrite the numerator in Equation 4 as $\mathbf{Pr}[E_{11} \wedge E_{12}]$, where $E_{12}$ denotes the process $p_{j_1}: d_k \overset{O_1}{\rightarrow} d_{j_1}$, and $E_{11}$ denotes the remaining part. Accordingly we let $E_{21}$ denote $(d_{j_1} \sim \bigcup_{z=1}^{j_1} B_z) \overset{O_2}{\rightarrow} (d_{j_2} \sim (\bigcup_{z=1}^{j_2} B_z))$ and let $E_{22}$ denote the process $p_{j_1}$ in the equation of $O_2$'s version. Since we are considering the probability of distributions while both $O_1$ and $O_2$ achieve the same distribution at the checkpoint $j_1$ with the same probability, the main influential factors are actually just $E_{11}$ and $E_{21}$. Now consider the $E_{11}$ and the attacker has observed the system at checkpoint $j_1$.

The buckets $\bigcup_{z=1}^{j_1} B_z \sim d_{j_1}$ will be tied with their corresponding keys until deleted, and sequential operations will update them to $\bigcup_{z=1}^{j_2} B_z \sim d_{j_2}$ with a bucket $B_{j_2}$ added. They are all non-incremental (see Corollary 6.1), *i.e.*, $\bigcup_{z=1}^{j_1} B_z \sim d_{j_1}$ just excludes some buckets whose keys are deleted in the process $p_{j_2}$. Since $S_{j_1}$(also $S_{j_2}$) for $O_1$ and $S_{j_1}$(also $S_{j_2}$) for $O_2$ are the same, the deleted keys are identical for these two sequences. $O_1$ and $O_2$ only differ from each other in the order of operations. Therefore, the distributions $d_{j_2} \sim \bigcup_{z=1}^{j_1} B_z$ under $O_1$ and $d_{j_2} \sim \bigcup_{z=1}^{j_1} B_z$ under $O_2$ are identical because they have the same *state* at the start and the end of the process. Besides, the other involved buckets can be represented as $\mathcal{R} \cap B_{j_2}$, where the $\mathcal{R}$ will update itself sequentially as the process runs. It is noted that these buckets will become the buckets $B_{j_2}$ finally when eHIFS reacts to the audit at checkpoint $j_2$. As our proposal describes, these buckets will participate in operations according to our algorithm in a single process, which will lead to the canonical distributions on storage media, *i.e.*, are strongly history independent.

Therefor the $E_{11}$ is identical to $E_{21}$. Considering the statements about the other parts of the Equation 4, we can come to a conclusion that "$O_1$" can be replaced with $O_2$, *i.e.*,

$$\mathbf{Pr}[d_{j_1} \overset{O_1}{\rightarrow} d_{j_2} | d_k \overset{O_1}{\rightarrow} d_{j_1}] = \mathbf{Pr}[d_{j_1} \overset{O_2}{\rightarrow} d_{j_2} | d_k \overset{O_2}{\rightarrow} d_{j_1}] \tag{5}$$

Then the remaining parts of expression in Equation 3 will have similar results, which will result in the final Equation 6, showing that eHIFS is strongly history independent according to the Definition in 2.2.

$$\begin{aligned}\mathbf{Pr}[d_k \overset{O_1}{\rightarrow} d_l] &= \mathbf{Pr}[d_k \overset{O_1}{\rightarrow} d_{j_1}] \times \mathbf{Pr}[d_{j_1} \overset{O_1}{\rightarrow} d_{j_2} | d_k \overset{O_1}{\rightarrow} d_{j_1}] \times \cdots \\ &\quad \times \mathbf{Pr}[d_{j_i} \overset{O_1}{\rightarrow} d_l | (d_k \overset{O_1}{\rightarrow} d_{j_1}) \wedge (d_{j_1} \overset{O_1}{\rightarrow} d_{j_2}) \\ &\quad \wedge \cdots \wedge (d_{j_i - 1} \overset{O_1}{\rightarrow} d_{j_i})] \\ &= \mathbf{Pr}[d_k \overset{O_2}{\rightarrow} d_{j_1}] \times \mathbf{Pr}[d_{j_1} \overset{O_2}{\rightarrow} d_{j_2} | d_k \overset{O_2}{\rightarrow} d_{j_1}] \times \cdots \\ &\quad \times \mathbf{Pr}[d_{j_i} \overset{O_2}{\rightarrow} d_l | (d_k \overset{O_2}{\rightarrow} d_{j_1}) \wedge (d_{j_1} \overset{O_2}{\rightarrow} d_{j_2}) \\ &\quad \wedge \cdots \wedge (d_{j_i - 1} \overset{O_2}{\rightarrow} d_{j_i})] \\ &= \mathbf{Pr}[d_k \overset{O_2}{\rightarrow} d_l]\end{aligned} \tag{6}$$

□