Generalizing Database Forensics

KYRIACOS E. PAVLOU and RICHARD T. SNODGRASS, University of Arizona

In this article we present refinements on previously proposed approaches to forensic analysis of database tampering. We significantly generalize the basic structure of these algorithms to admit new characterizations of the "where" axis of the corruption diagram. Specifically, we introduce *page-based partitioning* as well as *attribute-based partitioning* along with their associated corruption diagrams. We compare the structure of all the forensic analysis algorithms and discuss the various design choices available with respect to forensic analysis. We characterize the forensic cost of the newly introduced algorithms, compare their forensic cost, and give our recommendations.

We then introduce a comprehensive *taxonomy* of the types of possible corruption events, along with an associated *forensic analysis protocol* that consolidates all extant forensic algorithms and the corresponding type(s) of corruption events they detect. The result is a generalization of these algorithms and an overarching characterization of the process of database forensic analysis, thus providing a context within the overall operation of a DBMS for all existing forensic analysis algorithms.

Categories and Subject Descriptors: H.2.7 [**Database Management**]: Database Administration—*Security, integrity, and protection*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized access*

General Terms: Algorithms, Performance, Security

Additional Key Words and Phrases: Compliant records, forensic analysis algorithm, forensic cost, Monochromatic Algorithm, a3D Algorithm, page-based partitioning, attribute-based partitioning, corruption event taxonomy, forensic analysis protocol.

ACM Reference Format:
Pavlou, K. E. and Snodgrass, R. T. 2013. Generalizing database forensics. ACM Trans. Datab. Syst. 38, 2, Article 12 (June 2013), 43 pages.
DOI: http://dx.doi.org/10.1145/2487259.2487264

12

1. INTRODUCTION

Regulations and societal expectations have recently emphasized the need to mediate access to valuable databases, even access by insiders. Fraud occurs when a person tampers illegally with a database and tries to hide illegal activity. There are thousands of regulations [Gerr et al. 2003] that mandate how data should be managed and equally numerous laws for ensuring the correct storage, use, and maintenance of databases on extant DBMSes ([HIPAA, US Department of Health & Human Services 1996; Sarbanes-Oxley Act, U.S. Public Law No. 107–204, 116 Stat. 745 2002]).

Data owners would like to be assured that such tampering has not occurred, or if it does, that it will be quickly discovered. This need has spurred a high demand for

NSF grants IIS-0415101, IIS-0639106, IIS-0803229, IIS-1016205, and EIA-0080123 and a grant from Microsoft provided partial support for this work.
Authors' addresses: Kyriacos E. Pavlou, (Current address) Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801-2302, kpavlou@illinois.edu; Richard T. Snodgrass, Department of Computer Science, University of Arizona, Tucson, AZ 85721-0077, rts@cs.arizona.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2013 ACM 0362-5915/2013/06-ART12 $15.00
DOI: http://dx.doi.org/10.1145/2487259.2487264

integrated solutions for ensuring the integrity of databases, detecting data corruption even by insiders, and performing forensic analysis on such corruptions in a variety of application domains.

A previous paper by our research group on tamper detection proposed a new approach in which cryptographically strong one-way hash functions prevent an intruder, including an auditor or an employee or even an unknown bug within the DBMS itself, from silently corrupting the audit log [Malmgren 2007; Snodgrass et al. 2004]. This is accomplished by using a a database with *transaction time semantics* in which all data manipulated by transactions are cumulatively hashed as they become available to the system. This generates a hash chain whose value at each time instant represents all the data in the database. The hash values are periodically *notarized* and then *validated* in order to detect if the database audit log has been altered.

The question then arises, what should be done when an intrusion has been detected? At that point, all that is known is that at some time in the past, data somewhere in the database have been altered. *Database Forensics* is needed to ascertain *when* the intrusion occurred, *what* data were altered, and ultimately, *who* the adversary is.

In subsequent papers we introduced several algorithmic tools used in tamper detection and forensic analysis. Specifically, we have developed several forensic analysis algorithms, including Monochromatic [Pavlou and Snodgrass 2006], RGB [Pavlou and Snodgrass 2006], Tiled-Bitmap [Pavlou and Snodgrass 2010], RGBY [Pavlou and Snodgrass 2008], and a3D [Pavlou and Snodgrass 2008]. We have shown how the execution of each of these algorithms and their results can be succinctly captured in a *corruption diagram*. We have also introduced the concept of *forensic cost* for these algorithms.

These previous algorithms and their associated corruption diagrams utilize a single "where" dimension, that of commit time. This provides a very important clue as to what data was tampered with as well as perhaps who perpetrated the tampering. One limitation of these algorithms is that this clue only can go so far. It would be useful to provide more semantic information during forensic analysis. We show how to provide alternate characterizations of "where," specifically physically where in the database (which page or pages) as well as which values were corrupted. Doing so will provide more flexibility to the Chief Security Officer (CSO) in configuring forensic analysis to the semantics of the database and to the environment of the database and its associated threats and will provide more information in the event of tampering.

A second major limitation of previous work is the inadequate characterization of the *space* of possible corruptions and the concomitant lack of understanding of the *comprehensiveness* of extant tamper detection forensic analysis algorithms: the extent to which existing algorithms can identify all possible corruptions. For example, straightforward application of an existing forensic analysis algorithm can only identify multiple sites of corruption. The result has no additional information with regard to how the corruption affected the data: whether an attribute value was changed, or a timestamp was backdated or postdated, or even whether the schema of the database was changed.

Problem Statement. Existing techniques are not generalizable. What is needed are generalized forensic algorithms that support multiple characterization of the "where" aspect of the corruption. Also needed is a broader understanding of the types of corruptions possible as well as a general description of the process of database forensic analysis.

In this article, we build on this work and provide several refinements to extant tamper detection and forensic analysis techniques. Our contributions are the following.

—We extend the notion of hashing transactions based on timestamps, first to physical pages and then to any general time-correlatable field. The latter significantly

generalizes the basic structure of these algorithms to admit new characterizations of the "where" axis of the corruption diagram. Specifically, we introduce *page-based partitioning* as well as *attribute-based partitioning*.
—We compare the structure of all the forensic analysis algorithms and discuss the various design choices available with respect to forensic analysis. We compute the forensic cost of the newly introduced algorithms, compare their forensic cost, and give our recommendations.
—We introduce a comprehensive *taxonomy* of the types of possible corruption events.
—We introduce a *forensic analysis protocol* that consolidates all extant forensic algorithms. Employing this protocol achieves the detection of different types of corruption events characterized in the taxonomy.

The result of these four contributions is a generalization of the forensic algorithms and an overarching characterization of the process of database forensic analysis, thus providing a context within the overall operation of a DBMS. The importance of the contributions lies in the fact that this work presents a general identification of the steps that need to be taken from the time a tampering is detected until the corruption sites and the types thereof have been identified. By following these steps, anyone charged with the security of a database can thereby obtain a more complete explanation of what data were affected, in what ways the data were affected, and when this corruption transpired. This explanation is customizable to the needs of the user or to the demands of the application domain.

Section 2 discusses the audit system and different design choices required for the system's architecture. Section 3 reviews the key ideas behind corruption diagrams and forensic analysis algorithms. In particular we revisit perhaps the two most representative of the forensic analysis algorithms: the Monochromatic and a3D algorithms. In Sections 4 and 5 we show how forensic analysis algorithms can be generalized by partitioning the database data according to pages and attributes, respectively. Section 6 summarizes these new forensic analysis algorithms. We then cover the various design choices available with respect to forensic analysis, while Section 8 compares the structure of the complete collection of forensic analysis algorithms. Sections 9 and 10 characterize the forensic cost of the newly introduced algorithms, compare their forensic cost, and give recommendations. Section 11 introduces a formal definition of forensic analysis and provides a taxonomy of corruption event types along with an associated forensic analysis protocol. This is followed by related work. The article concludes with an overall summary and directions for future work. Finally, an electronic Appendix provides a detailed description of the step-by-step process of forensic analysis protocol to identify the type of a detected corruption.

2. THE AUDIT SYSTEM

In this section we describe how to audit a database and summarize the *tamper detection* approach we previously proposed and implemented [Snodgrass et al. 2004]. We give the gist of our approach, so that the refinements we subsequently propose can be understood. Table I lists the audit system execution phases, their subphases, and the actions performed during each. Figure 1 shows the architecture of the system and the actions performed during the *Normal Processing* execution phase.

The basic approach to normal processing differentiates between the *Total Chain Computation* subphase, in which transactions are hashed and resulting values are digitally notarized, and the *Tamper Detection and Partial Chain Computation* subphase, in which the hash values are recomputed and compared with those previously notarized. It is during validation of the total chain that tampering is detected, when the just-computed hash value doesn't match those previously notarized. Also, certain

Table I. Audit System Execution Phases, Subphases, and Actions

Execution Phases	Subphases	Actions
Normal Processing	Total Chain Computation	– Hashing to create total chain – Notarization of total chain
	Tamper Detection and Partial Chain Computation	– Re-hashing of total chain – Validation of total chain *If required by forensic algorithm:* – Hashing to create partial chains – Notarization of partial chains
Forensic Analysis	Corruption Region Analysis	– Running a forensic algorithm to determine where and when
	Manual Analysis	– Determining who and why

algorithms like a3D require the computation and notarization of one or more partial hash chains during the scan of the entire database that occurs during validation.

The actions for both normal processing subphases are illustrated in Figure 1. In Figure 1, the user application performs transactions on the database, each of which inserts, deletes, and updates rows of the current state. Behind the scenes, the DBMS maintains the audit log by rendering a specified relation as a *transaction-time* table. This instructs the DBMS to retain previous tuples during update and deletion, along with their insertion and deletion/update time (the start and stop timestamps [Snodgrass and Ahn 1986]), in a manner completely transparent to the user application [Bair et al. 1997]. (Thus the tuples of a transaction are those inserted, deleted, or updated by the transaction.) We note that IBM [IBM Corporation 2010], Oracle [Oracle Corporation. 2009] and Teradata [Teradata Corporation. 2012] DBMSes now support transaction-time tables in such a transparent manner.

An important property of all compliant data stored in the database is that they are *append-only*: modifications only add information; no information is ever deleted [Snodgrass and Ahn 1986]. Hence, if old information is changed in any way, then tampering has occurred. How this information is stored (e.g., in the log, in the relational store proper, in a separate "archival store" [Ahn and Snodgrass 1988]) is not critical in terms of tamper detection, as long as previous tuples are accessible in some way.

On each modification of a tuple, the DBMS and the Notarizer are responsible for *hashing* the tuples. (Our implementation uses the SHA-1 cryptographic hash function [US National Institute of Standards and Technology 2012] provided by the `beecrypt-4.1.2` library.) When that transaction commits, the DBMS obtains a time-stamp and computes a cryptographically strong one-way hash function of the data in the tuple and the timestamp. This type of hashing is termed *holistic* in order to distinguish it from the faster, but more complex, *opportunistic* and *incremental* hashing where tuples are hashed as soon as they are written to the database buffer [Snodgrass et al. 2004].

The DBMS then periodically performs a *notarization* by sending that hash value as a digital document to an independent external digital notarization service, such as Surety (www.surety.com), and obtaining a notary ID, thereby locking the contents and time of that digital document [Haber and Stornetta 1999]. The returned notary ID along with the initially computed hash values are stored in the secure master database [Malmgren 2007].

Later, say an adversary, let us call her Mala, gets access to the database. If she changes the data or a timestamp, the notary ID obtained from their notarization no longer corresponds to them. Mala cannot manipulate the data or the timestamp so that

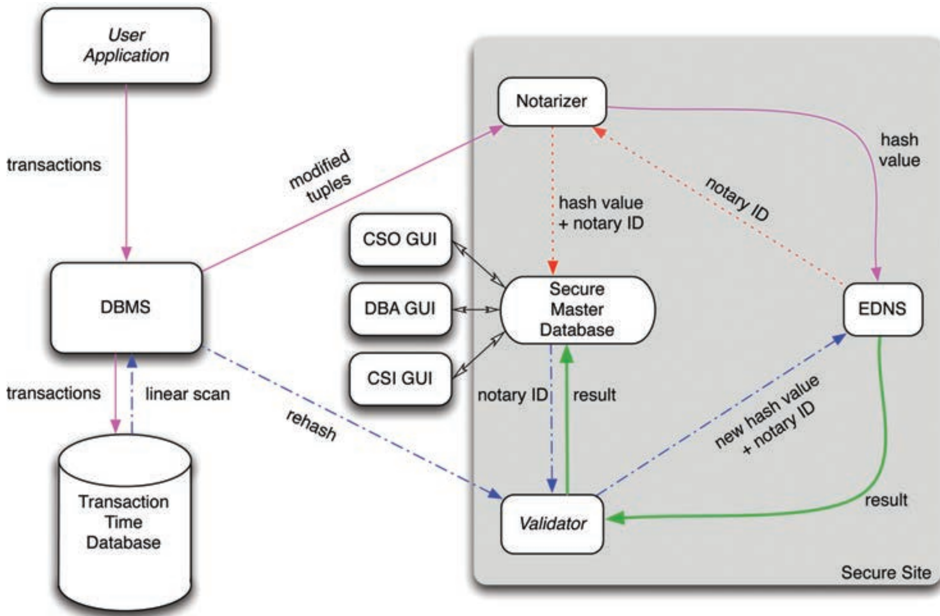


Fig. 1. Normal Processing comprises of (a) Computation of Total Chain and (b) Tamper Detection & Computation of Partial Chains.

the notary ID remains valid, because the hash function is one-way. Note that this holds even when Mala has access to the hash function itself. It is computationally infeasible to alter other tuples so that the same hash value results.

An independent validation service later scans the entire database (as illustrated in Figure 1). The Validator hashes the data and the timestamp of each tuple and provides the resulting hash value along with the previously obtained notary ID to the notarization service, which then checks the notarization time with the stored timestamp. The validation service then reports whether the database has been compromised.

If Mala decides to undo the corruption and is quick enough to revert the corrupted data to their original values before a validation check, such a temporary corruption will not be detected during tamper detection; the validation of the database will report that the database had not been compromised. Even though, at first glance, this appears to be a problem, we need to keep in mind that Mala's remedial action leaves the database in an legal state so we do not consider this attack as a corruption of the data stored in the the database. A full threat model concerning the audit system can be found elsewhere [Snodgrass et al. 2004]. Also, all important design choices discussed in this section will be summarized in Table V.

During the same execution of Tamper Detection and Partial Chain Computation the validator computes partial chains whose resulting values are then sent to be notarized (Figure 1). The creation and use of the partial chains for forensic analysis are discussed in Section 3.1.

For our purposes, the only detail important for forensic analysis is that at commit time, the transaction's hash value and the previous hash value are hashed together to obtain a new hash value. Thus, the hash values of the modified tuples of each individual transaction are linked in a sequence, then linked with each subsequent transaction, with the final value being essentially a hash of all changes to the database since the database was created. Hence, this chain is termed *total*.

Table II. Assumptions of Commit-Time-Based Monochromatic and a3D Algorithms

Monochromatic	a3D
Uses total cumulative chain but does not use partial chains	Uses total cumulative chain as well as partial chains which form a binary tree
Can only detect a single corruption with no false positives. If multiple corruptions exist it detects the one affect data with the oldest commit-time.	Can detect multiples corruptions with no false positives.
$V \in \mathbb{N}$	$V = 1$
$N = 1$	$N \in \mathbb{N}$

For more details on exactly how the tamper detection approach works, please refer to our previous paper [Snodgrass et al. 2004], which presents the threat model used by this approach, discusses performance issues, and clarifies the role of the external digital notarization service.

The validator provides a vital piece of information, that tampering has taken place, but doesn't offer much else. Since the hash value is the accumulation of every transaction ever applied to the database, we don't know when the tampering occurred, or what portion of the audit log was corrupted. We now turn to the details of how validation results can be leveraged to perform forensic analysis.

3. THE FORENSIC ANALYSIS ALGORITHMS

In this section we give an overview of important forensic tools and concepts that are central to the refinements discussed in the later sections. Table II summarizes the assumptions of the of Monochromatic and a3D Algorithms. A summary of the notation used along with definitions are given in Table III.

3.1. The Monochromatic Algorithm

Once the corruption has been detected, the *forensic analyzer* (a program) springs into action. The task of this analyzer is to ascertain, as accurately as possible, the *corruption region*: the bounds on "where" and "when" of the corruption.

From the last validation event, we have exactly one bit of information: validation failure. However, the forensic analyzer can use just the database itself to determine bounds on the corruption time (t_c) and the locus time (t_l).

The *Monochromatic Forensic Analysis Algorithm* yields the rectangular corruption region illustrated in Figure 2, with an area of 12 days^2 (two days wide by six days tall). Note that even though the two axes have different semantics, both are measured in days. To reach this result we must first note that the most recent *VE* before FVF is VE_3 and it was successful. This implies that the corruption event has occurred in this time period. Thus t_c is somewhere within the last I_V, which always bounds the "when" of the CE.

To bound the "where," the Monochromatic Algorithm can validate prior portions of the database, at times that were earlier notarized. Consider the very first notarization event, NE_1. The forensic analyzer can rehash all the transactions in the database in order, starting with the schema and then from the very first transaction (such data will have a commit time earlier than all other data), and proceeding up to the last transaction before NE_1. If that newly computed hash value matches the notarized hash value, the validation result will be true, and this validation will succeed, just like the original one would have, had we done a validation query then. Assume likewise that NE_2 through NE_7 succeed as well.

Table III. Summary of Notation Used
Page numbers of where each definition can be found in Pavlou and Snodgrass [2008] are also given.

Symbol	Name	Definition	Page #
CE	Corruption event	An event that compromises the database	30:6
VE	Validation event	The validation of the audit log by the notarization service	30:6
NE	Notarization event	The notarization of a document (hash value) by the notarization service	30:7
l_c	Corruption locus data	The corrupted data	30:7
t_n	Notarization time	The time instant of a NE	30:7
t_v	Validation time	The time instant of a VE	30:7
t_c	Corruption time	The time instant of a CE	30:6
t_l	Locus time	The time instant that l_c was stored	30:7
I_V	Validation interval	The time between two successive VEs	30:7
I_N	Notarization interval	The time between two successive NEs	30:7
R_t	Temporal detection resolution	Finest granularity chosen to express temporal bounds uncertainty of a CE	30:7
R_s	Spatial detection resolution	Finest granularity chosen to express spatial bounds uncertainty of a CE	30:7
t_{RVS}	Time of most recent validation success	The time instant of the last NE whose revalidation yielded a true result	30:12
t_{FVF}	Time of first validation failure	Time instant at which the CE is first detected	30:9
USB	Upper spatial bound	Upper bound of the spatial uncertainty of the corruption region	30:12
LSB	Lower spatial bound	Lower bound of the spatial uncertainty of the corruption region	30:12
UTB	Upper temporal bound	Upper bound of the temporal uncertainty of the corruption region	30:12
LTB	Lower temporal bound	Lower bound of the temporal uncertainty of the corruption region	30:12
V	Validation factor	The ratio I_V/I_N	30:14
N	Notarization factor	The ratio I_N/R_s	30:23

Of course, the original VE_1 and VE_2, performed during normal database processing, succeeded, but we already knew that. What we are focusing on here are validations of portions of the database performed by the forensic analyzer after tampering was detected. Computing the multiple hash values can be done in parallel by the forensic analyzer. The hash values are computed for each transaction during a single scan of the database and linked in commit order. Whenever a midnight is encountered as a transaction time, the current hash value is retained. When this scan is finished, these hash values can be sent to the notarization service to see if they match.

Now consider NE_8. The corruption diagram implies that the validation of all transactions occurring during day 1 through day 16 failed. That tells us that the "where" of this corruption event was the single I_N interval between the midnight notarizations of NE_7 and NE_8, that is, during day 15 or day 16. Note also that all validations after that, NE_9 through NE_{11}, also fail.

In general, we observe that revisiting and revalidating the total chain at past notarization events will yield a sequence of validation results that start out to be true and then at some point switch to false (TT...TF...FF). This single switch from true to false is a consequence of the cumulative nature of the total hash chain.

We term the time of the last NE whose revalidation yielded a true result (before the sequence of false results starts) the *time of most recent validation success* (t_{RVS}). This t_{RVS}

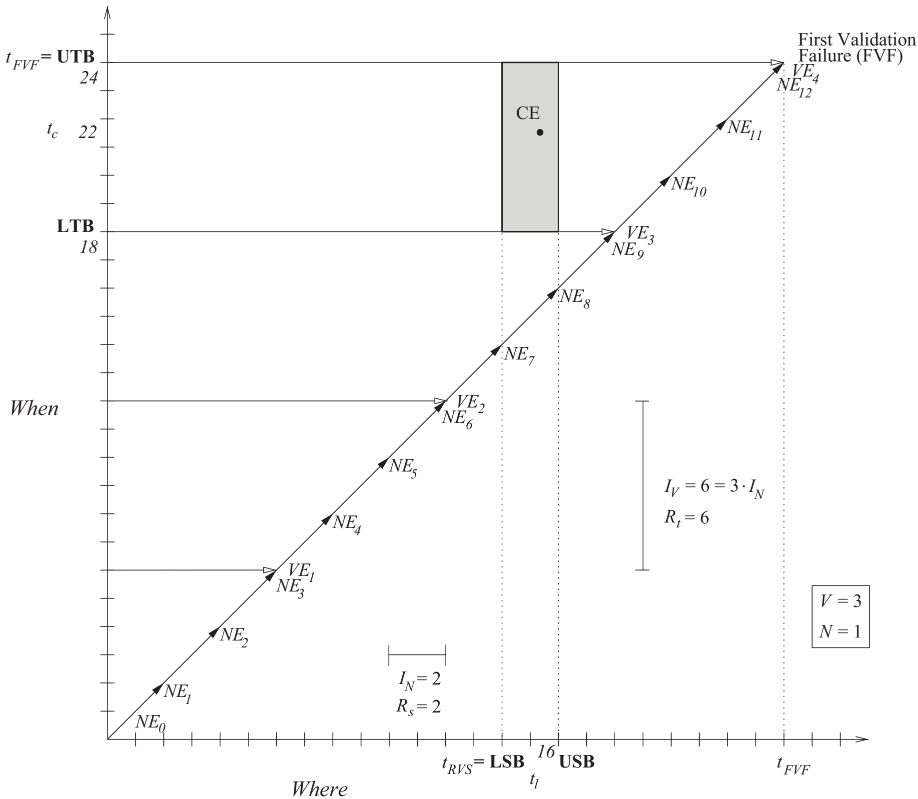


Fig. 2. Corruption diagram for the transaction-time-based Monochromatic Algorithm.

helps bound the where of the CE because the corrupted tuple belongs to a transaction which committed between t_{RVS} and next time database was notarized (whose validation now evaluates to false). t_{RVS} is marked on the Where axis of the of the corruption diagram, as seen in Figure 2.

In practice the Monochromatic Algorithm can quickly compute the bounds on the "where" of the CE by determining the interface between the true and false validation results, achieved by performing a binary search on notarization events. More details on the implementation of the Monochromatic Algorithm can be found elsewhere [Pavlou and Snodgrass 2008].

3.2. The a3D Algorithm

The Monochromatic Algorithm and its page-based counterpart cannot isolate the "where" of a corruption event to a resolution finer than R_s nor can it differentiate multiple corruptions. The a3D Algorithm [Pavlou and Snodgrass 2008], an extension that extensively utilizes partial chains for a more comprehensive analysis, is illustrated in Figure 3. (Note that the values for $V = 1$ and $N = 2$ are different from the ones in Figure 2 because the two figures illustrate different algorithms. The a3D Algorithm in Figure 3 requires that $V = 1$ while the Monochromatic Algorithm in Figure 2 requires that $N = 1$.) Even though the corruption diagram shows only *VE*s, it is implicit that these were preceded immediately by notarization events (not shown). The difference between a3D and the other algorithms (including Monochromatic) is a slowly increasing

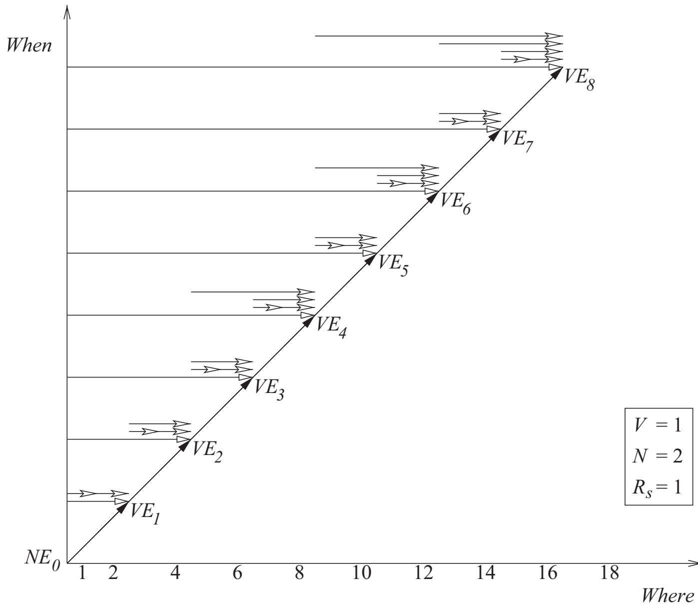


Fig. 3. Corruption diagram for the transaction-time-based a3D Algorithm.

number of (partial) chains at each validation. Thus in addition to the cumulative total chain (45-degree line in Figure 3) being maintained several smaller chains (shown with white-tipped arrows in Figure 3) that only hash specific parts of the data in the database (hence the name "partial") are computed during the validation scan.

The beauty of this algorithm is that it decides which chains to add based on the current time period induced by R_s. In this way the number of chains increases dynamically and thus maintains a binary tree/forest structure of hash chains covering the data in the database. This allows us to perform recursive binary search over the tree structure in order to locate the corruption(s).

Full details on the a3D Algorithm and an analysis of its forensic cost are provided elsewhere [Pavlou and Snodgrass 2008]. Fortunately, those details are not needed to understand the refinement and generalization of this a3D Algorithm to a page-based one.

As mentioned at the beginning of this article, the prior work just described is limited, in that they all provide a single characterization of the "where" dimension, that of commit time. In the following, we provide several refinements to extant tamper detection and forensic analysis techniques. Specifically, we build upon these algorithms to achieve significant generalizations and an overarching characterization of forensic analysis.

In the next section we elaborate on our previous approach to database forensics. We expand the basic structure of the forensic algorithms, so that they admit new characterizations of the "where" axis within the corruption diagram. Doing so will result in significant generalizations of the extant forensic analysis algorithms. Then in Section 11 we present a taxonomy and decision graph of corruptions, characterizing forensic analysis as a map from observables to types of corruptions in the taxonomy.

4. EXPANDING THE CONCEPT OF "WHERE"

The forensic analysis algorithms just described apply to a database for which the "where" axis is labeled by transaction commit time. (This axis also applies to the other

forensic algorithms in the literature [Pavlou and Snodgrass 2006, 2008].) In essence, those algorithms all view the data as partitioned on a very particular attribute within each tuple: the commit time.

In this section we generalize these algorithms by partitioning the database on any attribute that can be correlated with real time. We start by applying the same techniques to a database partitioned into pages, thereby analyzing each corruption from a different, *"spatial"* perspective. One environment where this might be especially effective is when the database is physically distributed, so that knowing which pages were altered could provide important information to identify who was responsible for the tampering.

4.1. Static-Numbered Page Hashing

In the Monochromatic Algorithm and also in the other forensic algorithm discussed here and elsewhere [Pavlou and Snodgrass 2008], the tuples of transactions are hashed and then linked to produce a total chain. Here we generalize the forensic algorithms to instead hash pages, specifically, the data in those pages. This marks a shift from a logical to a physical perspective in terms of implementation and forensic analysis. We discuss this in greater detail in Section 4.5.

This particular approach assumes that the records once initially stored on a particular page never move. This is a fair assumption, if one recalls that nothing is physically deleted from the database: instead at deletion a transaction stop timestamp is recorded. Hence, once a page is full it will not be modified again. (We note that in some transaction-time storage structures such as Time Split B-trees [Lomet and Salzberg 1989], tuples do move. For such structures, page hashing is not appropriate.) As a consequence, each record can be associated with a specific page number which does not change throughout its lifetime. This unchanging page number identifies the page in which the record was originally stored; for this reason we term this technique *static-numbered* page hashing. One such example would be an implementation that uses an unordered heap file [Ramakrishnan and Gehrke 2003]. (Note that the page number is *not* stored with the tuple; rather, it can be determined from where the tuple resides.)

To ensure consistency with our previous definitions, we define the spatial detection resolution, R_s, under this scheme, in terms of *pages*.

The first issue that needs to be addressed is when should page hashing occur (recall that page hashing produces the total chain used for tamper detection). There are several possibilities. Hashing of the tuples in a page can be:

(1) *per tuple*, that is, done individually for each modified tuple;
(2) *lazy*, that is, any time after page is written to disk;
(3) *holistic*, that is, only when the page becomes full in main memory;
(4) *incremental*, that is, whenever a tuple is written to a page cached in main memory;
(5) *hash-page-on-write*, that is, whenever the page is written to disk.

To see which option makes the most sense, we consider each one in turn.

Considering per-tuple hashing we simply note that it is inefficient especially in high performance databases and therefore is not appropriate.

Lazy hashing is also not appropriate because any time a page whose contents are not hashed is written to disk, correctness is compromised. The reason is that once the page is written to disk it can be corrupted. When time comes to hash the page the system will hash and then notarize the corrupted value.

In the case of the holistic option, hashing the page when it is full in main memory introduces a delay which compromises correctness and creates an irregular notarization interval. More specifically, a page may not become full for months, so this creates an unbounded amount of time before notarization and validation can occur; in the meantime tuples on the yet-to-be-hashed page on disk might be corrupted.

Incremental hashing is trickier. Let us see what happens if we try to apply it. As the tuples come in during normal processing they are written to pages in main memory. In a similar way to incremental hashing in commit-based partitioning, we will have to maintain a hash value for each page in memory a tuple is written to. The question which arises then is "when does the hash function produce a value?" Under incremental hashing in commit-based partitioning the hash value is produced as soon as the transaction commits.

In an equivalent construction of incremental hashing in page-based partitioning, the hash function will produce a value when the page is written to disk. But this is essentially the same as the Hash-Page-On-Write where the contents of the entire page are hashed once it is to be written to disk. Thus, there is no need to start a hash function early and hash incrementally. The only option that ensures correctness and preserves the consistency of the definitions is Hash-Page-On-Write.

We also need to consider where the hash values should be stored and how the hash values can be reproduced during the validation scan. It turns out these two concerns are intricately linked. In the commit-time-based algorithms, we could easily reproduce the hash value during validation because once a transaction commits it never changes and the timestamp, which is unique for each transaction, provides the desired order of hashing. (Also, a tuple sequence number had to be introduced to designate the order with which the tuples should be hashed within a particular transaction [Snodgrass et al. 2004].) In a page-based scheme, the complication is that the contents of a page (unlike those of a transaction) can change with time until it becomes full. So the page can be brought into memory and written out to disk (and thus hashed) multiple times before it is full. This implies that a single page can have multiple hash values associated with it, each capturing the contents of the page at different points in time. Hence, we cannot use the page number to order the pages in the order with which they were written out to disk/hashed. We need a way to somehow track this order and then be able to reproduce it during validation. Moreover, we also need to know for each of the potentially multiple hash values associated with a page, what were the contents of the page at the particular moment in time when they were hashed.

There are two ways to capture the order with which pages are hashed: save a page write timestamp on the page each time the page is hashed, or use a separate page pinned in main memory. We examine each in turn.

4.1.1. Page Write Timestamp (T-Only). Figure 4 provides an example of how this scheme works. (Please note that certain DBMSes implement the page structure such that the records are appended in the opposite order, that is, rec_N, $\text{rec}_{N-1}, \ldots$, rec_2, rec_1.) The figure should be read left to right and top to bottom as depicting one continuous sequence of events. Records are written to the page during normal processing, shown as $\text{rec}_1, \ldots, \text{rec}_5$. When the page is to be written out to disk the hash function hashes all the contents of the page and creates a hash value which it links with a hash value from the previous page write (of a potentially different page). The resulting hash value is maintained in main memory. Then the time of the page write—the timestamp t_i—is stored on the page just hashed, in a position immediately following the last record, and the page is written out to disk. At a later time when the same page is brought into memory to store more records, that is, $\text{rec}_6, \ldots, \text{rec}_8$, these will be appended after the timestamp t_i. When the page has to be written to disk again, a new hash is calculated corresponding to the records inserted since the last time the page was written to disk. This yields a new hash value which will be linked and stored in main memory and the new page write timestamp t_j will be appended after the last tuple (rec_8). Note that t_j may not be the successor of t_i since other pages may have been written out to disk in the time required for records rec_6, rec_7, rec_8, to be appended to the current page. In

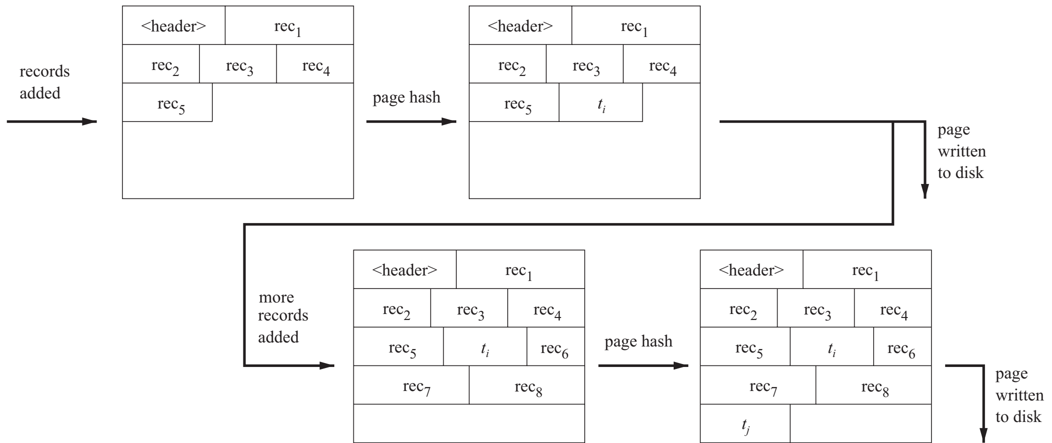


Fig. 4. Page structure exhibiting the page structure under the Hash-Page-On-Write scheme.

this way the page write timestamps delimit each addition of tuples in the database. The placement of the page write timestamps in the page captures the contents of the page at the time of hashing, whereas the values of the timestamps themselves identify the time of the page write event. The strictly monotonically increasing nature of the page write timestamps captures the order with which the pages were hashed/written out. This allows the faithful reconstruction of the hash chain values during validation.

4.1.2. Page Write Timestamp, Page Number, and Page Offset (TNO). The other approach is to use a page pinned in main memory to store the page number of the page just hashed along with an offset into the page where the last record ends, termed *TNO*. The page write timestamp is also stored since it anchors the page write event in clock time. This effectively captures the same information as the previous T-Only scheme (storing only a page write timestamp), however, under this scheme the information is explicit and saved in main memory rather than on disk. (This requires that redo and undo recovery actions must maintain the consistency of this pinned page across system failures.) Note that we could also store similar information on the log, though such an approach complicates forensic analysis because the forensic algorithms would be required to access the log.

4.1.3. Timing of Notarization. The next design choice we consider is when notarization occurs. We aim for consistency with the commit-time-based algorithms, therefore notarization events are scheduled after a set period of time. The notarization interval should then be defined as the time interval between two successive notarization events (*NE*s). It still holds that $I_N = N \cdot R_s$ where $N = 2^n$ for some $n \in \mathbb{N}$. For example, if $R_s = 4$ days and $N = 1$ then a notarization event occurs every four days. When it is time to perform a notarization the hash value produced from hashing the pages and which is stored and maintained in main memory is sent, along with a timestamp of the current time, to the external digital notarization service. There they will be notarized and returned along with a notary ID to be stored in the secure master database as the original protocol demands.

4.1.4. Implementation Considerations. During tamper detection the validator performs a linear scan the database. If a T-Only scheme is used, then the contents of the pages will be read and the timestamps stored within the pages will indicate the sequence

of page writes and hence the order with which the pages will be hashed and linked. Under this scheme the pages must be sorted (in multiple passes) according to the page write timestamp. The relative position of the page write timestamps within the page will indicate the appropriate content to be hashed. If the TNO scheme is used then the timestamps will indicate the order with which the pages must be hashed while the page number and the page offset will indicate which page is to be hashed and what part of the page must be hashed, respectively. The advantage of this latter scheme is that by using the page number we know exactly the order with which the pages must be brought into main memory and therefore no sorting is required.

A new hash value is thus constructed and sent with the corresponding notary ID (from the secure master database) to the notarization service for validation. Recall that a validation interval is the interval between two successive validation events and that the definition $I_V = V \cdot I_N$ still holds.

Storing timestamps on disk can give rise to a new type of corruption whereby the page write timestamps stored on the page are tampered with. We will discuss this type of corruption in Section 11.

There is no immediate performance penalty under these implementation schemes. Under the T-Only scheme, appending a timestamp to the page is does not incur further I/O. However, in the long term, the space overhead of appending a timestamp to the page after each page write will have an impact on I/O since pages become full more quickly. As an example, if tuples are 50 bytes on average and if a page is written after 10 tuples are inserted or modified on average, the additional timestamps (say each 8 bytes) will add about 1.6%, if the pages are full. If tuples are small and pages are large and the main-memory buffer is small, resulting in many page writes, then the overhead may become a problem, favoring the TNO scheme.

Under the TNO scheme the overhead is negligible. A single auxiliary page is pinned in main memory and used to store the page number, page offset, and page write timestamp. When the page becomes full then it treated as any other page: its contents are hashed, the page number, offset, and timestamp written to a clean page pinned in main memory and finally is written out to disk.

We now define a *page-based corruption diagram*—parallel to the previously introduced (transaction-time-based) corruption diagram in Figure 2—to illustrate this new technique. Figure 5 shows the page-based corruption diagram for the Monochromatic Algorithm. The x-axis is labeled in page numbers (as an order can be imposed on the pages) representing the spatial aspect while the y-axis, which captures the temporal aspect, is labeled with the enumeration of the page write events, denoted by WE. Each one of these events corresponds to a specific page being written to disk (marked by the small black circles on the trajectory of the write events). For example, at page write event 7, page 3 was written to disk. (Page write event 7 is on the y-axis; page 3 is on the x-axis.)

The meandering chain is the equivalent of the linear total chain in the commit-time-based corruption diagram. It shows the order with which the pages are hashed and linked cumulatively. The normal sequence of notarizations and validations occurs every two and every four days (because $I_N = N \cdot R_s = 1 \cdot 2$ days $= 2$ days and $I_V = V \cdot I_N = 2 \cdot 2$ days $= 4$ days), respectively.

The corruption diagram in Figure 5 also shows a corruption event affecting page 6 when it was written during page write event 4 (WE_4). Here a corruption is indicated by an open circle at the relevant write event. Note that specifying the page write event is crucial because the same page (with potentially different contents) can be written out to disk during more than one write event. For example, page 3 is written out to disk at WE_7 and at WE_{10}. As before, tamper detection will recompute the hashes at the different validation events. The validations VE_1 to VE_3 succeed and thus return

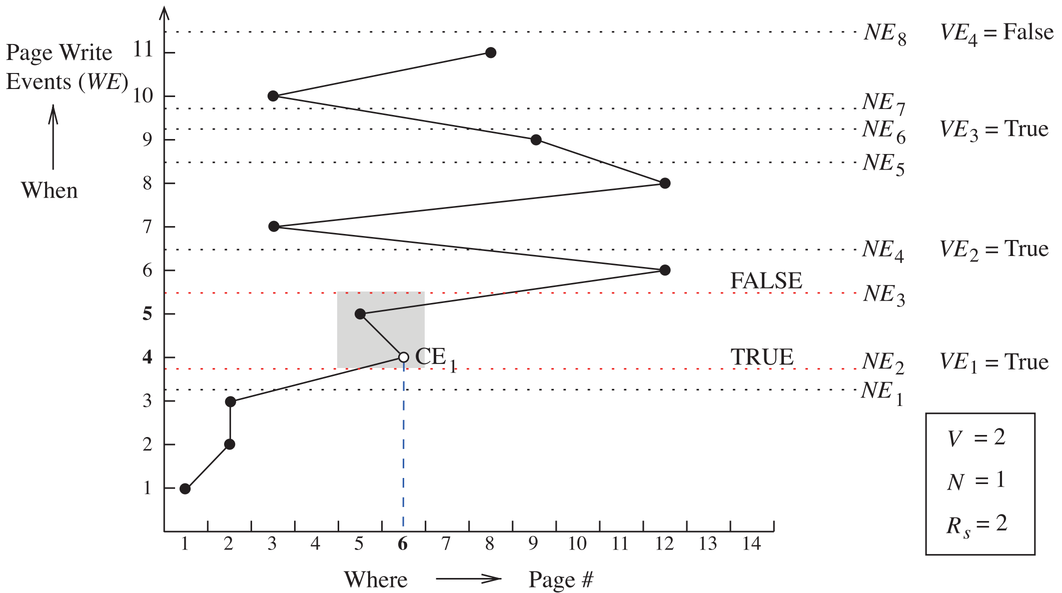


Fig. 5. The Monochromatic Forensic Analysis Algorithm extended to page hashing.

true, whereas at VE_4 a mismatch of hash values is observed and the result returned is false.

4.2. The Page-Based Monochromatic Algorithm

We can generalize the Monochromatic Forensic Analysis Algorithm to apply to this meandering total chain. The t_c of the corruption lies in the time interval between VE_3 and VE_4 since the former yielded a True result while the latter yielded a False result during validation. More information on how these bounds are found in Section 4.4.

The spatial analysis is more involved. As in previously introduced algorithms we can once again perform a binary search on the validation results of the previously notarized hash values in order to locate the interface of the transition from true to false (TT...TF...FF) validation results. This interface occurs between NE_2 and NE_3 (marked with TRUE and FALSE respectively) and allows us to bound the corruptions (shaded regions in Figure 5) in two different ways: in terms of page write event(s) and in terms of corruption locus, l_c (i.e., page(s)). At this point we know that the CE occurred some time between VE_3 and VE_4 (it was detected during VE_4) and corrupted pages that were written to disk between NE_2 and NE_3.

Note that the y-axis does not technically express a time dimension and for this reason the absolute spacing in the dotted horizontal lines which correspond to the notarization and validation events do not convey any temporal information. However, their *relative* spacing shows which pages were written during each notarization interval.

Forensic analysis returns two suspect pages: 5 and 6 (occurring at WE_4 and WE_5), which were written doing a 2 day period in accordance with the spatial resolution R_s (recall it is two days). The fact that the number of pages matches R_s is incidental. If three or one page was written during the interval of those two days then all those pages would be treated as suspect. Unlike the original corruption diagram for the Monochromatic Algorithm (e.g., Figure 2), the corruption region may not be (although here it is) a single visually contiguous rectangle, though in this example it is still of the same area because the suspect pages were written during the same notarization interval.

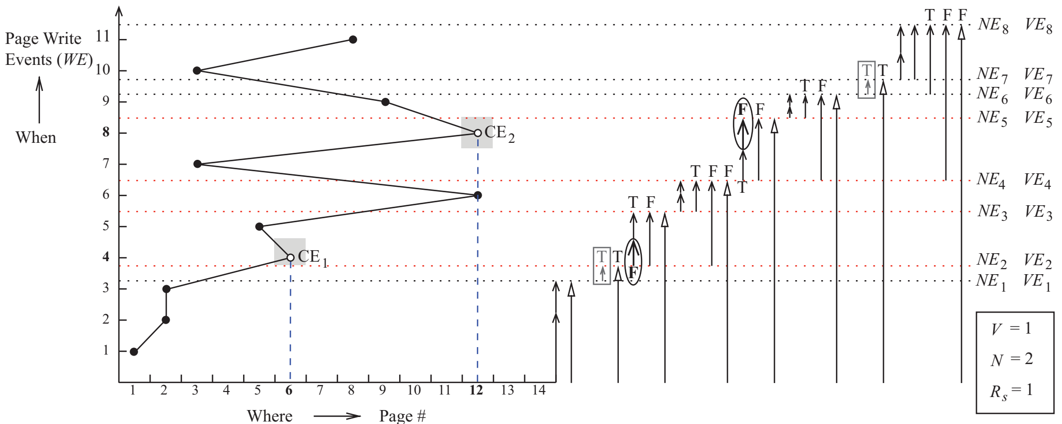


Fig. 6. The a3D Forensic Analysis Algorithm extended to page hashing.

4.3. The Page-Based a3D Algorithm

The next natural step is to introduce partial hash chains into the Page-based Monochromatic Algorithm which will help us create the Page-based a3D Algorithm. The partial hash chains are computed during the tamper detection phase when the validator performs a linear scan over the entire database. While the validator is hashing all the data in the database it can at the same time, and thus with no I/O overhead, compute hash chains which cover only parts of the database as per the specification of the a3D Algorithm parameters. Depending on the scheme chosen, T-Only or TNO, the validator will have sufficient information (page write timestamps, etc.) so as to hash the pages in the correct order. The resulting hash values of the partial chains are sent along with the new hash value, notary ID pair of the total chain to the external digital notarization service. The notarization service performs a validation on the total chain and if it succeeds it notarizes the partial chain hash values and returns a set of notary IDs back to be stored in the secure master database. As we can see the protocol for notarizing or validating any particular hash value remains the same. The important difference here is that during the tamper detection phase the system performs a validation and multiple notarizations.

During the forensic analysis phase instead of relying on a single total chain, we have additional previously notarized partial chains which we can now validate in order to achieve tighter bounds on the corruption event and even detect multiple corruptions (since these partial chains are not cumulative).

Figure 6 shows the page-based corruption diagram for the generalized a3D Algorithm with two corruptions. The first corruption affects page 6 and transpired during WE_4 while the other affects page 12 during the eighth write event. (In Figure 6 we have assumed the same sequence of writes as in Figure 5, but have introduced an additional corruption.) The partial hash chains seen on the right side of Figure 6 create a binary tree on top of the entire database in a way similar to the one shown in Figure 3. However, each of the partial chains hashes records according to the page the records belong to. Another difference from the Commit-time-based a3D Algorithm is that here we could have notarization intervals which are devoid of page writes, hence no pages were hashed. There are two such examples in Figure 6. Observe that no page write events occur in the two days between NE_1 and NE_2 and the two days between NE_6 and NE_7. For this reason there is no need to maintain all the partial hash chains that cover those time intervals. We just maintain a dummy hash chain for each such empty

interval. The dummy chains shown with an enclosing small rectangle in the figure. The validation value of these dummy chains is automatically and always set to true. This culls that part of the binary tree and hence speeds up forensic analysis since that subtree does not have to be investigated. This optimization does not endanger the correctness of the forensic analysis since during these time periods no pages were written to disk and hence their contents cannot be corrupted.

During forensic analysis a simple binary search on this tree identifies the location of the corruption(s). In Figure 6, the two partial chains that are circled (one computed during VE_2 and one computed during VE_5) are the ones that identify corruption events CE_1 and CE_2.

It is important to note that in Figure 6, there are two rectangular corruption regions which are not contiguous. In this case each corruption region signifies a single corruption, resulting in the identification of a total of two corruption events. This should not be confused with the two (contiguous) rectangular parts which together should be regarded as single corruption region shown in Figure 5. The difference stems from the fact that in Figure 5, two pages were written during the interval between NE_2 and NE_3. This implies that the spatial bounds identify two pages written consecutively constituting a single corruption region.

The above discussion thus shows how any "conventional" corruption diagram (that is, based on a where-axis of transaction time) can be extended to apply to static-numbered page hashing, thereby determining a "where" of a single page or a short write sequence of pages containing the page that was tampered.

4.4. Determining "When" from Page Number

In static-numbered page hashing as shown in Figures 5 and 6, the bounds on the "when" are not, strictly speaking, temporal. For example, in Figure 5, the bounds on the y-axis tell us only that a corruption occurred during a period when the fourth and fifth page write events took place. It does not inform us about when these two page write events occurred in real time. (This is in direct contrast to commit-time based corruption diagrams, for instance, Figures 2 and 3, where the "when" axis denotes real, that is, clock time, and so any conventional forensic analysis can bound the actual time a corruption occurred.) For this reason, we must find a way to determine the real time bounds on t_c from the suspect page numbers.

In order to obtain true temporal bounds on t_c, we must correlate the sequence of page write events with real time. We first find the times these events occurred. The way to achieve this is to employ the page write timestamps that were harvested and stored in the page when the page was hashed. Then we can plot real time against these times of page write events in a similar way as in the commit-time based corruption diagrams. This will allow us to put temporal bounds on t_c by using the clock time axis.

To illustrate this solution we introduce in Figure 7 an expanded 3D version of a page-based corruption diagram termed the *correlated page-based corruption diagram*. (We later describe how the implementation computes the corruption regions.) The (Page Number)×(Page Write Event)-plane is the page-based corruption diagram already given in Figure 6. The z-axis measures clock time while the (newly introduced) axis labelled "Time of Write Event" establishes the correlation between a write event and clock time. Observe that the partial chains which were situated on the right side of Figure 6 are now drawn in the (Clock Time)×(Time of Write Event)-plane. Also, note that now the partial hash chains of a particular *level*[1] are drawn as having equal lengths unlike the same chains in Figure 6. For example, any two chains of level zero

[1]*Level* was first defined for the a3D Algorithm as "the (zero-based) vertical position of [a hash] chain within a group of chains added at [a particular] *VE*" [Pavlou and Snodgrass 2008].

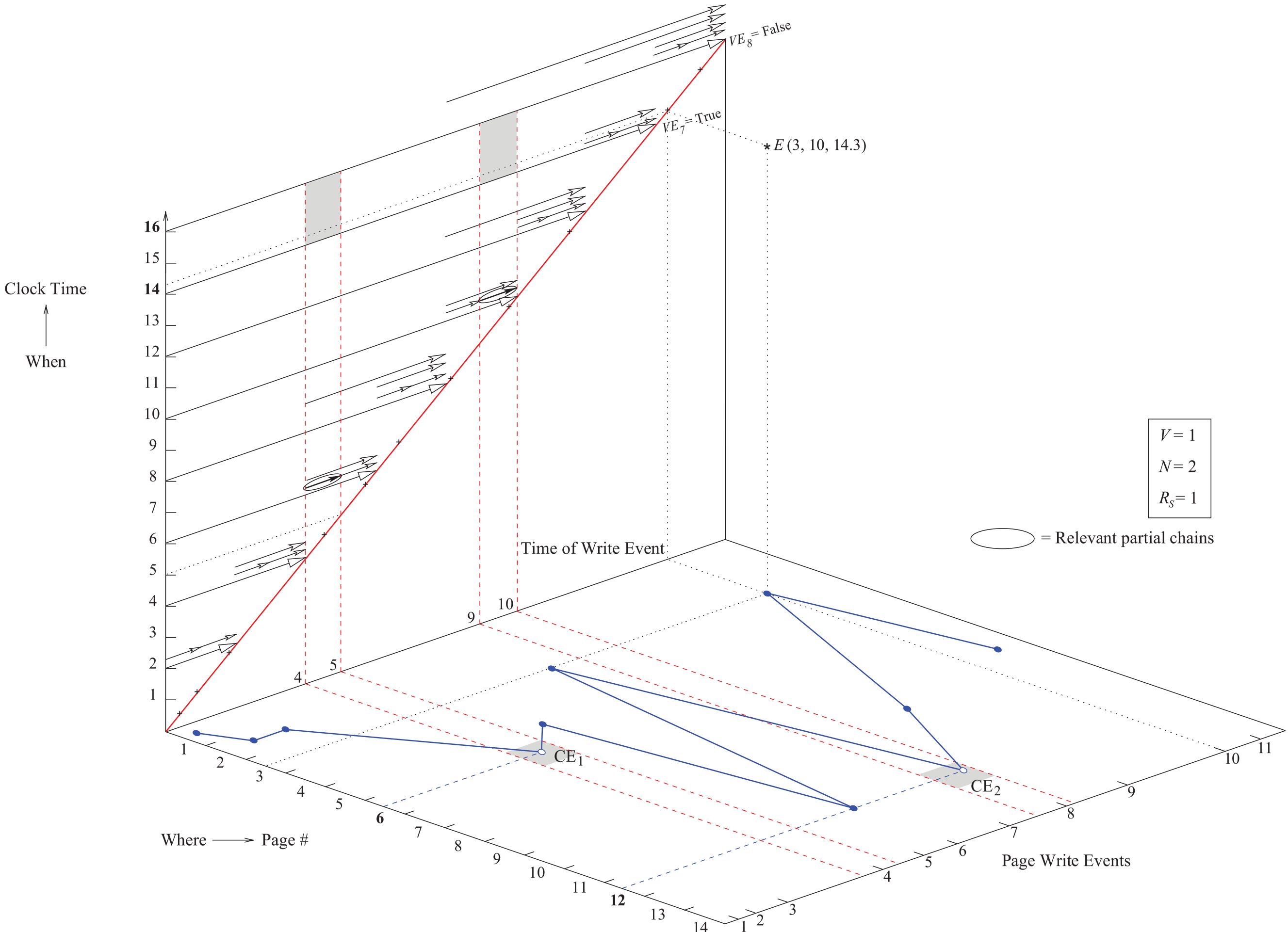


Fig. 7. The correlated page-based corruption diagram. The forensic analysis algorithm used is a3D. The temporal and spatial bounds of CE$_1$ and CE$_2$ are marked in bold font.

each covers one day ($R_s = 1$), so their lengths are necessarily equal. This is because the lengths of the partial chains reflect the time intervals between notarization events which which occur every I_N days. This was not the case in Figure 6, where the length of the partial chains was adjusted to cover the appropriate pages that were written during the time period the chains covered. This is because the page write events were equally spaced on the y-axis. In Figure 7 this irregularity in chains is no longer seen. Instead the intervals between page write events on the (Page Write Event)-axis are clearly shown as not being equally spaced, the reason being that the page write events here are correlated with time.

To see what information the coordinates of the diagram conveys, let us use point E as an example. (This point is not used in forensic analysis.) The coordinates of E are $(3, 10, 14.3)$. This should be interpreted as E marking page 3 which was written to disk at write event 10 which occurred at time 14.3.

During forensic analysis the binary search identifies two corrupted partial hash chains (shown circled in Figure 7), which reveal pages 6 and 12 as suspect. The corruption regions are shaded in the (Page Number)×(Page Write Event)-plane and the suspect pages 6 and 12 are marked with CE_1 and CE_2 respectively. Even though we can see from the (Page Write Event)-axis that page 6 was written at write event 4 and page 12 at write event 8, we have no direct indication on *when* the corruption occurred. (Technically there are two t_c's because we are assuming two corruption events CE_1 and CE_2 in this example. We will describe only CE_1 for convenience.) In order to obtain the temporal bounds we must first find when the write events occurred. The (Time of Write Event)-axis provides this information. Consider CE_1. The bounds on the (Page Write Event)-axis shown with red-dotted lines contain only WE_4. However, the same bounds when projected onto the (Time of Write Event)-axis produce bounds on the time the write event occurred, namely, WE_4 occurred during day $(4, 5]$. Notice that if these bounds are projected upwards onto the 45-degree action axis in the (Clock Time)×(Time of Write Event)-plane, they align with the corrupted partial hash chain identified during forensic analysis, as expected. Note also that care should be taken to avoid the pitfall of identifying the upper bound of the the interval $(4, 5]$ which aligns with the tip of the partial hash chain as 6 instead of the correct 5. Even though the partial hash chain was calculated during the validation scan which happened at time 6 it hashed pages which were written to disk only in the $(4, 5]$ time interval. Now we can find the true temporal bounds of CE_1. Observe that since VE_7=True and VE_8=False, the corruption transpired in the time interval between the two validations. According to the (Clock Time)-axis, the time of VE_7 is 14 and that of VE_8 is 16 (shown in bold font in Figure 7), therefore, $t_c \in (14,\ 16]$. (The corresponding corruption regions in the (Clock Time)×(Time of Write Event)-plane are also shaded.)

Finally, we conclude that CE_1 happened sometime in the interval $(14, 16]$ affecting page 6, which was written to the database some time in $(4, 5]$. A similar analysis for CE_2 produces the same temporal bounds $(14, 16]$ and page 12 as the corrupted page, which was written to the database sometime in $(9, 10]$.

So the question remains, how is Figure 7 constructed during forensic analysis? The first step after detecting tampering is to find the temporal bounds of the corruption. This is easy since the temporal bounds are always the times of the second to last and last validation events, that is, the most recent validation which returned true and the one first to return false. In our example these are VE_7 and VE_8, respectively. The next step is to run the a3D Algorithm. The algorithm will perform a recursive binary search on the binary tree of chains and identify the chains which return false when validated. By construction we already know what time intervals each of these chains covers. For example, the partial chain which returned false corresponding to CE_1 was added at the start of day 6 and covered the interval $(4, 5]$.

In order to get the spatial bounds of the corruption we have to able to answer the next crucial question: Which pages were written out to disk/hashed during the interval covered by the false partial chain? This answer is provided by the page write timestamps which were stored at the time each page was hashed and stored either on the page itself or in a separate page in main memory. The Time of Write Event-axis is comprised of these timestamps, which allows us to map the locus times t_l, that is, page write timestamps, into the corruption loci l_c, that is, page numbers. The forensic analysis ends by scanning the pages and looking for timestamps which fall in the interval covered by the identified partial chains. If the timestamp is in that interval the page storing the timestamp is considered suspect; otherwise it is ascertained to not have been altered.

4.5. Comparison with Commit-Time Approaches

There are some compelling reasons for why one should choose page-based algorithms over the commit-time-based ones.

One of the advantages of page-based algorithms is that a tuple sequence number is not explicitly maintained within the tuple so the inner workings of notarization and validation are completely imperceptible to the user. Moreover, if the TNO implementation is used then we can avoid the overhead of sorting the data during the validation scan since the order with which the pages have to be read back into main memory is explicitly stored in the pages that were pinned in main memory.

Furthermore, the implementation of hashing and of storing timestamps in pages can be moved below the level of the DBMS file manager, even into the OS/file system layer. Such an implementation decouples the function of the audit system from the DBMS, further reinforcing the shift from a logical to a physical perspective first mentioned in Section 4.1. The advantage of this is clear when the system is scaled to support multiple DBMSes. The representation of corruptions becomes uniform across all DBMSes: a list of page numbers is returned by forensic analysis for all corrupted databases. Recall that in general, a page-based scheme can be achieved without increasing forensic cost and with minor space overhead. This opens up many avenues to explore in our future work in order to understand the full architectural implications of such a modification.

Paged-based algorithms can be modified to provide added features to forensic analysis. Defining R_s, I_N, and I_V in terms of number of page write events rather than number of days would allow us to bound the manual work required once forensic analysis is finished. The number of pages we would have to examine would always be a fixed integral multiple of R_s. That is not the case with commit-time-based algorithms where within two separate R_s-day intervals there can exist a different number of transactions. By making this modification in the way R_s is measured, manual work is rendered *load insensitive*. This has also the desirable side-effect of making the notarization and validation intervals irregular with respect to time because page write events do not occur at regular time intervals. Hence, the adversary cannot predict when notarization and validation events occur making them difficult to intercept. In order to avoid cases where pages are not written to disk often enough, hence resulting in overly long notarization/validation intervals, we can put an artificial limit on the time a page can reside in main memory. When that time limit is reached then the pages are flushed to disk.

5. PARTITIONING ON ATTRIBUTES

We now turn to extending the partitioning of the data according to an identified explicit attribute. Note that we are now talking about a single table whose schema includes that attribute. This is in direct contrast to the commit time- and page-based schemes which are *schema agnostic*. Attributes, on the other hand, are restricted to a single table and therefore only pertain to a section of the data. The advantage

of the attribute-based scheme over the others is that it allows for finer control over what part of the data is under audit, by utilizing the database schema. For example, we may want to partition on zip code, which provides a semantic partitioning of the data. Another example is partitioning on the *manager* attribute. Again, the CSO would determine from a variety of considerations whether to utilize partitioning on attributes and if so, on which attributes.

5.1. Naïve Approach

Let us assume that the domain of the identified attribute is discrete and finite, for instance, zip code. If we try to apply analogous definitions (from commit time- and page-based partitioning) to an attribute-based scheme then we run into problems. R_s needs to be defined in terms of "groups of tuples" or *granules*, in the same way transactions and pages were used in commit time and page-based partitioning, respectively. Moreover, these granules need some sort of timestamp associated with them so that they can be chronologically ordered (akin to commit time or page write time).

However, there is no "natural" way to do this with respect to a chosen attribute. And by "natural" we mean "native to the system." However, let us try and do it artificially by introducing some user-defined way of grouping the tuples.

Suppose we group the attributes within the tuples by the order they are modified, into groups of size R_s. In other words, every R_s tuples constitute a granule.

We could also use the time the attribute is written to the database cache (this is what we mean when we mentioned above "tuples that are modified") as a way to chronologically order the groups (or even order the tuples within the group).

However, the approach just described *does not* achieve the desired partition because closer inspection reveals that the scheme, in reality, does not make use of attributes at all. It doesn't matter whether we are using zip code or another attribute like age; we could just as easily describe this by dealing only with the tuples themselves.

This is revealed when we try to do forensic analysis: the spatial bounds obtained, inform us that a granule was corrupted, having R_s tuples. This last sentence says nothing about the specific attribute which we are supposedly partitioning on! In other words, there is nothing in this partitioning scheme that takes advantage of the chosen attribute in order to produce forensic information by characterizing the corrupted tuples in terms of the value the attribute takes in these tuples. Hence, the scheme we have been using so far *cannot* be applied directly in order to construct a total chain based on attribute partitioning.

5.2. A Natural Extension to Attribute-Based Partitioning

In order to leverage the existence of attributes and of their particular domain, which can be appropriately partitioned, we clearly need a different approach. The naïve approach we tried in the previous section does not work so we have devised a new one which seems to be the most natural extension from commit-time- and page-based partitioning to an attribute-based one.

The first change required is for the total chain, under the new attribute-based scheme, to be replaced by multiple nonoverlapping chains maintained in parallel. For a chosen attribute in a specific table we partition the values in the domain associated with that attribute. Let the chosen attribute be zip code and let there be ten subsets yielded by the associated partitioning. We thus maintain ten hash chains in parallel, one for each subset. The reason for not having any overlap between the subset(s) that each chain corresponds to, is that experience with creating the Tiled Bitmap and RGBY Algorithms has shown that if we are not careful, overlap between hash chains can introduce complications (e.g., false positives) during forensic analysis [Pavlou and Snodgrass 2008]. As the algorithms introduced in the present article do not introduce

false positives, we will just mention such complications only when comparing our new algorithms with Tiled Bitmap and RGBY.

The next step is to assign each tuple in the table to the corresponding hash chain according to the value it contains. This means that if a specific tuple has the value 85705 as zip code, that tuple is hashed into the hash chain corresponding to the subset containing the value 85705. Recall that each *version* of the tuple is written only once. A modification of the tuple, including that of its zip code, results in a new tuple being inserted. So each tuple is hashed once and included in only one hash chain. This implements an *initial* partition of tuples into different hash chains according to their attribute value. The number of subsets in the partition of the domain and hence the number of hash chains is decided by the DBA and thus is configurable. We term this number the *domain resolution*, R_d.

We assume each of the ten hash chains is constructed and maintained according to the original commit-time-based partitioning. This is because page-based partitioning has a high implementation overhead. Recall that whenever a page is written to disk, the page write timestamp of the chain that page belongs to has to be written in the page itself. In this attribute-based scheme, a single page can have tuples with zip code values belonging to all of the ten subsets/hash chains. This implies that we could potentially have to store and maintain hash values for each one of those ten hash chains within a single page. The space overhead and the logistics of tracking which hash values in the page correspond to which hash chain make this approach less than appealing.

The tuples within one hash chain can be ordered chronologically according to their commit time. Under this scheme notarizations and validations will happen simultaneously for all chains, if all chains have the same notarization and validation intervals, or independently for each chain, if the intervals differ across chains. In either case, this creates an overhead for the scheme where the number of contacts with the external digital notarization service is bounded above by a multiple of R_d since notarizations and validations have to be performed for each one of the hash chains.

The initial tamper detection stage will also work, with some modifications. All hash chains will need to be validated periodically and validation failure in any one of the chains will identify which hash chain/zip code set the corrupted tuple belongs to. After this initial step, forensic analysis can proceed as in the commit time-based scheme.

Due to this independence between an attribute-based and a commit time-based scheme, one can potentially use a different forensic algorithm for each of the R_d independent total hash chains. The overall database audit scheme will have a single domain resolution R_d, which then determines the number of chains, with *each* such chain potentially having a particular spatial detection resolution R_s, notarization interval I_N, validation interval I_V, and forensic analysis algorithm. For example, assume we partition the domain of the attribute zip code into ten subsets (i.e., $R_d = 10$). The first nine resulting total chains could use a Tiled Bitmap Forensic Analysis Algorithm with $R_s = 2$ days and $I_N = I_V = 16$ days. The tenth total chain is unusual: the distribution of zip code values in the tuples is skewed, in that it has higher density in the subset corresponding to this tenth chain. Due to this skew we wish that the forensic analysis on the tenth chain provide tighter spatial bounds on the corruption region computed by forensic analysis and with no false positives. Thus we specify for this particular chain the a3D Algorithm with $R_s = 1$ day, and $I_N = I_V = 8$ days. The increased accuracy of the tenth chain comes at the expense of a higher forensic cost, assuming the database is online for a sufficient length of time, for instance, one year [Pavlou and Snodgrass 2008]. This independence provides an additional means for an attribute-based scheme to allow for finer control over the auditing of data in the database.

There are some remaining issues that need to be addressed, especially with respect to the nature of the domain of the attribute chosen for the partition. The above description

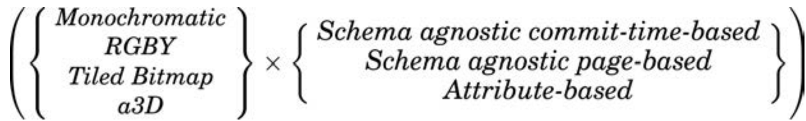


Fig. 8. The types of forensic analysis algorithms.

pertains to domains that are discrete and finite. (Note these domains include ones with *categorical* values, e.g., "nationality.") Hence, the next natural extension is to ask how does implementation change if the domain values are continuous?

If the domain is continuous and finite, the solution is to first partition the domain into intervals. The number of intervals defines the number of total chains maintained in parallel, although the correspondence might not be one-to-one. A chain could be associated with *several* such intervals. Note that there is nothing that prevents us from applying this scheme of forming intervals to discrete finite domains, thus grouping different values together to form hash chains containing tuples with multiple domain values. The domain values each hash chain contains could be informed by the application semantics and/or the cost for maintaining the chains. For example, instead of maintaining a chain for each different nationality we could instead have five chains grouping nationalities by continent.

The way the data are partitioned under the attribute-based partitioning scheme merits special attention. The data are first partitioned according to the attribute domain and for each such resulting subset of data, the data are further partitioned according to transaction commit-time. Therefore, the data are subjected to two nested partitions: attribute and then commit-time, which we term *composite* partitioning.

6. SUMMARY

In the previous section we formulated a way to partition the data based on any attribute. The initial naïve approach did not work, suggesting that a new scheme had to be devised. This led to a natural extension of commit-time-based and page-based partitioning to attribute-based partitioning.

The set-theoretic formulation given in Figure 8 captures all the types covered so far. The leftmost column features the four familiar forensic analysis algorithms. Each algorithm can be applied to a database partitioned according to the three schemes of the middle column.

These schemes of partitioning the records in a database can easily be further extended to apply to more general attributes. We can distinguish between three kinds of fields on which a database may be partitioned.

—*Implicit* attributes. These pertain to data maintained by the DBMS, specifically, transaction ID, commit time, insertion time, and deletion time [Lorentzos 2009].
—*Virtual* attributes. These pertain to data or aspects maintained by the file system, for instance, the physical order in which data may be organized/stored (by increasing commit time) and the page number of the page the tuple was originally stored.
—*Explicit* attributes. These pertain to actual attributes in the schema of the database.

Table IV summarizes the different ways the data in a database can be partitioned: according to commit time, page writes, or attribute (the columns of the table). Note that the column denoted "Attribute-Based" represents all the forensic analysis algorithms which can be used in conjunction with an attribute-based partitioning scheme. For each such partitioning the table provides a number of crucial properties and features and how they are defined (the rows of this table). Along any particular row, the important aspect of each cell is italicized.

Table IV. Ways of Partitioning the Data

	Partitioning		
	Commit-Time-Based	Page-Based	Attribute-Based
Tables Affected	Any number	Any number	One or several of those containing the designated attribute
R_s	Time interval	Time interval	Time interval
R_d	N/A	N/A	Number of subsets of domain values
Segment	One of the contiguous *periods* induced by R_s, starting from a particular anchor. Contiguous periods form a chronologically ordered partition.	One of the contiguous *periods* induced by R_s, starting from a particular anchor. Contiguous periods form a chronologically ordered partition.	One of the contiguous *periods* induced by R_s, starting from a particular anchor. Contiguous periods form a chronologically ordered partition.
Granule	Encompasses all tuples with *commit times* within the associated segment (one granule has tuples from many transactions committing in that segment).	Encompasses all tuples whose *physical location* is in a page mentioned within the associated segment.	Encompasses all tuples with *commit times* within the associated segment (one granule has tuples from many transactions committing in that segment).
Hashing order	Transactions hashed in order of increasing *commit time*.	Granules hashed in chronological *order of* "page write" *event* of the page. Granules not hashed in order of page number.	Transactions hashed in order of increasing *commit time*.
Segment Completion Event	When the last *tuple* in the granule associated with that segment commits	When the last *page* write event in the segment occurs.	When the last *tuple* in the granule associated with that segment commits
Notarization Factor (N)	Specified by DBA	Specified by DBA	Specified by DBA
Validation Factor (V)	Specified by DBA	Specified by DBA	Specified by DBA
I_N	$N \times R_s$	$N \times R_s$	$N \times R_s$
I_V	$V \times I_N$	$V \times I_N$	$V \times I_N$
Notarization	Occurs *as soon as N* granules are hashed.	Occurs *as soon as N* granules are hashed.	Occurs *as soon as N* granules are hashed.
Validation	Occurs as soon as V notarizations have occurred.	Occurs as soon as V notarizations have occurred.	Occurs as soon as V notarizations have occurred.

Table V. Design Choices

With respect to	*Options*	
	Qualifier	*Explanation*
Algorithms	Monochromatic	Tuples are hashed cumulatively into a single hash chain.
	RGBY	In addition to the total chain, tuples are hashed into repeating & partially overlapping shorter (partial) chains.
	Tiled Bitmap	In addition to the total chain, tuples are hashed into repeating groups of hash chains, covering and creating a bitmap over the data.
	a3D	Tuples are hashed in such a way so as to create a binary tree of hash chains over the data.
Hashing	*Per Tuple*	Each modified tuple is hashed individually.
	Incremental	Tuples are hashed as soon as they are written to the buffer. No associated overhead of restarting the hash function.
	Holistic	Tuples are hashed as soon as transaction commits.
	Lazy	Tuples are hashed any time after transaction commits.
	Hash-Page-On-Read	A hash of the tuples is logged on each data page read.
	Hash-Page-On-Write	The page is hashed whenever it is written to disk.
Clustering	*Clustered*	The records are physically ordered on disk.
	Unclustered	The records are not physically ordered on disk.
Partitioning	*Single*	A single field is used to partition the data.
	Orthogonal	Two or more fields create independent partitions of the data.
	Composite	Two or more fields create a nested partition of the data.

Of particular interest are the notions of *segment*, induced by R_s as a partition of the spatial dimension, and of *granule* as the group of tuples included in a segment. The table also provides useful information on how the hashing is implemented and when notarization and validation events occur. It is interesting to see both the portions of each aspect (row) that are common across partitioning approaches and those portions (rendered in italics) that are unique to a partitioning approach.

Note that issuing an `ALTER TABLE` command is allowed since it does not change the integrity of the chains so far. If the command is used just to rename the table then the notarizer and validator do not need to be altered. However, a command used to change columns in the table is handled in the usual way: a new temporary copy of the original table is made. The alteration is performed on the copy, and then the original table is deleted and the new one is renamed. The notarizer and and validator must be made aware which columns to hash in the rows prior to the alteration. There is only one case where we disallow the alter command and that is when the command affects a partitioning attribute.

7. DESIGN CHOICES

Table V summarizes the different design choices the (CSO) must make when deciding on the features of the forensic analysis to be applied to the particular database (or even to a particular table). The choices required are with respect to Algorithms, Hashing, Clustering, and Partitioning. Please note that Table V definitions of the different hashing choices refer to commit-time-based partitioning and do not contradict the definitions given in Section 4.1 which refer to page-based partitioning. In a sense,

"page" is the equivalent of "transaction" as a convenient way of grouping tuples together under the two partitioning schemes.

All choices are orthogonal (i.e., independent of each other) with the sole exception of Hash-Page-On-Write which is associated with single partitioning and more specifically with page-based partitioning. The criteria used when making these choices are cost, spatial/temporal detection resolution, and feasibility. (Striking a balance between the first two can be challenging. Of course this also depends on how sensitive is the data stored in the database, and hence how much one is willing to spend to protect it.) For example, an algorithm like a3D provides a greater forensic power but at a higher cost. In the case of hashing, per tuple schemes are usually very expensive. On the other hand, trying to use lazy hashing with a page-based partitioning scheme compromises correctness (vid. Section 4.1). Clustering can provide greater forensic power in allowing us to distinguish between postdating and backdating corruption events. Orthogonal and composite clustering schemes provide greater forensic strength but again at a higher cost.

8. COMPARISON OF FORENSIC ANALYSIS ALGORITHMS

Table VI provides a convenient comparison of the structure of hash chains used by the forensic analysis algorithms. Italicized words highlight the differences between corresponding algorithms under different partitioning schemes. For example, the length and width of chains for commit-time based algorithms grow with time whereas those of page-based algorithms grow with the number of page writes.

The first column titled "Algorithm" shows all the possible algorithms grouped by partition scheme. The next column summarizes the number of levels in each algorithm which is an initial indication of the number and complexity of the corresponding algorithm. We need to generalize the notion of "level" to apply to our new algorithms [Pavlou and Snodgrass 2008]. To the prior definition of *level* we must add that we choose the numbering such that 2^{level} equals the number of hash chains present at that level. Then this definition can be applied to the RGBY, Tiled Bitmap, and a3D Algorithms. However, the Monochromatic Algorithm presents a slight problem: this algorithm technically has no levels because levels are only defined for the partial chains added during a validation event. So for Monochromatic we interpret "levels" loosely: even though it has no partial chains it does have a single cumulative chain. We state this in Table VI as having "(1)" level.

The "Topology of Chains" column captures the position of the chains relative to each other whereas the next column over indicates whether the partial chains are linked, which has significant implications for the forensic strength of the algorithms. In the case of the Monochromatic Algorithm, even though the total chain is cumulative and linear and can be considered as linked this chain is not partial by definition. For the RGBY Algorithm the total chain is linear with the partial chains overlapping nonlinked. The Tiled-Bitmap Algorithm makes use of a linear total chain and repeating pattern of partial hash chains called a tile. These partial chains within a tile are linked in such a way so as to form a bitmap over the database data. All a3D Algorithms have a total linear chain and partial which together form a binary tree over the database. The partial chains are not linked.

The next two columns are named "Length of Chain(s)" and "Width of Chain(s)" respectively. The *length* of a hash chain is the portion of the data stored in the database that the chain's hash value captures. The *width* of the hash chain is the partition: that subset of the domain of the partition attribute for which the chain hashes tuples with an attribute value in that subset. The length of a Commit-time-based Monochromatic Algorithm increases with time as more and more records are hashed; the chain always

Table VI. Comparison of the Chain Structure of Forensic Analysis Algorithms

Algorithm	Number of Levels	Topology of Chain(s)	Are Partial Chains Linked?	Length of Chain(s)	Width of Chain(s)	Number of Chains in a Level
Commit-Time-Based						
Monochromatic	(1)	linear	N/A	grows with *time*	grows with *time*	1
RGBY	1	linear/overlapping	No	During creation it grows with *time*; otherwise it is constant.	During creation it grows with *time*; otherwise it is constant.	2
Tiled-bitmap	fixed within tile	linear/bitmap within tile	Yes			2^L
a3D	variable	binary tree	No			2^L
Page-Based						
Monochromatic	(1)	linear	N/A	grows with *number of page writes*	grows with *number of page writes*	1
RGBY	1	linear/overlapping	No	During creation it grows with *number of page writes*; otherwise it is constant.	During creation it grows with *number of page writes*; otherwise it is constant.	2
Tiled-bitmap	fixed within tile	linear/bitmap within tile	Yes			2^L
a3D	variable	binary tree	No			2^L
Attribute-Based						
Monochromatic	(1)	linear	N/A	grows with *time*	grows with *time*	R_d
RGBY	1	linear/overlapping	No	During creation it grows with *time*; otherwise it is constant.	During creation it grows with *time*; otherwise it is constant.	$2 \cdot R_d$
Tiled-bitmap	fixed within tile	linear/bitmap within tile	Yes			$2^L \cdot R_d$
a3D	variable	binary tree	No			$2^L \cdot R_d$

cover the entire database. Its width also increases with time because the partition attribute is commit time and the total chain includes records with commit time values over the entire time domain. The RGBY, Tiled Bitmap, and a3D Commit-time-based Algorithms only consider partial chains. The length and width of these partial chains only increase during the time of their creation. Once they are fully formed their length and width do not change again.

The page-based algorithms follow an analogous pattern. Here the length and width of the total chain in the Monochromatic Algorithm changes with the number of page writes occurring and not with time. Even though page writes are correlated with time, time can increase whereas the chains will not increase accordingly unless a page write occurs. For the rest of these algorithms, the partial chain length and width increase with the number of page write events during the chain's creation but remain constant once they are created. If no page writes occur during the chain's creation then then these partial chains technically do not exist.

The attribute-based case is exactly the same the commit-time-based case. The length of the partial chains increases with time since tuples are hashed into the chain as time passes. The width however remains fixed with time: even though more tuples are hashed and included in a particular partial chain the value of the partition attribute is an element of the subset of the partition attribute domain corresponding to that chain.

The number of chains in a level as given in the last column of Table VI where the level of a chain is denoted by L. The number of chains in a level provides a crucial parameter to the cost formulas of the forensic algorithms: the greater the number of chains the higher the cost [Pavlou and Snodgrass 2008]. In the case of natural extensions of attribute-based algorithms, we assume the same algorithm is used in each of the R_d chains. Note that in only the attribute-based schemes, all the hash chains of the R_d independent partition subsets are treated together.

Because the attribute-based schemes add an orthogonal partitioning on the attribute, the total number of chains is increased by a factor of R_d, thereby obtaining counts of chains such as $2^L \cdot R_d$.

Overall, the three groups of forensic algorithms (commit-time, page, and attribute-based) follow a consistent parallel construction. Attribute-based and commit-time-based algorithms differ in the number of chains per level they maintain. The additional multiple total chains have to be maintained in parallel in the case of attribute-based algorithms in order to produce tighter bound on the corruption event. The same is true if we compare page-based and attribute-based algorithms.

Page-based algorithms also differ from both commit-time-based and attribute based in the way the length and width of the chains increases. In page-based algorithms, the chains' length and/or width changes with page writes and not time as is the case for commit-time and attribute-based algorithms.

9. FORENSIC COST

In this section we generalize the forensic cost model developed elsewhere [Pavlou and Snodgrass 2008] so that it is applicable to the new forensic algorithms developed in this article.

9.1. Forensic Cost of Commit-Time-Based Algorithms

Forensic cost is a sum of four components, each representing a cost that we would like a forensic analysis algorithm to minimize, and each weighted by a separate constant factor: α, β, γ, and δ. The first two components capture number of calls to the external digital notatization server (each such call will have monetary cost); the other two components capture the discernability of the algorithm: how exactly can it characterize

a tampering.

$$FC(D, N, V, \kappa) = \alpha \cdot NormalProcessing(D, N, V) + \beta \cdot ForensicAnalysis(D, N, V, \kappa) + \gamma \cdot Area_P(D, N, V, \kappa) + \delta \cdot Area_U(D, N, V, \kappa)$$

Each of the normal processing, and forensic analysis costs is measured as the number of times the digital notarization service is contacted during each execution phase.

The forensic cost is defined as a function of D (the number of days since the database went online, normalized in terms of number of R_s units), N, the notarization factor (with $I_N = N \cdot R_s$), V, the validation factor (with $V = I_V/I_N$), and κ, the number of *corruption sites*. A corruption site differs from a CE because a single CE can result in a corruption of multiple data sites.

Each i of the κ corruption sites is associated with three areas. $Area_{Pi}$ is the area where the algorithm positively identifies that a corruption site exists. (We mention in passing that false positives will increase this area, and so are included in forensic cost, though the algorithms developed in this article never produce false positives.) $Area_{Ui}$ is the area where we have no information as to whether a corruption site exists or not. $Area_{Ni}$ is the area where the algorithm establishes that no corruption site exists. For each corruption site, the sum of the areas, denoted by $TotalArea = Area_{Pi} + Area_{Ui} + Area_{Ni}$, corresponds to the horizontal trapezoid in the corruption diagram which is bound by the action line, the *When*-axis, and the two horizontal lines of the last two validation event scans. Hence, $TotalArea = (V \cdot N) \cdot (D - (1/2) \cdot V \cdot N)$. The forensic cost model only features $Area_P = \sum_i Area_{Pi}$ and $Area_U = \sum_i Area_{Ui}$ because minimizing these areas maximizes $Area_N$ with the overall effect of minimizing forensic cost.

Under this model we have already computed the forensic cost of commit-time-based algorithms [Pavlou and Snodgrass 2008]. The Monochromatic Algorithm has forensic cost $O(\kappa \cdot V \cdot D)$ under worst-case distribution of corruption sites while the a3D Algorithm has cost $O(\kappa \cdot N + D + \kappa \cdot \lg D)$.

As we will see in detail in the following, the forensic cost model does not depend on the underlying partitioning of data. For this reason the forensic cost of page-based algorithms remains the same as that of commit-time-based ones. The commit-time-based algorithms have been implemented and evaluated using actual values elsewhere [Pavlou and Snodgrass 2008]. In the case of attribute-based algorithms, the above statements are also true but because of the composite nature of the partitioning we end up having multiple instances of commit-time-based algorithms maintained in parallel. This is reflected in the forensic cost attribute-based algorithms which otherwise have the same forensic cost components as the commit-time-based algorithms except that the *Normal Processing* and $Area_U$ components will be scaled by R_d (the domain resolution) which is also the number of instances of commit-time-based algorithms we have to maintain in parallel.

9.2. Forensic Cost of Page-Based Algorithms

In order to compute the forensic cost of paged-based forensic analysis algorithms, we can use same formulas as above. This is due to the fact that the forensic cost model does not depend on the underlying partitioning of the data. Hence, the forensic cost remains the same for commit-time and page-based algorithms.

In page-based algorithms, the temporal bounds on the corruption event are reported in the exact same way as in the commit-time-based algorithms. The difference between the two groups lies is the way the *final* spatial bounds are reported. For (commit-time) a3D the spatial bounds are given in terms of a time interval. This means that any further analysis to find exactly where the corruption lies will have to look for all the tuples in the transactions which committed in the given time interval. For Page-based

a3D the spatial bounds are initially also given in terms of time, that is page write time. We can deduce this from the start and stop times of the hash chain which evaluated to false during forensic analysis. In this case, any further analysis will have to look for all the tuples in the pages which were written in that time interval. The page-based algorithm goes one step further and can actually find those pages from information kept during the validation scan of the entire database. The final spatial results are given in terms of a set of pages. Actually, the commit-time-based algorithms can be extended to operate in an analogous way. They can report the spatial results in terms of transaction IDs. (This implies that commit-time-based algorithms can have their own type of 3D corruption diagram.)

9.3. Generalization of Forensic Cost

We must now generalize the forensic cost model in order to create a model capable of characterizing the forensic cost of all the new algorithms, including the attribute-based ones. We augment the model by introducing the domain resolution R_d in all of the components of the model except for the forensic analysis component. In order to be consistent with the analysis given elsewhere [Pavlou and Snodgrass 2008], we also assume here that the four constant factors are equal to one.

$$\begin{aligned} FC(D, N, V, \kappa, R_d, p_c) = \; & NormalProcessing(D, N, V, R_d) \\ & + ForensicAnalysis(D, N, V, \kappa, p_c) \\ & + Area_P(D, N, V, \kappa, R_d) + Area_U(D, N, V, \kappa, R_d). \end{aligned}$$

The introduction of R_d is crucial because it captures the increase in the Normal Processing cost of the attribute-based algorithms and the decrease in the areas resulting from forensic analysis. The new parameter p_c is the number of partitions (out of a total of R_d) that have corruption sites, thus $1 \leq p_c \leq R_d$. It specifies on which partition(s) forensic analysis must be performed. Note that if $R_d = 1$ then the resulting algorithms, and thus their forensic cost, are identical to the commit-time and page-based ones. This property is termed *single-partition reducibility*.

9.4. Forensic Cost of the Attribute-Based Monochromatic Algorithm

Attribute-based forensic analysis algorithms are different from the other two algorithm groups. In the Attribute-based Monochromatic Algorithm, we are dealing with composite partitioning: attribute partitioning with nested commit-time partitioning. The attribute-based algorithms use the R_d subsets of the attribute domain to partition the data. Then, for each of the R_d subsets in the partition a conventional Monochromatic Algorithm is applied.

As in our initial analysis [Pavlou and Snodgrass 2008], here we assume a worst-case distribution for κ corruption sites, where all sites occur within a single total chain affecting tuples with partition attribute value in a single partition domain subset that is, $p_c = 1$. This guarantees that the $Area_U$ term in the forensic cost formula is not zero. Also, each site is added to the earliest untampered I_N. This distribution minimizes $Area_N$ and maximizes $Area_U$. Details on how the sites are added can be found elsewhere [Pavlou and Snodgrass 2008].

We also assume a uniform partition of the attribute domain. For this reason, the two types of area in the forensic cost must be divided by R_d in order to maintain an accurate comparison between the algorithms. In order to derive the asymptotic complexity we make, here and in the following sections, the simplifying assumptions that $1 \leq \kappa \leq D$, $1 \leq N \leq D$, $1 \leq V \leq D$, and $1 < R_d \leq D$. We first study the Attribute-based

Monochromatic Algorithm when $\kappa = 1$.

$$\begin{aligned} FC_{attr_mono}(D, 1, V, 1, R_d, 1) &= R_d \cdot NormalProcessing_{mono} \\ &\quad + 1 \cdot ForensicAnalysis_{mono} \\ &\quad + Area_{P\,mono}/R_d \\ &= R_d \cdot (D + D/V) \\ &\quad + 2 \cdot \lg D \\ &\quad + V/R_d \\ &= O(R_d \cdot D). \end{aligned} \tag{1}$$

The first three lines of the of the first equality show clearly the relationship between the components of the original forensic cost model and the general one. The Normal Processing cost of the attribute-base Monochromatic Algorithm is R_d times the Normal Processing cost of the original Commit-time-based Monochromatic. The Forensic Analysis component for the Attribute-based Monochromatic is the same as that of the commit-time based because a worst-case distribution of corruption sites implies that $p_c = 1$. The $Area_P$ of the attribute-based algorithms is only $1/R_d$ of the $Area_P$ of the commit-time-based ones because each area in the attribute-based algorithms is restricted to a single partition and hence captures only $1/R_d$ of the entire data.

On the first line of the second equality of equation (1), D is the number of notarization and D/V is the number of validations for a single Monochromatic Algorithm. Since we are maintaining R_d of them, one for each partition subset, we multiply the two terms. On the fifth line the term $2 \cdot \lg D$ is the cost of doing a binary search during the forensic analysis phase. Under worst-case distribution of corruption sites all sites are located within a single partition so we need only perform forensic analysis on a a single Monochromatic Algorithm. The term V/R_d on the sixth line corresponds to the $Area_P$ of the single corruption site we have. Hence no $Area_U$ is present. We now consider the case where $\kappa \geq 2$.

$$\begin{aligned} FC_{attr_mono}(D, 1, V, \kappa \geq 2, R_d, 1) &= R_d \cdot (D + D/V) \\ &\quad + 2 \cdot \lg D \\ &\quad + (1/R_d) \cdot (V + (\kappa - 1) \cdot (TotalArea - V)) \\ &= O(R_d \cdot D + \kappa \cdot V \cdot D/R_d). \end{aligned} \tag{2}$$

The only difference in the case of $\kappa \geq 2$ is the existence of $Area_U$ associated with $\kappa - 1$ sites because the Monochromatic Algorithm can only detect the corruption site affecting a tuple with the oldest commit time.

9.5. Forensic Cost of the Attribute-Based a3D Algorithm

For the Attribute-based a3D Algorithm we assume in a similar manner that all κ corruption sites have a worst-case distribution within a single partition subset so that only one of the R_d binary trees is affected. The corruption sites have worst-case distribution inside the binary tree, namely, they tamper every other leaf node.

$$\begin{aligned} FC_{attr_a3D}(D, N, 1, \kappa, R_d, 1) &= R_d \cdot NormalProcessing_{a3D} \\ &\quad + 1 \cdot ForensicAnalysis_{a3D} \\ &\quad + Area_{P\,a3D}/R_d \\ &= R_d \cdot (D/N + \mathcal{N}(D) + D/N - (1 + \lfloor \lg(D/N) \rfloor)) \\ &\quad + \mathcal{V}(\kappa) \\ &\quad + Area_{P\,a3D}/R_d \end{aligned}$$

Table VII.
Settings used in experimental validation of forensic cost assuming worst-case distribution of corruption sites (Values in **bold** are nonconfigurable)

Parameters	Algorithm					
	Commit-time/Page-based				Attribute-based	
	Monochromatic	RGBY	Tiled Bitmap	a3D	Monochromatic	a3D
R_d	1	1	1	1	4, 8	4, 8
R_s	1	1	1	1	1	1
N	**1**	**1**	8	8	**1**	8
V	8	**2**	1	**1**	8	**1**
D	40, 200, 1000					

$$
\begin{aligned}
&= R_d \cdot (D/N + 2 \cdot D - 1 + D/N - (1 + \lfloor \lg(D/N) \rfloor)) \\
&\quad + 2 \cdot \kappa \cdot (\lceil \lg D \rceil - \lceil \lg \kappa \rceil) + (1 + [\kappa \neq 2^i]) \cdot 2^{\lfloor \lg \kappa \rfloor + 1} - 1 \\
&\quad + (1/R_d) \cdot (\kappa \cdot N) \\
&= \Theta(\kappa \cdot N/R_d + R_d \cdot D + \kappa \cdot \lg D). \qquad (3)
\end{aligned}
$$

On the first line of the second equality of Equation (3), D/N is the number of validations for a single a3D Algorithm while $\mathcal{N}(D) + D/N - (1 + \lfloor \lg(D/N) \rfloor)$ is the number of notarizations. $\mathcal{N}(D)$ is a recursive formula which gives the number of chains in the binary tree and $D/N - (1 + \lfloor \lg(D/N) \rfloor)$ is the number of total chain hash values not part of the tree. The derivation of $\mathcal{N}(D)$ can be found elsewhere [Pavlou and Snodgrass 2008]. We maintain R_d of these binary trees so again all terms are multiplied by R_d. The term $\mathcal{V}(\kappa)$, on the second line of the second equality is the cost of performing forensic analysis on the binary tree. $\mathcal{V}(\kappa)$ is a recursive formula which computes the number of hash chains validated by the algorithm when κ corruption sites exist under a worst-case distribution. The derivation of $\mathcal{V}(\kappa)$ can also be found elsewhere [Pavlou and Snodgrass 2008]. All the κ corruption sites can be positively identified by a3D so $Area_U = 0$ while $Area_P$ is κ times the area N/R_d of a single site.

We can use big theta notation to describe the cost of Attribute-based a3D because similar analysis under a best-case distribution of corruption sites (not shown) yields the same asymptotic cost. This is not the case for the Attribute-based Monochromatic Algorithm, whose cost under best-case distribution is $O(R_d \cdot D + \kappa \cdot V/R_d)$.

9.6. Forensic Cost Evaluation of the Forensic Analysis Algorithms

We proceed to compare the forensic analysis algorithms and give recommendations. The values used to create the cost plots in Figures 9–13 are predicted values computed from the forensic cost formulas derived above. We have used all the terms with the recursions accurately computed, rather than the asymptotic behavior of the cost. We expect the predicted forensic cost values to be accurate because the attribute-based algorithms' costs are comprised of distinct cost components identical to ones found in commit-time-based ones, albeit with small rearrangements that constitute scaling of the commit-time cost components. The accuracy of the commit-time-based algorithms' forensic cost has been previously verified by comparing predicted values to actual ones computed from an implementation of the commit-time-based algorithms [Pavlou and Snodgrass 2008]. Hence, the predicted forensic cost for the attribute-based algorithms should be very close to what would be observed in practice.

Any artifacts on the cost curves of the following plots are due to the way they were generated in `gnuplot`. Even though we obtained values for all points in the stated domain, only certain values are marked with the distinguishing point symbol (e.g., ■, ×, ○) in order to avoid cluttering the image. Thus each cost was drawn twice using two data sets of differing cardinalities and the resulting curves may not always coincide.

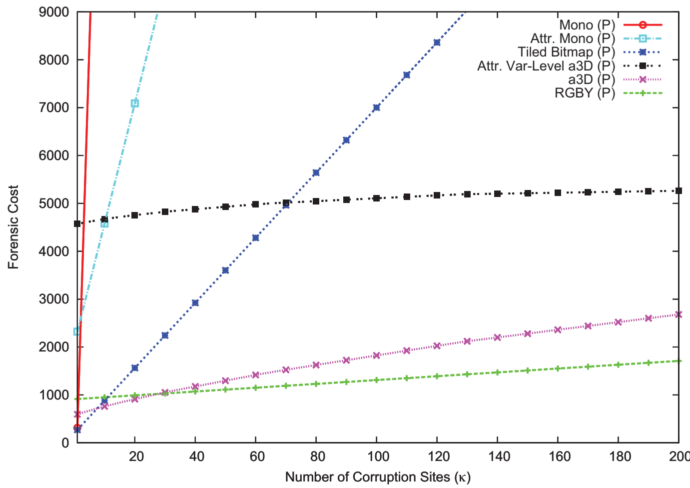
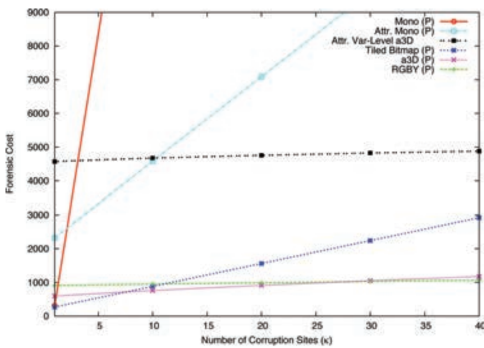
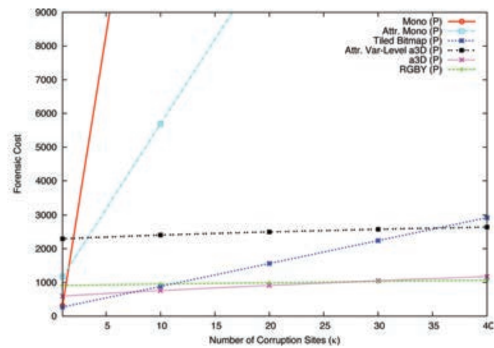


Fig. 9. The Forensic Cost versus κ for $D = 256$ and $R_d = 8$.



(a) $R_d = 8$



(b) $R_d = 4$

Fig. 10. The Forensic Cost against κ for $D = 256$ (detail).

Table VII summarizes the key parameter values used to generate the cost plots. Figures 9 and 10 investigate the impact a growing number of corruption sites on forensic cost for a fixed $D = 200$. All algorithms exhibit a linear behavior with respect to κ except the two a3D Algorithms, which are $O(\kappa - \kappa \cdot \lg \kappa)$. Even though this is a decreasing function on κ, only its ascending slightly leveling limb is visible because κ is bounded above by D.

We observe that as κ increases RGBY becomes the cheapest algorithm. However, RGBY suffers from false positives. Fortunately the second cheapest algorithm is a3D which can detect multiple corruptions with no false positives. Even though the Attribute-based a3D in only the third cheapest algorithm it outperforms all the rest of the algorithms. Thus a new attribute-based algorithm such as a3D can outperform even a commit-time-based algorithm like Tiled Bitmap. The advantage of Attribute-based a3D over RGBY is the same as that of a3D: a3D algorithms do not suffer from false positives. Attribute-based a3D has the added advantage of providing semantic information associated with the corruption. Once we identify which one of the R_d subsets contains the corruption site then we immediately know the range of the partition attribute values in the affected tuples. (This semantic information is not captured by our forensic cost model.) By incurring a slightly higher forensic cost, Attribute-based

a3D provides semantic information about the nature of the corrupted data which can help identify the perpetrator at a later stage.

Figure 10 displays the forensic cost curves for the first 40 corruption sites (before most of the algorithms completely diverge from each other) and for two different sizes of the domain partition ($R_d = 4, 8$). Figure 10(a) is a detail of Figure 9 with $R_d = 8$. When the number of corruption sites is small ($\kappa \leq 7$) Tiled Bitmap is the cheapest algorithm. Then for $8 \leq \kappa \leq 27$ a3D becomes cheapest and for values of $\kappa \geq 28$, a3D overtakes RGBY making the latter the algorithm with lowest cost. Even though none of the attribute-based algorithms can compete purely in terms of forensic cost, if additional semantics are required then for $\kappa \leq 10$ Attribute-based Monochromatic is preferable to Attribute-based a3D. For $\kappa \geq 11$ the situation is reversed. Remarkably, because the forensic cost of the Attribute-based a3D Algorithm increases slowly, the algorithm can be cheaper than Tiled Bitmap for certain values of κ. This is clearly seen in Figure 9 when κ reaches the value 71.

Reducing the R_d value (the number of partitions) from 8 to 4, as shown in Figure 10(b), has no effect on the page-based algorithms. However, as R_d decreases the Attribute-based Monochromatic Algorithm converges towards its page-based counterpart. For $R_d = 1$ the two algorithms are identical. The same applies to the Attribute-based a3D Algorithm which converges to the Page-based a3D Algorithm. This is the reason why the Attribute-based a3D Algorithm becomes cheaper than Tiled Bitmap at a smaller κ value when R_d is reduced from 8 to 4.

We now turn to the effect of the number of days D on the forensic cost for different values of κ. We show results for $\kappa = 1, 2, 30$, and 100. These values were chosen because major shifts in forensic costs are observed at these or close to these values, as also seen in Figures 9 and 10(a). Note that curves in these plots do not begin at $D = 1$. They begin at $D = \max\{I_V, \kappa\}$ because enough days must elapse before validation can occur or because a sufficient number of days is needed to accommodate all the corruption sites.

Figure 11 shows cost plots of forensic cost versus D for $\kappa = 1$ and 2. When a single corruption exists the Monochromatic Algorithm is the cheapest algorithm for the first 136 days. For days 137–141 Monochromatic and Tiled Bitmap share the same cost while past day 142 Tiled Bitmap becomes cheapest. This is to be expected since $\kappa = 1$ is a special case for the Monochromatic Algorithm: recall $Area_U$ is zero. A more accurate picture emerges in Figure 11(b), where the Monochromatic Algorithm becomes the most expensive of the page-based algorithms getting closer to the cost of the Attribute-based Monochromatic Algorithm. In the long term, the two attribute-based algorithms are the most expensive algorithms overall (for $\kappa = 1$ and 2). The reason for this is that the $R_d \cdot D$ term in the asymptotic behavior of the algorithms dominates the forensic cost. This is the price to be paid in exchange for the extra semantic information furnished by these algorithms.

If the number of corruption sites is increased to 30, as shown in Figure 12, then the cost of the Monochromatic Algorithm becomes so large that for the sake of clarity it is not included in the cost plot. To fully capture the forensic cost behavior we have extended the domain values to 1000 days. RGBY starts as the cheapest algorithm and on day it overtakes a3D. The a3D Algorithm remains the cheapest algorithm until day 975 when the Tiled Bitmap Algorithm becomes cheapest. The Attribute-based a3D is the algorithm of choice if additional semantic information is sought.

Figure 13(a) shows the effect on forensic cost of a large number of corruption sites ($\kappa = 100$) while Figure 13(b)shows the effect of halving the domain resolution ($R_d = 4$). Both figures show that the forensic cost curves exhibit a similar behavior to the ones in Figure 12. The main difference lies in the time it takes for the RGBY Algorithm to cross the a3D Algorithm, Tiled Bitmap to cross a3D, and Attribute-based a3D and Monochromatic to cross Tiled Bitmap. The bigger the κ the longer it takes for the

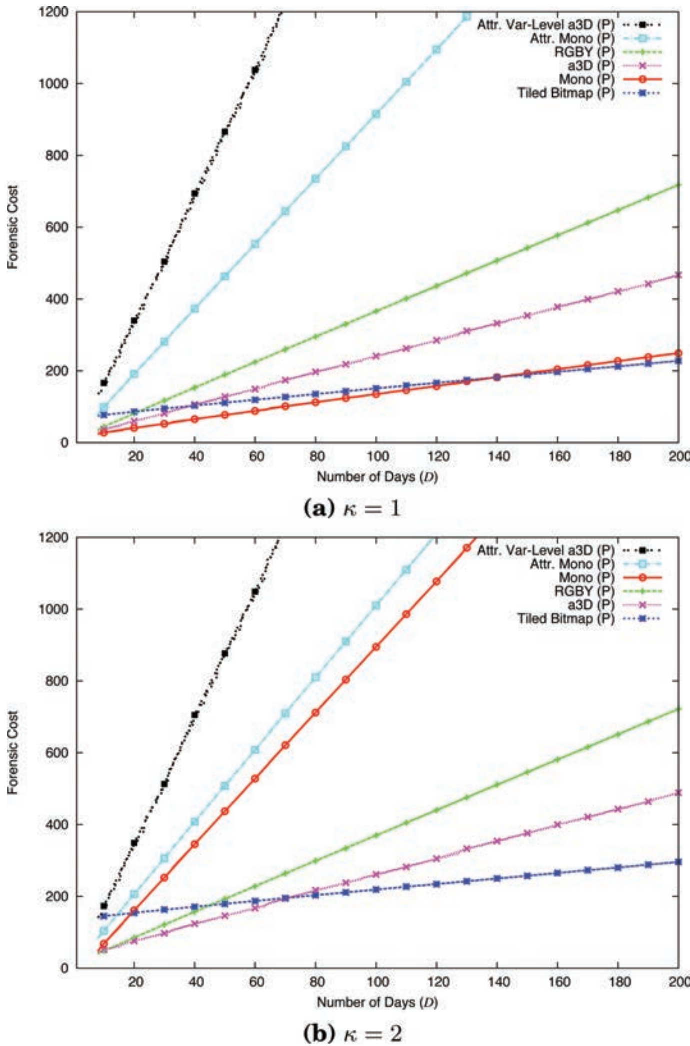


Fig. 11. The Forensic Cost against D for $R_d = 8$.

crossing to occur. Also, the cost of the Attribute-based Monochromatic Algorithm is sufficiently large that it starts at a higher value than Tiled Bitmap. Reducing R_d has the opposite effect; the crossings happen sooner.

10. RECOMMENDATIONS

As the expected number of corruption sites increases the algorithms with lowest forensic cost are Tiled Bitmap, RGBY, and a3D depending on the specific value of κ. This is of course the case if minimizing forensic cost is the only criterion used to choose an algorithm. If the existence of false positives is unacceptable then we must opt for a3D. If extra semantic information is required or if only part of the database suffices to be under audit then we have to turn to the Attribute-based a3D Algorithm which is the cheapest algorithm fulfilling these requirements.

Long-term cost is sensitive to κ. For $\kappa = 1$ the Monochromatic Algorithm rivals a3D and even manages to be cheaper than Tiled Bitmap at least for a while. Tiled Bitmap

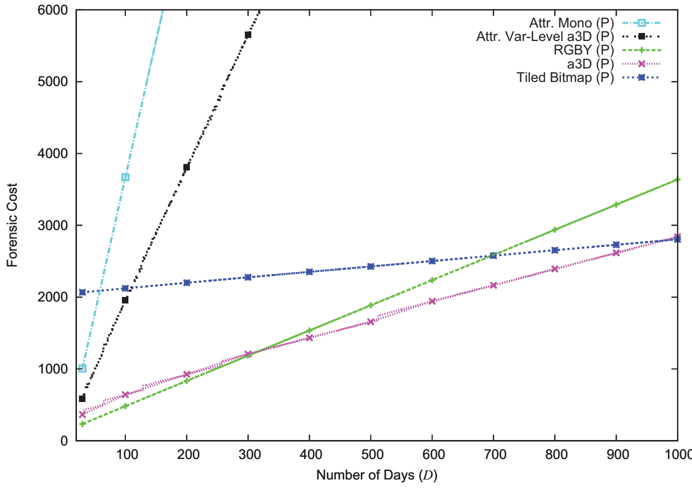


Fig. 12. The Forensic Cost against D for $R_d = 8$ and $\kappa = 30$.

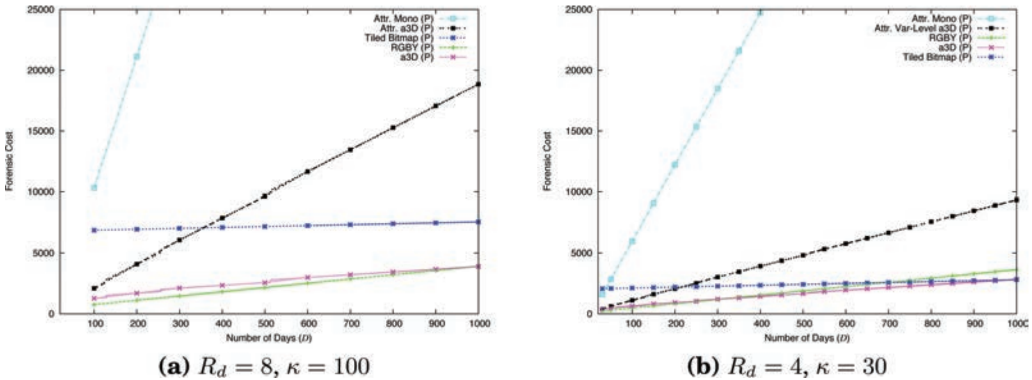


Fig. 13. The Forensic Cost against D for large κ and small R_d.

is consistently the cheapest algorithm in the long term regardless of the number of corruption sites. This algorithm is closely followed by a3D which has obvious advantages over Tiled Bitmap (easier to implement and no false positives). The only effect of increasing the number of corruption sites has is to postpone the time at which Tiled Bitmap and a3D become the cheapest algorithms.

If an attribute-based algorithm is required then the situation is slightly different than before. The Attribute-based Monochromatic Algorithm has the cheapest cost in the long term for a small number of corruption sites ($\kappa = 1, 2$). If κ increases then Attribute-based a3D becomes once again the cheapest among attribute-based algorithms in the long term.

Varying R_d can only affect the attribute-based algorithms. Our results show that R_d does not have as wide a range of useful values as we initially had hoped. Experiments with R_d values as high as 128 (not shown) make the forensic cost of these algorithms rival that of the Monochromatic Algorithm.

Overall, our findings corroborate the cost analysis of the commit-time-based algorithms found elsewhere [Pavlou and Snodgrass 2008]. The Tiled Bitmap and the RGBY provide forensic analysis with low associated forensic cost but suffer from false

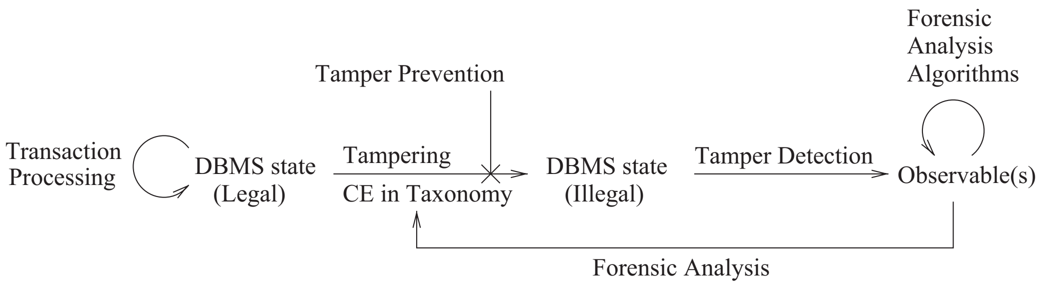


Fig. 14. The relationship between Observables, Taxonomy, Forensic Analysis Algorithms, and Forensic Analysis.

positives. False positives are significant factor which needs to taken into account when choosing an forensic algorithm. The rate of false positives has been quantified for both Tiled Bitmap and RGBY [Pavlou and Snodgrass 2008]. In the worst case scenario where corruption sites alternate with corruption-free areas of width I_N, RGBY can produce up to 50% false positives. The number of false positives in the Tiled Bitmap Algorithm, under worst case conditions where only the first granule of each tile is tampered, could be significantly higher than the number of false positives observed in the RGBY Algorithm: for each tile of size N granules we have $N-1$ false positives. The choice between the two depends on how acceptable false positives are and ultimately rests with the Chief Security Officer.

The Monochromatic Algorithm although cheap has limited forensic capabilities since it cannot detect multiple corruptions. If false positive results are unacceptable and long term forensic cost is tolerated then the a3D Algorithm is the algorithm of choice. Attribute-based variants of these algorithms should be used if it is important to obtain information on the corrupted attribute values of the data during forensic analysis.

One question remains. When does one choose page-based algorithms instead of the commit-time-based ones, especially in light of the fact that both sets of algorithms exhibit the same forensic cost? Here the answer revolves around differential advantages in implementation. A page-based approach is preferable over a commit-time-based one when scalability of the system to support multiple DBMSes with a uniform presentation of the results of forensic analysis is very important. Additional factors include performance, such as the smaller overhead of linear scans in page-based algorithms, and information hiding by not exposing of the page-based algorithms' inner workings to the DBMS file manager layer.

We now turn our focus to the variety of different types of corruptions and present a general forensic analysis protocol to detect them.

11. A TAXONOMY AND A DECISION GRAPH OF CORRUPTIONS

In this section we step back to attempt to understand and characterize the *space* of possible corruptions and the *comprehensiveness* of extant tamper detection and forensic analysis algorithms to be able to identify the full range of such corruptions.

To achieve this we use the high-level model shown in Figure 14. Initially the database exists in a legal state. When a corruption occurs the database transitions from a legal to an illegal state. The specific details of this illegal state are solely dependent on the type of corruption that transpired. Hence, one of the first things one needs to consider is the set of types of corruption events. To do so, we propose a *taxonomy* of what can go wrong. Then sufficient tamper detection tools and techniques must be developed so that the result of their application to the database can yield a set of *observables*.

Observables are clues which help detect corruptions as well as reveal the details of the illegal state of the DBMS. We can then define *forensic analysis* as a map from the observables to the elements in the taxonomy. Ideally, we would like to have a one-to-one correspondence, between the observables and elements (kinds of corruptions) in the taxonomy, so that forensic analysis can unequivocally identify each possible corruption. For this to happen, the available tools should be able then to determine the causal factors (what, where, when) of the corruption, such that subsequent manual analysis can determine who and why, given the illegal state of the database.

Figure 15 is a UML [Booch et al. 2005] taxonomy which shows all the places within the DBMS where a *single* tampering can occur along with the resulting type of corruption event. This diagram concerns a single corruption event, therefore some of the conclusions are drawn by the process of elimination. The situation is more complex when multiple corruptions are involved.

Tampering could affect data on disk, the schema of the data on disk, the transaction log (on disk), or even data stored in main memory. In the case of main memory tampering, corruption could affect data brought into main memory from disk or the hash chain values evaluated from the data. Tampering the hash values in main memory poses a challenge to our current approach.

Before we delve into the taxonomy in more detail observe that all subclasses are disjoint (denoted by ⓓ) and that all superclass/subclass relationships are total (denoted by the double line descending from the little triangle below the superclass into the constituent subclasses).

If data are corrupted on disk then this could affect page write timestamps, a tuple attribute, or even an entire tuple. Timestamps are stored on disk (and within pages) only under a page hashing/partitioning scheme. If such a timestamp is corrupted then we must identify if it has been postdated (changed to later time) or backdated (changed to earlier time).

If an attribute value of a tuple is corrupted then we must distinguish whether the database is partitioned on this attribute. This is important because these corruptions produce different observables during forensic analysis. Even though in most cases we can identify Non-Partition Attribute corruptions, there are situations in which we cannot distinguish between these and corruptions of an attribute in a tuple stored on non-static-numbered pages. In fact, no forensic analysis algorithm can yet distinguish between all possible types of corruptions when using non-static-numbered page hashing schemes.

Corruption of a partition field can lead to either a timestamp (transaction commit time) corruption or a corruption of any other attribute value. If a partition attribute is corrupted, and given that the domain of the partition attribute is ordered (either naturally or artificially), then we need to discover whether the value resulting from the corruption is lesser or greater than the original. Timestamp corruption is a special case of Attribute corruption, in which we need to distinguish between corruptions that changed the original timestamp to an earlier time (Backdated) or a later time (Postdated).

Moving back up the tree, if Data Corruption affects an entire tuple then there are four possibilities: the data in the tuple is deleted, the header of the data is deleted, just the tuple header is corrupted, or an entire tuple is inserted. All these will make the data of the tuple inaccessible.

Finally, if a Corruption Event affects the schema of the database then we could face corruption of the specific Table and Column schemas. Of course corruption of other schema information is also possible.

In the Appendix we give a forensic analysis protocol to identify leaves in the taxonomy. Currently, the protocol does not include tools or techniques that allow for the analysis of transaction log corruption, main memory corruption, or page number

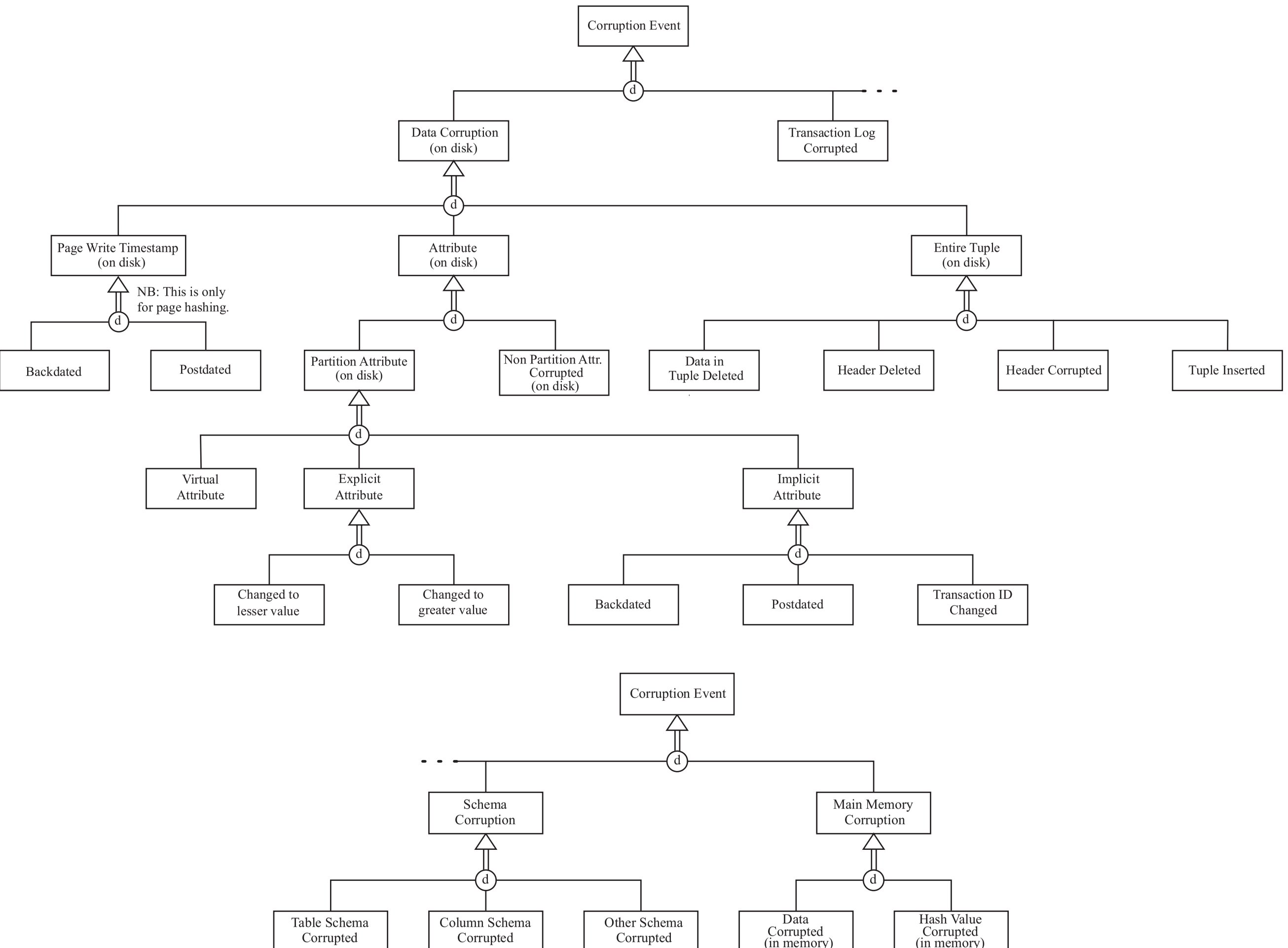


Fig. 15. The Corruption Event Taxonomy. (NB: Due to space limitations the diagram was split into two parts, given one above the other.)

corruption (a virtual attribute corruption) under non-static-numbered page partitioning. In the case of entire tuple corruption, the forensic analysis algorithms will detect a corruption but there are no techniques yet to distinguish between the corruption's different subtypes. Our protocol can distinguish between the remaining types of corruption.

12. RELATED WORK

Recent years have seen an increase in research activity surrounding the issues of compliance, tamper-resistance, appropriate use, and provenance of business records. In this section we focus on the relevant issues of tamper-resistance, tamper detection, and forensic analysis of records in high-performance relational databases.

Various long-term high-integrity retention techniques for fine granularity business records (database tuples) have been proposed based on WORM storage servers. One such example is a *log-consistent compliant database architecture* (LDA) that extends immutability to relational tuples [Mitra 2008; Mitra et al. 2009]. This system stores a database snapshot on WORM at audit time, then uses an additional *compliance log* stored on WORM to record database modifications. This snapshot plus the compliance log let the auditor verify that the new database state is compliant.

A more efficient architecture is the *transaction log on WORM* (TLOW) approach for supporting long-term immutability of relational tuples [Hasan and Winslett 2011]. TLOW stores the current database instance in ordinary storage and the transaction log on WORM storage, while dispensing with the compliance log altogether. The audit process uses hash values representing the data in database snapshots and logs rather than the data themselves. In a similar fashion as in LDA, after a successful audit, the auditor writes a snapshot of the current database state to WORM storage. However, in TLOW the auditor also stores on WORM a hash of the contents of this snapshot obtained using a cryptographically strong hash function. The auditor then signs both the snapshot and its hash value. For the next audit, the auditor checks the signature on the hash of the previous snapshot and generates new hashes from the current database state and from the log. An audit is successful if the hash from the old database snapshot plus the hash of all the new tuples introduced in the transaction log is equal to the hash of the current database instance. TLOW has several advantages over LDA. It requires no changes to the DBMS kernel so legacy applications can run on it without any changes. Moreover, TLOW imposes less than 1% overhead in transaction throughput under a TPC-C benchmark and with a special *audit helper* add-on module audit time is reduced to two hours.

AuditGuard is a new framework that allows auditing of a relational database that is subject to retention policies [Lu et al. 2012]. The challenge arises from the fact that such policies while trying to address privacy can run contrary to the ability to provide meaningful answers to auditing queries. The authors present the historical data model they employ by describing the structure of the audit log, transaction-time relations and how essentially one can be derived from the other (audit-fields notwithstanding). They define and describe declarative rules that can express two types of retention rules, namely, redaction and expunction. They then present, implement, and evaluate two models: the Tuple-Independent (TI) and Tuple-Correlated (TC) model which allow audit queries on tables containing incomplete information or uncertainty. Implementations of both the TI and TC models are evaluated and compared.

We emphasize that tamper prevention is not the only way to address the problem posed by unauthorized database tampering. Weitzner et al. [2008] argue that access control and cryptography are not capable of protecting information privacy and that there is a true dearth of mechanisms for effectively addressing information leaks. They propose that as an alternative information accountability "must become a primary

means through which society addresses appropriate use" [Weitzner et al. 2008]. *Information accountability*, in this context, assumes that information should be transparent so as to easily determine whether a particular use is appropriate under a given set of rules. The following works elaborate their systems and techniques along these lines.

Basu [2006] presents a method of forensic tamper detection of sensitive data in SQL Server. His method does not provide tamper-prevention measures, but shows how tampering can be detected and how to localize the affected data. The solution is based on creating an interwoven chain of hash values which will ensure that if a particular row is modified, inserted, or deleted from the audit log table, the detection algorithm can find a mismatch between newly calculated hash values and those stored in the special columns `HReserved` and `VReserved`. Even though this method has some nice advantages (e.g., lack of overhead resulting form maintaining the data in nonencrypted form, no special deployment strategy required), it suffers from some serious drawbacks. The hash functions used are not cryptographically strong (the hash values are computed using `BINARY_CHECKSUM` and `CHECKSUM_AGG`), and the algorithm has limited forensic strength (in that certain delete operations cannot be detected).

An entirely different approach to tamper detection that can encompass database forensics is *database watermarking*. In general, digital watermarking for the purpose of integrity verification is called *fragile watermarking* whereas *robust watermarking* is used for copyright protection. When using a robust watermarking scheme, the goal is to embed in data a watermark that is resilient against attacks that aim at removing the watermark or at making it undetectable. In a fragile watermarking scheme, the embedded watermark is such that it becomes "broken" by modifications. This characteristic of the watermark allows for the detection and localization of the tampered data. Agrawal and Kiernan [2002] have devised a robust watermarking scheme for databases which uses watermark bits embedded in the data by modifying some bits of some attributes. The modifications happen according to a secret embedding key.

A different robust watermarking scheme for databases was devised by Sion et al. [2003]. Whereas Agrawal's watermarking scheme is tuple-based, Sion's scheme is based on securely dividing tuples into nonintersecting subsets. Each subset has a single watermark bit embedded into its tuples by modifying the distribution of tuple values. The same watermark bit is embedded repeatedly across several subsets and the majority voting technique is employed to recover the embedded bits. This scheme is robust against subset attacks, data resorting, and data transformation. The major drawback with this scheme is the fact that it is not suitable for database relations that are frequently updated, due to the high overhead associated with re-watermark the updated relations.

Guo, Jajodia, Li, and Liu formulated a fragile watermarking scheme for databases [Guo et al. 2006; Li et al. 2004]. Their scheme is based on a watermark that is *invisible*, meaning that the embedded watermark does not introduce any distortions to the data (hence, this approach is also suitable for categorical data). Furthermore, the watermark can be *blindly verified* meaning that the original unmarked database relation is not required for verification. Finally, during verification, the extracted watermark indicates the locations of alterations down to the granularity of a range of tuples. The watermark is based on the hash values of the tuples' primary key value, their attribute values, and a secret embedding key. Unauthorized tampering of data is detected when forensic analysis yields a mismatch between the originally embedded watermark and the extracted watermark. This scheme wrests control on tuple placement from the DBMS and suffers from false positives.

All these database forensic techniques are valuable but most of them are evaluated using a probabilistic analysis. Furthermore, watermark embedding techniques may distort data and have a high overhead. Although watermarking techniques can provide

spatial bounds on the tampered data (by localizing them), they provide no temporal bounds on when the tampering transpired. These are concerns we wish to avoid.

Database tampering evinces three basic concerns: (i) how to detect tampering, (ii) how to determine roughly where and when tampering occurred, and (iii) how to further narrow the corruption regions and/or make forensic analysis or normal processing more efficient via successive refinements and generalizations. Our original paper on tamper detection [Snodgrass et al. 2004] was concerned with the first issue. Subsequent papers [Pavlou and Snodgrass 2006, 2008, 2010] elaborated on the second issue. In contrast, the present article, particularly Sections 4 and 11, on expanding the concept of "where" through page hashing and partitioning on attributes, is solidly in the mold of issue (iii). Moreover, the present article introduces a taxonomy of corruption types and a protocol which demonstrates how the forensic algorithms can distinguish between different types of corruption.

13. SUMMARY AND FUTURE WORK

Integrated solutions for ensuring the integrity of databases, detecting data corruption even by insiders, and performing forensic analysis on such corruptions are of great interest in a variety of application domains.

The goal of the this work is to create (a) a generalization of forensic analysis algorithms that provides more semantically relevant information about detected tampering, (b) an overarching characterization of the process of database forensic analysis, and (c) a general approach to this task that provides a context within the overall operation of a database for all existing forensic analysis algorithms.

We have expanded the notion of "Where" by introducing page-based partitioning along with attribute-based partitioning. Thus we can speak of *attribute-based partitioning* and an *attribute-based corruption diagram* that characterizes a database as partitioned on different attributes, be they time (recorded in the transaction start and stop implicit attributes), pages (determined by the page number of the page the tuple resides in), or any tuple attribute that can be correlated with time (and thus ordered). We have illustrated these approaches by applying both to two forensic algorithms: the Monochromatic and a3D Algorithms.

The expansion of forensic analysis algorithms necessitated the development of a generalized forensic cost model which could characterize all algorithms. For a small number of expected corruptions the algorithms with lowest forensic cost are the Tiled-Bitmap and a3D, whereas for a large number of corruptions the use of RGBY is recommended. If the existence of false positives is undesirable then one must opt for the a3D Algorithm. If additional semantic information is needed one can turn to the Attribute-based a3D Algorithm. In case the long-term cost of the algorithms is of importance, then the Tiled-Bitmap algorithm must be used. Page-based algorithms are preferable to commit-time-based ones when scalability, performance, and information hiding take priority over other criteria.

In subsequent sections we introduced a taxonomy of various types of corruption events and a definition of forensic analysis. Then based on that definition, we constructed a forensic analysis protocol which described the available tools and methods required to distinguish between the different types of corruption.

The developed algorithms can help ensure record compliance for financial and medical institutions. They can serve as an unbiased witness to any type of database storing sensitive information. These may include court-submitted data from police databases or biological research results. The latter can be of particular use to a biosciences lab because it can ensure nondeviation from protocols thus providing a certain type of provenance for their final results. Furthermore, they can be utilized for

the improvement of software and protect databases from bugs silently corrupting the system by potentially providing hints for isolating the piece of code responsible.

The techniques proposed do not just protect data but also provide continuous assurance as to the quality of the data, since the system will be able to detect corruption shortly after tampering and automate to a great extent the work required in the aftermath of a database corruption. The techniques also highlight the advantages over approaches relying heavily on information restriction through either hardware which can have prohibitive costs for small institutions, have a limited shelf-life and are relatively complex; or require cryptography which does not adequately offer remedies after a leak.

Several interesting issues remain to be resolved. The first is to address the problem of temporary corruption. The discussion in Section 2 assumes a nonzero regret interval which allows for the detection of a temporary corruption. Techniques are needed that can detect a temporary corruption under the assumption of an empty regret interval.

Regarding the alternative implementations of page-based forensic algorithms we need to explore the architectural opportunities provided by the shift from a logical to a physical perspective in the audit system and forensic analysis as well as specific implementation issues such as addressing the complication of records spanning multiple pages and the handling of data skew in attribute-based techniques.

It would be desirable to incorporate semantic information, such as the that provided by the attribute-based algorithms, into the forensic cost model. The next task is the creation of new forensic analysis techniques to map observables to all the individual types of corruption in the taxonomy. This would serve as good basis for the creation of a forensic analysis protocol which would describe the tools and techniques required to analyze multiple corruption events.

Another challenging problem is to create a *non-static-numbered* page hashing approach. This is important because it would be applicable to DBMS storage structures where tuples can migrate between pages.

Finally, we need to consider the corruption of data in main memory. This is rather hard to achieve since a well-maintained operating system will make this line of attack unattractive. Nevertheless, we need consider it for the sake of completeness and because it has some interesting implications. A full analysis of the corruption of hash values in main-memory is also required in order to complete the characterization of the forensic analysis decision graph.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library. The appendix presents a step-by-step discussion of the *forensic analysis protocol* which allows us to identify many types of corruption events.

ACKNOWLEDGMENTS

We thank Alejandro Estrella-Balderrama for computing the initial level-planar graph embedding for the taxonomy (Figure 15) and the associated forensic analysis protocol. We also thank Christian Collberg, Peter Downey, Nirav Merchant, Soumyadeb Mitra, Radu Sion, Joseph Watkins, and Marianne Winslett for numerous and very helpful discussions on compliant databases and on tamper detection and prevention. Finally, we thank the reviewers for their helpful comments.

REFERENCES

AGRAWAL, R. AND KIERNAN, J. 2002. Watermarking relational databases. In *Proceedings of the International Conference on Very Large Databases*. VLDB Endowment, 155–166.

AHN, I. AND SNODGRASS, R. T. 1988. Partitioned storage structures for temporal databases. *Inf. Syst. 13*, 4, 369–391.

BAIR, J., BÖHLEN, M., JENSEN, C. S., AND SNODGRASS, R. T. 1997. Notions of upward compatibility of temporal query languages. *Business Informatics (Wirtschafts Informatik) 39*, 1, 25–34.

BASU, A. 2006. Forensic tamper detection in SQL server. http://www.sqlsecurity.com/chipsblog/archivedposts.

BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. 2005. *The Unified Modeling Language User Guide* 2nd Ed. Addison-Wesley Professional.

GERR, P. A., BABINEAU, B., AND GORDON, P. C. 2003. Compliance: The effect on information management and the storage industry. Res. rep., Enterprise Storage Group.

GUO, H., LI, Y., LIU, A., AND JAJODIA, S. 2006. A fragile watermarking scheme for detecting malicious modifications of database relations. *Inf. Sci. 176*, 10, 1350–1378.

HABER, S. AND STORNETTA, W. S. 1999. How to time-stamp a digital document. *J. Cryptology 3*, 99–111.

HASAN, R. AND WINSLETT, M. 2011. Efficient Audit-based Compliance for Relational Data Retention. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11)*. ACM, New York, 238–248.

HIPAA, US DEPARTMENT OF HEALTH & HUMAN SERVICES. 1996. The Health Insurance Portability and Accountability Act. https://www.cms.gov/Regulations-and-Guidance/HIPAA-Administrative-Simplification/HIPAAGenInfo/index.html.

IBM CORPORATION. 2010. A matter of time: Temporal data management in DB2 for z/OS. White paper, IBM.

LI, Y., GUO, H., AND JAJODIA, S. 2004. Tamper detection and localization for categorical data using fragile watermarks. In *Proceedings of the 4th ACM Workshop on Digital Rights Management*. ACM, New York, 73–82.

LOMET, D. AND SALZBERG, B. 1989. Access methods for multiversion data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, 315–324.

LORENTZOS, N. A. 2009. *Encyclopedia of Database Systems*. Springer, Chapter on Value Equivalence.

LU, W., MIKLAU, G., AND IMMERMAN, N. 2012. Auditing a database under retention policies. *VLDB J.*, 1–26.

MALMGREN, M. 2007. An infrastructure for database tamper detection and forensic analysis. Honors thesis, University of Arizona. http://www.cs.arizona.edu/projects/tau/tbdb/MelindaMalmgrenThesis.pdf.

MITRA, S. 2008. Trustworthy and cost effective management of compliance records. PhD dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign.

MITRA, S., WINSLETT, M., SNODGRASS, R. T., YADUVANSHI, S., AND AMBOKAR, S. 2009. An architecture for regulatory compliant database management. In *Proceedings of the IEEE International Conference on Data Engineering*. 162–173.

ORACLE CORPORATION. 2009. Oracle Database 11g Workspace Manager overview. Oracle Corporation. http://www.oracle.com/technetwork/database/twp-appdev-workspace-manager-11g-128289.pdf.

PAVLOU, K. E. AND SNODGRASS, R. T. 2006. Forensic analysis of database tampering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 109–120.

PAVLOU, K. E. AND SNODGRASS, R. T. 2008. Forensic analysis of database tampering. *ACM Trans. Datab. Syst. 33*, 4, 30:1–30:47.

PAVLOU, K. E. AND SNODGRASS, R. T. 2010. The tiled bitmap forensic analysis algorithm. *IEEE Trans. Knowl. Data Eng. 22*, 4, 590–601.

RAMAKRISHNAN, R. AND GEHRKE, J. 2003. *Database Management Systems* 3rd Ed. McGraw-Hill.

SARBANES-OXLEY ACT, U.S. PUBLIC LAW NO. 107–204, 116 STAT. 745. 2002. The Public Company Accounting Reform and Investor Protection Act. (2002).

SION, R., ATALLAH, M., AND PRABHAKAR, S. 2003. Rights protection for relational data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, 98–109.

SNODGRASS, R. T. AND AHN, I. 1986. Temporal databases. *IEEE Comput. 19*, 9, 35–42.

SNODGRASS, R. T., YAO, S. S., AND COLLBERG, C. 2004. Tamper detection in audit logs. In *Proceedings of the International Conference on Very Large Databases*. 504–515.

TERADATA CORPORATION. 2012. Teradata transforms global database technology. http://www.teradata.com/News-Releases/2012/Teradata-Transforms-Global-Database-Technology/.

US NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. 2012. Federal Information Processing Standards Publication 180-4: Secure Hash Standard. (March 2012). http://www.csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf.

WEITZNER, D. J, ABELSON, H., BERNERS-LEE, T., FEIGENBAUM, J., HENDLER, J., AND SUSSMAN, G. J. 2008. Information accountability. *Comm. ACM 51*, 6, 82–87.

Received January 2012; revised September 2012, January 2013; accepted March 2013