Forensic Analysis of Database Tampering

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

Regulations and societal expectations have recently expressed the need to mediate access to valuable databases, even by insiders. One approach is tamper detection via cryptographic hashing. This article shows how to determine when the tampering occurred, what data was tampered with, and perhaps, ultimately, who did the tampering, via forensic analysis. We present four successively more sophisticated forensic analysis algorithms: the Monochromatic, RGBY, Tiled Bitmap, and a3D algorithms, and characterize their "forensic cost" under worst-case, best-case, and average-case assumptions on the distribution of corruption sites. A lower bound on forensic cost is derived, with RGBY and a3D being shown optimal for a large number of corruptions. We also provide validated cost formulæ for these algorithms and recommendations for the circumstances in which each algorithm is indicated.

Categories and Subject Descriptors: H.2.0 [**Database Management**]: General—*Security, integrity, and protection*

General Terms: Algorithms, Performance, Security

Additional Key Words and Phrases: a3D algorithm, compliant records, forensic analysis algorithm, forensic cost, Monochromatic algorithm, Polychromatic algorithm, RGBY algorithm, Tiled Bitmap algorithm.

ACM Reference Format:

Pavlou, K. E. and Snodgrass, R. T. 2008. Forensic analysis of database tampering. ACM Trans. Datab. Syst. 33, 4, Article 30 (November 2008), 47 pages. DOI = 10.1145/1412331.1412342 http://doi.acm.org/10.1145/1412331.1412342

1. INTRODUCTION

Recent regulations require many corporations to ensure trustworthy long-term retention of their routine business documents. The US alone has over 10,000 regulations [Gerr et al. 2003] that mandate how business data should be managed [Chan et al. 2004; Wingate 2003], including the Health Insurance Portability and Accountability Act: HIPAA [1996], Canada's PIPEDA

NSF grants IIS-0415101, IIS-0639106, and EIA-0080123 and a grant from Microsoft provided partial support for this work.

Authors' address: K. E. Pavlou and R. T. Snodgrass, Department of Computer Science, University of Arizona, Tucson, AZ 85721-0077; email: {kpavlou, rts}@cs.arizona.edu.

[2000], Sarbanes-Oxley Act [2002], and PITAC's advisory report on health care [Agrawal et al. 2007]. Due to these and to widespread news coverage of collusion between auditors and the companies they audit (e.g., Enron, WorldCom), which helped accelerate passage of the aforementioned laws, there has been interest within the file systems and database communities about built-in mechanisms to detect or even prevent tampering.

One area in which such mechanisms have been applied is *audit log security*. The Orange Book [Department of Defense 1985] informally defines audit log security in Requirement 4: "Audit information must be selectively kept and protected so that actions affecting security can be traced to the responsible party. A trusted system must be able to record the occurrences of security-relevant events in an audit log ... Audit data must be protected from modification and unauthorized destruction to permit detection and after-the-fact investigations of security violations."

The need for audit log security goes far beyond just the financial and medical information systems mentioned previously. The 1997 U.S. Food and Drug Administration (FDA) regulation "part 11 of Title 21 of the Code of Federal Regulations; Electronic Records; Electronic Signatures" (known affectionately as "21 CFR Part 11" or even more endearingly as "62 FR 13430") requires that analytical laboratories collecting data used for new drug approval employ "user independent computer-generated time stamped audit trails" [FDA 2003].

Audit log security is one component of more general *record management systems* that track documents and their versions, and ensure that a previous version of a document cannot be altered. As an example, *digital notarization services* such as Surety (`www.surety.com`), when provided with a digital document, generate a *notary ID* through secure one-way hashing, thereby locking the contents and time of the notarized documents [Haber and Stornetta 1991]. Later, when presented with a document and the notary ID, the notarization service can ascertain whether that specific document was notarized, and if so, when.

Compliant records are those required by myriad laws and regulations to follow certain "processes by which they are created, stored, accessed, maintained, and retained" [Gerr et al. 2003]. It is common to use Write-Once-Read-Many (WORM) storage devices to preserve such records [Zhu and Hsu 2005]. The original record is stored on a write-once optical disk. As the record is modified, all subsequent versions are also captured and stored, with metadata recording the timestamp, optical disk, filename, and other information on the record and its versions.

Such approaches cannot be applied directly to high-performance databases. A copy of the database cannot be versioned and notarized after each transaction. Instead, audit log capabilities must be moved into the DBMS. We previously proposed an innovative 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 [Snodgrass et al. 2004]. This is accomplished by hashing data manipulated by transactions and periodically *validating* the audit log database to detect when it has been altered.

The question then arises, what do you do when an intrusion has been detected? At that point, all you know is that at some time in the past, data somewhere in the database has been altered. *Forensic analysis* is needed to ascertain *when* the intrusion occurred, *what* data was altered, and ultimately, *who* the intruder is.

In this article, we provide a means of systematically performing forensic analysis after an intrusion of an audit log has been detected. (The identification of the intruder is not explicitly dealt with.) We first summarize the originally proposed approach, which provides exactly one bit of information: has the audit log been tampered with? We introduce a schematic representation termed a *corruption diagram* for analyzing an intrusion. We then consider how additional validation steps provide a sequence of bits that can dramatically narrow down the when and where. We examine the corruption diagram for this initial approach; this diagram is central in all of our further analyses. We characterize the *forensic cost* of this algorithm, defined as a sum of the external notarizations and validations required and the area of the uncertainty region(s) in the corruption diagram. We look at the more complex case in which the *timestamp* of the data item is corrupted, along with the data. Such an action by the intruder turns out to greatly increase the uncertainty region. Along the way, we identify some configurations that turn out not to improve the precision of the forensic algorithms, thus helping to cull the most appropriate alternatives.

We then consider computing and notarizing additional sequences of hash values. We first consider the Monochromatic Algorithm; we then present the RGBY, Tiled Bitmap, and a3D Algorithms. For each successively more powerful algorithm, we provide an informal presentation using the corruption diagram, the algorithm in pseudocode, and then a formal analysis of the algorithm's asymptotic run time and forensic cost. We end with a discussion of related and future work. The appendix includes an analysis of the forensic cost for the algorithms, using worst-case, best-case, and average-case assumptions on the distribution of corruption sites.

2. TAMPER DETECTION VIA CRYPTOGRAPHIC HASH FUNCTIONS

In this section, we summarize the *tamper detection* approach we previously proposed and implemented [Snodgrass et al. 2004]. We just give the gist of our approach, so that our forensic analysis techniques can be understood.

This basic approach differentiates two execution phases: *online processing*, in which transactions are run and hash values are digitally notarized, and *validation*, in which the hash values are recomputed and compared with those previously notarized. It is during validation that tampering is detected, when the just-computed hash value doesn't match those previously notarized. The two execution phases constitute together the *normal processing phase* as opposed to the *forensic analysis phase*. Figure 1 illustrates the two phases of normal processing.

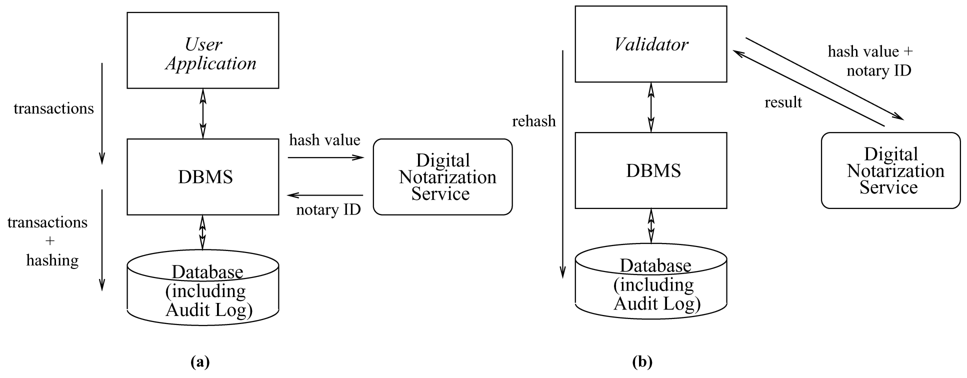


Fig. 1. Online processing (a) and Audit log validation (b).

In Figure 1(a), the user application performs transactions on the database, which insert, delete, and update the 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), in a manner completely transparent to the user application [Bair et al. 1997]. An important property of all data stored in the database is that it is *append-only*: modifications only add information; no information is ever deleted. Hence if old information is changed in any way, then tampering has occurred. Oracle 11g supports transaction-time tables with its workspace manager [Oracle Corporation 2007]. The Immortal DB project aims to provide transaction time database support built into Microsoft SQL Server [Lomet et al. 2005]. How this information is stored (in the log, in the relational store proper, in a separate "archival store" [Ahn and Snodgrass 1988]) is not that critical in terms of forensic analysis, as long as previous tuples are accessible in some way. In any case, the DBMS retains for each tuple hidden Start and Stop times, recording when each change occurred. The DBMS ensures that only the current state of the table is accessible to the application, with the rest of the table serving as the audit log. Alternatively, the table itself could be viewed by the application as the audit log. In that case, the application only makes insertions to the audited table; these insertions are associated with a monotonically increasing Start time.

We use a *digital notarization service* that, when provided with a digital document, provides a *notary ID*. Later, during audit log validation, the notarization service can ascertain, when presented with the supposedly unaltered document and the notary ID, whether that document was notarized, and if so, when.

On each modification of a tuple, the DBMS obtains a timestamp, computes a *cryptographically strong one-way hash function* of the (new) data in the tuple and the timestamp, and sends that hash value, as a digital document, to the notarization service, obtaining a notary ID. The DBMS stores that ID in the tuple.

Later, an intruder gets access to the database. If he changes the data or a timestamp, the ID will now be inconsistent with the rest of the tuple. The intruder cannot manipulate the data or timestamp so that the ID remains valid, because the hash function is one-way. Note that this holds even when the intruder has access to the hash function itself. He can instead compute a new hash value for the altered tuple, but that hash value won't match the one that was notarized.

An independent audit log validation service later scans the database (as illustrated in Figure 1(b)), hashes the data and the timestamp of each tuple, and provides it with the ID to the notarization service, which then checks the notarization time with the stored timestamp. The validation service then reports whether the database and the audit log are consistent. If not, either or both have been compromised.

Few assumptions are made about the threat model. The system is secure until an intruder gets access, at which point he has access to everything: the DBMS, the operating system, the hardware, and the data in the database. We still assume that the notarization and validation services remain in the trusted computing base. This can be done by making them geographically and perhaps organizationally separate from the DBMS and the database, thereby effecting correct tamper detection even when the tampering is done by highly motivated insiders. (A recent FBI study indicates almost half of attacks were by insiders [CSI/FBI 2005].)

The basic mechanism just described provides correct tamper detection. If an intruder modifies even a single byte of the data or its timestamp, the independent validator will detect a mismatch with the notarized document, thereby detecting the tampering. The intruder could simply re-execute the transactions, making whatever changes he wanted, and then replace the original database with his altered one. However, the notarized documents would not match in time. Avoiding tamper detection comes down to inverting the cryptographically-strong one-way hash function. Refinements to this approach and performance limitations are addressed elsewhere [Snodgrass et al. 2004].

A series of implementation optimizations minimize notarization service interaction and speed up processing within the DBMS: opportunistic hashing, linked hashing, and a transaction ordering list. In concert, these optimizations reduce the run time overhead to just a few percent of the normal running time of a high-performance transaction processing system [Snodgrass et al. 2004]. For our purposes, the only detail that is 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 value of each individual transaction is linked in a sequence, with the final value being essentially a hash of all changes to the database since the database was created. 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 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. (Actually,

Table I. Summary of Notation Used

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

the validator does provide a very vague sense of when: sometime before now, and where: somewhere in the data stored before now.)

It is the subject of the rest of this article to examine how to perform a forensic analysis of a detected tampering of the database.

3. DEFINITIONS

We now examine tamper detection in more detail. Suppose that we have just detected a *corruption event* (*CE*), which is any event that corrupts the data and compromises the database. (Table I summarizes the notation used in this article. Some of the symbols are introduced in subsequent sections.)

The corruption event could be due to an intrusion, some kind of human intervention, a bug in the software (be it the DBMS or the file system or somewhere in the operating system), or a hardware failure, either in the processor or on the disk. There exists a one-to-one correspondence between a CE and its *corruption time* (t_c), which is the actual time instant (in seconds) at which a CE has occurred.

The CE was detected during a validation of the audit log by the notarization service, termed a *validation event* (*VE*). A validation can be scheduled (that is, is periodic) or could be an *ad hoc VE*. The time (instant) at which a *VE* occurred is termed the *time of validation event*, and is denoted by t_v. If validations are periodic, the time interval between two successive validation events is termed the *validation interval*, or I_V. Tampering is indicated by a *validation failure*, in which the validation service returns *false* for the particular query of a hash value and a notarization time. What is desired is a *validation success*, in which the notarization service returns *true*, stating that everything is OK: the data has not been tampered with.

The validator compares the hash value it computes over the data with the hash value that was previously notarized. A *notarization event* (*NE*) is the notarization of a document (specifically, a hash value) by the notarization service. As with validation, notarization can be scheduled (is periodic) or can be an *ad hoc notarization event*. Each *NE* has an associated *notarization time* (t_n), which is a time instant. If notarizations are periodic, the time interval between two successive notarization events is termed the *notarization interval*, or I_N.

There are several variables associated with each corruption event. The first is the data that has been corrupted, which we term the *corruption locus data* (l_c).

Forensic analysis involves *temporal detection*, the determination of the corruption time, t_c. Forensic analysis also involves *spatial detection*, the determination of where, that is, the location in the database of the data altered in a CE. (Note that the use of the adjective "spatial" does not refer to a spatial database, but rather where in the database the corruption occurred.)

Recall that each transaction is hashed. Therefore, in the absence of other information, such as a previous dump (copy) of the database, the best a forensic analysis can do is to identify the particular transaction that stored the data that was corrupted. Instead of trying to ascertain the corruption locus data, we will instead be concerned with the *locus time* (t_l), the time instant that locus data (l_c) was originally stored. The locus time specifically refers to the time instant when the transaction storing the locus data commits. (Note that here we are referring to the specific *version* of the data that was corrupted. This version might be the original version inserted by the transaction, or a subsequent version created through an update operation.) Hence the task of forensic analysis is to determine two times, t_c and t_l.

A CE can have many l_cs (and hence, many t_ls) associated with it, termed *multi-locus*: an intruder (hardware failure, etc.) might alter many tuples. A CE having only one l_c (such as due to an intruder hoping to remain undetected by making a single, very particular change) is termed a *single-locus CE*.

The finest spatial granularity of the corrupted data would be an explicit attribute of a tuple, or a particular timestamp attribute. However, this proves to be costly and hence we define R_s, which is the finest granularity chosen to express the uncertainty of the spatial bounds of a CE. R_s is called the *spatial detection resolution*. This is chosen by the DBA.

Similarly, the finest granularity chosen by the DBA to express the uncertainty of the temporal bounds of a CE is the *temporal detection resolution*, or R_t.

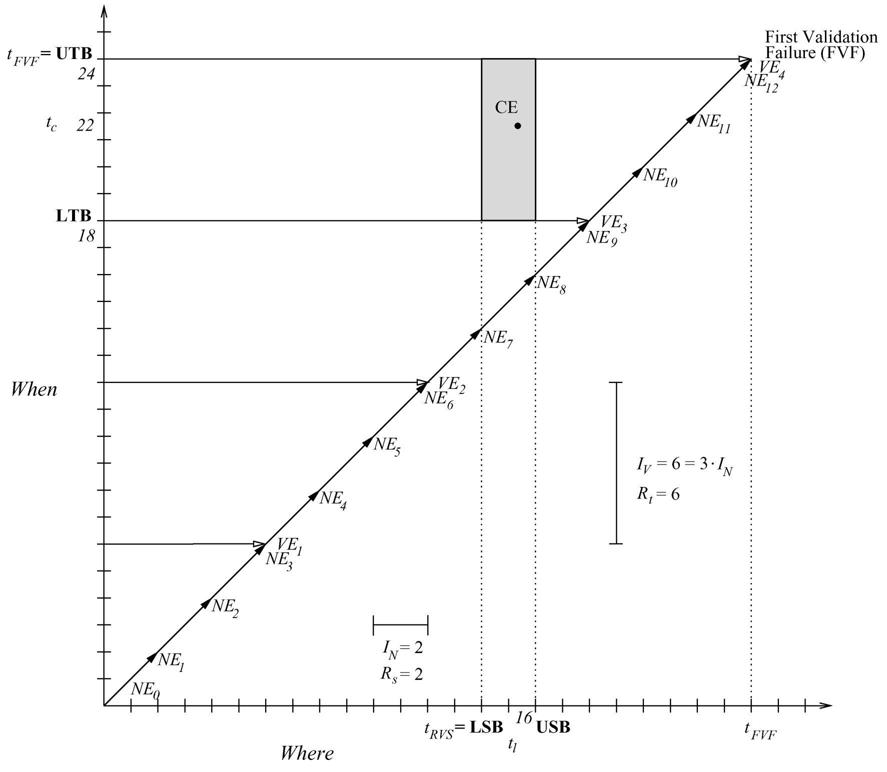


Fig. 2. Corruption diagram for a data-only single-locus retroactive corruption event.

4. THE CORRUPTION DIAGRAM

To explain forensic analysis, we introduce the *Corruption Diagram*, which is a graphical representation of CE(s) in terms of the temporal-spatial dimensions of a database. We have found these diagrams to be very helpful in understanding and communicating the many forensic algorithms we have considered, and so we will use them extensively in this article.

Definition. A *corruption diagram* is a plot in $\mathbb{R}^2$ having its ordinate associated with real time and its abscissa associated with a partition of the database according to transaction time. This diagram depicts corruption events and is annotated with hash chains and relevant notarization and validation events. At the end of forensic analysis, this diagram can be used to visualize the regions ($\subset \mathbb{R}^2$) where corruption has occurred.

Let us first consider the simplest case. During validation, we have detected a corruption event. Though we don't know it (yet), assume that this corruption event is a single-locus CE. Furthermore, assume that the CE just altered the data of a tuple; no timestamps were changed.

Figure 2 illustrates our simple corruption event. While this figure may appear to be complex, the reader will find that it succinctly captures all the important information regarding what is stored in the database, what is notarized, and what can be determined by the forensic analysis algorithm about the corruption event.

The x-axis represents when the data are stored in the database. The database was created at time 0, and is modified by transactions whose commit time is monotonically increasing along the x-axis. (In temporal database terminology [Jensen and Dyreson 1998], the x-axis represents the transaction time of the data.) In this diagram, time moves inexorably to the right.

This axis is labeled "*Where*." The database grows monotonically as tuples are appended (recall that the database is append-only). As above, we designate where a tuple or attribute is in the database by the time of the transaction that inserted that tuple or attribute. The unit of the x-axis is thus (transaction-commit) time. We delimit the days by marking each midnight, or, more accurately, the time of the last transaction to commit before midnight.

A 45-degree line is shown and is termed the *action line*, since all the action in the database occurs on this line. The line terminates at the point labeled "FVF," which is the validation event at which we first became aware of tampering. The *time of first validation failure* (or t_{FVF}) is the time at which the corruption is first detected. (Hence the name: a corruption diagram always terminates at the *VE* that detected the corruption event.) Note that t_{FVF} is an instance of a t_v, in that t_{FVF} is a specific instance of the time of a validation event, generically denoted by t_v. Also note that in every corruption diagram, t_{FVF} coincides with the current time. For example, in Figure 2 the *VE* associated with t_{FVF} occurs on the action line, at its terminus, and turns out to be the fourth such validation event, VE_4.

The actual corruption event is shown as a point labeled "CE," which always resides above or on the action line, and below the last *VE*. If we project this point onto the x-axis, we learn where (in terms of the locus of corruption, l_c) the corruption event occurred. Hence the x-axis, which being ostensibly commit time, can also be viewed as a spatial dimension, labeled in locus time instants (t_l). This is why we term the x-axis the *where axis*.

The y-axis represents the temporal dimension (actual time-line) of the database, labeled in time instants. Any point on the action line thus indicates a transaction committing at a particular transaction time (a coordinate on the x-axis) that happened at a clock time (the same coordinate on the y-axis). (In temporal database terminology, the y-axis is valid time, and the database is a *degenerate bitemporal database*, with valid time and transaction time totally correlated [Jensen and Snodgrass 1994]. For this reason, the action line is always a 45-degree line. Projecting the CE onto the y-axis tells us when in clock time the corruption occurred, that is, the corruption time, t_c. We label the y-axis with "*When*." The diagram shows that the corruption occurred on day 22 and corrupted an attribute of a tuple stored by a transaction that committed on day 16.

There is a series of points along the action line denoted with "*NE*." These (naturally) identify notarization events, when a hash value was sent to the notarization service. The first notarization event, NE_0, occurs at the origin, when the database was first created. This event hashes the tuples containing the database schema and notarizes that value.

Notarization event NE_1 hashes the transactions occurring during the first two days (here, the notarization interval, I_N, is two days), linking these hash values together using *linked hashing*. This is illustrated with the upward-right-pointing arrow with the solid black arrowhead originating at NE_0 (since the linking starts with the hash value notarized by NE_0) and terminating at NE_1. Each transaction at commit time is hashed; here, the where (transaction commit time) and when (wall-clock time) are synchronized; hence this occurs on the diagonal. The hash value of the transaction is linked to the previous transaction, generating a linked sequence of transactions that is associated with a hash value notarized at midnight of the second day in wall-clock time and covering all the transactions up to the last one committed before midnight (hence NE_1 resides on the action line). NE_1 sends the resulting hash value to the digital notarization service.

Similarly, NE_2 hashes two days' worth of transactions, links it with the previous hash value, and notarizes that value. Thus the value that NE_{12} (at the top right corner of Figure 2) notarizes is computed from all the transactions that committed over the previous 24 days.

In general, all notarization events (except NE_0) occur at the tip of a corresponding *black* hash chain, each starting at the origin and cumulatively hashing the tuples stored in the database between times 0 and that NE's t_n.

Also along the action line are points denoted with "*VE*." These are validation events for which a validation occurred. During VE_1, which occurs at midnight on the sixth day (here, the validation interval, I_V, is six days), rehashes all the data in the database in transaction commit order, denoted by the long right-pointing arrow with a white arrowhead, producing a linked hash value. It sends this value to the notarization service, which responds that this "document" is indeed the one that was previously notarized (by NE_3, using a value computed by linking together the values from NE_0, NE_1, NE_2, and NE_3, each over two days' worth of transactions), thus assuring us that no tampering has occurred in the first six days. (We know this from the diagram, because this *VE* is not at the terminus.) In fact, the diagram shows that VE_1, VE_2, and VE_3 were successful (each scanning a successively larger portion of the database, the portion that existed at the time of validation). The diagram also shows that VE_4, immediately after NE_{12}, failed, since it is marked as FVF; its time t_{FVF} is shown on both axes.

In summary, we now know that at each of the *VE*s up to but not including FVF succeeded. When the validator scanned the database as of that time (t_v for that *VE*), the hash value matched that notarized by the *VE*. Then, at the last *VE*, the FVF, the hash value didn't match. The corruption event, CE, occurred before midnight of the 24th day, and corrupted some data stored sometime during those twenty four days. (Note that as the database grows, more tuples must be hashed at each validation. Given that *any* previous hashed tuple could be corrupted, it is unavoidable to examine *every* tuple during validation.)

5. FORENSIC ANALYSIS

Once the corruption has been detected, a *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. For us to learn anything more, we have to go to other sources of information.

One such source is a backup copy of the database. We could compare, tuple-by-tuple, the backup with the current database to determine quite precisely the where (the locus) of the CE. That would also delimit the corruption time to after the locus time (one cannot corrupt data that has not yet been stored!). Then, from knowing where and very roughly when, the chief information officer (CIO) and chief security officer (CSO) and their staff can examine the actual data (before and after values) to determine who might have made that change.

However, it turns out that the forensic analyzer can use just the database itself to determine bounds on the corruption time and the locus time. The rest of this article will propose and evaluate the effectiveness of several forensic analysis algorithms.

In fact, we already have one such algorithm, the *trivial forensic analysis algorithm*: on validation failure, return the upper-left triangle, delimited by the when and action axes, denoting that the corruption event occurred before t_{FVF} and altered data stored before t_{FVF}.

Our next algorithm, termed the *Monochromatic Forensic Analysis Algorithm* for reasons that will soon become clear, yields the rectangular corruption region illustrated in the diagram, with an area of 12 days^2 (two days by six days). We provide the trivial and Monochromatic Algorithms as an expository structure to frame the more useful algorithms introduced later.

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. (The transaction timestamp stored in each tuple indicates when the tuple should be hashed; a separate tuple sequence number stored in the tuple during online processing indicates the order of hashing these tuples within a transaction.) If that *de novo* 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.

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

In light of these observations, we define four values:

—the *lower temporal bound*: $LTB := \max(t_{FVF} - I_V, t_{RVS})$,
—the *upper temporal bound*: $UTB := t_{FVF}$,
—the *lower spatial bound*: $LSB := t_{RVS}$, and
—the *upper spatial bound*: $USB := t_{RVS} + I_N$.

These define a corruption region, indicated in Figure 2 as a narrow rectangle, within which the CE must fall. This example shows that, when utilizing the Monochromatic Algorithm, the notarization interval, here $I_N = 2$ days, bounds the where, and the validation interval, here $I_V = 6$ days, bounds the when. Hence for this algorithm, $R_s = I_N$ and $R_t = I_V$. (More precisely,

$$R_t = UTB - LTB = \min(I_V, t_{FVF} - t_{RVS}),$$

due to the fact that R_t can be smaller than I_V for late-breaking corruption events, such as that illustrated in Figure 3.)

The CE just analyzed is termed a *retroactive corruption event*: a CE with locus time t_l appearing before the next to last validation event. Figure 3 illustrates an *introactive corruption event*: a CE with a locus time t_l appearing after the next to last validation event. In this figure, the corruption event occurred on day 22, as before, but altered data on day 21 (rather than day 16 in the previous diagram). NE_{10} is the most recent validation success. Here, the corruption region is a trapezoid in the corruption diagram, rather than a rectangle, due to the constraint mentioned earlier that a CE must be on or above the action line ($t_c \geq t_l$). This constraint is reflected in the definition of *LTB*.

It is worth mentioning here that these CEs are ones that only corrupt data. It is conceivable that a CE could alter the timestamp (transaction commit time) of a tuple. This creates two new independent types of CEs, termed *postdating* or *backdating* CEs, depending on how the timestamp was altered. An analysis of timestamp corruption will be provided in Section 7.

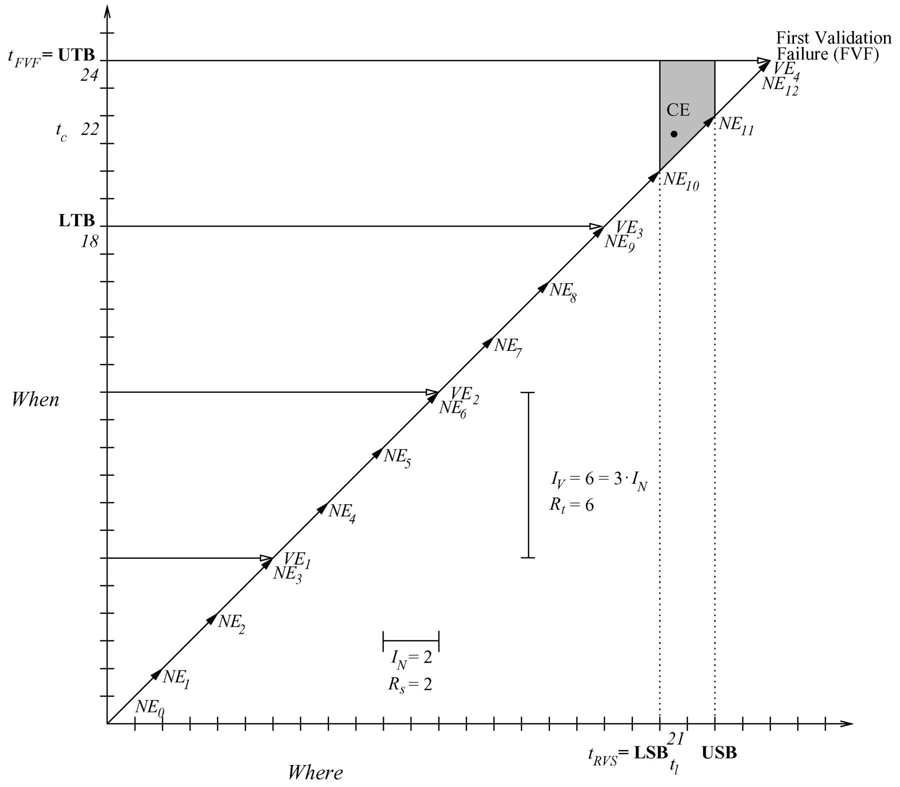


Fig. 3. Corruption diagram for a data-only single-locus introactive corruption event.

6. NOTARIZATION AND VALIDATION INTERVALS

The two corruption diagrams we have thus far examined assumed a notarization interval of $I_N = 2$ and validation interval of $I_V = 6$. In this case, notarization occurs more frequently than validation and the two processes are in phase, with I_V a multiple of I_N. In such a scenario, we saw that the spatial uncertainty is determined by the notarization interval, and the temporal uncertainty by the validation interval. Hence we obtained tall, thin CE regions. One naturally asks, what about other cases?

Say notarization events occur at midnight every two days, as before, and validation events occur every three days, but at noon. So we might have NE_1 on Monday night, NE_2 on Wednesday night, NE_3 on Friday night, VE_1 on Wednesday at noon, and VE_2 on Saturday at noon. VE_1 rehashes the database up to Monday night and checks that linked hash value with the digital notarization service. It would detect tampering prior to Monday night; tampering with a t_l after Monday would not be detected by VE_1. VE_2 would hash through Friday night; tampering on Tuesday would then be detected. Hence we see that a nonaligned validation just delays detection of tampering. Simply speaking, one can validate only what one has previously notarized.

If the validation interval were shorter than the notarization interval, that is $I_N = 2$, $I_V = 1$, say every day at midnight, then a validation on Tuesday at midnight could again only check through Monday night.

Our conclusion is that the validation interval should be equal to or longer than the notarization interval, should be a multiple of the notarization interval, and should be aligned, that is, validation should occur immediately after notarization. Thus we will speak of the *validation factor* V such that $I_V = V \cdot I_N$. As long as this constraint is respected, it is possible to change V, or both I_V and I_N, as desired. This, however, will affect the size of the corruption region and subsequently the cost of the forensic analysis algorithms, as emphasized in Section 9.

7. ANALYZING TIMESTAMP CORRUPTION

The previous section considered a *data-only* corruption event, a CE that does not change timestamps in the tuples. There are two other kinds of corruption events with respect to timestamp corruption. In a *backdating corruption event*, a timestamp is changed to indicate a previous time/date with respect to the original time in the tuple. We term the time a timestamp was backdated *to* the *backdating time*, or t_b. It is always the case that $t_b < t_l$. Similarly, a *postdating corruption event* changes a timestamp to indicate a future time/date with respect to the original commit time in the tuple, with the *postdating time* (t_p) being the time a timestamp was postdated to. It is always the case that $t_l < t_p$. Combined with the previously introduced distinction of retroactive and introactive, these considerations induce six specific corruption event types.

$$\left\{ \begin{array}{c} \textit{Retroactive} \\ \textit{Introactive} \end{array} \right\} \times \left\{ \begin{array}{c} \textit{Data-only} \\ \textit{Backdating} \\ \textit{Postdating} \end{array} \right\}$$

For backdating corruption events, we ask that the forensic analysis determine, to the extent possible, when (t_c), where (t_l), and to where (t_b). Similarly, for postdating corruption events, we want to determine t_c, t_l, and t_p. This is quite challenging given the only information we have, which is a single bit for each query on the notarization service.

It bears mention that neither postdating nor backdating CEs involve movement of the actual tuple to a new location on disk. Instead, these CEs consist entirely of changing an insertion-date timestamp attribute. (We note in passing that in some transaction-time storage organizations the tuples are stored in commit order. If an insertion date is changed during a corruption event, the fact that that tuple is out of order provides another clue, one that we don't exploit in the algorithms proposed here.)

Figure 4 illustrates a retroactive postdating corruption event (denoted by the forward-pointing arrow). On day 22, the timestamp of a tuple written on day 10 was changed to make it appear that that tuple was inserted on day 14 (perhaps to avoid seeming that something happened on day 10). This tampering will be detected by VE_4, which will set the lower and upper temporal bounds of the CE, shown in Figure 4 as $LTB = 18$ and $UTB = 24$. The Monochromatic

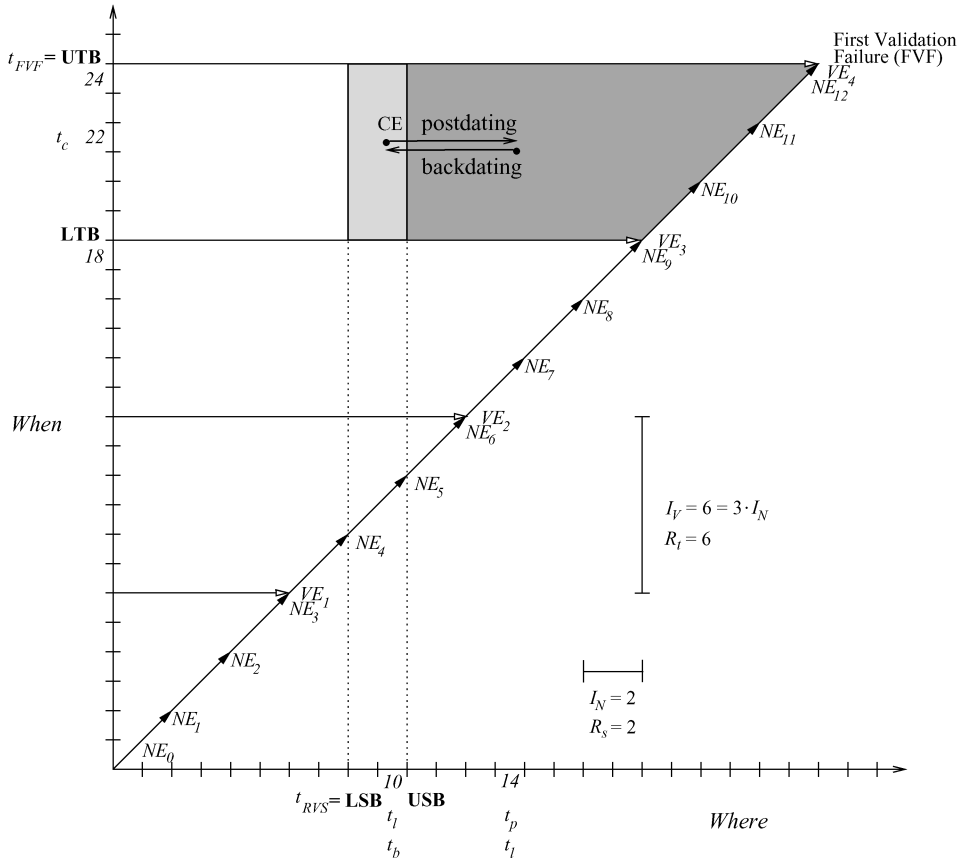


Fig. 4. Corruption diagram for postdating and backdating corruption events.

Algorithm will then go back and rehash the database, querying with the notarization service at NE_0, NE_1, NE_2, ... It will notice that NE_4 is the most recent validation success, because the rehashed sequence will not contain the tampered tuple: its (altered) timestamp implies it was stored on day 14. Given that the query at NE_4 succeeds and that at NE_5 fails, the tampered data must have been originally stored sometime during those two days, thus bounding t_l to day 9 or day 10. This provides the corruption region shown as the left-shaded rectangle in the figure.

Since this is a postdating corruption event, t_p, the date the data was altered to, must be after the local time, t_l. Unfortunately, all subsequent revalidations, from NE_5 onward, will fail, then giving us absolutely no additional information as to the value of t_p. The "to" time is thus somewhere in the shaded trapezoid to the right of the corruption region. (We show this on the corruption diagram as a two-dimensional region, representing the uncertainty of t_c and t_p. Hence the two shaded regions denote just three uncertainties, in t_c, t_l, and t_p.)

Figure 4 also illustrates a retroactive backdating corruption event (backward-pointing arrow). On day 22, the timestamp of a tuple written on day 14 was changed to make it appear that the tuple in question was inserted on day 10 (perhaps to imply something happened before it actually did). This tampering will be detected by VE_4, which will set the lower and upper temporal bounds of the CE (as in the postdating case). Going back and rehashing the data at $NE_0, NE_1, \ldots$ the Monochromatic Algorithm will compute that NE_4 is the most recent validation success. The rehashing up to NE_5 will fail to match its notarized value, because the rehashed sequence will erroneously contain the tampered tuple that was originally was stored on day 14. Given that the query at NE_4 succeeds and that at NE_5 fails, the new timestamp must be sometime within those two days, thus bounding t_b to day 9 or day 10. The left-shaded rectangle in the figure illustrates the extent of the imprecision of t_b.

Since this is a backdating corruption event, the date the data was originally stored, t_l, must be after the "to" time, t_b. As with postdating CEs, all subsequent revalidations, from NE_5 onward, will fail, then giving us absolutely no additional information as to the value of t_l. The corruption region is thus the shaded trapezoid in the figure.

While we have illustrated backdating and postdating corruption events separately, the Monochromatic Algorithm is unable to differentiate these two kinds of events from each other, or from a data-only corruption event. Rather, the algorithm identifies the RVS, the most recent validation success, and from that puts a two-day bound on *either* t_l or t_b. Because the black link chains that are notarized by *NE*s are cumulative, once one fails during a rehashing, all future ones will fail. Thus future *NE*s provide no additional information concerning the corruption event.

To determine more information about the corruption event, we have little choice but to utilize to a greater extent the external notarization service. (Recall that the notarization service is the only thing we can trust after an intrusion.) At the same time, it is important to not slow down regular processing. We'll show how both are possible.

8. FORENSIC ANALYSIS ALGORITHMS

In this section we provide a uniform presentation and detailed analysis of forensic analysis algorithms. The algorithms presented are the original Monochromatic Algorithm, the RGBY Algorithm, the Tiled Bitmap Algorithm [Pavlou and Snodgrass 2006b], and the a3D Algorithm. Each successive algorithm introduces additional chains during normal processing in order to achieve more detailed results during forensic analysis. This comes at the increased expense of maintaining—hashing and validating—a growing number of hash chains. We show in Section 9 that the increased benefit in each case more than compensates for the increased cost.

The Monochromatic Algorithm uses only the cumulative (black) hash chains we have seen so far, and as such it is the simplest algorithm in terms of implementation.

The RGBY Algorithm introduced here is an improvement of the original RGB Algorithm [Pavlou and Snodgrass 2006a]. The main insight of the previously

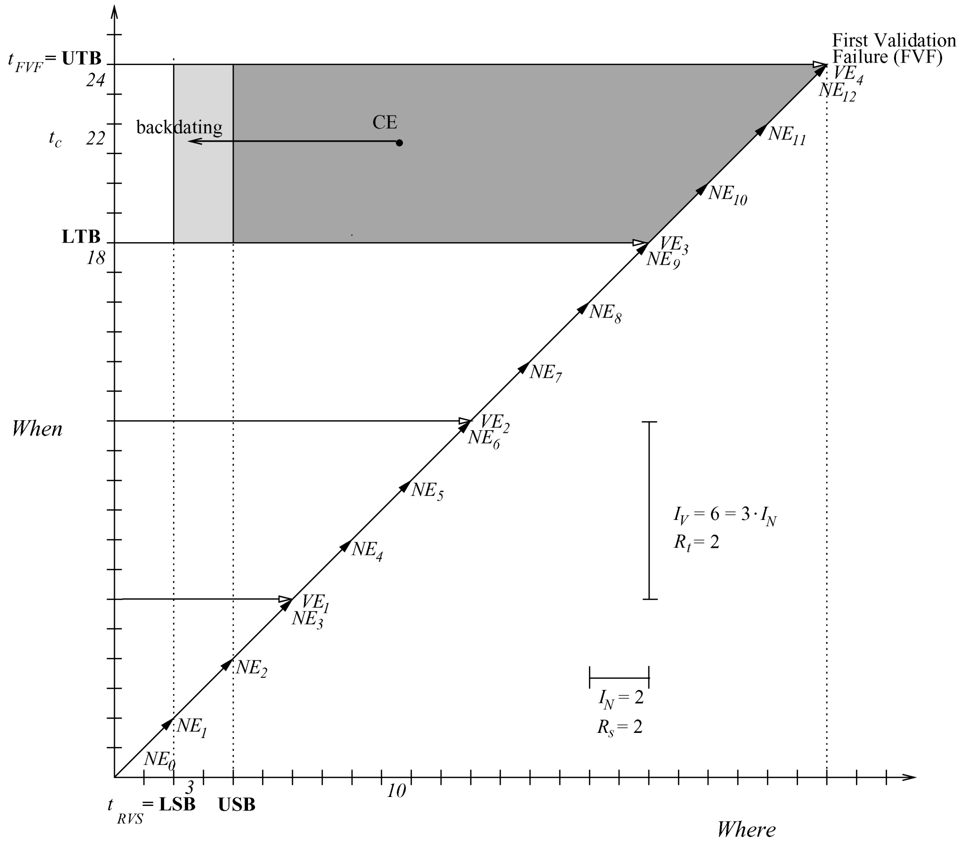


Fig. 5. Corruption diagram for a backdating corruption event.

presented *Red-Green-Blue forensic analysis algorithm* (or simply, the *RGB Algorithm*) is that during notarization events, in addition to reconstructing the entire hash chain (illustrated with the long right-pointed arrows in prior corruption diagrams), the validator can also rehash *portions* of the database and notarize those values, separately from the full chain. In the RGB Algorithm, three new types chains are added, denoted with the colors red, green, and blue, to the original (black) chain in the so-called Monochromatic Algorithm. These hash chains can be computed in parallel; all consist of linked sequences of hash values of individual transactions in commit order. While additional hash values must be computed, no additional disk reads are required. The additional processing is entirely in main memory. The RGBY Algorithm retains the red, green, and blue chains and adds a yellow chain. This renders the new algorithm more regular and more powerful.

The Tiled Bitmap Algorithm extends the idea of the RGBY Algorithm of using partial chains. It lays down a regular pattern (a *tile*) of such chains over contiguous segments of the database. What is more, the chains in the tile form a bitmap that can be used for easy identification of the corruption region [Pavlou and Snodgrass 2006b].

```
// input:     $t_{FVF}$ is the time of first validation failure
//            $I_N$ is the notarization interval
//            $V$ is the validation factor
// output:    types of and bounds on CE
procedure Monochromatic($t_{FVF}$, $I_N$, $V$):
1:    $I_V \leftarrow V \cdot I_N$
2:    $t_{RVS} \leftarrow$ find_$t_{RVS}(t_{FVF}, I_N)$
3:    $USB \leftarrow t_{RVS} + I_N$
4:    $LSB \leftarrow t_{RVS}$
5:    $UTB \leftarrow t_{FVF}$
6:    $LTB \leftarrow \max(t_{FVF} - I_V, t_{RVS})$
7:    if $t_{RVS} \geq (t_{FVF} - I_V)$ then report Introactive CE
8:    else if $t_{RVS} < (t_{FVF} - I_V)$ then report Retroactive CE
9:    report Data-only CE, $LSB < t_l \leq USB$, $LTB < t_c \leq UTB$
10:   report Postdating CE, $LSB < t_l \leq USB$, $LTB < c \leq UTB$, $USB < t_p \leq t_{FVF}$
11:   report Backdating CE, $LSB < t_b \leq USB$, $LTB < t_c \leq UTB$, $USB < t_l \leq t_{FVF}$

// input:     $t_{FVF}$ is the time of first validation failure
//            $I_N$ is the notarization interval
// output:    Schema Corruption if it exists
//            $t_{RVS}$ is the time of most recent validation success
procedure find_$t_{RVS}(t_{FVF}, I_N)$:
1:    $left \leftarrow 1$
2:    $right \leftarrow t_{FVF}$
3:    $t_{RVS} \leftarrow \lfloor (left + right)/2 \rfloor$
      // since $t_{RVS}$ may not coincide with a $NE$
4:    if $(t_{RVS} \bmod I_N) \neq 0$ then $t_{RVS} \leftarrow t_{RVS} - (t_{RVS} \bmod I_N)$
5:    while $(\neg\, BlackChains[\max(1 + (t_{RVS}/I_N), 0)] \vee BlackChains[t_{RVS}/I_N])$
            $\wedge\ (right \geq left)$ do
6:        if $\neg\, BlackChains[t_{RVS}/I_N]$ then
7:            if $t_{RVS} = 0$ then
8:                report "Schema Corruption: cannot proceed..."
9:                exit
10:           if $t_{RVS} - I_N < 0$ then $right \leftarrow 0$ else $right \leftarrow t_{RVS} - I_N$
11:       else
12:           if $t_{RVS} + I_N > t_{FVF}$ then $left \leftarrow t_{FVF}$ else $left \leftarrow t_{RVS} + I_N$
13:       $t_{RVS} \leftarrow \lfloor (left + right)/2 \rfloor$
14:       if $(t_{RVS} \bmod I_N) \neq 0$ then $t_{RVS} \leftarrow t_{RVS} - (t_{RVS} \bmod I_N)$
15:   return $t_{RVS}$
```

Fig. 6. The Monochromatic Algorithm.

The a3D Algorithm introduced here is the most advanced algorithm in the sense that it does not lay repeatedly a "fixed" pattern of hash chains over the database. Instead, the lengths of the partial hash chains change (decrease or increase) as the transaction time increases, in such as way that at each point in time a complete binary tree (or forest) of hash chains exists on top of the database. This enables forensic analysis to be sped up significantly.

8.1 The Monochromatic Algorithm

We provide the pseudocode for the Monochromatic Algorithm in Figure 6. This algorithm takes three input parameters, as indicated next. t_{FVF} is the *time of* *first validation failure*, that is, the time at which the corruption of the log is first detected. In every corruption diagram, t_{FVF} coincides with the current time. I_N is the notarization interval, while V, called the validation factor, is the ratio of the validation interval to the notarization interval ($V = I_V/I_N$, $V \in \mathbb{N}$). The algorithm assumes that a single CE transpires in each example. The resolutions for the Monochromatic Algorithm are $R_s = I_N$ and $R_t = I_V = V \cdot I_N$. (The DBA can set the resolutions indirectly, by specifying I_N and V.) Hence if a CE involving a timestamp transpires and t_l and t_p/t_b are both within the same I_N, such a (backdating or postdating) corruption cannot be distinguished from a data-only CE, and hence it is treated as such.

The algorithm first identifies t_{RVS}, the time of most *recent validation success*, and from that puts an I_N bound on *either* t_l or t_b. Then depending on the value of t_{RVS} it distinguishes between introactive and retroactive CEs. It then reports the (where) bounds on t_l and t_p (or t_b) of both data-only and timestamp CEs, since it cannot differentiate between the two. These bounds are given in terms of the upper spatial bound (*USB*) and the lower spatial bound (*LSB*). The time interval where time of corruption t_c lies is bounded by the lower and upper temporal bounds (*LTB* and *UTB*).

It is worth noting here that the points $(t_l,\ t_c)$ and $(t_p,\ t_c)$—or $(t_b,\ t_c)$—must always share the same *when*-coordinate, since both refer to a single CE. The algorithm reports multiple possibilities for the CEs, since the algorithm can't differentiate between all the different types of corruption. Also, the bounds are given in a way that is readable and quite simple. The results are captured by a system of linear inequalities whose solution conveys the extent of the corruption region.

The find_t_{RVS} function, which is used on line 2 in the Monochromatic procedure of Figure 6, finds the time of most recent validation success by performing binary search on the cumulative black chains. It revisits past notarizations and, by validating them, it decides whether to recurse to the right or to the left of the current chain.

In this algorithm we use an array *BlackChains* of Boolean values to store the results of validation during forensic analysis. The Boolean results are indexed by the subscript of the notarization event considered: the result of validating NE_i is stored at index i, that is, *BlackChains*$[i]$. Since we do not wish to precompute all this information, the validation results are computed lazily, that is, whenever needed. On line 7 we report only if there is schema corruption and no other special checks are made in order to deal with this special case of corruption.

Note that on lines 6 and 11 these are the only possibilities for the validation results of the *NE*s in question. No other case ever arises, since the results of the validations of the cumulative black chains, considered from right to left, always follow a (single) change from false to true.

The running time of the Monochromatic Algorithm is dominated by the simple binary search required to find t_{RVS}. It ultimately depends on the number of cumulative black hash chains maintained. Hence the running time of the Monochromatic Algorithm is $O(\lg(t_{FVF}/I_N))$.

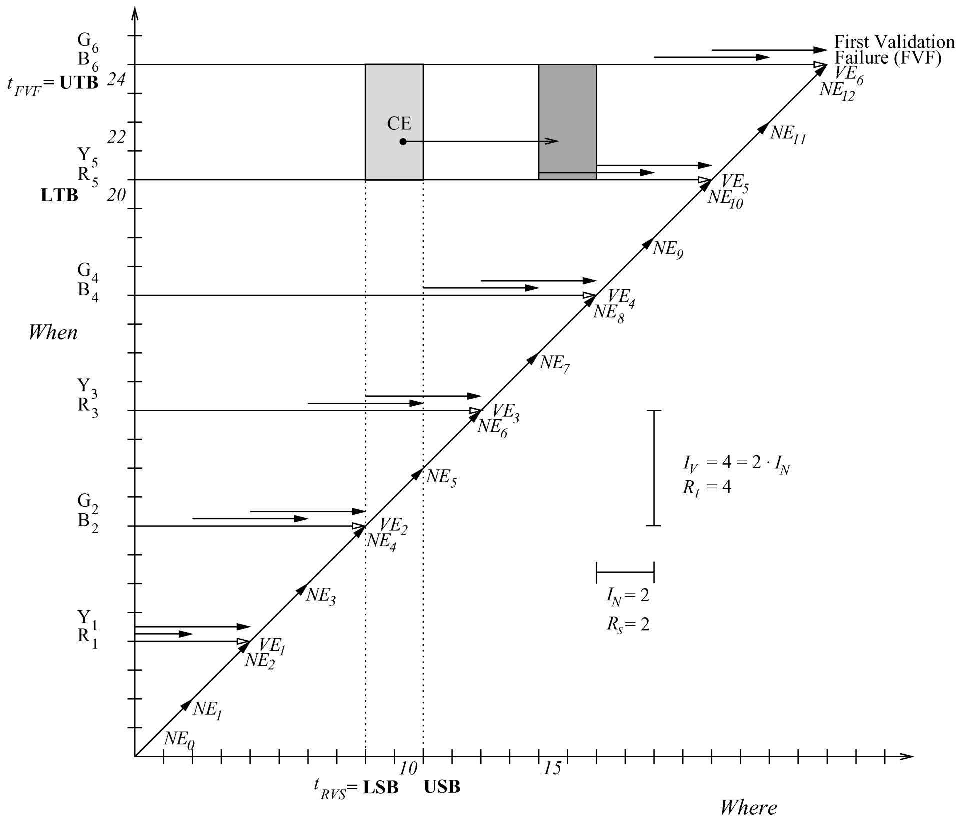


Fig. 7. Corruption diagram for the RGBY Algorithm.

8.2 The RGBY Algorithm

We now present an improved version of the RGB Algorithm that we call the *RGBY Algorithm*. RGBY has a more regular structure and avoids some of RGB's ambiguities. The RGBY chains are of the same types as in the original RGB Algorithm. The black cumulative chains are used in conjunction with new *partial hash chains*, that is, chains that do not extend all the way back to the origin of the corruption diagram. Another difference is that these partial chains are evaluated and notarized during a validation scan of the entire database and, for this reason, they are shown running parallel to the *Where* axis (instead of being on the action axis) in Figure 7. The introduction of the partial hash chains will help us deal with more complex scenarios, for example, multiple data-only CEs or CEs involving timestamp corruption.

The partial hash chains in RGB are computed as follows. (We assume throughout that the validation factor $V = 2$ and I_N is a power of two.)

— for odd i the *Red* chain covers $NE_{2\cdot i-3}$ through $NE_{2\cdot i-1}$.
— for even i the *Blue* chain covers $NE_{2\cdot i-3}$ through $NE_{2\cdot i-1}$.
— for even i the *Green* chain covers $NE_{2\cdot i-2}$ through $NE_{2\cdot i}$.

In this new algorithm we simply introduce a new *Yellow* chain, computed as follows:

—for odd i the *Yellow* chain covers $NE_{2 \cdot i-2}$ through $NE_{2 \cdot i}$.

In Figure 7 the colors of the partial hash chains are denoted along the *When* axis with the labels **R***ed*, **G***reen*, **B***lue*, and **Y***ellow* (the figure is still in black and white). We use subscripts to differentiate between chains of the same color in the corruption diagram. Each chain takes its subscript from the corresponding *VE*. In the pseudocode we use instead a two-dimensional array called *Chain*. It is indexed as $Chain[color, number]$, where *number* refers to the subscript of the chain, while *color* is an integer between 0 and 3 with the following meaning:

—if $color = 0$ then *Chain* refers to a *Blue* chain.

—if $color = 1$ then *Chain* refers to a *Green* chain.

—if $color = 2$ then *Chain* refers to a *Red* chain.

—if $color = 3$ then *Chain* refers to a *Yellow* chain.

We also introduce the following comparisons:

$$Chain[color_1, number_1] \prec Chain[color_2, number_2] \text{ iff}$$
$$(number_1 < number_2) \vee (number_1 = number_2 \wedge color_1 < color_2),$$
$$Chain[color_1, number_1] = Chain[color_2, number_2] \text{ iff}$$
$$(number_1 = number_2 \wedge color_1 = color_2).$$

The algorithm requires that $V = 2$. This is because the chains are divided into two groups: red/yellow, added at odd-numbered validation events, and blue/green, added at even-numbered validation events. Note that the find_t_{RVS} routine from the Monochromatic Algorithm is used here. As with the Monochromatic Algorithm, the spatial detection resolution is equal to the validation interval ($R_s = I_V$) and the temporal detection resolution is equal to the notarization interval ($R_t = I_N$).

In this algorithm (shown in Figure 8), as well as in all subsequent ones, instead of using an array *BlackChains* to store the Boolean values of the validation results, as that used in find_t_{RVS}, we use a helper function called val_check. This function takes a hash chain as a parameter and returns the Boolean result of the validation of that chain.

During the normal processing, the cumulative black hash chains are evaluated and notarized. During a *VE*, the entire database is scanned and validated, while the partial (colored) hash chains are evaluated and notarized.

On line 2 we initialize a set that accumulates all the corrupted granules (in this case, days). Line 3 computes t_{RVS} and lines 4–7 set the temporal and spatial bounds of the oldest corruption. On lines 9–10, we compute what is the most recent partial chain (*lastChain*), while on lines 11–13 we compute the rightmost chain covering the oldest corruption (*currChain*). In Figure 7 the oldest corruption is in the I_N covering days 9 and 10, so *currChain* is $Yellow_3$. The "while" loop on line 14 linearly scans all the partial chains to the right of t_{RVS}, that is, from *currChain* to *lastChain*, and checks for the pattern ...TFFT... in order to identify the corrupted granules. To achieve this, the algorithm must check the validation result of *chainChain* and its immediate successor. Lines

```
// input:     $t_{FVF}$ is the time of first validation failure
//            $I_N$ is the notarization interval
// output:    $C_{set}$ is the set of corrupted granules
//            $UTB$, $LTB$ are the temporal bounds on $t_c$
procedure RGBY($t_{FVF}$, $I_N$):
1:  $I_V \leftarrow 2 \cdot I_N$        // $V = 2$
2:  $C_{set} \leftarrow \emptyset$
3:  $t_{RVS} \leftarrow$ find_$t_{RVS}(t_{FVF}, I_N)$
4:  $USB \leftarrow t_{RVS} + I_N$
5:  $LSB \leftarrow t_{RVS}$
6:  $UTB \leftarrow t_{FVF}$
7:  $LTB \leftarrow \max(t_{FVF} - I_V, t_{RVS})$
8:  $C_{set} \leftarrow C_{set} \cup \{t_{RVS} + 1\}$
9:  $v \leftarrow (t_{FVF}/I_V)$
10: $lastChain \leftarrow Chain[1 + v \bmod 2, v]$
11: $n \leftarrow (LSB/I_N)$
12: $s \leftarrow \lceil (n/2.0) \rceil + 1$
13: $currChain \leftarrow Chain[(n+3) \bmod 4, s]$
14: while $currChain \preceq lastChain$ do
15:     if $(currChain.color = \text{Green}) \vee (currChain.color = \text{Yellow})$ then
16:         $succChain.number \leftarrow currChain.number + 1$
17:     else $succChain.number \leftarrow currChain.number$
18:     $succChain.color \leftarrow (currChain.color + 1) \bmod 4$
19:     if $\neg$ val_check($currChain$) then
20:         if $\neg$ val_check($succChain$) then
21:             if $currChain.color = \text{Blue} \vee currChain.color = \text{Red}$ then
22:                 $C_{set} \leftarrow C_{set} \cup \{2 \cdot (currChain.number - 1) \cdot I_N + 1\}$
23:             else $C_{set} \leftarrow C_{set} \cup \{2 \cdot currChain.number \cdot I_N - I_N + 1\}$
24:     $currChain \leftarrow succChain$
25: return $C_{set}$, $LTB < t_c \leq UTB$
```

Fig. 8. The RGBY Algorithm.

15–18 compute this successor denoted by *succChain*. If both the validation of *currChain* and *succChain* return false, then we have located a corruption, and the appropriate granule is added to C_{set} (lines 21–23).

The RGBY Algorithm was designed so that it attempts to find more than one CE. However, the main disadvantage of the algorithm is that it cannot distinguish between three contiguous corruptions and two corruptions with an intervening I_N between them. In both cases, the pattern of truth values of the validated partial chains is ...TFFFFT .... Hence in the latter case, the algorithm will report all three $I_V \times I_N$ rectangles as corrupted. This is not desirable because it introduces a false positive result. (Appendix B explains this in more detail.)

The running time of the RGBY Algorithm is $O(\lg(t_{FVF}/I_N) + (t_{FVF}/I_V)) = O(t_{FVF}/I_V)$. The $\lg(t_{FVF}/I_N)$ term arises from invoking find_t_{RVS}. The (t_{FVF}/I_V) term is due to the linear scan of all the colored partial chains, which in the worst case would be twice the number of *VE*s.

8.3 The Tiled Bitmap Algorithm

Appendix C presents an improved version of the Polychromatic Algorithm [Pavlou and Snodgrass 2006a] called the *Tiled Bitmap Algorithm*. The

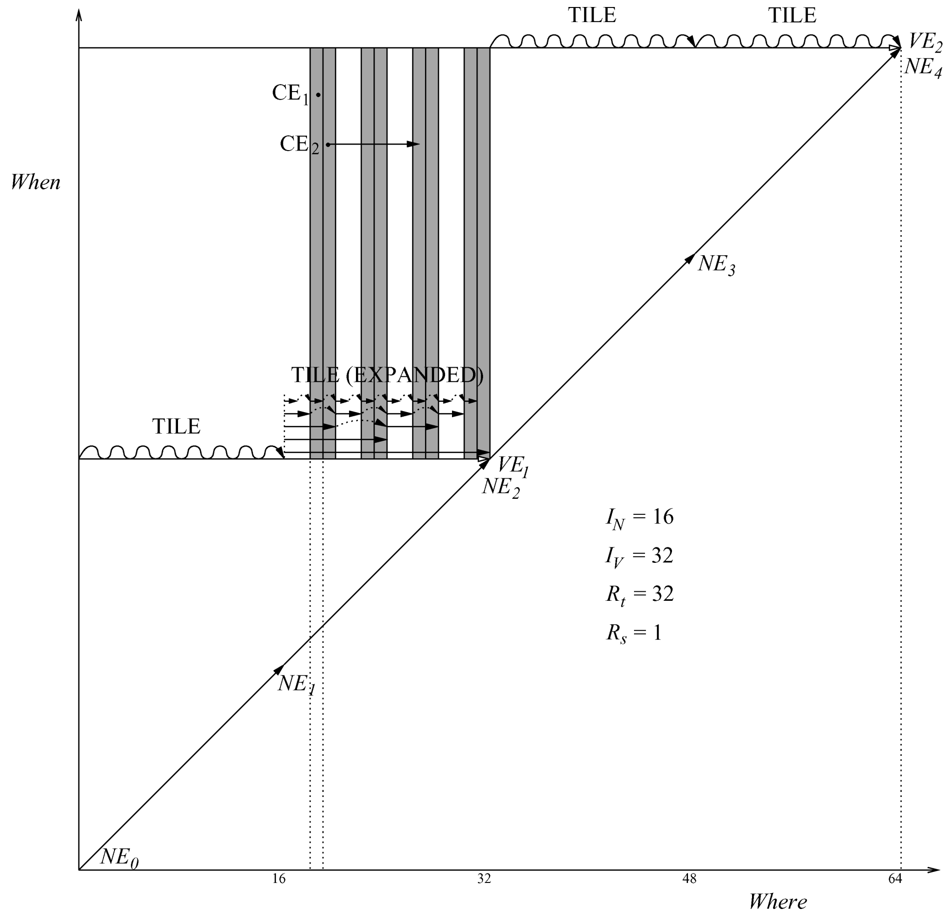


Fig. 9. Corruption diagram for the Tiled Bitmap Algorithm.

original Polychromatic Algorithm utilized multiple *Red* and *Blue* chains while retaining the *Green* chain from the RGB Algorithm. These two kinds of chains and their asymmetry complicated this algorithm. The Tiled Bitmap Algorithm relocates these chains to be more symmetric, resulting in a simpler pattern.

The algorithm also uses a logarithmic number of chains for each tile of duration I_N. The spatial resolution in this case can thus be arbitrarily shrunk with the addition of a logarithmic number of chains in the group. The result is that, for this algorithm, and not for the previous two, R_s can be less than I_N. More specifically, the number of chains that constitute a tile is $1 + \lg(I_N/R_s)$. We denote the ratio I_N/R_s by N, the *notarization factor*. We require N to be a power of 2. (NB: In the previous two algorithms $N = 1$.) This implies that for all the algorithms, $I_N = N \cdot R_s$ and $R_t = V \cdot I_N = V \cdot N \cdot R_s$. Also, because of the fact that R_s can vary, we define D to be the number of R_s units in the time interval from the start until t_{FVF}, that is, $D = t_{FVF}/R_s$.

As an example, in Figure 9, $R_s = 1$, $I_N = N = 2^4 = 16$, $V = 2$, $R_t = 32$, and $D = 64$. If we wanted an R_s of, say, 90 minutes (1/16 day), we would need another four chains: $1 + \lg(I_N/R_s) = 1 + \lg(16/\frac{1}{16}) = 9$. (Appendix C explains this figure in much more detail.)

In all of the algorithms presented thus far, discovering corruption (CEs or postdating intervals) to the right of t_{RVS} is achieved using a linear search that visits potentially all the hash chains in this particular interval. Due to the nature of these algorithms, this linear search is unavoidable. The Tiled Bitmap algorithm reduces the size of the linear search by just iterating on the longest partial chains ($c(0)$) that cover each tile. The running time of the Tiled Bitmap Algorithm is shown in Appendix C to be $O(D)$.

In addition, the Tiled Bitmap Algorithm may handle multiple CEs but it potentially overestimates the degree of corruption by returning the candidate set with granules which may or may not have suffered corruption (false positives). The number of false positives in the Tiled Bitmap Algorithm could be significantly higher than the number of false positives observed in the RGBY Algorithm. Figure 9 shows that the Tiled Bitmap Algorithm will produce a candidate set with the following granules (in this case, days): 19, 20, 23, 24, 27, 28, 31, 32. The corruptions occur on granules 19, 20, and 27, while the rest are false positives. In order to overcome these limitations, we introduce the next algorithm.

8.4 The a3D Algorithm

We have seen that the existence of multi-locus CEs can be better handled by summarizing the sites of corruption via candidate sets, instead of trying to find their precise nature. We proceed now to develop a new algorithm that avoids the limitations of all the previous algorithms and at the same time handles the existence of multi-locus CEs successfully. We call this new algorithm the a3D Algorithm for reasons that will become obvious when we analyze it. The a3D Algorithm is illustrated in Figure 10. 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 the Tiled Bitmap Algorithm and a3D is that, in the latter, each chain is contiguous, that is, it has no gaps. It was the gaps that necessitated the introduction of the candidate sets. Figure 10 shows that the corruption regions in the a3D Algorithm each correspond to a single corruption. All existing corruptions at granules 4, 7, and 10 are identified with no false positives. The difference between a3D and the other algorithms is a slowly increasing number of chains at each validation. In Figure 10, the chains are named using letters B for the *black* cumulative chains, and P for the partial chains. Observe that there is one diagonal full chain at VE_1, and two partial chains. VE_2 has a full black chain (B_2, with the subscript the day—R_s unit—of the validation event), retains the chains ($P_{2,0,2}$ and $P_{2,0,3}$), and adds a longer partial chain ($P_{2,1,1}$). (We will explain these three subscripts shortly.) We add another chain at VE_4 ($P_{4,2,1}$) and another chain at VE_8 ($P_{8,3,1}$).

The a3D Algorithm assumes that, given an R_s, $t_{FVF} \neq 0$, $D = t_{FVF}/R_s$, and $V = 1$ (which implies that $R_t = I_N$).

The beauty of this algorithm is that it decides what chains to add based on the current day/R_s unit. In this way, the number of chains increases dynamically,

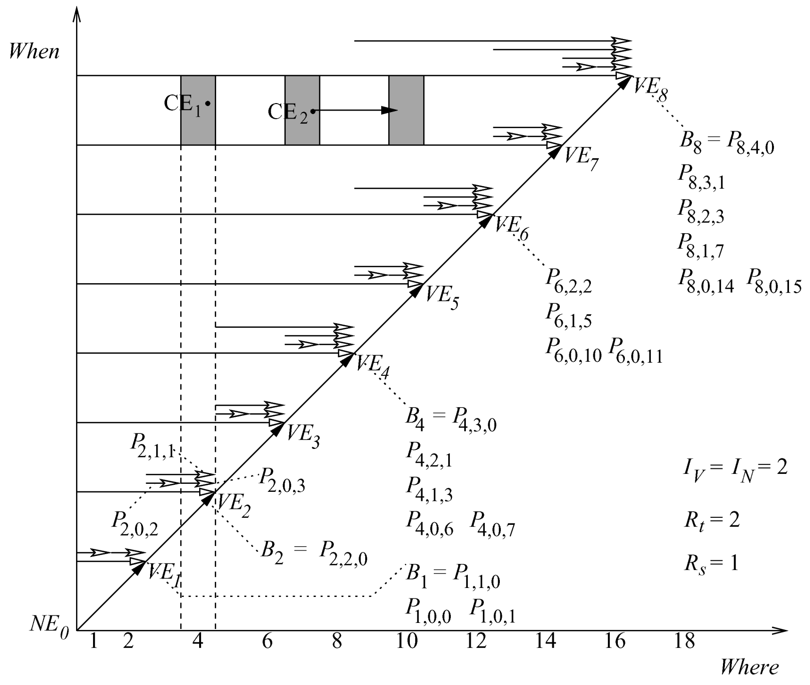


Fig. 10. Corruption diagram for the a3D Algorithm.

which allows us to perform binary search in order to locate the corruption. If we dissociate the decision of how many chains to add from the current day, then we are forced to repeat a certain fixed pattern of hash chains, which results in the drawbacks seen in the Tiled Bitmap Algorithm.

During normal processing, the algorithm adds partial hash chains (shown with white-tipped arrows). These partial chains are labeled as P with three subscripts. The first subscript is the number m of the current *VE*, such as $P_{4,2,1}$ added at VE_4. The second subscript, *level*, identifies the (zero-based) vertical position of the chain P within a group of chains added at VE_m. This subscript also provides the length of the partial chain as 2^{level}. For example, chain $P_{4,2,1}$ has length $2^2 = 4$. The final subscript, *comp* (for component), determines the horizontal position of the chain: all chains within a certain *level* have a position *comp* that ranges from 0 to $2^{level} - 1$. For example, hash chain $P_{4,2,1}$ is the second chain at level 2. The first chain at level 2 is $P_{2,2,0}$, which just happens to be the black chain B_2; the third chain at this level is $P_{6,2,2}$; and the fourth chain is $P_{8,2,3}$.

The addition of partial hash chains allows the algorithm to perform a bottom-up creation of a *binary tree* whose nodes represent the hash chains (see Figure 11). Depending on when the CE transpires, there maybe nodes missing from the complete tree, so in reality we have multiple binary trees that are subtrees of the next complete tree. In Figure 11, the nodes/chains missing are those in the shaded region, while there are three complete subtrees each rooted at $B_4 = P_{4,3,0}$, $P_{6,2,2}$, and $P_{7,1,6}$, respectively.

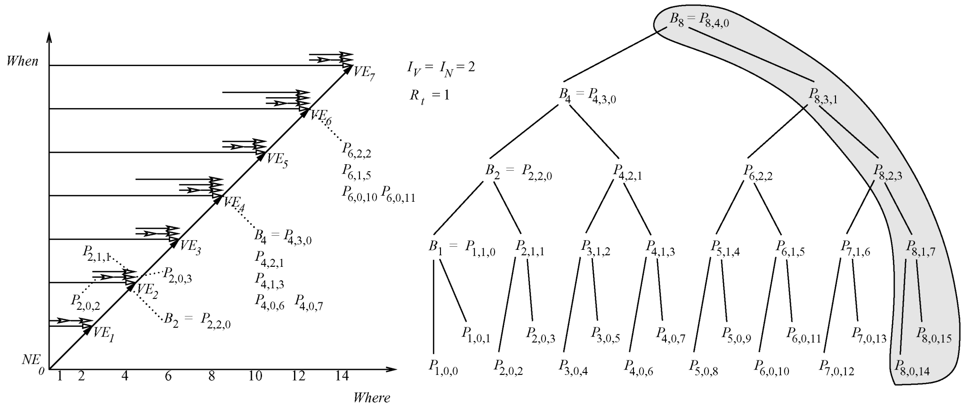


Fig. 11. The a3D Algorithm performs a bottom-up creation of a binary tree.

The a3D Algorithm is given in Figure 12. Note that when val_check is called with a hash chain $P[m, \mathit{level}, \mathit{comp}]$ for whom m is a power of 2, $\mathit{level} \geq \lg(N)$, and $\mathit{comp} = 0$, these chains are actually black chains whose validation result can be obtained through $\mathit{BlackChains}[m]$. All black chains appear only on the leftmost path from the root to the leftmost child; however, not all chains on this path are black.

The a3D function evaluates the *height* of the complete tree, regardless of whether we have a single tree or a forest (line 5). Then it calls the recursive a3D_helper function, which performs the actual search. In the recursive part of a3D_helper, the function calls itself (lines 8–9, 11–12) with the appropriate hash (sub-)chain only if the current chain does not exist or evaluates to false (line 6). In this case, we are relying on short-circuit Boolean evaluation for correctness. All of the compromised granules are accumulated into C_{set}.

The running time of the algorithm is dominated by the successive calls to the recursive function a3D_helper. The worst-case running time is captured by the recursion $T(D) = 2 \cdot T(D/2) + O(1)$, that is, we have to recurse to both the left and right children. The solution to this recursion gives us $T(D) = \Theta(D)$, so the algorithm is linear in the number of R_s units. In the best case, the algorithm recurses on only one of the two children, and thus the running time is $O(\lg D)$.

The algorithm takes its name from the fact that, for a given D, the algorithm makes in the worst case $3 \cdot D$ number of notarization contacts, as in the following:

$$\begin{aligned} \mathit{Total\ Number\ of\ Notarizations} &= \mathit{number\ of\ chains\ in\ tree} \\ &\quad + \mathit{number\ of\ black\ chains\ not\ in\ tree} \\ &= \mathcal{N}(D) \\ &\quad + D/N - (1 + \lfloor \lg(D/N) \rfloor) \end{aligned} \tag{1}$$

```
// input:     $t_{FVF}$ is the time of first validation failure
//            $I_N$ is the notarization interval
//            $R_s$ spatial detection resolution
// output:    $C_{set}$ is the set of corrupted granules
//            $UTB$, $LTB$ are the temporal bounds on $t_c$
procedure a3D($t_{FVF}$, $I_N$, $R_s$):
1:   $C_{set} \leftarrow \emptyset$
2:   $D \leftarrow t_{FVF}/R_s$
3:   $N \leftarrow I_N/R_s$
4:   $m\_max \leftarrow 2^{\lceil \lg(D/N) \rceil}$
5:   $height \leftarrow \lg N + \lg(m\_max)$
6:   $C_{set} \leftarrow$ a3D_helper($P[m\_max, height, 0]$, $C_{set}$, $N$)
7:   $min \leftarrow C_{set}[0]$
8:   if $min < t_{FVF} - I_N$ then $LTB \leftarrow t_{FVF} - I_N$
9:   else $LTB \leftarrow min$
10:  $UTB \leftarrow t_{FVF}$
11:  return $C_{set}, LTB < t_c \leq UTB$
```

```
// input:     $P[m$, level, comp$]$ is a hash chain that was evaluated on $VE_m$
//              and whose length depends on $level$
//            $N$ is the notarization factor
//            $C_{set}$ an empty set in which the corrupted granules will be accumulated
// output:    $C_{set}$ is the set of corrupted granules
procedure a3D_helper($P[m, level, comp]$, $C_{set}$, $N$):
1:   if $level = 0$ then
2:     if exists($P[m, level, comp]$) then
3:       if $\neg$ val_check($P[m, level, comp]$) then $C_{set} \leftarrow C_{set} \cup \{comp\}$
4:     return $C_{set}$
5:   else
6:     if $\neg$ exists($P[m, level, comp]$) $\vee$ $\neg$ val_check($P[m, level, comp]$) then
7:       if $\neg$ ($level \leq \lg N$) then
8:           return a3D_helper ($P[\frac{1}{2} \cdot (m + m - (2^{level}/N)), level - 1, 2 \cdot comp]$, $C_{set}$, $N$)
9:           return a3D_helper ($P[m, level - 1, 2 \cdot comp + 1]$, $C_{set}$, $N$)
10:      else
11:          return a3D_helper ($P[m, level - 1, 2 \cdot comp]$, $C_{set}$, $N$)
12:          return a3D_helper ($P[m, level - 1, 2 \cdot comp + 1]$, $C_{set}$, $N$)
```

Fig. 12. The a3D Algorithm.

where

$$\mathcal{N}(D) = \begin{cases} 0, & D = 0 & (i) \\ 2^{i+1} - 1 = 2 \cdot D - 1, & D = 2^i, i \in \mathbb{N}, D > 0 & (ii) \\ \mathcal{N}(2^{\lfloor \lg D \rfloor}) + \mathcal{N}(D - 2^{\lfloor \lg D \rfloor}), & D \bmod 2 = 0 \wedge D \neq 2^i, D > 0 & (iii) \\ \mathcal{N}(D-1), & D \bmod 2 = 1 & (iv) \end{cases}$$

$\mathcal{N}$ is the number of hash chains, which is the same as the number of nodes in the complete binary tree or the forest.

Case (iv) of this recursion shows that, for odd D, $\mathcal{N}(D)$ is always equal to the number of hash chains of the previous even D. For this reason, we only need consider the case when D is even. What case (iii) essentially does at each stage of the recursion is to decompose D into a sum of powers of 2; each such power under the action of $\mathcal{N}$ yields $2^{i+1} - 1$ notarizations. (This is also the number of nodes in the subtree of height i.) Thus to evaluate this recurrence we examine the binary representation of D. Each position in the binary representation where there is a '1' corresponds to a power of 2 with decimal value 2^i. Summing the results of each one of these decimal values under the action of $\mathcal{N}$ gives the desired solution to $\mathcal{N}(D)$. This solution can be captured mathematically using Iverson brackets [Graham et al. 2004, p. 24] (here, & is a bit-wise *AND* operation):

$$\mathcal{N}(D) = \sum_{i=0}^{\lfloor \lg D \rfloor} (2^{i+1} - 1) \cdot \ [D \& 2^i \neq 0].$$

The total number of notarizations is bounded above by the number $3 \cdot D$. This loose bound can be derived by simply assuming that the initial value of D is a power of 2. Assuming also that the complete binary tree has height $H = \lg D$, then

$$\begin{aligned} \textit{Total Number of Notarizations} &\leq 2 \cdot D - 1 + D/N - (1 + \lfloor \lg(D/N) \rfloor) \\ &< 2 \cdot D + D/N \qquad \text{minimum value of } N = 1 \\ &\leq 3 \cdot D. \end{aligned}$$

8.5 Summary

We have presented four forensic analysis algorithms: Monochromatic, RGBY, Tiled Bitmap, and a3D.

Assuming worst case scenarios, the running time of the Monochromatic Algorithm is $O(\lg D)$; for the rest it is $O(D)$. Each of these algorithms manages the trade-off between effort during normal processing and effort during forensic analysis; the algorithms differ in the precision of their forensic analysis. So, while the Monochromatic Algorithm has the fastest running time, it offers no information beyond the approximate location of the earliest corruption. The other algorithms work harder, but also provide more precise forensic information. In order to more comprehensively compare these algorithms, we desire to capture this tradeoff and resulting precision in a single measure.

9. FORENSIC COST

We define the forensic cost as a function of D (expressed as the 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* (the total number of t_l's, t_b's, and t_p's). A corruption site differs from a CE because a single timestamp CE has two corruption sites.

$$\begin{aligned} FC(D, N, V, \kappa) = \ &\alpha \cdot \textit{NormalProcessing}(D, N, V) \\ &+ \beta \cdot \textit{ForensicAnalysis}(D, N, V, \kappa) \\ &+ \gamma \cdot \textit{Area}_P(D, N, V, \kappa) + \delta \cdot \textit{Area}_U(D, N, V, \kappa) \end{aligned}$$

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 component, *NormalProcessing*, is the number of notarizations and validations made during normal processing in a span of D days. The second component, *ForensicAnalysis*, is the cost of forensic analysis in terms of the number of validations made by the algorithm to yield a result. Note that this is different from the running time of the algorithm. The rationale behind this quantity is that each notarization or validation involves an interaction with the external digital notarization service, which costs real money.

The third and fourth components informally indicate the manual labor required after automatic forensic analysis to identify exactly where and when the corruption happened. This manual labor is very roughly proportional to the uncertainty of the information returned by the forensic analysis algorithm. It turns out that there are two kinds of uncertainties, formalized as different areas (to be described shortly). That these components have different units than the first two components is accommodated by the weights.

In order to make the definition of forensic cost applicable to multiple corruption events, we need to distinguish between three regions within the corruption diagram. These different areas are the result of the forensic analysis algorithm identifying the corrupted granules. This distinction is based on the *information content* of each type.

—$Area_P$ or *corruption positive area* is the area of the region in which the forensic algorithm has established that corruption has definitively occurred.

—$Area_U$ or *corruption unknown area* is the area of the region in which we don't know *if* or *where* a corruption has occurred.

—$Area_N$ or *corruption negative area* is the area of the region in which the forensic algorithm has established that *no* corruption has occurred.

Each corruption site is associated with these three types of regions of varying area. More specifically, each site induces a partition of the horizontal trapezoid bound by the latest validation interval into three types of forensic area. Figure 13 shows this for a specific example of the RGBY Algorithm with two corruption events (CE$_1$, CE$_2$) and three corruption sites ($\kappa = 3$). For each corruption site, the sum of the areas, denoted by $TotalArea = Area_P + Area_U + Area_N$, corresponds to the horizontal trapezoid as shown. Hence $TotalArea = (V \cdot N) \cdot (D - (1/2) \cdot V \cdot N)$. Moreover, the forensic cost is a function of the number of corruption sites κ, each associated with the three areas $Area_P$, $Area_U$, $Area_N$. Hence in evaluating the forensic cost of a particular algorithm, we have to compute $Area_P$ and $Area_U$ for all κ, for example, $Area_P(D, N, V, \kappa) = \sum_\kappa Area_P$. The stronger the algorithm, the less costly it is, with smaller $Area_P$ and $Area_U$. It is also desirable that $Area_N$ is large but, since *TotalArea* is constant, this is achieved automatically by minimizing $Area_P$ and $Area_U$.

We now proceed to compute the forensic cost of our algorithms. We ignore the weights, since these constant factors will not be relevant when we use order notation.

9.1 The Monochromatic Algorithm

In the Monochromatic Algorithm, the spatial detection resolution (R_s) is the notarization interval, I_N, that is, $N = 1$. Recall that the Monochromatic

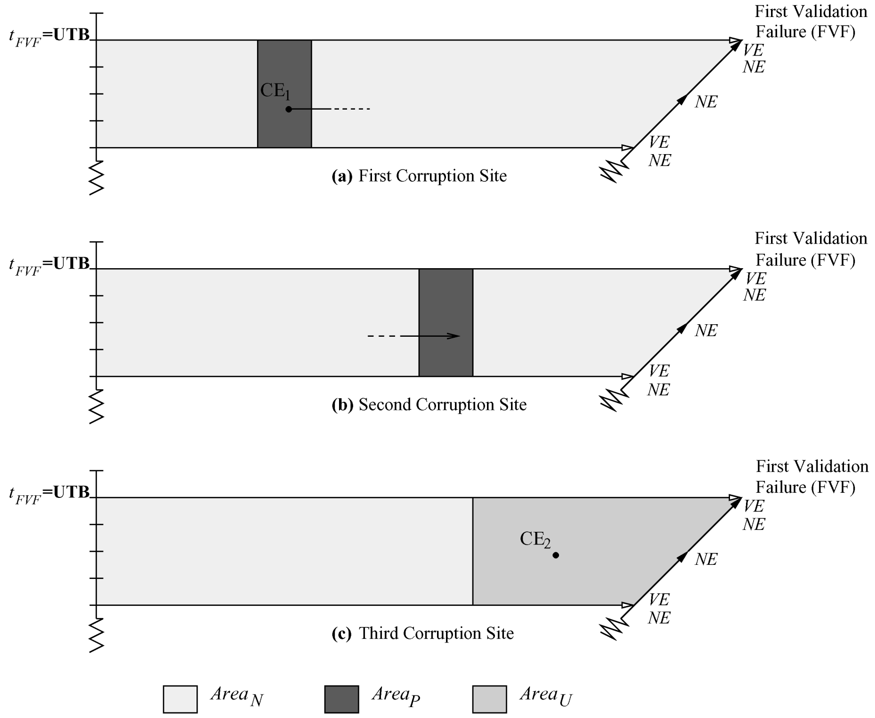

t_{FVF}=UTB
First Validation Failure (FVF)
VE
NE
NE
VE
NE
CE$_1$
(a) First Corruption Site
t_{FVF}=UTB
First Validation Failure (FVF)
VE
NE
NE
VE
NE
(b) Second Corruption Site
t_{FVF}=UTB
First Validation Failure (FVF)
VE
NE
CE$_2$
NE
VE
NE
(c) Third Corruption Site
$Area_N$
$Area_P$
$Area_U$

Fig. 13. Three types of forensic area for RGBY and $\kappa = 3$.

Algorithm can only detect a single corruption site, even though there could be κ of them in a single corruption diagram.

$$\begin{aligned} \mathit{NormalProcessing}_{mono} &= \mathit{Number\ of\ Notarizations} \\ &\quad + \mathit{Number\ of\ Validations} \\ &= D + D/V \end{aligned}$$

In forensic analysis calculations, we require D to be a multiple of V because t_{FVF} is a multiple of I_V and only at that time instant can the forensic analysis phase start. $\mathit{ForensicAnalysis}_{mono} = 2 \cdot \lg D$, since t_{RVS} is found via binary search on the black chains; the factor of two is because a pair of contiguous chains must be consulted to determine which direction to continue the search.

For the detected corruption site, $Area_P$ has a different shape depending on the position of the corruption site. As the corruption site moves from left to right (from earlier days to later days), the shape of the region changes from rectangular, to trapezoidal, and finally, to triangular. However, since we are dealing with worst-case scenario, the upper bound of $Area_P$ is $V \cdot N^2 = V$. Figure 14 shows such a worst-case distribution of κ corruption sites and how each site partitions the horizontal trapezoid into different forensic areas. In the worst-case, the corruption detected occurs within the first I_N, which makes $Area_P = 0$ of all other corruption sites because they cannot be detected. This

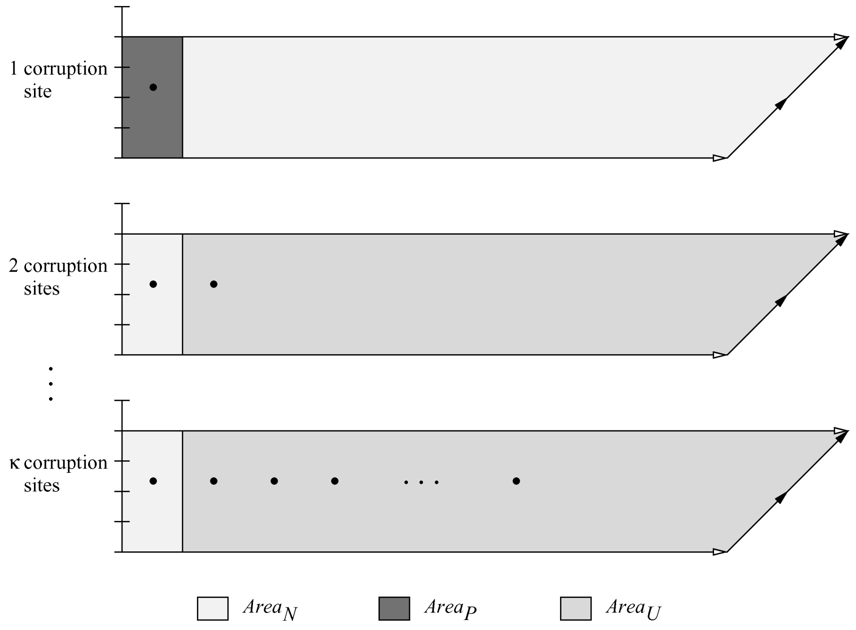


Fig. 14. Three types of forensic area for Monochromatic and κ corruption sites.

Table II. The Forensic Areas for $1 \leq \kappa \leq D$ Corruption Sites (Monochromatic)

# Corruption Sites ($1 \leq \kappa \leq D$)	$Area_P$	$Area_U$	$Area_N$
1	V	0	$TotalArea - V$
2	0	$TotalArea - V$	V
$\vdots$	$\vdots$	$\vdots$	$\vdots$
κ	0	$TotalArea - V$	V

results in:

$$\begin{aligned} Area_U &= (\kappa - 1) \cdot (TotalArea - V) \\ &= (\kappa - 1) \cdot [V \cdot (D - (1/2) \cdot V) - V] \\ &= (\kappa - 1) \cdot V \cdot (D - (1/2) \cdot V - 1). \end{aligned}$$

as shown in Table II.

Hence the forensic cost for the Monochromatic Algorithm is as follows:

$$\begin{aligned} FC_{mono}(D, I_N, V, \kappa) &= (D + D/V + 2 \cdot \lg D) \\ &\quad + \Big(V + \sum_{i=2}^{\kappa} (TotalArea - V)\Big). \end{aligned}$$

Table III. Summary of the Forensic Cost Assuming Worst-case Distribution of Corruption Sites

Algorithm	Worst-Case Forensic Cost	
	$(\kappa = 1)$	$(1 < \kappa \leq D)$
Monochromatic	$O(V + D)$	$O(\kappa \cdot V \cdot D)$
RGBY	$O(\kappa + D)$	
Tiled Bitmap	$O(\kappa \cdot V \cdot N^2 + (D \cdot \lg N)/N + \lg D)$	
a3D	$O(\kappa \cdot N + D + \kappa \cdot \lg D)$	

We consider the forensic cost of a single corruption site ($\kappa = 1$) separately from the cases where $\kappa > 1$, the reason being that the area breakdown is different in the two cases. Note that for $\kappa = 1$, $\sum_{i=2}^{\kappa}(\mathit{TotalArea} - V)$ is an empty sum, equal to zero.

$$\begin{aligned}
FC_{mono}(D, 1, V, 1) &= (D + D/V + 2 \cdot \lg D) + V \\
&= O(V + D) \\
FC_{mono}(D, 1, V, \kappa \geq 2) &= (D + D/V + 2 \cdot \lg D) \\
&\quad + (V + (\kappa - 1) \cdot (\mathit{TotalArea} - V)) \\
&= O(\kappa \cdot V \cdot D) \qquad (2)
\end{aligned}$$

In order to arrive at the order notation we make, here and in the following sections, the simplifying assumptions that $1 \leq V \leq \kappa \leq D$ and $1 \leq N \leq \kappa \leq D$.

9.2 Summary

In Appendix D we perform a similar worst-case forensic cost analysis for the RGBY, Tiled Bitmap, and a3D Algorithms; Appendices E and F provide best-case and average-case analyses, respectively, for all four algorithms.

We summarize the forensic cost for worst-case distribution of corruption sites of the algorithms in Table III. Tables IV and V summarize the forensic cost for average-case and best-case forensic cost, respectively. In the first two tables, we consider the number of corruptions $\kappa = 1$ separately from $1 < \kappa \leq D$ for the Monochromatic Algorithm. Recall that the forensic cost is a function of D, N, V, and κ and that for some of the algorithms N or V may be fixed.

In Table III we see in the rightmost column that Monochromatic depends on the product of κ and D, whereas RGBY depends on their sum. Tiled Bitmap has a complex combination of N and V and a3D adds a $\lg D$ multiplier to κ. If we consider the case when $\kappa = 1$ for all algorithms, we see that they are generally linear in D, except for Tiled Bitmap.

Observe that Table IV (average case) mirrors almost exactly the forensic cost of the worst-case distribution shown in Table III. This is not the case with Table V where the Monochromatic Algorithm is very cheap under best-case distribution and thus has a clear advantage over Tiled Bitmap and a3D. However,

Table IV. Summary of the Forensic Cost Assuming Average-case Distribution of Corruption Sites

Algorithm	Average-Case Forensic Cost	
	$(\kappa = 1)$	$(1 < \kappa \leq D)$
Monochromatic	$O(V + D)$	$O(\kappa \cdot V \cdot D)$
RGBY	$O(\kappa + D)$	
Tiled Bitmap	$O(\kappa \cdot V \cdot N^{\lg 3} + (D \cdot \lg N)/N + \lg D)$	
a3D	$O(\kappa \cdot N + D + \kappa \cdot \lg D)$	

Table V. Summary of the Forensic Cost Assuming Best-case Distribution of Corruption Sites

Algorithm	Best-Case Forensic Cost
	$(1 \leq \kappa \leq D)$
Monochromatic	$O(\kappa \cdot V + D)$
RGBY	$O(\kappa + D)$
Tiled Bitmap	$O(\kappa \cdot V \cdot N + (D \cdot \lg N)/N + \lg D)$
a3D	$O(\kappa \cdot N + D + \kappa \cdot \lg D)$

this only happens in the unlikely case where the position of the κ corruption sites allows the Monochromatic Algorithm to definitively identify them all.

In Table V (best case), for the Monochromatic Algorithm, we see that the forensic cost is lowest under best-case distribution, while for average-case and worst-case the forensic costs are asymptotically the same. Specifically, the number of granules D starts as an additive factor $O(\kappa \cdot V + D)$ in the best case and becomes a multiplicative factor $O(\kappa \cdot V \cdot D)$ in the average and worst cases. The forensic cost of the RGBY algorithm under all assumed distributions remains the same and is equal $O(\kappa + D)$. The forensic cost of the Tiled Bitmap Algorithm differs under each distribution. The difference lies with the exponent of N in the first term, which starts from 1, goes to $\lg 3$, and then to 2. The a3D Algorithm, like the RGBY Algorithm, is very stable, with the same forensic cost of $O(\kappa \cdot N + D + \kappa \cdot \lg D)$ under any assumed distribution of corruption events.

9.3 Audit System Implementation

A full implementation of the audit system was built on top of the TUC (Temporal Upward Compatibility), CLK (Clock), and STP (Stamper) modules that were previously added to an underlying Berkeley DB system so that it could support transaction time [Snodgrass et al. 2004]. A notarization service requester utility and a database validator utility were implemented as separately running programs; both send requests to and receive responses from an external digital notarization service, in this case Surety. The existence of the audit system imposed a 15% increase in time in the worst case when tuples are sufficiently small (10 bytes). The impact on time when record size and record number were larger was even less than 15%. Hence the overhead for runtime hashing is small. The Monochromatic forensic analysis algorithm has been incorporated into this system, and the rest of the algorithms are in the process of being incorporated. We can make the following observations. We have an idea of what the overhead is for the Monochromatic Algorithm because the normal processing part was evaluated before [Snodgrass et al. 2004]; the only part that is missing is the forensic analysis phase, which revalidates past hash chains. All algorithms have the same number of I/O operations (and hence performance in terms of time) as the Monochromatic for the normal processing phase, since for all algorithms the database is scanned entirely only during validations.

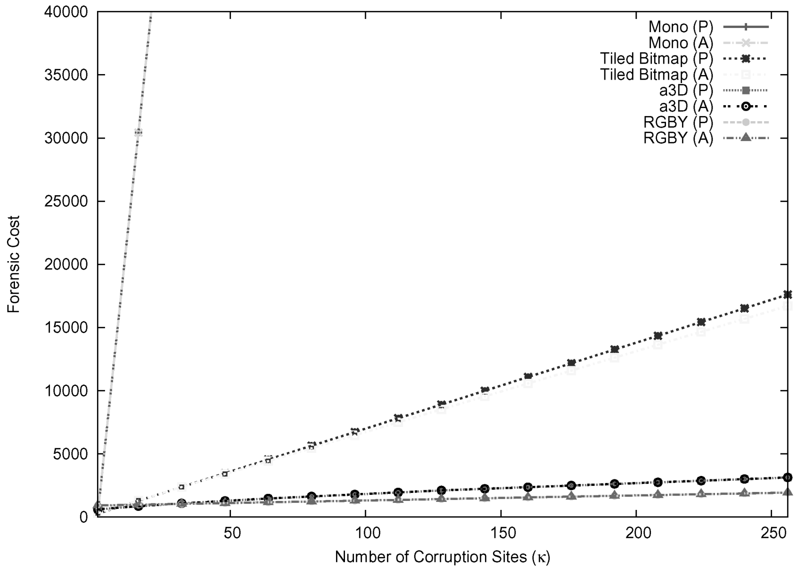
9.4 Illustrating and Validating the Forensic Cost

We have implemented the Monochromatic, RGBY, Tiled Bitmap, and a3D Algorithms in C. The entire implementation is approximately 1480 lines long and the source code is available at `http://www.cs.arizona.edu/projects/tau/tbdb/`. The forensic cost has been validated experimentally by inserting counters in the appropriate places in the code. More specifically, the forensic cost and its normal processing component have been validated for values $1 \leq D \leq 256$ and $1 \leq \kappa \leq 256$ as shown in Figures 15, 16, and 17.

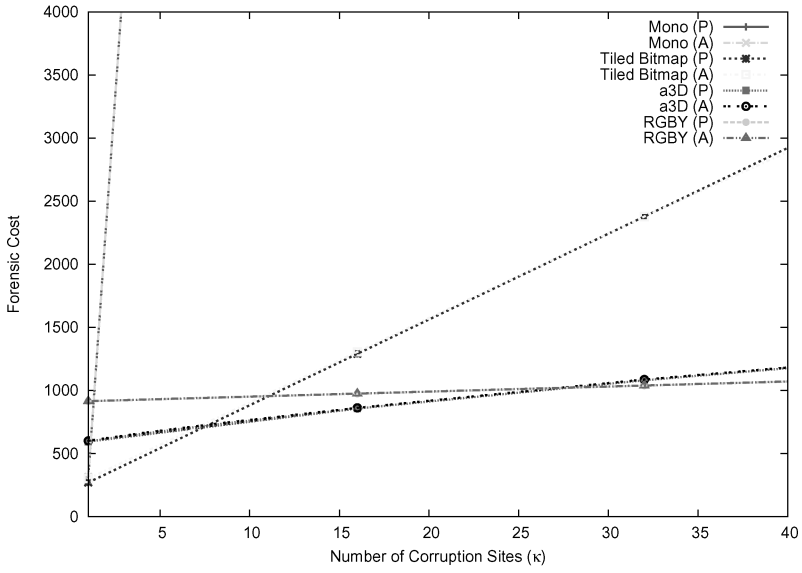
To examine the effects of the various parameters on the theoretical cost, we provide graphs showing the growth of forensic cost against these parameters. These are drawn on the same set of axes as the graphs derived experimentally. In all graphs, we have uniformly used $D = 256$ and $R_s = 1$. For the Monochromatic Algorithm, N is required to be 1, so $I_N = 1$, and we also set $V = 8$. For the RGBY Algorithm, N is also required to be 1 and V is required to be 2, so this dictates $R_t = 2$. For the Tiled Bitmap Algorithm, we set $N = 8$, which implies four chains, and we also set $V = 1$. For the a3D Algorithm, we similarly set $N = 8$ and $V = 1$. All algorithms have $R_t = 8$ except for RGBY. The settings used in the experimental validation are summarized in Table VI.

Rather than using the cost formulas in order notation to create the graphs, we used the more involved (and more accurate) cost functions derived for each algorithm: equation (2) in Section 9.1 for the Monochromatic Algorithm, equation (3) in Appendix D.1 for the RGBY Algorithm, and equation (4) in Appendix D.2 for the Tiled Bitmap Algorithm. For the a3D Algorithm we used the even more precise recursive formula equation (5) in Appendix D.3 to calculate the cost of normal processing instead of the one shown within (6) in Appendix D.3. Also, for values of D that are not a multiple of $V \cdot I_N$ we use the largest multiple of $V \cdot I_N$ less than D to calculate the cost of the forensic analysis stage.

Note that all cost plots show both the predicted forensic cost (denoted by "(P)" in the plot legend) and the actual forensic cost values (denoted by "(A)" in the plot legend). The different types of symbols on the curves were added for clarity and correspond to a subset of the actual data points.



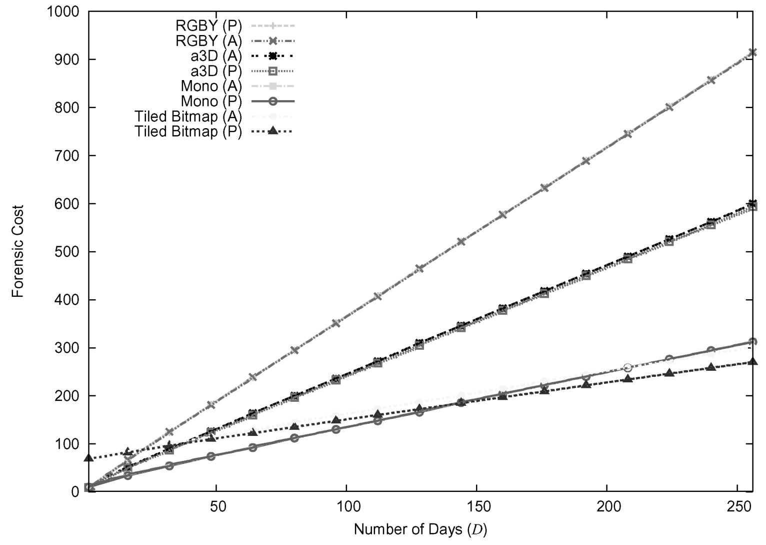
(a) $1 \leq \kappa \leq 256$



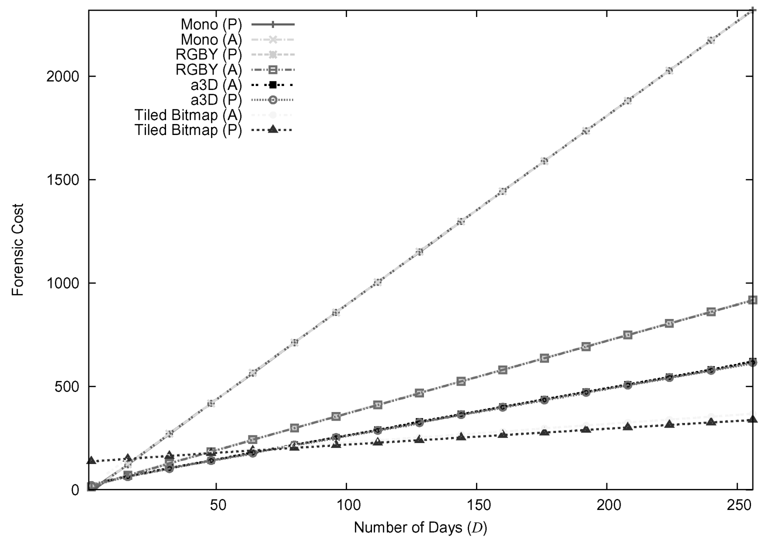
(b) $1 \leq \kappa \leq 40$

Fig. 15. Forensic cost against κ for $D = 256$.

We start by examining the growth of forensic cost of the algorithms against κ, as shown in the graphs in Figure 15. Figure 15(a) shows how the forensic cost increases with κ. Most of the predicted forensic costs are very close to the actual values and in these cases (e.g., a3D algorithm) the two lines overlap. Figure 15(b) is a magnification of the region in Figure 15(a) where κ takes values between 1 and 40. The cheapest algorithm for $\kappa = D$ is RGBY, while the most expensive is Monochromatic. Observe in Figure 15(b) that for $\kappa = 1$ the



(a) $\kappa = 1$



(b) $\kappa = 2$

Fig. 16. Forensic cost against D for $\kappa = 1$ and $\kappa = 2$.

costs of Monochromatic and Tiled Bitmap are comparable and have the lowest value *of all other* algorithms.

Even more interestingly, the RGBY algorithm starts off as being the most expensive algorithm and then becomes the cheapest. This can be explained by observing that ambiguity in the corruption region (large $Area_U$ for $\kappa > 1$) increases the cost of the Monochromatic Algorithm. The Tiled Bitmap Algorithm suffers considerably from false positives (for every true corruption site there exist $N - 1$ false positives) On the other hand the comparison of a3D to RGBY

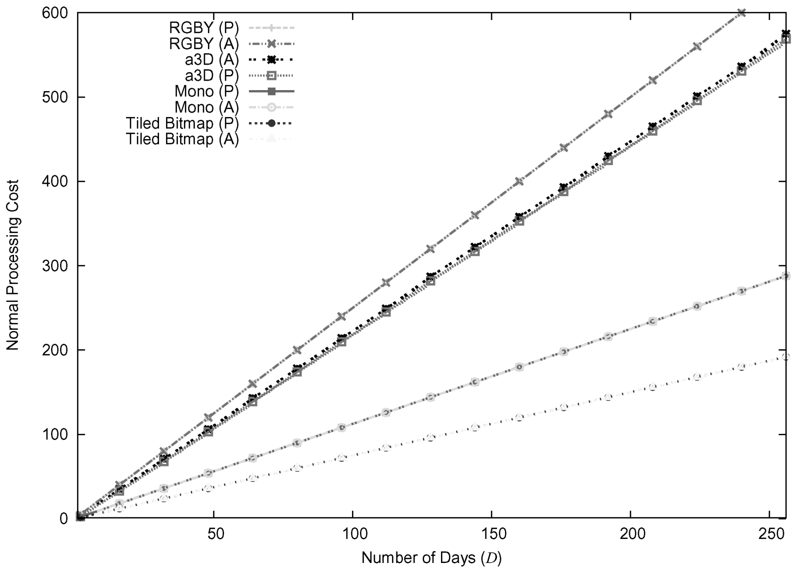


Fig. 17. Cost of *Normal Processing* against D.

Table VI. Settings Used in Experimental Validation of Forensic Cost Assuming Worst-case Distribution of Corruption Sites (values in **bold** are non-configurable)

	Algorithm			
Parameters	Monochromatic	RGBY	Tiled Bitmap	a3D
R_s	1	1	1	1
N	**1**	**1**	8	8
V	8	**2**	1	**1**
D	256	256	256	256

is more subtle. a3D starts below RGBY but eventually becomes more expensive. The reason is that, initially, RGBY has to linearly scan all its partial chains, whereas a3D does not (this happens only for $D/2 \leq \kappa \leq D$). However, the overhead of validating the a3D tree outweighs the impact of the presence of false positives produced by RGBY.

Figure 16 shows how the forensic cost increases with time D for different numbers of corruption sites, namely, $\kappa = 1$, and $\kappa = 2$. As expected for $\kappa = 1$ (Figure 16(a)), the Monochromatic Algorithm has the lowest forensic cost for roughly the first half of the range of values of D. For the second half, the Tiled Bitmap Algorithm becomes the cheapest because it is able to definitively identify all the corruption negative areas through a logarithmic number of chains.

When $\kappa = 2$ the Tiled Bitmap Algorithm is cheapest, as shown in Figure 16(b). This is because Tiled Bitmap need only identify a single additional tile, whereas a3D (the second cheapest algorithm) has to process an entire subtree of maximal height. Also, for this value of κ the Monochromatic Algorithm becomes the most expensive algorithm.

Table VII. Sample Forensic Costs for the Four Implemented Algorithms

Forensic Cost	Monochromatic	RGBY	Tiled Bitmap	a3D
Predicted	512352	1934	17610	3128
Actual	512354	1935	16714	3134

Finally, Figure 17 shows how the cost of only the normal processing phase varies with the number of days D. Note that this cost is independent of κ. Here we see clearly that the most expensive normal processing phase belongs to the RGBY Algorithm. a3D is the next most expensive algorithm, while Tiled Bitmap is the cheapest in terms of normal processing. This suggests that shifting the cost/amount of work done to the normal processing phase may not always pay off during forensic analysis.

Table VII shows the predicted and actual forensic costs for the four implemented algorithms when $D = \kappa = 256$. The two values in each of the four cases differ by about 6% for Tiled Bitmap; for the other three, the actual and predicted were nearly identical.

10. A LOWER BOUND FOR FORENSIC COST

We wish to derive a realistic lower bound for the *Forensic Cost* measure in terms of κ corruption sites, validation factor V, notarization interval I_N, and D days before the corruption is first detected.

The optimal value for $Area_U$ is 0, whereas the optimal value for $Area_P$ is $\kappa \cdot V \cdot I_N \cdot w$, where w is the width and $V \cdot I_N = I_V$ is the height of the rectangle in which the corruption site is located.

Keep in mind that these algorithms do not aim at reducing the t_c uncertainty. The size of the uncertainty of this temporal dimension depends solely on the size of the validation interval (I_V) and therefore it can be reduced if and only if I_V is reduced. No other factor involving external notarization can have any impact on it. This is due to the fact that the t_c uncertainty is bounded above by the current time—the time the CE was discovered—and is also bounded below by the last time we checked the database, the last *VE* time. This is by definition the validation interval. Any new strategies introduced that are purely native to the system cannot be trusted and thus violate the working premise of this approach: no extra assumptions should be made about the system.

To obtain bounds on the *Normal Processing* and *Forensic Analysis* phases, we start with a rather optimistic scenario. Suppose that we had a priori knowledge (both when and where) of the exact κ granules to be corrupted in the future. Then the optimal algorithm would notarize κ hash chains of length one, each covering the granule to be corrupted. Similarly, the forensic analysis would validate those κ hash chains and that would find all corruption sites, each bounded by a rectangle of height $V \cdot I_N$ and width $w = 1$ granule (where a granule is a unit of R_s). Thus a lower bound would be

$$FC_{\texttt{lower bound}_1} = (\kappa + \kappa) + \kappa \cdot V \cdot I_N \cdot 1 = 2 \cdot \kappa + \kappa \cdot V \cdot I_N.$$

However, we do not feel it is fair to tax our algorithms with the burden of precognition. While we still assume that κ and D are known, we have no a priori knowledge of where the corruption sites are going to occur (within D). Given this information, we seek to find the optimal value for $n \geq \kappa$, the number of notarizations during normal processing. The cost of *Normal Processing* $= n$, while the cost of *Forensic Analysis* $= \kappa \cdot \lg n$. This is because, for every κ, we must perform binary search in order to locate it. The width w for $Area_P$ is $D/2^n$ because each notarization provides a single bit of information in the where dimension.

$$\begin{aligned} FC_{\texttt{lower bound}_2} &= (n + \kappa \cdot \lg n) \\ &\quad + \kappa \cdot V \cdot I_N \cdot D/2^n \end{aligned}$$

If we assume that the width w can take a minimum value of $w = 1$, then it follows that $n = \lg D$. If we substitute this value of n into the previous FC expression, we get:

$$FC_{\texttt{lower bound}_2} = (\lg D + \kappa \cdot \lg\lg D) + \kappa \cdot V \cdot I_N.$$

This lower bound makes fewer assumptions about the information available and therefore $FC_{\texttt{lower bound}_1} \leq FC_{\texttt{lower bound}_2}$. Our final lower bound is:

$$\begin{aligned} FC_{\texttt{lower bound}} &= \kappa \cdot V \cdot I_N + \kappa \cdot \lg\lg D + \lg D\ , \quad \kappa \leq D \\ &= O(\kappa \cdot V \cdot N + \lg D). \end{aligned}$$

Table VIII compares this lower bound with the worst-case forensic cost of our algorithms, characterized for "small κ" and "large κ." In particular, we eliminate κ by assuming it is either equal to $O(1)$ or $O(D)$, respectively. Note that for the Monochromatic Algorithm, if $\kappa = O(1)$, we assume that $V \ll D$, thus simplifying the cost from $O(V+D)$ to $O(D)$. Tables IX and X repeat this comparison with average- and best-case forensic costs derived in Appendices E and F. The lower bound across all three tables is the same because the only way for the lower bound to decrease in the best-case and average-case analysis is for the binary search to be faster during forensic analysis. All other components are essential, that is, n notarizations are required, and the width w must equal 1. Binary search in the best-case takes $O(1)$, eliminating the $\lg\lg D$ factor, which in asymptotic notation is irrelevant. In the average-case forensic cost, the average running time of binary search is $O(\lg n)$ so the lower bound remains the same. Hence there is only the notion of a single lower bound.
For best-case forensic cost, the Monochromatic algorithm (which requires $N = 1$) and the RGBY algorithm (which requires $N = 1$ and $V = 2$) are optimal for large κ. Observe also that the asymptotic forensic cost of both the RGBY Algorithm and that of the a3D Algorithm (which requires $V = 1$) for all possible cases (worst, best, average) is optimal for large κ and is close to optimal for small κ.

11. RECOMMENDATIONS

Given the forensic cost formulæ and the insights from the previous sections, our recommendation is that it is best to provide users with three algorithms:

Table VIII. Worst-case Forensic Cost and Lower Bound

Algorithm	Worst-Case Forensic Cost	
	Small κ ($\kappa = O(1)$)	Large κ ($\kappa = O(D)$)
Monochromatic	$O(D)$	$O(V \cdot D^2)$
RGBY	$O(D)$	
Tiled Bitmap	$O(V \cdot N^2 + (D \cdot \lg N)/N + \lg D)$	$O(V \cdot N^2 \cdot D)$
a3D	$O(N + D)$	$O(N \cdot D)$
Lower Bound	$O(V \cdot N + \lg D)$	$O(V \cdot N \cdot D)$

Table IX. Average-case Forensic Cost and Lower Bound

Algorithm	Average-Case Forensic Cost	
	Small κ ($\kappa = O(1)$)	Large κ ($\kappa = O(D)$)
Monochromatic	$O(D)$	$O(V \cdot D^2)$
RGBY	$O(D)$	
Tiled Bitmap	$O(V \cdot N^{\lg 3} + (D \cdot \lg N)/N + \lg D)$	$O(V \cdot N^{\lg 3} \cdot D)$
a3D	$O(N + D)$	$O(N \cdot D)$
Lower Bound	$O(V \cdot N + \lg D)$	$O(V \cdot N \cdot D)$

Table X. Best-case Forensic Cost and Lower Bound

Algorithm	Best-Case Forensic Cost	
	Small κ ($\kappa = O(1)$)	Large κ ($\kappa = O(D)$)
Monochromatic	$O(D)$	$O(V \cdot D)$
RGBY	$O(D)$	
Tiled Bitmap	$O(V \cdot N + (D \cdot \lg N)/N + \lg D)$	$O(V \cdot N \cdot D)$
a3D	$O(N + D)$	$O(N \cdot D)$
Lower Bound	$O(V \cdot N + \lg D)$	$O(V \cdot N \cdot D)$

Monochromatic, a3D and, depending on the application requirements, RGBY or Tiled Bitmap. The reason for considering RGBY and Tiled Bitmap as optional is that both these algorithms, unlike the Monochromatic and a3D, suffer from false positives. The RGBY Algorithm has the same optimal characteristics as a3D and is the cheapest when a large number of corruptions is expected. On the other hand, the Tiled Bitmap Algorithm has the lowest forensic cost in the long term for a fixed small number of corruptions but suffers from more false positives than RGBY, which translates into more human effort when trying to pinpoint the exact corruptions at a later stage. The Tiled Bitmap Algorithm is also indicated when efficiency during normal processing is critical.

If only two algorithms are to be used, then both Monochromatic and a3D should be implemented. If only one algorithm is needed, the choice would be again between the Monochromatic and a3D Algorithms. The Monochromatic Algorithm is by far the simplest one to implement and it is best-suited for cases when multiple corruptions are not anticipated or when only the earliest corruption is desired. The a3D Algorithm is the second easiest algorithm to implement and it is the only algorithm that exhibits all three of the most desirable characteristics: (i) it identifies multiple corruptions, (ii) it does not produce false positives, and (iii) it is stable and optimal for large κ (and near optimal for small κ). Hence this algorithm is indicated in situations where accuracy in forensic analysis is of the utmost importance.

12. RELATED WORK

There has been a great deal of work on records management, and indeed, an entire industry providing solutions for these needs, motivated recently by Sarbanes-Oxley and other laws requiring audit logs. In this context, a "record" is a version of a document. Within a document/record management system (RMS), a DBMS is often used to keep track of the versions of a document and to move the stored versions along the storage hierarchy (disk, optical storage, magnetic tape). Examples of such systems are the EMC Centera Compliance Edition Content Addressed Storage System,[1] the IBM System Storage DR series,[2] and NetApp's SnapLock Compliance.[3] Interestingly, these systems utilize magnetic disks (as well as tape and optical drives) to provide WORM storage of compliant records. As such, they are implementations of *read-only file systems* (also termed *append-only*), in which new files can only be added. Several designs of read-only file systems have been presented in the research literature [Fu et al. 2000; Mazières et al. 1999]. Both of these systems (as well as Ivy [Muthitacharoen et al. 2002]) use cryptographic signatures so that programs reading a file can be assured that it has not been corrupted.

Hsu and Ong [2004] have proposed an end-to-end perspective to establishing trustworthy records, through a process they term *fossilization*. The idea is that once a record is stored in the RMS, it is "cast in stone" and thus not modifiable. An index allows efficient access to such records, typically stored in some form of WORM storage. Subsequently, they showed how the index itself could be fossilized [Zhu and Hsu 2005]. Their approach utilizes the WORM property provided by the systems just listed: that the underlying storage system supports

[1]`http://www.emc.com/products/detail/hardware/centera.htm` (accessed April 28, 2008)
[2]`http://www.ibm.com/systems/storage/disk/dr/` (accessed April 28, 2008)
[3]`http://www.netapp.com/us/products/protection-software/snaplock.html` (accessed April 28, 2008)

reads from and writes to a random location, while ensuring that any data that has been written cannot be overwritten.

This is an appealing and useful approach to record management. We have extended this perspective by asserting that every tuple in a database is a record, to be managed. The challenge was two-fold. First, a record in a RMS is a heavyweight object: each version is stored in a separate file within the file system. In a DBMS, a tuple is a lightweight object, with many tuples stored on a single page of a file storing all or a good portion of a database. Secondly, records change quite slowly (examples include medical records, contacts, financial reports), whereas tuples change very rapidly in a high-performance transactional database. It is challenging to achieve the functionality of tracked, tamper-free records with the performance of a DBMS.

This raises the interesting question: since record management systems often use relational databases internally, how effective can these systems really be? Given the central role of audit logs in performing auditing of interactions with the records (tracking, modifications, exposure), the audit logs themselves are as valuable as the records they reference. It is critical that such audit logs and tracking information be correct and unalterable. It is not sufficient to say, "the records we store in our RMS are correct, because we store all interactions and tracking data in a separate audit log." The integrity of the underlying database is still in question. While Zhu and Hsu [2005] provide a partial answer through their fossilized index (mentioned previously), the rest of the database might still be tampered.

Johnston [2006] goes back thousands of years to show that in many cases, tamperproofing is economically inferior to tamper detection: "Often, it's just not practical to try to stop unauthorized access or to respond to it rapidly when detected. Frequently, it's good enough to find out some time after the fact that trespassing took place".

The first work to show that records management could be effectively merged with conventional databases was that by Barbará et al. [2000] on using checksums to detect data corruption. By computing two checksums in different directions and using a secret key, they were able to dramatically increase the effort an intruder would have to make to tamper the database. Our paper on tamper detection removed one assumption, that the system could keep a secret key that would not be seen by insiders [Snodgrass et al. 2004]. We showed that cryptographic techniques coupled with a carefully-considered architectural design and an external digital notarization service could solve one part of the puzzle: detecting tampering. In this article we consider another part of the puzzle: forensic analysis once tampering has been detected.

Computer forensics is now an active field, with over fifty books published in the last ten years[4] and another dozen already announced for 2008. However, these books are generally about preparing admissible evidence for a court case, through discovery, duplication, and preservation of digital evidence. There are few computer tools for these tasks, in part due to the heterogeneity of the data.

[4]`http://www.e-evidence.info/books.html` (accessed April 28, 2008)

One substantive example of how computer tools can be used for forensic analysis is Mena's book [2003]. The more narrow the focus, the more specialized tools there are that can help. Carvey and Kleiman's book [2007] covers just variants of that operating system and explains how to use the author's *Forensic Server Project* system to obtain data from a Windows system in a forensically sound manner. Closer to home, Schneier and Kelsey [1999] describe a secure audit log system, but do not consider forensic analysis of such audit logs.

Goodrich et al. [2005] introduce new techniques for using indexing structures for data forensics. The usual way of detecting malicious tampering of a digital object using cryptographic one-way hashes to store a cryptographic hash of the item and then to use it later as a reference for comparison. The approach of Goodrich et al. goes beyond the single bit (true/false) of information provided by a hash: they store multiple hashes (and attempt to minimize the required number of such values) to pinpoint which of a given set of items has been modified. They encode authentication information in the topology of the data structure of items (not in the stored values themselves) so that alterations can be detected. This is important because this approach requires no additional space other than the cryptographic master key used by the auditing agent. Their techniques are based on a new reduced-randomness construction for nonadaptive combinatorial group testing (CGT). In particular, they show how to detect up to d defective items out of a total of n items, with the number of tests being $O(d^2 \lg n)$. Moreover, they provide forensic constructions of several fundamental data structures, including binary search trees, skip lists, arrays, linked lists, and hash tables.

Several differences exist between Goodrich's approach and the one outlined in the current article.

—The threat model in Goodrich et al. does not allow changes in the topology of the data structure, whereas ours places no such restrictions.

—The objective in Goodrich et al. is to minimize the number of hashes stored that would make it possible to identify d corruptions, given a particular data structure. In the current article, we seek a structure of hash chains with the lowest forensic cost, which includes both normal processing and forensic analysis components.

—Their CGT method is probabilistic whereas ours is deterministic.

—Goodrich et al.'s work applies to main-memory structures, whereas ours applies to disk-resident data items.

—There exists an upper bound on the number of modified items that can be detected, for example, for a balanced binary search tree storing n elements, the bound is $O(n^{1/3} / \log^{2/3} n)$. Our approach can detect up to $\kappa = n$ corruptions.

Goodrich's approach in constructing forensic data structures might be generalizable to detecting changes in key values stored in a B-tree. This could then provide some information about data values, thereby possibly reducing the number of hashes needed and thus the forensic cost.

Earlier, we introduced the approach of using cryptographic hash functions for tamper detection [Snodgrass et al. 2004] and introduced the first forensic analysis algorithms for database tampering [Pavlou and Snodgrass 2006a]. The present article significantly extends that research, with pseudocode for one previous algorithm (Monochromatic) and three new algorithms: the RGBY (a refinement of the previous RGB Algorithm), Tiled Bitmap (a refinement of the previous Polychromatic Algorithm), and a3D forensic analysis Algorithms.

We refine the definition of forensic strength to arrive at a notion of "forensic cost" that encompasses multiple corruption events. The objective is to minimize this cost in order to achieve a higher forensic strength. The definition of forensic cost retains some of the key characteristics of the original definition [Pavlou and Snodgrass 2006a], while adopting a more sophisticated treatment of the region and uncertainty areas returned by the forensic algorithms, and incorporating the notions of temporal and spatial resolution. We characterize and validate experimentally the forensic cost for all four algorithms presented in this article. We also present a lower bound that considers multiple corruption events.

13. SUMMARY AND FUTURE WORK

New laws and societal expectations are raising the bar concerning stewardship of stored data. Corporate and governmental databases are now expected and in many cases required to mediate access, to preserve privacy, and to guard against tampering, even by insiders.

Previously-proposed mechanisms can detect that tampering has occurred. This article considers the next step, that of determining *when*, *what*, and hence indirectly providing clues as to *who*, through the use of various forensic analysis algorithms that utilize additional information (in this case, partial hash chains) computed during periodic validation scans of the database.

We introduced corruption diagrams as a way of visualizing corruption events and forensic analysis algorithms. We presented four such algorithms, the Monochromatic, RGBY, Tiled Bitmap, and a3D Algorithms, and showed through a formal forensic cost comparison (with worst-case, best-base, and average-case assumptions), validated with an implementation, that each successive algorithm adds extra work in the form of main-memory processing, but that the resulting additional precision in the obtained information more than counterbalances this extra work. Finally, we provided a lower bound for forensic cost and showed that only the a3D Algorithm is optimal for a large number of corruptions and close to optimal in all cases, without producing false positives.

Our recommendation is that, at an initial stage, it is best to provide users with the Monochromatic and a3D Algorithms. The Monochromatic Algorithm is the easiest to implement and is indicated when multiple corruptions are not anticipated or when only the earliest corruption site is desired. The a3D Algorithm is stable with optimal forensic cost (for large κ), is able to determine the "where", and the "when" of a tampering quite precisely and efficiently, and is able to effectively handle multiple corruption events. The other two algorithms produce false positives and can be provided as dictated by the application requirements. The RGBY Algorithm has optimal cost (for large κ) and is cheapest when many corruption sites are anticipated. The Tiled Bitmap Algorithm has the lowest forensic cost in the long term for a fixed number of corruptions and is also indicated when efficiency during normal processing is critical.

We are integrating these algorithms into a comprehensive enterprise solution for tamper detection and analysis that manages multiple databases with disparate security risks and requirements. Also, we are examining the interaction between a transaction-time storage manager and an underlying magnetic-disk-based WORM storage. As archival pages are migrated to WORM storage, they would be thus protected from tampering, and so would not need to be rescanned by the validator. It is an open question how to extend the forensic analysis algorithms to accommodate schema corruption.

Our challenge is in a sense the dual of that considered by Stahlberg et al. [2007]. As mentioned in Section 2, we utilize a transaction-time table to retain previous states and perform forensic analysis on this data once tampering is detected. Stahlberg considers the problem of forensic analysis uncovering data that has been previously deleted, data that shouldn't be available. It is an open question as to how to augment our approach for forensic analysis to accommodate *secure deletion*.

Finally, it might make sense to augment database storage structures, such as indexes, in a manner similar to that proposed for main-memory structures by Goodrich et al. [2005], to aid in forensic analysis.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library. The appendix discusses the subtleties involved in the forensic analysis of introactive corruption events, and demonstrates how false positives arise in the RGBY Algorithm. It also describes the Tiled Bitmap Algorithm, discusses the notion of a candidate set, and gives the running time of the algorithm. Finally, it analyzes the forensic cost for the algorithms, using worst-case, best-case, and average-case assumptions on the distribution of corruption sites.

ACKNOWLEDGMENTS

Huilong Huang, Qing Ju, Melinda Malmgren, and Shilong (Stanley) Yao contributed to the implementation. We thank Travis Wheeler for his help with the characterization of forensic cost and thank the reviewers for their concrete suggestions which helped improve the presentation of the material.

REFERENCES

AGRAWAL, R., GRANDISON, T., JOHNSON, C., AND KIERNAN, J. 2007. Enabling the 21st century healthcare IT revolution. *Comm. ACM*, *50*, 2, 34–42.

AHN, I. AND SNODGRASS, R. T. 1988. Partitioned storage structures for temporal databases. *Inform. 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. *Bus. Inform. 39*, 1, 25–34.

BARBARÁ, D., GOEL, R., AND JAJODIA, S. Using checksums to detect data corruption. In *Proceedings of the International Conference on Extending Database Technology*, Lecture Notes in Computer Science, vol. 1777, Springer, Berlin, Germany.

CARVEY, H. AND KLEIMAN, D. 2007. Windows Forensics and Incident Recovery, Syngres.

CHAN, C. C., LAM, H., LEE, Y. C., AND ZHANG, X. 2004. *Analytical Method Validation and Instrument Performance Verification*, Wiley-IEEE.

CSI/FBI. 2005. Tenth Annual Computer Crime and Security Survey, http://www.cpppe.umd.edu/Bookstore/Documents/2005CSISurvey.pdf (accessed April 25, 2008).

DEPARTMENT OF DEFENSE. 1985. Trusted Computer System Evaluation Criteria. DOD-5200.28-STD, http://www.dynamoo.com/orange (accessed April 25, 2008).

F.D.A. 2003. Title 21 Code of Federal Regulations (21 CFR Part 11) Electronic records; Electronic Signatures, http://www.fda.gov/ora/compliance_ref/part11/ (accessed April 28, 2008).

FU, K., KAASHOEK, M. F., AND MAZIÈRES, D. 2000. Fast and secure distributed read-only file system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Association, Berkeley, CA, 181–196.

GERR, P. A., BABINEAU, B., AND GORDON, P. C. 2003. Compliance: The effect on information management and the storage industry. Tech. rep. Enterprise Storage Group. http://www.enterprisestrategygroup.com/ESGPublications/ReportDetail.asp?ReportID=201 (accessed May 4, 2008).

GOODRICH, M. T., ATALLAH, M. J., AND TAMASSIA, R. 2005. Indexing information for data forensics. In *Proceedings of the Conference on Applied Cryptography and Network Security (ACNS)*. Lecture Notes in Computer Science, vol. 3531, Springer, Berlin, Germany, 206–221.

GRAHAM, R. L., KNUTH, D. E., AND PATASHNIK, O. 2004. *Concrete Mathematics*, 2nd Ed., Addison–Wesley.

HABER, S. AND STORNETTA, W. S. 1991. How to time-stamp a digital document. *J. Cryptology*, *3*, 2, 99–111.

HIPAA. 1996. The Health Insurance Portability and Accountability Act. U.S. Dept. of Health & Human Services. http://www.cms.hhs.gov/HIPAAGenInfo/ (accessed April 25, 2008).

HSU, W. W. AND ONG, S. 2004. Fossilization: a process for establishing truly trustworthy records. Tech. rep. RJ 10331, IBM.

JENSEN, C. S. AND DYRESON, C. E., Eds. 1998. A consensus glossary of temporal database concepts—(February 1998 Version). In *Temporal Databases: Research and Practice*, Etzion, O., Jajodia, S., and Sripada S., Eds. Springer, 367–405.

JENSEN, C. S. AND SNODGRASS, R. T. 1994. Temporal specialization and generalization. *IEEE Trans. Knowl. Data Eng. 6*, 6, 954–974.

JOHNSTON, R. G. Tamper-indicating seals. 2006. *Am. Sci. 94*, 6, 515–524.

LOMET, D., BARGA, R., MOKBEL, M. F., SHEGALOV, G., WANG, R., AND ZHU, Y. 2005. Immortal DB: transaction time support for SQL server. *In Proceedings of the International ACM Conference on Management of Data (SIGMOD)*, ACM, New York, 939–941. http://research.microsoft.com/research/db/immortaldb/ (accessed April 25, 2008).

MAZIÈRES, D., KAMINSKY, M., KAASHOEK, M. F., AND WITCHEL, E. 1999. Separating key management from file system security. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, 124–139.

MENA, J. 2003. *Investigative Data Mining for Security and Criminal Detection*. Butterworth Heinemann.

MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. 2002. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (SOSDI).* ACM, New York, NY, 31–44.

ORACLE CORPORATION. 2007. Oracle Database 11g Workspace Manager Overview. Oracle White Paper, http://www.oracle.com/technology/products/database/workspace_manager/pdf/twp_AppDev_Workspace_Manager_11gR1.pdf (accessed April 28, 2008).

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

PAVLOU, K. E. AND SNODGRASS, R. T. 2006b. The pre-images of bitwise AND functions in forensic analysis. Tech. rep., TIMECENTER.

PIPEDA. 2000. Personal Information Protection and Electronic Documents Act. Bill C-6, Statutes of Canada, http://www.privcom.gc.ca/legislation/02_06_01_01_e.asp.

Sarbanes-Oxley Act. 2002. U.S. Public Law No. 107–204, 116 Stat. 745, The Public Company Accounting Reform and Investor Protection Act.

Schneier, B. and Kelsey, J. 1999. Secure audit logs to support computer forensics. *ACM Trans. Inform. Syst. Sec. 2*, 2, 159–196.

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 (VLDB)*. Toronto, Canada. Morgan Kaufmann, San Francisco, CA, 504–515.

Stahlberg, P., Miklau, G., and Levine, B. N. 2007. Threats to privacy in the forensic analysis of database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Beijing, China. ACM, New York, NY, 91–102.

Wingate, G., Ed. 2003. *Computer Systems Validation: Quality Assurance, Risk Management, and Regulatory Compliance for Pharmaceutical and Healthcare Companies*. Informa Healthcare.

Zhu, Q. and Hsu, W. W. 2005. Fossilized index: The linchpin of trustworthy non-alterable electronic records. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Baltimore, Md. ACM, New York, NY, 395–406.

Received December 2007; revised June 2008; accepted August 2008.