Forensic Analysis of Database Tampering

Kyriacos Pavlou
Department of Computer Science
University of Arizona
Tucson, AZ
kpavlou@email.arizona.edu

Richard T. Snodgrass
Department of Computer Science
University of Arizona
Tucson, AZ
rts@cs.arizona.edu

ABSTRACT

Mechanisms now exist that detect tampering of a database, through the use of cryptographically-strong hash functions. This paper addresses the next problem, that of determining who, when, and what, by providing a systematic means of performing forensic analysis after such tampering has been uncovered. We introduce a schematic representation termed a "corruption diagram" that aids in intrusion investigation. We use these diagrams to fully analyze the original proposal, that of a linked sequence of hash values. We examine the various kinds of intrusions that are possible, including retroactive, introactive, backdating, and postdating intrusions. We then introduce successively more sophisticated forensic analysis algorithms: the monochromatic, RGB, and polychromatic algorithms, and characterize the "forensic strength" of these algorithms. We show how forensic analysis can efficiently extract a good deal of information concerning a corruption event.

1. INTRODUCTION

Due in part to recent federal laws (e.g., Health Insurance Portability and Accountability Act: HIPAA, Canada's PIPEDA, Sarbanes-Oxley Act) and standards (e.g., Orange Book for security), and in part due 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 need for audit log security goes far beyond just the financial and medical information systems mentioned above. 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" [9].

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 [5]. 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 (10,000 in the US) to follow certain "processes by which they are created, stored, accessed, maintained, and retained" [4]. It is common to use Write-Once-Read-Many (WORM) storage devices to preserve such records [16]. 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 [15]. 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* is the intruder.

In this paper, we provide a means of systematically performing forensic analysis after an intrusion of an audit log has been detected. We first summarize the originally proposed

SIGMOD'06, June 27–29, 2006, Chicago, Illinois, USA.

approach, which provides exactly one bit of information: has the audit log been tampered? 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 strength" of this algorithm, defined as the reduction in area of the uncertainty region 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 decrease the forensic strength. Along the way, we identify some configurations that turn out not to improve the forensic strength, thus helping to cull the most appropriate alternatives.

We then consider computing and notarizing additional sequences of hash values. For each successively more powerful forensic analysis algorithm, we provide a formal/diagrammatic analysis of its forensic strength. The above-mentioned algorithm is the monochromatic algorithm; we also consider the RGB algorithm and the polychromatic algorithm. This last algorithm can efficiently extract a good deal of information concerning a corruption event. We end with a discussion of related and future work.

2. TAMPER DETECTION

In this section we summarize the *tamper detection* approach we previously proposed and implemented [15]. We just give the gist of our approach, so that our forensic analysis techniques can be understood.

There are several related ideas that in concert allow tamper detection.

- The first insight is that the DBMS can maintain the audit log in the background, 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 [2]. 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 10g[1] supports transaction-time tables with its workspace manager [13]. The Immortal DB project[2] aims to provide transaction time database support built into Microsoft SQL Server [10]. How this information is stored (in the log, in the relational store proper, in a separate "archival store" [1]) is not that critical in terms of forensic analysis, as long as previous tuples are accessible in some way.

- The second insight is that the data modified (inserted/updated/deleted) by a transaction can be cryptographically hashed to generate a secure one-way hash of the transaction.

[1] `http://www.oracle.com/technology/products/database/workspace_manager/index.html`

[2] `http://www.research.microsoft.com/research/db/immortaldb/`

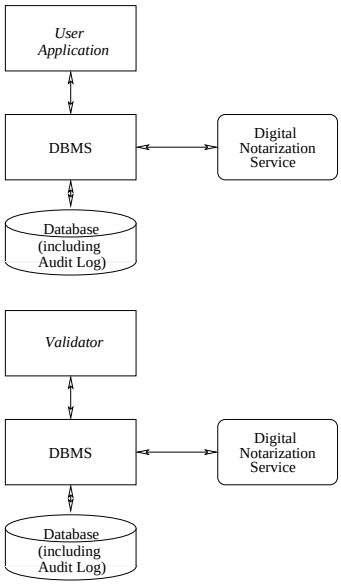

Figure 1: Normal Operation and Audit Log Validation

- The third insight is to digitally notarize this hash value with an external notarization service. So even if the intruder has full access to the database itself, the DBMS, and even the operating system and hardware, the intruder cannot change the hash value. This makes it exceedingly difficult to make a series of changes to the audit log that generate the same hash value.

- The final insight is a series of implementation optimizations that minimize notarization service interactions and speed up normal processing within the DBMS: opportunistic hashing, linked hashing, and a transaction ordering list, that in concert reduce the runtime overhead to just a few percent of the normal running time of a high-performance transaction processing system. 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 [15], which presents the threat model used by this approach, discusses performance issues, and clarifies the role of the external notarization service.

This basic approach differentiates two execution phases: *normal processing*, in which transactions are run and hash values are digitally notarized, and *validation*, in which the hash values are recomputed and compared with that previous notarized. It is during validation that tampering is detected, when the just-computed hash value doesn't match those previously notarized. Figure 1 illustrates these two phases.

Initially, our database is running fine, processing many transactions per second. Periodically, say every night at midnight, it sends a hash value to the digital notarization service, receiving back a notarization ID that it inserts into the hash sequence. At some point, we decide to run the validator (which, by the way, can be an ordinary user application, though we have to be a little careful, so that a potential intruder doesn't have control over it). The validator, to our mortification, reports that our database has been tampered. The DBA or Chief Information Officer (CIO) is contacted, along with the Chief Security Officer (CSO), and forensic analysis is initiated. Additional steps might include lockdown of the facility, temporary tightening of the firewall, and other steps to limit subsequent damage to the database.

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, 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 paper to examine how to perform a forensic analysis of a detected tampering of the database.

3. SOME DEFINITIONS

Let's now get more precise. We have just detected a *corruption event* (or *CE*), which is any event that corrupts the data and compromises the database. 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 (*NS*), termed a *validation event* (or *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. 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 NS returns *true*, stating that everything is OK: the data has not been tampered.

The validator compares the hash value it computes over the data with the hash value that was previously notarized. A *notarization event* (or *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.

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. The finest granularity of the corruption data locus would be an explicit attribute of a tuple, or a particular timestamp attribute. We term this data that has been corrupted the *corruption locus data* (l_c).

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_c's (and hence, many t_l's) 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*.

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 paper.

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. There is a lot going on in this diagram, but 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 [7], the x-axis represents the transaction time of the data.) Hence, 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

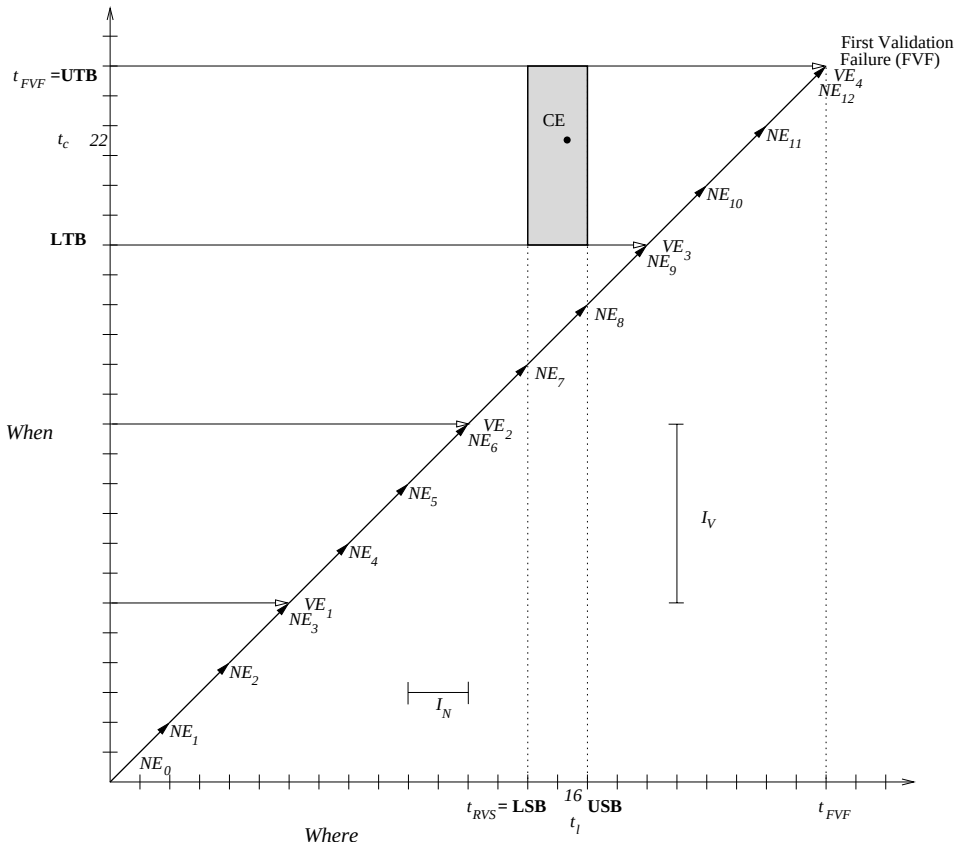

Figure 2: The corruption diagram for a data-only single-locus corruption event

marking each midnight, or, more accurately, the time of the last transaction to commit before midnight.

A 45-degree line is shown, terminating at the point labeled "FVF." This is the validation event at which we first became aware of tampering. (Hence the name: a corruption diagram always terminates at the *VE* that detected the CE.) We term this the *time of first validation failure* (or t_{FVF}): the time at which the corruption of the log is first detected. Note that t_{FVF} is an instance of a t_v. Also note that in every corruption diagram, t_{FVF} coincides with the current time.

The 45-degree line is termed the *action axis*, as all the action in the database occurs on this line. So for example, the *VE* associated with t_{FVF} occurs on the action axis, at its terminus. (This 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 axis, 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 timeline) of the database, labeled in time instants. Any point on the action axis 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 [8]. For this reason, the action axis 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 axis 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 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 oc-

curs 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 axis). 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} notarizes is computed from all the transactions that committed over the previous 24 days.

Also along the action axis 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, as 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. (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.) 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 24^{th} day, and corrupted data stored sometime during those twenty four days.

5. FORENSIC ANALYSIS

Once the corruption has been detected, a *forensic analyzer* 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 this 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 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 CIO and CSO and their staffs 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 paper 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, 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 RGB and polychromatic algorithms introduced later.

The most recent *VE* before FVF, VE_3, occurred at the *time of most recent validation success* (t_{RVS}), as shown on the where axis. This implies that the corruption event occurred *after* the RVS, that is, $t_c > t_{RVS}$. The corruption event occurred before the most recent validation event, the first validation failure, that is, $t_c < t_{FVF}$. Thus the times of *VE*s bound the "when."

To bound the "where" the monochromatic analyzer 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 start time earlier than all other data), and proceeding up to the last transaction before NE_1 (the transaction timestamps stored in the tuples indicate when these tuples should be hashed). 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 for a notarization check.

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

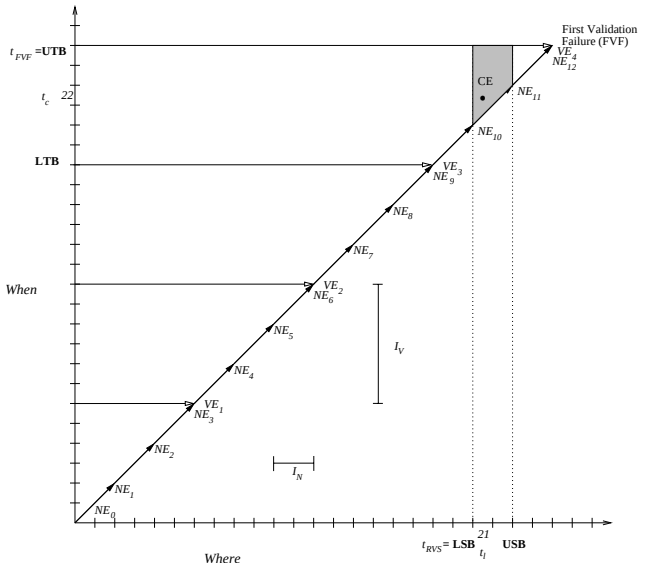

Figure 3: The corruption diagram for a data-only single-locus introactive corruption event

16. Note also that all validations after that, NE_9 through NE_{11}, also fail.

Define the *lower temporal bound* **LTB** $:= t_{FVF} - I_V$, the *upper temporal bound* **UTB** $:= t_{FVF}$, the *lower spatial bound* **LSB** $:= t_{RVS}$, and the *upper spatial bound* **USB** $:= t_{RVS} + I_N$. This defines a corruption region, indicated here 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."

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 axis.

Recall that a CE with locus time t_l appearing after at least one future validation event doesn't make sense. This last case should not be confused with the case where the corruption changes time-related data (timestamps) so that they appear to have happened in the future (see the next section for this scenario). Due to the way t_l is defined, it only depends on the true (original) time of the transaction, not the time it was changed to.

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_2. VE_2 would hash through Friday night; tampering on Tuesday *would* then be detected. Hence, we see that a non-aligned 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, 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, periodically be simultaneous with notarization. When the two do align, validation should occur immediately after notarization. Thus we will speak of the *validation factor* V such that $I_V = V \cdot I_N$.

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. In a *backdating corruption event*, one or more timestamps are 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 one or more timestamps to indicate a future time/date with respect to the original time in the tuple, with the *postdating time* (t_p) being the time a timestamp was postdated to. It is always the case that $t_l < t_p$. This induces six specific corruption event types.

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

For backdating corruption events, we ask that the forensic analysis determine, to the extent possible, "when" (t_c),

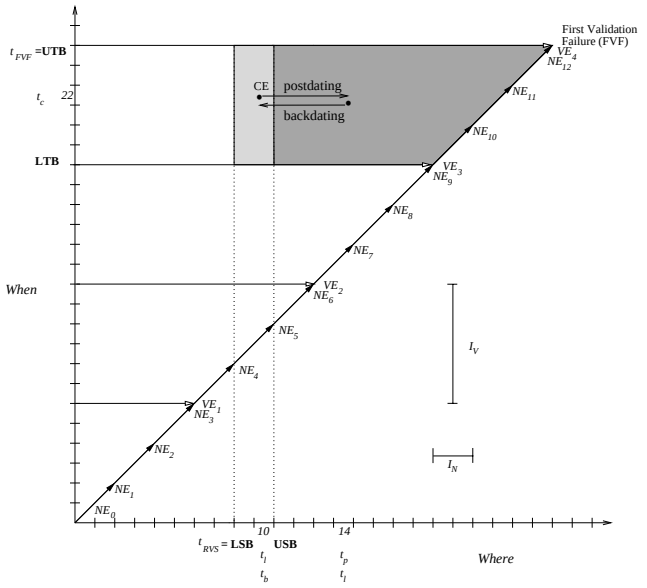

Figure 4: The corruption diagram for postdating and backdating corruption events

"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. 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. The monochromatic algorithm will then go back and rehash the database, querying with the notarization service at NE_0, NE_1 ... 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. Going back and rehashing the data at NE_0, NE_1, ... 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 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. THE RGB FORENSIC ALGORITHM

The central insight of the monochromatic algorithm was that the data could be rehashed when tampering was detected, and the hash value(s) sent to the notarization service to be checked. In fact, this could be done by the forensic analyzer at every notarization event, thus bounding the "where" to within one notarization interval (in the above examples, a particular two days). Thus, extra work during the last validation event, the one that detected the tampering, provides us with a corruption region. However, we also saw that we couldn't tightly bound *both* t_l and t_b or t_p.

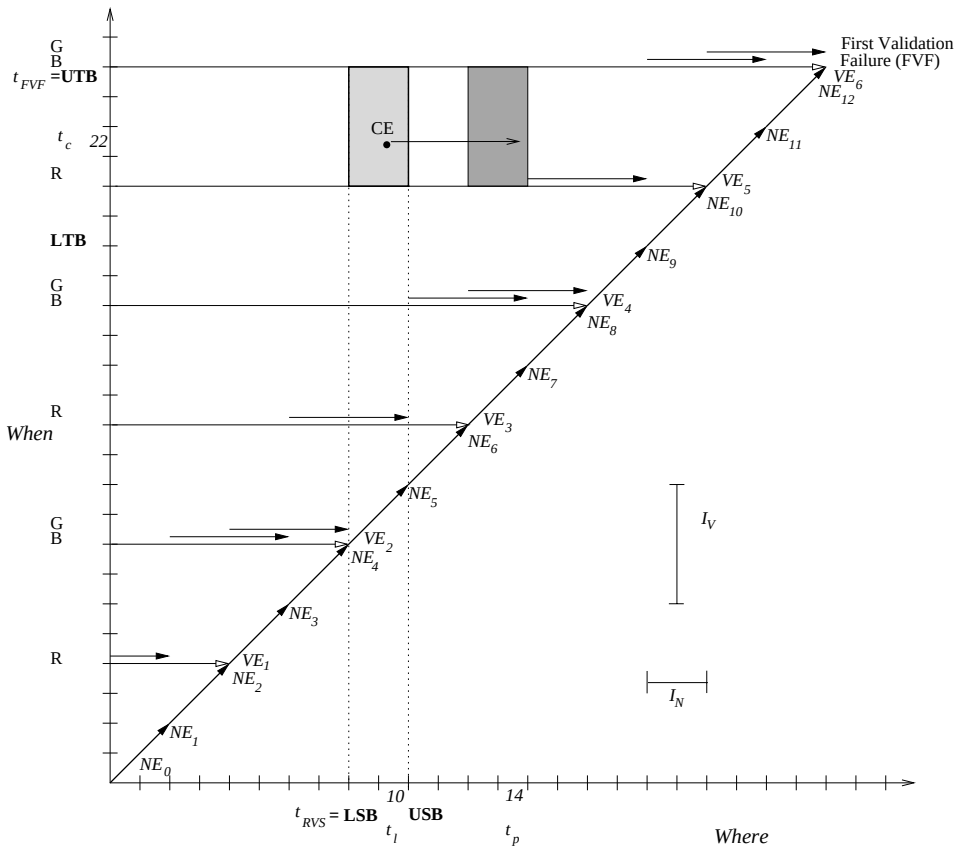

Figure 5: The corruption diagram for a postdating corruption event using the RGB algorithm

The insight of the *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, we add three additional chains, 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.

Figure 5 illustrates a postdating corruption event with the additional chains notarized during the validation events. Note that the validation factor has been changed to 2: $I_V = 2 \cdot I_N$ (this is required by the algorithm). The postdating corruption is the same as with Figure 4: 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.

The RGB algorithm adds several hash chains to the corruption diagram. For odd i, VE_i computes a *red* hash chain from $NE_{2 \cdot i-3}$ to $NE_{2 \cdot i-1}$. Hence, VE_1 computes a red hash chain over NE_0–NE_1 (a short chain), VE_3 over NE_3–NE_5, and VE_5 over NE_7–NE_9. (Note the "R" on the far left designating the red hash chains.) The resulting hash value is notarized with the digital notarization service. During forensic analysis, this value can be checked. This is shown in the corruption diagram with a black arrowhead for the red hash chains.

For even i, two hash chains are computed: a *blue* chain over $NE_{2 \cdot i-3}$–$NE_{2 \cdot i-1}$ and a *green* chain over $NE_{2 \cdot i-2}$–$NE_{2 \cdot i}$. VE_2 computes a blue chain over NE_1–NE_3 and a green chain over NE_2–NE_4, VE_4 blue over NE_5–NE_7 and green over NE_6–NE_8, and VE_6 blue over NE_9–NE_{11} and green over NE_{10}–NE_{12}. All of these are black arrows, as they represent additional notarizations.

There is a distinct pattern here. The original black hash chain covers the "where" axis, as do the red and blue chains in combination. The black chain gets longer and longer, whereas the other three are of fixed length of $I_V = 2 \cdot I_N$. The green hash chain covers the last two notarizations before this validation event. The monochromatic algorithm computes one hash value, and hence performs one query, per validation; the RGB algorithm performs that query as well as 1.5 additional notarizations per validation on average.

Using only the original black hash chain, the RGB algorithm can compute the (t_c,t_l) corruption region, which is the left shaded region. To compute t_p, the other three chains are used. The query for the red chain at VE_1 succeeds (denoted $Red_1 = T$). Similarly, the blue and green queries at VE_2 succeed ($Blue_2 = Green_2 = T$). But $Red_3 = F$ because a

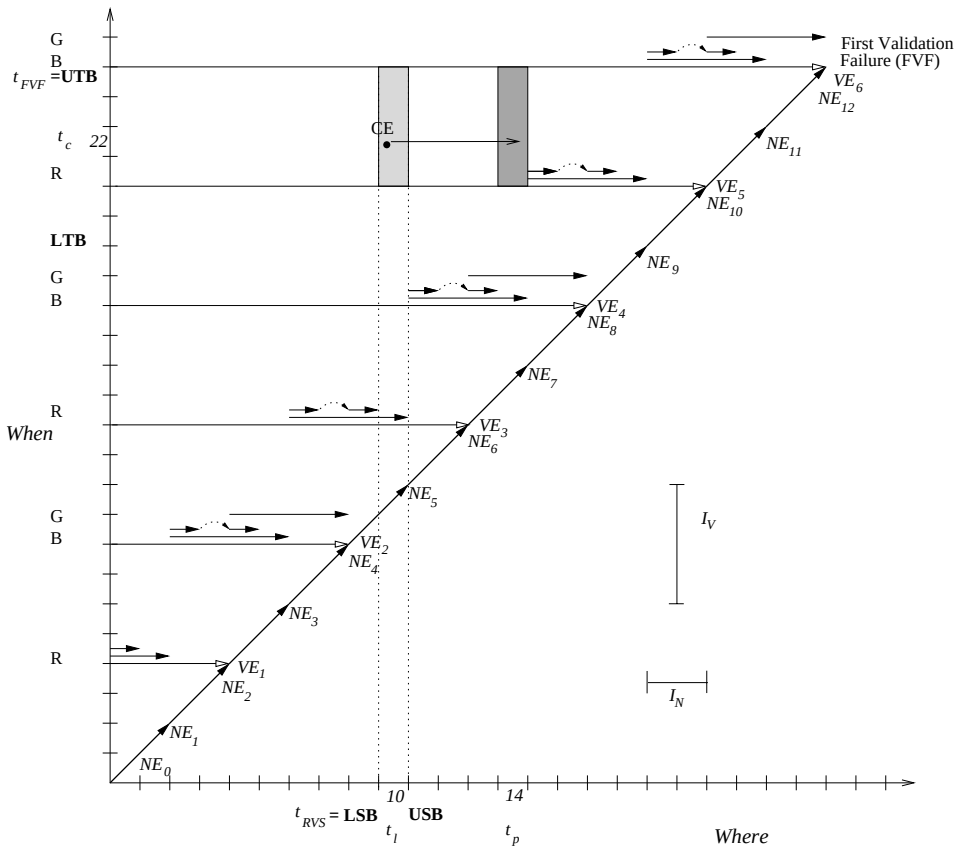

Figure 6: The corruption diagram for a postdating corruption event using the polychromatic algorithm

tuple originally stored during NE_3–NE_5 has been tampered (recall that this tampering is a postdating corruption event, where the timestamp of a tuple originally stored during day 10 is changed to day 14). Similarly, $Blue_4 = Green_4 = F$ because a tuple was added to those transactions on day 14. Finally, $Red_5 = Blue_6 = \text{Green}_6 = T$.

In this case, the red failure doesn't tell us anything, as we already know that t_l is on day 9 or day 10. But the blue and green failures tell us that t_p must be on day 13 or day 14, hence the t_p shaded region on the right. Both regions will each be of area $I_V \cdot I_N = 8$ days.

Had the corruption postdated the tuple to day 15, then $Blue_4 = T$ and $Green_4 = Red_5 = F$, and the combination of green and red would have narrowed t_p to two days, day 15 or day 16. (Note that t_l and t_p must be in different validation periods.)

A similar analysis applies to backdating. In fact, the algorithm is not able to differentiate postdating from backdating: either way, two regions will be obtained. For data-only single-locus corruption events, only one region will result from the analysis.

9. THE POLYCHROMATIC ALGORITHM

Recall that the red and blue chains in concert cover the "where" axis. They allow the uncertainty of t_p to be reduced to I_N, or two days. The polychromatic algorithm extends these partial hash chains to shrink the corruption regions. Specifically, with k partial red and blue chains, the uncertainty can be reduced to $I_N/2^k$. In Figure 6, we add one red chain Red^1 and one blue chain $Blue^1$ to reduce the uncertainty to one day. While a day may comprise many transactions and even more tuples, knowing which particular day was corrupted may provide valuable context for identifying the intruder.

The first additional red chain, Red_1^1, extends for just the first day. (We'll denote the original red hash chain as Red_1^0.) All the other additional chains will be composed of two separate days, linked together. The first additional blue chain, $Blue_2^1$, is composed of days 3 and 5; the linking is denoted with a dotted curved arrow. So VE_2 will link all the transactions occurring on day 3 with all the transactions occurring on day 5, computing a single hash value which is then notarized with the digital notarization service. Similarly, Red_3^1 will link day 7 and day 9, and $Blue_4^1$ will link day 11 and 13, and so on. The net effect is that each validation event causes 2.5 notarizations.

Once we establish a rough bound on the "where," t_l, using the black chains (here, within the interval I_N, or day 9 or day 10), these additional hash chains enable t_l to be determined to within one day. The algorithm computes $Red_3^0 = F$ and $Red_3^1 = T$, thereby locating t_l to day 10. The algorithm computes $Green_4 = F$ and $Blue_4^0 = F$ and $Blue_4^1 = T$ to establish t_p on day 14.

In the general case, let $I_V = 2^k$ days. (I_N can be any smaller power of two number of days.) The polychromatic algorithm requires an additional $k-1$ red hash chains and an equal number of blue hash chains to reduce the width of the corruption region to one granule (day). In the above example, $I_V = 4$ and two red hash chains are required. The cost is $0.5 + k$ notarizations per validation event.

The polychromatic algorithm needs to determine the day in which all the hash chains overlapping that day fail. We saw in the above example that for day 10 the hash chains were $Black_5$ (from NE_5) and Red_3^0 from VE_3 and for day 14 the relevant hash chains were $Blue_4^0$ and $Green_4$, both from VE_4.

We'll first define the RGB algorithm and then add the red and blue partial chains for the polychromatic algorithm. For a given day d, let $g = \lfloor (d-1)/I_V \rfloor + 1$. This partitions the x-axis into intervals of size I_V, aligned with the origin. The green hash chains are the even intervals. Similarly, $rb = \lfloor (d - 1 + I_V/2)/I_V \rfloor + 1$ partitions the x-axis also into intervals of size I_V, but offset by $I_V/2$, corresponding to the red and blue hash chains. Finally, the black hash chains are partitioned by $\lceil d/I_N \rceil$. Then t_l for data-only or postdating corruption events is within day d if the following predicate is satisfied.

$$Black_{\lceil d/I_N \rceil - 1} \wedge \neg Black_{\lceil d/I_N \rceil} \wedge (g \text{ is even} \Rightarrow \neg Green_g) \wedge (rb \text{ is odd} \Rightarrow \neg Red^0_{rb}) \wedge (rb \text{ is even} \Rightarrow \neg Blue^0_{rb})$$

This expression also finds t_b for backdating CEs, and t_p for postdating CEs and t_l for backdating CEs. For Figure 5, these predicates will be satisfied for days 9 and 10 for t_l and for days 13 and 14 for t_p. This completes the RGB algorithm.

For the partial chains maintained by the polychromatic algorithm, we first map the day into an integer between 0 and $I_V - 1$. Let $m = (d - 1 - I_V/2) \mod I_V$. The time is within day d if the following two predicates are also satisfied.

$$rb \text{ is odd} \Rightarrow \bigwedge_{i=1}^{k-1} \left(\left\lfloor \frac{m}{(I_V/2^{i+1})} \right\rfloor \text{ is even} \Rightarrow \neg Red^i_{rb} \right)$$

$$rb \text{ is even} \Rightarrow \bigwedge_{i=1}^{k-1} \left(\left\lfloor \frac{m}{(I_V/2^{i+1})} \right\rfloor \text{ is even} \Rightarrow \neg Blue^i_{rb} \right)$$

Since $I_V = 2^k$, the Red^{k-1} or $Blue^{k-1}$ partial hash chains contain single-day segments. Thus, only one day will satisfy the three predicates above. For Figure 6, these predicates will be satisfied for day 10 for t_l and day 14 for t_p.

10. COMPARISON

In this paper we have defined four forensic analysis algorithms.

Trivial: the corruption and backdating/postdating date is the entire upper-left triangle.

Monochromatic: The corruption region is reduced to $I_V \cdot I_N$ days, at the cost of additional queries during forensic analysis, but the postdating date could be anywhere to the right of t_l.

RGB: The backdating/postdating date is known to within I_N days, at the cost of additional partial hash chain notarizations during normal processing and additional queries during forensic analysis.

Polychromatic: This algorithm ensures that t_l and t_p (or t_b) are known to one day (or hour or second) at the cost of $k-1$ additional partial hash chain notarizations during normal processing and additional queries during forensic analysis.

More effort during normal processing and forensic analysis yields higher precision. This is useful, as it gives the CSO a knob to vary the cost and precision. Note that these algorithms do not differ on the times that they access the database: all four require that the entire database be read once during each notarization event. Rather, these algorithms differ mainly in the number of hash values computed, as well as the number of notarizations and queries of the digital notarization service. As the latter two actions costs real money, it will be those that we tally.

In this section, we go further, by characterizing the forensic strength of each of these four algorithms. We focus on postdating corruption events, as those are the most difficult to analyze. We define *forensic strength* of an algorithm in terms of three rather disparate components: (1) the effort (work) of the forensic analysis (to estimate t_c, t_l, and t_p), (2) the region-area (a measure of the quality of the result of the algorithm and again, of t_c, t_l, and t_p), and (3) the uncertainty (an additional measure of quality). These components are in some way compensatory, in that with more effort, high quality (or smaller region or uncertainty) results. Indeed, for each successive algorithm, it turns out that the work increases and the uncertainty (or region area) decreases. If these two effects balance out for two candidate algorithms, the two algorithms will be considered to have similar forensic strength.

We chose to combine these three components as a product, which has the desirable property just stated. Our intuition is that in general, with twice the work, one would expect, for a similar forensic strength, twice the accuracy, that is, half the uncertainty (focusing just on these two components for the moment). Say for simplicity that the two components were roughly the same size, S. With an additive definition of forensic strength, the former algorithm would have a strength of $S + S = 2S$, while the latter algorithm would have a strength of $0.5S + 2S = 2.5S$. With our multiplicative definition, the former's strength is $S * S = S^2$ and the latter's, $0.5S * 2S = S^2$. Hence, the multiplicative definition better matches our intuition.

An alternative definition of forensic strength would be some kind of weighted sum or product of the components. A problem with any weighted scheme is that it introduces the complication of having to propose specific values for the weights, which makes the comparison more difficult. That said, other definitions of forensic strength may certainly be pertinent.

We define the *inverse forensic strength* as the product of (a) the normal processing, in units of number of notarizations and validations plus the cost of forensic analysis, in units

of number of queries on the notarization service, (b) the area of the corruption region, in units of days^2, and (c) the uncertainty of t_p, in units of days. However, the inverse forensic strength increases if the work or the uncertainty increases. So we define the forensic strength as the reciprocal of the inverse forensic strength. For convenience, we focus on the inverse forensic strength (*IFS*), which we want to minimize.

We'll characterize *IFS* as a function over D, the number of days before the first validation failure was detected, I_N, the notarization interval, and V, the validation factor ($V = I_V/I_N$) and consider the worst-case.

$$\begin{aligned} IFS(D, I_N, V) &= (NumNotarizes + ForensicAnalysis) \\ &\quad \cdot RegionArea \cdot PostdatingUncertainty \end{aligned}$$

For the trivial algorithm, a notarization occurs at each notarization event and a query occurs at each validation event; the three times are known only to be less than D.

$$\begin{aligned} IFS_{\text{trivial}} &= (\lceil D/I_N \rceil + \lceil D/(V \cdot I_N) \rceil) \cdot \frac{1}{2} D^2 \cdot D \\ &= O(D^4/I_N) \end{aligned}$$

In this and following complexity analyses, we assume that $D \gg I_V$ and $D \gg V$.

The monochromatic algorithm finds the first validation failure. To do so, it can use binary search on the notarization events, thus requiring $\lceil \lg(D/I_N) \rceil$ queries, while reducing the uncertainty of t_c to I_V and of t_l to I_N.

$$\begin{aligned} NumNotarizes_{\text{mono}} &= \lceil D/I_N \rceil + \lceil D/(V \cdot I_N) \rceil \\ ForensicAnalysis_{\text{mono}} &= \lceil \lg(D/I_N) \rceil \end{aligned}$$

$$\begin{aligned} IFS_{\text{mono}} &= (\lceil D/I_N \rceil + \lceil D/(V \cdot I_N) \rceil + \\ &\qquad \lceil \lg(D/I_N) \rceil) \cdot \\ &\qquad (I_N \cdot V \cdot I_N) \cdot D \\ &= O(V \cdot D^2 \cdot I_N) \end{aligned}$$

As $D^2/I_N \gg V \cdot I_N$, the monochromatic algorithm has strictly greater forensic strength.

The RGB algorithm adds red, blue, and green partial hash chains to reduce the uncertainty of t_p to I_N. It performs a binary search on the black hash chains to find the first one that failed and a linear search on the red, green, and blue partial hash chains to find the right region. The uncertainty region is now I_V by I_N and the uncertainty in the postdate is I_N.

$$\begin{aligned} NumNotarizes_{\text{RGB}} &= \lceil D/I_N \rceil + 1.5 \cdot \lceil D/(V \cdot I_N) \rceil \\ ForensicAnalysis_{\text{RGB}} &= \lceil \lg(D/I_N) \rceil + 2 \cdot \lceil D/(V \cdot I_N) \rceil \end{aligned}$$

$$\begin{aligned} IFS_{\text{RGB}} &= (\lceil D/I_N \rceil + 1.5 \cdot \lceil D/(V \cdot I_N) \rceil + \\ &\qquad \lceil \lg(D/I_N) \rceil + 2 \cdot \lceil D/(V \cdot I_N) \rceil) \cdot \\ &\qquad (I_N \cdot V \cdot I_N) \cdot I_N \\ &= O(V \cdot D \cdot I_N^2) \end{aligned}$$

As $D \gg I_N$, the RGB algorithm has strictly greater forensic strength than the monochromatic algorithm.

The polychromatic algorithm adds some queries during normal processing and linear search on the multiple red and blue partial hashes to get the uncertainties of t_l and t_p down to one day.

$$\begin{aligned} NumNotarizes_{\text{poly}} &= \lceil D/I_N \rceil + 0.5 \cdot \lceil D/(V \cdot I_N) \rceil + \\ &\qquad \lg(I_N) \cdot \lceil D/(V \cdot I_N) \rceil \\ ForensicAnalysis_{\text{poly}} &= \lceil \lg(D/I_N) \rceil + 2 \cdot \lceil D/(V \cdot I_N) \rceil + \\ &\qquad 2 \cdot (\lceil \lg(D/(V \cdot I_N)) \rceil) \end{aligned}$$

$$\begin{aligned} IFS_{\text{poly}} &= (\lceil D/I_N \rceil + 0.5 \cdot \lceil D/(V \cdot I_N) \rceil + \\ &\qquad \lg(I_N) \cdot \lceil D/(V \cdot I_N) \rceil + 2 \cdot \lceil \lg(D/(V \cdot I_N)) \rceil + \\ &\qquad \lceil D/I_N \rceil + 2 \cdot (\lceil \lg(D/(V \cdot I_N)) \rceil)) \cdot \\ &\qquad I_N \cdot V \cdot 1 \\ &= O((V + \lg(I_N)) \cdot D) \end{aligned}$$

As $V \cdot I_N^2 \gg V + lg(I_N)$, the polychromatic algorithm has strictly greater forensic strength than all of the other forensic analysis algorithms introduced here. The short summary is that while each successive algorithm adds extra work during both normal processing and forensic analysis, the resulting decrease in uncertainty more than counterbalances this extra work.

Given the cost functions for the three components for each of these algorithms, it is easy, for any given set of specific weights, to compute the forensic strength of each algorithm and to compare the various algorithms using an alternative definition of forensic strength, such as a weighted sum or product.

11. RELATED WORK, SUMMARY, AND FUTURE 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[3], the IBM System Storage DR series[4], and NetApp's SnapLock Compliance[5]. 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 [3, 11]. Both of these systems (as well as Ivy [12]) use cryptographic signatures so that programs reading a file can be assured that it has not been corrupted.

Hsu and Ong have proposed an end-to-end perspective to establishing trustworthy records, through a process they term *fossilization* [6]. The idea is that once a record is stored in the RMS, it is "cast in stone" and thus not modifiable. An

[3] `http://www.emc.com/products/systems/centera_ce.jsp`
[4] `http://www.ibm.com/servers/storage/disk/dr/`
[5] `http://www.netapp.com/products/software/snaplock.html`

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 [16]. Their approach utilizes the WORM property provided by the systems just listed: that the underlying storage system supports 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 extend this perspective by asserting that every tuple in a database is a record, to be managed. The challenge is two-fold. First, a record in a RMS is a heavy-weight object: each version is stored in a separate file within the file system. In a DBMS, a tuple is a light-weight 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.

In the present paper, we introduced corruption diagrams as a way of visualizing corruption events and forensic analysis algorithms. We presented four such algorithms, trivial, monochromatic, RGB, and polychromatic, showing through a formal forensic strength comparison 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. The polychromatic algorithm in particular is able to determine the "where", the "when," and the "to when" of a tampering quite precisely.

There is much left to do. First, a lower bound for forensic strength is needed. We conjecture that the lower bound is at least $\Omega(D)$. We're working to improve on the polychromatic algorithm, by improving its performance and by eliminating some simplifying assumptions. We are investigating multi-locus corruption events as well as complex corruption events that simultaneously tamper with data and timestamp values. We are extending the polychromatic algorithm to efficiently accommodate these multiple corruption events, while also enabling much finer resolution of detection. We are also considering how to differentiate backdating from postdating corruption events. We are developing a comprehensive cost function of the polychromatic forensic analysis algorithm that takes into account its space requirements and the time to compute the additional hash functions. We are implementing this algorithm to validate its cost function. Finally, we are examining the interaction between a transaction-time storage manager and an underlying magnetic-disk-based WORM storage (such as those discussed above). 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.

12. ACKNOWLEDGMENTS

NSF grants IIS-0100436, IIS-0415101, and EIA-0080123 and a grant from Microsoft provided partial support for this work. We thank the reviewers for their insightful questions.

13. REFERENCES

[1] I. Ahn and R. T. Snodgrass, "Partitioned Storage Structures for Temporal Databases," *Information Systems*, Vol. 13, No. 4, December 1988, pp. 369–391.

[2] J. Bair, M. Böhlen, C. S. Jensen, and R. T. Snodgrass, "Notions of Upward Compatibility of Temporal Query Languages," *Business Informatics (Wirtschafts Informatik)* 39(1):25–34, February, 1997.

[3] K. Fu, M. F. Kaashoek and D. Mazières, "Fast and secure distributed read-only file system," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pp. 181–196, October 2000.

[4] P. A. Gerr, B. Babineau, and P. C. Gordon, "Compliance: the effect on information management and the storage industry," Enterprise Storage Group Technical Report, May 2003.

[5] S. Haber and W. S. Stornetta, "How To Time-Stamp a Digital Document," *Journal of Cryptology* 3:99–111, 1999.

[6] W. W. Hsu and S. Ong, "Fossilization: A process for establishing truly trustworthy records," IBM Research report RJ 10331, 2004.

[7] C. S. Jensen and C. E. Dyreson (eds), "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), Springer-Verlag, pp. 367–405, 1998.

[8] C. S. Jensen and R. T. Snodgrass, "Temporal Specialization and Generalization," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 6, December 1994, pp. 954–974.

[9] Lab Compliance, `www.labcompliance.com/-e-signatures/overview.htm`, viewed November 14, 2005.

[10] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu, "Immortal DB: transaction time support for SQL server," in *Proceedings of the International ACM Conference on Management of Data (SIGMOD)*, pp. 939–941, June 2005.

[11] D. Mazières, M. Kaminsky, M. F. Kaashoek and E. Witchel, "Separating key management from file system security," in *Proceedings of the ACM Symposium on Operating Systems Principles*, pp. 124–139, December 1999.

[12] A. Muthitacharoen, R. Morris, T. M. Gil and B. Chen, "Ivy: A Read/Write Peer-to-Peer File System," in *Proceedings of USENIX Operating Systems Design and Implementation*, 2002.

[13] Oracle Corporation, "Oracle Database 10g Workspace Manager Overview," Oracle White Paper, May 2005.

[14] B. Schneier and J. Kelsey, "Secure Audit Logs to Support Computer Forensics," ACM *Transactions on Information and System Security* 2(2):159–196, May 1999.

[15] R. T. Snodgrass, S. S. Yao, and C. Collberg, "Tamper Detection in Audit Logs," in *Proceedings of the International Conference on Very Large Databases*, pp. 504–515, Toronto, Canada, September 2004.

[16] Q. Zhu and W. W. Hsu, "Fossilized Index: The Linchpin of Trustworthy Non-Alterable Electronic Records," in *Proceedings of the ACM International Conference on Management of Data*, pp. 395–406, Baltimore, Maryland, June 2005.