Efficient Audit-based Compliance for Relational Data Retention

UIUC Dept. of CS Tech Report UIUCDCS-R-2009-3044, March 2009

Ragib Hasan
rhasan@illinois.edu
University of Illinois at Urbana-Champaign

Marianne Winslett
winslett@illinois.edu
University of Illinois at Urbana-Champaign

Soumyadeb Mitra
soumyadeb@gmail.com
Data Domain

Abstract

The Sarbanes-Oxley Act inspired research on long-term high-integrity retention of business records, based on the long-term immutability guarantees that WORM storage servers offer for files. Researchers recently proposed a *Log-compliant DBMS Architecture* (LDA) that extends those immutability guarantees to relational tuples, using an approach that imposes a 10-20% performance penalty on TPC-C benchmark runs.

In this paper, we present the *transaction log on WORM* (TLOW) approach for supporting long-term immutability for relational tuples. TLOW incurs less than 1% runtime overhead on TPC-C benchmarks with Berkeley DB, which is much less than for LDA. TLOW requires no changes to the DBMS kernel, and audit time is comparable to that of LDA: 2.7% of transaction time, i.e. ten days for a yearly audit on the platform we used. We also introduce the *audit helper* (AH) add-on to TLOW, which decreases the cost of a yearly audit on our platform to two hours. We provide a proof of correctness for TLOW, which exposes a subtle threat. The proof also illustrates a non-obvious problem with LDA, which we show how to correct.

1 Introduction

The drumbeat of financial accounting scandals, from ENRON in 2000 to Satyam Infotech in 2008, has prompted the introduction of regulations intended to guarantee the integrity of business records. For example, Wall Street firms are subject to Securities and Exchange Commission Rule 17a-4, and all medium and large US public companies are subject to the Sarbanes-Oxley Act. These two regulations are intended to ensure the preservation of immutable copies of all business email, spreadsheets, instant messages, and reports during a multi-year mandatory retention period. The penalties for non-compliance include hefty fines and potential jail terms.

To help companies comply with these regulations, all major storage vendors (IBM, EMC, NetApp, HP, Sony, Toshiba, and more) sell so-called *WORM storage servers*, providing a version of write-once-read-many semantics for storing files. On such devices, files are *term-immutable*; that is, once they are committed to the device, they are read-only for the duration of a predeclared retention period. The WORM semantics imply that during its retention period, not even an insider with system administrator privileges can delete or alter a file. To help ensure that WORM storage servers are trustworthy, they run their own file server code and no user programs; they offer a narrow interface for users and administrators that returns an error message to file system requests that would violate term-immutability. WORM storage servers also have special backup facilities and clocks, to help prevent attacks that target those facilities. Some products also support eradication of files at the end of their retention periods and can enforce "litigation holds" to ensure that subpoenaed files are not destroyed.

Because they support term-immutability at the file level, WORM storage servers are not directly useful for providing term-immutability for fine-grained data such as database tuples. Researchers have noted that it is impractical to provide term-immutability by making every tuple a separate file, making a new copy of the database file on every update, or moving the DBMS functionality into the WORM storage server [16]. A DBMS that supports term-immutability must consist of untrusted code that communicates with the (trusted) WORM storage server over as narrow an interface as possible.

Recently, researchers have proposed a scheme for supporting term-immutability in databases [16]. Their approach, called the *log-consistent DBMS architecture* (LDA), turns every tuple insert, delete, and update request into the creation of a new version of the tuple. LDA stores a DB snapshot on WORM at audit time, then uses an additional log stored on WORM to record database modifications. LDA uses the log and snapshot at the next audit to check for tampered content in the current DB state. LDA has a small window of vulnerability (e.g., 5 minutes) between the time that a transaction commits and when its writes become term-immutable. Depending on the implementation, LDA requires small changes in the DBMS kernel, or none at all. LDA imposes 10-20% overhead in transaction throughput for TPC-C and, as discussed in Section 7, audits are slow.

In this paper, we propose a much more efficient architecture for supporting term-immutable relational tuples. Our *transaction log on WORM* (TLOW) architecture provides term-immutability for tuples with no changes to the DBMS kernel. Our main contributions are: (1) we provide a proof of correctness for TLOW, which illustrates subtle points concerning windows of vulnerability after a transaction commits, together with a small correctness problem with LDA that we show how to fix; (2) we use the TPC-C benchmark to show that TLOW imposes less than 1% overhead in transaction throughput; and (3) we introduce an *audit helper* (AH) add-on that reduces the cost of a yearly audit to just two hours on the platform we used, rather than the several days required for an LDA audit.

The rest of the paper is organized as follows: we discuss the threat model for long-term retention of data in compliance with the Sarbanes-Oxley Act and SEC Rule 17a-4 in Section 2. We present the TLOW architecture and its AH audit helper in Sections 3 and 5, with a proof of correctness for TLOW in Section 4. We discuss forensic analysis in Section 6, provide experimental results in Section 7, discuss related work in Section 8, and conclude in Section 9.

2 Threat Model

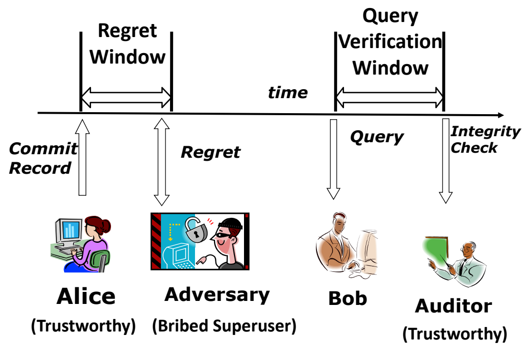


Figure 1: Threat model parameters. The *regret interval* is the minimum time interval between when a tuple is committed and when someone may wish to tamper with it. The *query validation interval* is the time gap between querying and validation of the query results.

Primary threat: forgery of history. We use the regret-based threat model for databases, as introduced in [16]. This threat model captures the way that falsification of historical records typically happens in the situations targeted by Sarbanes-Oxley and SEC Rule 17a-4. These regulations require the presence and enforcement of policies and procedures outside the DBMS to ensure that routine business records (email, financial transactions, and so) are captured at the time they are generated, and untampered software is used to process them. Once records are captured, the threat is that an adversary such as a bribed superuser will eventually regret the existence of one of these records, and alter or delete it. An adversary may also try to insert a backdated record. For example, an executive might request the insertion of phony backdated product orders, or retroactively change the quantities on existing product orders, to increase sales totals for the previous quarter. Our task is to foil these attacks when possible, and to ensure that such tampering can be detected when it does occur. When tampering is detected at an audit, regulations require the company to launch an investigation, with potential prosecution for those responsible for the tampering; thus adversaries want to perform undetectable tampering.

How to forge history. To forge history, an adversary Mala can tamper with the database file and/or log files (up to the limits, if any, imposed by the servers where the files reside). Since Mala may be a system administrator, she may be able to assume any user's identity and read, overwrite, append to, or delete any file, including database data, indexes, logs, and metadata, up to the limits imposed by the storage server where the files reside. She can change the database's contents with a file editor, or by using a non-compliant DBMS to overwrite the contents of the database. She can crash the DBMS or storage server. She can tamper with the execution environment, such as by tampering with the clock on the DBMS or storage server (within limits discussed below), and change any other environmental parameters.

Since Mala can take on anyone's identity, in theory she can successfully impersonate the DBMS when talking to a storage server, and issue any command during her impersonation. We assume that the DBMS and storage server prevent such attacks *on the transaction log, while the DBMS is up*. For example, they may use some combination of TPMs, mutual authentication, a secure communication channel, and non-advisory locking on the storage server side, so that only the DBMS can write to the transaction log while the DBMS is up. [1] We call this property *transaction integrity*; without it, Mala can append arbitrary material to the transaction log while the DBMS is running, so that the log contents no longer

[1]Alternatively, the DBMS might occasionally sign the log files that it has written, using a key that Mala cannot obtain. But for reasons that will become clear later, we prefer a solution approach that does not require scanning the transaction log.

faithfully mirror the intent of the executing transactions. Such attacks are not part of our threat model per se, as they cannot be used to forge history, but they are clearly undesirable in any enterprise, and they muddy the question of what a "correct" final state is. While the DBMS is down, Mala can append arbitrary material to the transaction log that refers to transactions active at the time of the crash, or to new transactions.

We assume that Mala cannot tamper with data while it resides in the DBMS page cache. Page cache attacks can be prevented by kernel patches that keep processes, even those owned by root, from getting read-write access to other processes' memory [1]. The only way to bypass such patches is to replace the kernel and reboot the DBMS server machine, which is hard to carry out without being detected by an audit. Thus we assume that when a new transaction arrives at the DBMS (even one submitted by Mala), the DBMS correctly executes it.

We assume that Mala does not block, alter, or significantly delay the messages sent between the DBMS and the storage servers, for the following reasons. First, we assume that Mala does not want to interfere with the ordinary, non-threatening transactions being processed by the DBMS, so she does not have a motive to block those packets. Second, as discussed below, we assume a nonzero regret interval at the DBMS level. This implies that Mala will not decide to delay packets for a regret interval, and then inspect them to see if she objects to their contents. Overall, the effect of these limits on Mala's tampering is that DBMS transactions that insert new tuples will be able to insert them and commit normally.

We assume negligible delays in communication between the DBMS and storage servers and in performing simple writes to storage, such as the appending of a new block to a log. In practice, a short delay (e.g., a second or less) is unlikely to cause a problem; but a lengthy delay will look like a DBMS crash to auditors, who will expect to find certain crash-specific information on the storage. If in fact no crash occurred, the absence of that information will look like a tampering attempt to the auditor.

Threats other than history forgery. In addition to history forgery attempts, the DBMS and storage servers are subject to all the traditional threats of ordinary life, and measures must be taken to deal with those as well. For example, the storage server must be backed up regularly. Mala can launch a denial-of-service attack against the DBMS. Mala can create any *new* applications that she likes, under any user identity, and these applications can submit transactions to the DBMS, which will faithfully execute them. For example, Mala might inappropriately alter the *current* state of the database, by giving herself a big raise. She can vandalize the database state and/or logs in an easily detectable manner, launch a cross-site scripting attack against the database, or tamper with the DBMS or application software. More generally, compliance is a many-layered endeavor: every layer of the system needs to have its own safeguards to protect against attacks at that level. Sarbanes-Oxley auditors require a risk-management-based approach to guarding against these attacks, and we assume that policies and procedures are in place for that purpose. For example, to address the problem of tampering with DBMS and application software, one can check signatures on the hashes of the executables [5, 23]. (Tampering with major software applications is difficult anyway, because source code is usually not available.) Controls and techniques outside the scope of this paper are needed to help guard against, detect, analyze, and clean up after these traditional attacks.

Trusted source of information about crashes. We assume that an auditor has a trustworthy source of information about the time at the WORM server that each crash or shutdown occurred since the previous audit, and the time that normal transaction processing began or resumed. We believe that this assumption is reasonable because the crash of a production system will be a visible inconvenience at the application layer, and applications can note the relevant times in an error log on WORM. The auditor does *not* need to know the times of crashes that occur during crash recovery; this is important, since such crashes are not easily observable at the application layer.

The regret interval. As shown in Figure 1, at some point an adversary starts to regret the current database state and wishes to tamper with it. The *regret interval* is the minimum time interval we can assume between when a tuple is committed to the database and an adversary tries to tamper with it. For post-hoc insertion of tuples, the regret interval is the minimum time interval we can assume between when a tuple was *not* committed to the database and when an adversary tries to insert it with a backdated start time. In current legal interpretations of email compliance, the regret interval is zero, meaning that email must be archived on WORM before it is delivered to its recipient. The architects of LDA argued that to ensure good DBMS performance, we need a non-zero regret interval [16]. Current practice in industry is to dump a snapshot of the database contents and current log files to WORM periodically, making the regret interval at least a day long. Thus, a regret interval of a few seconds or minutes at the DBMS level represents a significant advance over current industrial practice.

Although the DBMS regret interval will be non-zero, the application level may be able to enforce a zero regret interval, e.g., by sending all arriving invoices to WORM storage before entering them into the company's accounting system.

The query verification interval. The LDA authors also introduced the concept of a *query verification interval*, which is the interval between the time a transaction reads data and the time when we determine whether that data had been tampered with. For example, Mala may use a file editor to overwrite her salary in the DB, doubling it right before a legitimate application gives every employee a 10% raise. Then she can untamper the old version of the tuple, leaving the new version in place with its incorrect salary. Under LDA, the query verification window is always the time until the next audit. The effects of Mala's tampering may have spread far and wide before the next yearly audit, and LDA audits are so expensive that it is impractical to audit frequently. In this paper, we introduce techniques for high-performance auditing, which in turn make frequent informal audits practical.

Leveraging WORM storage servers. Companies rely on WORM storage servers to counter the threat of history forgery. While not completely tamperproof when faced with a determined and skilled system administrator with a screwdriver and physical access to the server, the servers do raise the bar quite high for tampering. Further, they cost approximately the same amount as ordinary storage servers, which is very important: the high cost of retaining all business records for (in most cases) seven years in a tamper-evident manner must be balanced against the benefit to society of increased accuracy in financial reporting and accountability for inaccurate financial reporting. Thus in this work, we also leverage WORM storage servers as our trusted computing base. We do require one extension to current interfaces, namely, the ability to append to term-immutable files in a particular directory or volume, from the time that they are created until the first time they are closed. After the first closure, the file is read-only for the remainder of its retention period. This simple extension makes it possible to append to a log file, while the older part of the same file is already term-immutable.

We do trust that the WORM server operates properly. Mala can log in as root on the WORM server, and perform any action permitted to her there, but no action of hers will cause the WORM server to overwrite or delete unexpired files or to append to non-appendable files. We trust that Mala cannot overcome the anti-tampering provisions of WORM server clocks (e.g., SnapLock's "Compliance Clock, a secure time mechanism"). We exploit our trust in the WORM server's clock, to provide a reality check on the commit time timestamps produced by the DBMS server. In particular, we require the DBMS and WORM storage servers to keep their clocks roughly synchronized, by limiting their drift to less than $r/2$ time units at all times, where r is the length of the regret interval.

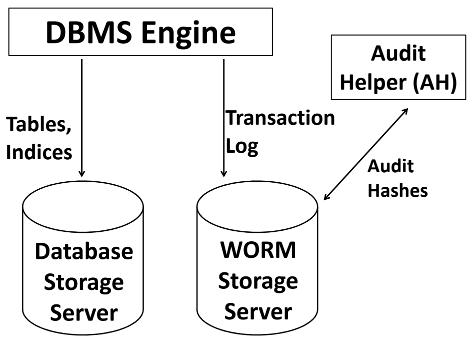
Leveraging transaction-time databases. An auditor must be able to distinguish between data tampering and legitimate modifications. For this reason, we retain all versions of a database tuple, using a layer of software above the DBMS that turns it into a transaction-time database [9, 12, 13, 19, 22]. In general, a transaction-time DBMS translates every tuple insertion, deletion, or modification into the insertion of a new version of the tuple, which is placed on the same page as the old tuple if possible. Unknown to legacy applications, each tuple has a *start time* attribute (or the equivalent), giving the commit time of the transaction that inserted the tuple. Deleted tuples are identified by a special end-of-life version, whose start time is the commit time of the transaction that deleted the tuple. Legacy applications can run on a transaction-time database with no changes, and see only the most recent version of each undeleted tuple. As needed, other applications can view past states of the database by including an extra clause in their queries, using standard temporal SQL constructs.

For a transaction-time database, the threats described above take the form of insertion of tuples with start times that already have passed, removal of tuples that have not yet expired, overwrites of existing versions of tuples, and updates to deleted tuples or tuple versions that are not the most recent.

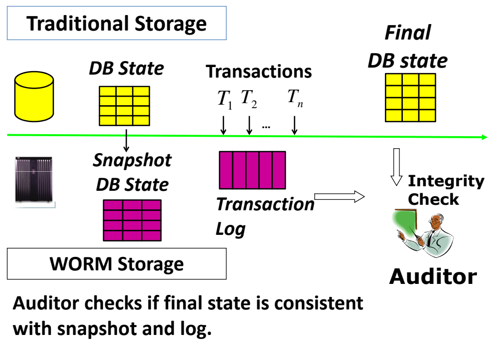
3 The Transaction Log on WORM (TLOW) DB Architecture

In designing a new architecture for supporting term-immutable tuples, our design goals were to preserve transaction throughput, minimize space overhead, avoid changes to the DBMS kernel, and make audits as fast as possible. The resulting *Transaction Log on WORM* (TLOW) DB architecture runs a transaction-time layer atop an ordinary DBMS. As shown in Figure 2(a), TLOW stores the current DB instance on ordinary storage and the transaction log on WORM storage. DBMSs already contain a function that can be called periodically to flush the current transaction log to disk, close its file, and open a new log file. For a regret interval of r time units (e.g., 1 minute), the DBMS must perform this operation once every $r/2$ time units. Depending on the DBMS, this can be accomplished by tuning the appropriate parameter, or by having a small application on the DBMS server platform that sleeps for $r/2$ time units, makes the call, and goes back to sleep.

Intuitively, it seems that TLOW DBs should have a zero regret interval: once the transaction log is on WORM, subsequent audits should work correctly. The proof of Theorem 4.1 shows that this is incorrect, unless the WORM and DBMS servers have perfectly synchronized clocks. The periodic creation of a new log file is



(a) Transaction-log-on-WORM (TLOW) DBMS architecture



(b) Audit process

Figure 2: TLOW DBMS architecture, and audit scheme. (a) TLOW architecture: the transaction log resides on WORM storage, along with a snapshot of the database contents as of the last audit. The current database contents are kept on ordinary storage. (b) The auditor uses $\mathcal{L}$ and the old snapshot to validate the current database state, then creates a new snapshot and discards the old one.

necessary for detecting clock tampering attacks on the DBMS server that can compromise the next audit.

Legacy applications can run on a TLOW DB with no changes whatsoever. All traditional SQL statements will work correctly, including deletions. New time-aware applications can query past states of the database by including an additional clause at the end of their SQL queries, describing the time point at which the queries should be run.

As shown in Figure 2(b), at the end of a successful audit, the auditor writes a snapshot of the current database state and a cryptographic hash of its contents to WORM storage, and signs them. The old transaction logs can be discarded at that point. For the next audit, the auditor checks the signature on the hash of the previous snapshot and generates new hashes from the current instance and from the log. Intuitively, the audit succeeds if the hash from the old DB snapshot D_o, plus the hash of all the new tuples introduced in the transaction log $\mathcal{L}$, is equal to the hash of the current instance D_c. Slightly abusing notation, we can write this as the following *tuple completeness condition:* [16]

$$H(D_c) = H(D_o) \,\cup\, H(\mathcal{L})$$

As does LDA, we use a cryptographically strong incremental hash function for sets that has the following properties.

- **Input**: The hash function H operates on a set $\{a_1, \ldots, a_n\}$.
- **Incremental**: Given a_n and $H(\{a_1, \ldots, a_{n-1}\})$, one can efficiently compute $H(\{a_1, \ldots, a_n\})$.
- **Commutative**: The value of H is independent of the order of the items in the set.
- **Cryptographically Secure** (Pre-image Resistant): Given a set $\{a_1, \ldots, a_n\}$, one cannot efficiently find $\{b_1, \ldots, b_m\}$ ($\neq \{a_1, \ldots, a_n\}$) such that
$$H(\{a_1, \ldots, a_n\}) = H(\{b_1, \ldots, b_m\}).$$

In our implementation, we used the ADD_HASH function proposed by Bellare and Micciancio [4]:

$$ADD\_HASH(a_1,\, \ldots,\, a_n) = \sum_{1 \le i \le n} h(a_1),$$

where h is a big (512 bits or more) secure one-way hash function and the sum is taken modulo a large number.

Using such a hash function, the auditor can incrementally compute a hash over $D_o \cup \mathcal{L}$ and D_c. Pre-image resistance ensures that $H(D_o \cup \mathcal{L}) = H(D_c)$ if and only if $D_s \cup \mathcal{L} = D_c$. Each hash operation takes $O(1)$ time. The completeness check requires a single pass over D_c and $\mathcal{L}$, plus a read of the signed hash of the previous snapshot. The total auditing cost, including the time to write out the new signed snapshot and signed hash, is $O(|D_c| + |\mathcal{L}|)$. This asymptotic complexity is the same as for LDA, but we will show how to reduce the audit cost below that of LDA by a factor of 100.

A *state reversion attack* occurs if Mala tampers with the values in the database, causing transactions to read incorrect data (and then possibly write incorrect data based on what they read), and then untampers the data before the next audit. Mala's salary-doubling tamper/untamper example in Section 2 is a state reversion attack; note that the tuple completeness check will not detect that her final salary value is incorrect. One solution to the state reversion attack is to adopt the hash-page-on-read approach originally proposed (for slightly different purposes) for LDA [16]; in essence, the DBMS logs a hash of the tuples on each data page read, and the auditor reconstructs the content that should have been on that page at that point, and computes its hash. This task is hard because a page that is read may contain timestamps instead of transaction IDs (explained below), and vice versa; may contain tuples from transactions that subsequently abort; and may split in two before the next audit. This approach is also expensive, imposing an approximate 10% overhead on TPC-C processing on the platform where we ran our experiments, plus over a day of audit time each year. While the audit helper approach presented in Section 5 can be used to shrink the audit costs to an acceptable level, the 10% TPC-C overhead remains. While moving the hashing functionality to the DBMS kernel can reduce the cost to perhaps 8%, we prefer not to complicate the kernel with a new task if possible.

The major question is whether the cost of perfect detection of state reversion attacks in public companies and Wall Street firms is worth the benefit it will bring to society in detecting and deterring fraud. With the suspicion that it is not, we propose a lower-cost probabilistic alternative that leverages the use of frequent internal audits, which in turn are made possible by our new audit techniques that run much faster. The key idea is that any state reversion attack that is underway will be detected by a tuple completeness check. If tuple completeness checks are performed frequently, e.g., as part of a daily backup process, we will detect most state reversion attacks, because most tampering needs to persist for a significant amount of time to be effective. For example, a phony backdated purchase order will need to be left in place for an extended period of time, so that it shows up in all accounting runs for that fiscal period.[2]

This approach will not detect any state reversion attacks that are launched and completed before the next tuple completeness check. For example, if Mala doubles and then reverts her salary, all on the day that raises are given out, we will probably not detect her attack. Note also that a state reversion attack can only be carried out by a highly skilled user such as a DBA who can overwrite the DB state by invoking a non-compliant copy of the DBMS, or using a file editor. If this user is also in charge of the informal audit, then she has a conflict of interest. Separation-of-duty constraints should be employed to ensure that reports of tampering from informal audits will reach someone who is unlikely to be involved in carrying out a state reversion attack – though we can never rule out the possibility of collusion between Mala and the report recipient. All in all, policy-makers must decide whether this level of protection against state-reversion attacks is sufficient, or society needs the higher-cost hash-tuple-on-read approach.

[2]We are referring to backdating that can be detected at the DBMS level. Application-level backdating, such as an ordinary purchase order insertion that gives a date in the previous fiscal quarter, can be detected by an application-level audit that compares the DBMS commit timestamp on the purchase order tuples with the date given in the purchase order tuples themselves.

In addition to checking tuple completeness and (if desired) for state reversion attacks, the auditor must also verify the integrity of all indexes, database pages, and the transaction log, using the ordinary integrity-checking utilities included with a commercial DBMS. This includes checking that the slot pointers on each page are set up correctly, the tuples are in sorted order across the pages (if that is required), the different versions of a tuple are all threaded together in commit-time order (if the DBMS uses version threading; our implementation does not), and all other stored metadata is correct (the magic number on the page, the count of tuples on the page, etc.). The auditor must also check that all indexes are set up properly and point to the appropriate tuples; otherwise, a tuple can be hidden from queries by suitable manipulations of the indexes.

For performance reasons, a transaction-time DB often uses the transaction ID T as a temporary commit time value in a tuple, and does a lazy update of the commit time later [12, 13]. Once T has committed, the next time that any page it wrote goes to disk, all occurrences of T's transaction ID on that page will be replaced by T's timestamp. Since some pages that T wrote may already be on disk when T commits, those pages will not have their timestamps filled in until the next time they are read into memory, which may occur much later. All must be filled in by the next audit, or the audit will fail; if needed, they can be filled in during the scan of the current instance during audit. If a timestamp is not filled in before one regret interval has passed, then Mala may wish to tamper with the timestamp, but we will prove that such an attack is detected at the next audit. Lazy timestamping may cause a tuple in $\mathcal{L}$ to contain a transaction ID rather than a commit time.

A transaction T may overwrite the same tuple t several times. For example, T may have several SQL statements that all update t. In a transaction-time database, these three updates result in only one new version of the tuple, rather than three new versions. After each of these writes, t's dirty page may be sent back to disk when the DBMS's buffers fill up, only to be brought into memory again. The auditor must hash only the final values written by T.

TLOW assumes ARIES-style logging and recovery. If transaction processing is interrupted by a crash, the DBMS will not accept new transactions until crash recovery has finished. To recover from the crash, the DBMS performs normal UNDO/REDO recovery. In most implementations, this involves first redoing all committed transactions (oldest first), then undoing all uncommitted transactions (newest first). The final effect is the same as if all transactions that had not yet committed at the time of the crash were aborted, and then transaction processing continued with the next transaction to arrive. If a crash occurs during recovery, the DBMS restarts recovery, as usual. Recovery must be completed before the next audit starts. The only compliance-related twist to recovery is that a new file for $\mathcal{L}$ must be created at the beginning of crash recovery, and again when the DBMS is restarted after normal shutdown or when it starts to accept new transactions after crash recovery.

Auditor details. Pseudocode for all auditor activities can be found at the end of the paper. The auditor's work begins with a variety of sanity checks on the contents of the log files; we focus here on those associated with crashes, which are described in the GlobalSanityCheck routine. Let k be the time of the first crash or shutdown that the auditor has not yet analyzed, and k' be the time when the DBMS starts accepting new transactions, taken from her trusted list of such times. A log file is part of the recovery log, written $\mathcal{R}$, if its life span [create time, last write time] falls entirely inside the interval $[k, k']$. The auditor ignores all $\mathcal{R}$ files during the remainder of the audit. We reserve the notation $\mathcal{L}$ for log files generated during normal transaction processing, outside of recovery.

Tampering is indicated by any log file whose lifespan *partially* overlaps $[k, k']$. More precisely, any log file whose last write time is after k and before k' and contains anything other than recovery-related information (e.g., CLRs) indicates tampering, as does any log file whose create time is before k' and whose last write time is after k'. For example, Mala may have appended additional records to an existing log file after a crash. If any of these conditions is violated, the audit fails, and forensic analysis should be performed to clean up the state before proceeding. We leave the design of forensic analysis routines to future work, along with their incorporation into the recovery and audit processes. After the sanity checks are complete, the auditor can continue on to the main tasks of the audit, being sure to ignore all $\mathcal{R}$ files.

The auditor parses the log records in $\mathcal{L}$ to find all the new tuples (more precisely, versions of tuples) inserted by transactions. For each such tuple t, she extracts its timestamp field ts. If ts is the timestamp of a transaction whose COMMIT record she has already scanned, or that committed before the last audit, then she ignores t. If ts is a transaction ID or timestamp that she has not seen before, she determines the corresponding transaction ID tid. She starts a new data structure for tid and records t's key (exclusive of timestamp) and content there. If she has seen ts before but not seen a corresponding COMMIT record, she adds t's key k (exclusive of timestamp) and content to the data structure for the tid corresponding to ts, overwriting any previous content stored there for k.

The auditor does not actually hash any of the new tuples for T until she sees the COMMIT record for T, which records T's transaction ID tid and timestamp ts. For recovery purposes, all of T's new tuples appear on $\mathcal{L}$ before that COMMIT record. Thus at that point, the auditor hashes all the new tuples she has recorded in the data structure for tid, and then deletes the data structure for tid. Once she has scanned past the COMMIT record for T, the auditor ignores T's subsequent writes to t in $\mathcal{L}$. If these updates change t's content (exclusive of timestamp), they are tampering attempts. If they only change t's timestamp, they are irrelevant because in her hash, the auditor only uses the timestamp she found in the COMMIT record for T. The auditor never hashes new tuples from transactions that abort; any tid data structures remaining at the end of her scan are for aborted transactions. When the auditor finishes scanning one file of $\mathcal{L}$, she moves on to the next file (in create time order).

At the end of its retention period, a tuple version may be shredded. We adopt the shredding approach proposed for LDA [16], which we do not discuss further in this paper.

4 Proof of Correctness for TLOW

Intuitively, two transaction-time databases are equivalent if they contain the same tuple versions, up to shuffling of timestamps. We must formalize this notion of "shuffling".

Definition The **timestamp normal** form of a transaction-time DB is created by replacing all occurrences of the smallest timestamp in the entire DB by 0, the next smallest by 1, and so on.

The sequence of timestamps created during normalization gives the serialization order of the transactions that originally created the DB.

Definition Two transaction-time DBs are **equivalent** if after timestamp normalization, both DBs contain the same set of tuples.

There can be several different serial orders of a set of transactions that all result in the same final state. However, for our purposes it suffices to consider a single serial order.

Definition Suppose that a database is created by running a set of transactions that commit in the order $T_1, \ldots, T_n$. The resulting database instance is **correct** if and only if it is equivalent to one obtained by running transactions $T_1, \ldots, T_n$ in serial order on an initially empty database.

Theorem 4.1. *Suppose that no crashes or state-reversion attacks occur. If a TLOW audit succeeds, then the current DB instance is correct.*

Proof. Because the audit succeeded, the auditor's sanity check of $\mathcal{L}$ (routine SanityCheck) must have succeeded. Thus there is at most one COMMIT record for each transaction. In addition, the timestamps for the COMMIT records in $\mathcal{L}$ must be in strictly increasing order. This serial order $T_1, \ldots, T_n$ is the one used to construct the correct final state.

Consider the case where no previous audits have taken place. The audit has succeeded, so the auditor's hash over the new tuples in $\mathcal{L}$ is the same as the hash over the current DB state. We must prove that the auditor's hash over the new tuples in $\mathcal{L}$ is also the same as a hash of the tuples obtained by running $T_1, \ldots, T_n$ serially on an initially empty database. If these two hashes are the same, then by the properties of the auditor's hash function, the current DB state is equivalent to a correct final state.

Suppose that the two hashes disagree. In that case, there must be one tuple in the correct final state that is not included in the auditor's hash over the new tuples in $\mathcal{L}$, or vice versa. Let us consider the first such tuple t, in the sense that it was written by transaction T_i, and no transaction T_j, where $j < i$ in the serialization order, wrote such a tuple.

First we consider the case where the extra tuple t is in the correct final state, but is not included in the auditor's hash over the new tuples in $\mathcal{L}$. Consider what happens when t's transaction T_i (which we will call T for simplicity) runs.

We trust the TLOW DBMS to carry out its functions in its usual manner, including concurrency control. Since T is the first transaction in serialization order to exhibit behavior that differs from that needed to obtain the correct final state, and no state-reversion attacks occur, all the transactions that committed before T must have written the correct final values for the tuples that they inserted. Otherwise the audit would have failed, due to a mismatch between the tampered final state and the new tuples in $\mathcal{L}$.

Further, because the DBMS carries out its concurrency control functions correctly, it would have aborted T if T had read any dirty values written by transactions that eventually aborted. Since T was not aborted, this problem did not occur. This means that T must have read correct values when it ran under the TLOW architecture, because no state-reversion attacks occur while it runs, so any tampering of the values it reads would have caused the audit to fail. More precisely, T will read the same values as if $T_1, \ldots, T_i = T$ were run one by one against an initially empty database, with no attacks. Since T reads the same values in both configurations, it will write the same final values in both cases.[3]

Recall that we trust the TLOW DBMS code to carry out insertions of new tuples in its usual manner. Thus when T writes a tuple, that write is promptly reflected in a dirty page in the DBMS buffer pool. Recall that we also trust the DBMS buffers; in other words, Mala does not tamper with the dirty pages while they sit in memory. Thus, since t is a final value written by T, the DBMS put a copy of a dirty page p containing t (or, depending on how the DBMS performs logging, a copy of the dirty parts of p, which must include t) on $\mathcal{L}$ before it wrote a COMMIT record for T to $\mathcal{L}$. Since T committed, that part of $\mathcal{L}$ has reached disk by the time of the audit. Thus the auditor will eventually scan p in $\mathcal{L}$. (If only the dirty parts of p are logged, the argument will be the same; so we argue only the case where p itself is logged.)

When the auditor parses p, she sees either t or else a tuple t' that is identical to t except that it contains a transaction ID T instead of T's timestamp ts. Next we argue that the auditor will include the new tuple t/t' for T in her hash appropriately.

Suppose that the auditor does not add t/t' to her data structure for T. The only possible reason is that she thinks that T has already committed. However, this cannot be the case, because the auditor's initial sanity check determined that there was only one COMMIT record for T. Further, according to the threat model, Mala does not regret any tuple that is not yet committed, so Mala has no reason to add a spurious COMMIT record for T to $\mathcal{L}$. We conclude that the auditor does add the key and content for t/t' to her data structure for T.

Because T does eventually commit, the auditor will eventually scan a COMMIT record for T and hash all of the tuples she has collected for T. The only possible reason for t to be omitted from the hash is if the auditor thinks that t is not a final value for T. In other words, she thinks that T overwrites the value of t, exclusive of key and timestamp. For the auditor to reach this conclusion, there must have been a page image p' that follows p on $\mathcal{L}$, precedes the COMMIT record for T, and contains a new value for t (exclusive of the timestamp). Either Mala appended this image to $\mathcal{L}$; or p went to disk, was tampered there, was read back in, was dirtied anew, and was sent to $\mathcal{L}$ again before T committed; or Mala created this image by tampering with the buffer pool. We trust the buffer pool, so this last attack is not possible. Let us consider Mala's motivation for carrying out one of the two other attacks.

According to the threat model, the first possible motivation is that Mala overwrote t because she regretted its existence. However, she cannot regret the existence of t

[3]We assume here that T does not make calls to the outside environment, e.g., to read the time of day, other than to obtain the transaction's timestamp.

before t is even committed, so she cannot have overwritten t for this reason before T committed. If she overwrote t after T committed, the auditor will have ignored the overwrite when computing her hash, and included t in her hash.

Second, at some point (call it time 0 on the WORM server) Mala may have regretted the absence of some other tuple t' from the database and wanted to create a backdated version of t' by tampering with t. For this attack to be effective, she must set back the DBMS server clock to the time she would like t' to be stamped with, which must be at least one regret interval in the past. (*Note that this precludes a zero regret interval.*) So that t' will be included in the auditor's hash, Mala must get a copy of t' onto $\mathcal{L}$, followed by a COMMIT record for t'. If Mala acts as fast as possible, she can write the COMMIT record for t' immediately after time 0 on the WORM server, with timestamp at most $-r$.

We trust the WORM storage server's clock, because Mala cannot tamper with it. Thus we trust the create time and last write time for each file of $\mathcal{L}$. Consider the file f of $\mathcal{L}$ in use at time 0 on the WORM server, i.e., at the point where Mala appends the page containing t' and the new COMMIT record. The create time of f must be greater than $-r/2$, because it is still in use after time 0, when Mala appends material to $\mathcal{L}$. The timestamp of the COMMIT record she appends is at most $-r$. Thus the auditor's sanity check will determine that the clocks on the WORM server and (presumably) the DBMS server differ by more than $r/2$ time units, and the audit will fail. Since the audit did not fail, Mala cannot have mounted such an attack. We conclude that the auditor will include the correct value for t in her hash in this case.

The LDA approach is also subject to DBMS clock tampering attacks, and its authors proposed to thwart them by flushing the LDA compliance log buffers to disk every r time units. The argument given above shows that the LDA approach needs to flush those buffers every $r/2$ time units, instead. TLOW also avoids the LDA requirement that the DBMS and WORM storage server clocks be roughly synchronized.

Another possibility is that Mala's goal in inserting the page image containing t' that overwrote t was to cause a side effect other than getting rid of t or backdating t'. In that case, her goal must have been to get rid of some other tuple s that was committed to the database at least one regret interval ago, or to create some backdated tuple s' that is not currently in the database. No matter what integrity constraints may be present in a transaction-time database, no insertion can cause the creation of another backdated tuple s'; as argued above, Mala cannot backdate new tuples by more than one regret interval. On the other hand, the insertion of t' can cause another tuple s to be deleted, if certain integrity constraints are present. However, s will not be *shredded*. It will still be visible to temporal queries. If Mala wants to get rid of s, she will have to overwrite it or shred it, and the bogus overwrite of t by t' does not help her toward that goal. The final possibility is that Mala wrote the record for reasons outside the scope of our threat model, in which case it is also outside the scope of this theorem. We conclude that the auditor does include t in her hash.

However, perhaps the auditor does not use the right timestamp for t in her hash. As discussed earlier, the auditor must find T's timestamp and substitute it into t before computing the hash. She will find T's correct timestamp in $\mathcal{L}$, because we trust the DBMS to insert new tuples correctly, we trust that Mala will not be able to alter the timestamp in the buffer pool, and because the auditor's $\mathcal{L}$ sanity check found only one timestamp for T. We conclude that the auditor does include t in her hash correctly, a contradiction of our initial assumption that t is excluded from her hash of the tuples inserted by T.

Now suppose that the extra tuple t is in the current DB state, but is not in the equivalent correct final state. Since the audit succeeded, t must have been included in the auditor's hash. Suppose that the copy of t used in the hash came from a page image p on $\mathcal{L}$, and p said that transaction T wrote t. Because t was included in the auditor's hash, a COMMIT record for T must follow p on $\mathcal{L}$ as well; otherwise T would not have been included in the serialization order. Since the auditor performed a sanity check on $\mathcal{L}$, $\mathcal{L}$ must have only one COMMIT record for T. Thus T qualifies as a "submitted transaction" in the theorem statement, and the correct final state must have included an invocation of T.[4] Suppose that $T = T_i$ in the serialization order. The COMMIT record for T tells the auditor what timestamp to use for t in her hash. Because the auditor included t in her hash, it must appear on $\mathcal{L}$ before the COMMIT record for T.

One possibility is that t is not in the equivalent correct final state because t was subsequently overwritten by T. In that case, the auditor did not recognize that the value for t on p was subsequently overwritten, and therefore included it in her hash. If the auditor overlooked the subsequent value, then one possibility is that it was not logged. However, this cannot happen, because we assume that the DBMS operates correctly and Mala does not tamper with the buffer pool.

A second possibility is that the subsequent value for T appears too late on $\mathcal{L}$ and the auditor ignored it. In this case, since we trust the DBMS to log pages correctly, Mala must have inserted a spurious COMMIT record for T. But then the auditor's sanity check would have observed two COMMIT records for T, and the audit would

[4]Note that without transaction integrity, T may include tuple insertions that originate from Mala, in addition to those intended by T's author.

have failed, which it did not.

A third possibility is that Mala altered the database state to contain t' instead of t, and did not tamper with $\mathcal{L}$. Since the audit did not fail, Mala must have reverted the page contents to their correct content before the audit. However, this violates our assumption that Mala did not carry out any state reversion attacks.

The only remaining possible reason that t is not in the equivalent correct final state is that Mala inserted the page image p for t on $\mathcal{L}$. According to our threat model, one possible reason for her behavior is that she wants to insert a copy of t that is backdated by at least one regret interval. However, the arguments given earlier show that although she can tamper with the DBMS server's clock, she cannot backdate t by an entire regret interval without detection. Another possibility is that t is overwriting another value written by T, and Mala regrets the existence of that other value. However, Mala cannot regret that other value, because it is not even committed yet. Further, the argument given earlier shows that Mala cannot turn back the DBMS clock far enough to successfully overwrite t. A third possibility is that the insertion of t will cause a side effect that meets Mala's needs, by creating a tuple s that is backdated by at least one regret interval, or by overwriting or shredding an existing tuple. As described earlier, these side effects are impossible. Nor can the spurious page image on $\mathcal{L}$ cause any ongoing or previous transaction to abort, as the DBMS will not be aware of the page image. We conclude that if Mala inserted such a page image, it was for reasons outside the scope of our threat model.

Now consider the case where k previous audits have occurred. Since audits are rare and major events, the auditor can consult a trusted third party to see who was responsible for the previous audit, and obtain the public key of that auditor. With that public key, the auditor can check the digital signature on the snapshot from the previous audit, to verify that the snapshot is the one produced by the previous auditor. Since the snapshot, its associated hash H, and the corresponding digital signature are on WORM storage in non-appendable files, none of them can have been tampered with since they were created. We trust the previous auditor to have correctly created the snapshot and its audit hash H_o, and correctly signed them. Thus the hash value stored with the previous snapshot is trustworthy. The auditor adds this hash value to that computed over $\mathcal{L}$. The remainder of the argument proceeds as for the first audit. □

Theorem 4.2. *Suppose that no state reversion attacks occur, the organization provides transaction integrity, and the auditor knows when each DBMS crash and shutdown occurred and when normal transaction processing started or resumed. If the audit succeeds, then the final database state is equivalent to a correct final state.*

Proof. We begin by presenting our high-level argument. Let $(T_1, \ldots, T_n)$ be the serialization order of the transactions that committed since the last audit.

1. Because the audit succeeded, the auditor's hash $H_{\mathcal{L}}$ over the transaction log files (omitting all $\mathcal{R}$ files) is the same as the auditor's hash H_{final} over the actual final database state.
2. Suppose that we remove all the $\mathcal{R}$ files and the records of any crashes, creating a no-crash version of the logs. Then we run the auditor's routine again. The auditor will compute the same hash value H_{final} as before; call it $H_{noCrashLogs}$. Let the *no-crash final database state* be the result of replaying the no-crash version of the logs. Assume for the moment that the no-crash final database state is the same as the final database state.
3. Since $H_{noCrashLogs}$ is equal to the hash over the no-crash final database state, the previous theorem tells us that the no-crash final database state is equivalent to that obtained by running the transactions in S one by one, starting with the snapshot produced by the previous audit.
4. Therefore the actual final database state is also equivalent to that obtained by running transactions $T_1, \ldots, T_n$ one by one on the snapshot produced by the previous audit. The theorem follows.

We have two remaining tasks: extracting the serialization order $T_1, \ldots, T_n$ and showing that a DBMS run that produced the no-crash version of the logs would have produced the actual final database state. We start by arguing that the auditor extracts the serialization order correctly.

The auditor's trusted information about crash/shutdown and start /restart times allows her to distinguish between $\mathcal{L}$ and $\mathcal{R}$ files, and discard any $\mathcal{L}$ files that Mala may have created before the DBMS resumed admitting new transactions. The auditor's sanity check has eliminated the possibility that any pair of $\mathcal{L}$ or $\mathcal{R}$ files have overlapping life spans, or that any $\mathcal{L}$ files were written to after a DBMS crash occurred but before the subsequent recovery was completed. The auditor's sanity checks ensure Mala cannot have appended a COMMIT record to any $\mathcal{L}$ file after a crash, and each transaction has at most one COMMIT record. Thus the $\mathcal{L}$ files do correctly show which transactions committed and what pages they intended to write. As argued in the previous theorem, the auditor will correctly extract the sequence of COMMIT records from the $\mathcal{L}$ files, giving her the equivalent serial ordering $T_1, \ldots, T_i, \ldots, T_n$ that should be used to construct the correct final state.

Because the organization provides transaction integrity and Mala cannot have appended a COMMIT record to an $\mathcal{L}$ file after a crash, all new tuples inserted by committed transactions were already on the log at the time of the crash. As argued in the proof of the previous theorem, the auditor will be able to extract those page writes from the log correctly.

As discussed in the proof of the previous theorem, the timestamps for transactions that committed within $r/2$ time units before a crash, according to their timestamps on $\mathcal{L}$, cannot differ from the corresponding WORM server clock times by more than r time units.

Our next task is to argue that the no-crash version of the logs could have produced the actual final database state.

Let N be the no-crash version of the sequence of log files, obtained by omitting all the $\mathcal{R}$ files based on the list of crash/restart times. If the auditor accepted the original set of log files without demanding forensic analysis, then she will also accept N, as its sanity checks are a subset of her original checks. Further, N writes the same final tuple values as did the original runs that produced the actual final database state. The final tuple values written by committed transactions are all that the auditor considers when computing the hash over a database state. Assuming that N could have been produced by an actual run, then the previous theorem applies, telling us that the actual final database state is equivalent to that produced by running $T_1, \ldots, T_n$ serially in isolation on the snapshot produced by the previous audit.

Finally, N could have been produced by an actual run, as follows. Suppose that during the original run, at each point where a crash occurred, instead the applications chose to abort their active transactions. If they did not submit any new transactions until the time when the DBMS came back up in the original run, then they could have produced exactly the log N and the corresponding final state. □

If the auditor does not know when crashes occurred or normal transaction processing resumed, then in the worst case, the auditor may not recognize that any crashes have occurred. The previous proof shows us what vulnerabilities this introduces. Suppose a crash occurred at time k. The set of vulnerable transactions is all those who had started but not committed or aborted by time k. Mala can choose whether to commit or abort each of them, and can add additional writes to those that do commit.

5 Speeding up Audits with an Audit Helper (AH)

Since audits are likely to be infrequent (e.g., once a year or once a quarter), $\mathcal{L}$ may be extremely long and the audit may be very slow. Our experiments presented in Section 7 verify this concern; for example, if our implementation of TLOW runs a 10-warehouse TPC-C around the clock for one year on the platform we used, then the audit will take 10 days to process the log, plus the time to scan the current database instance and verify its integrity. The instance is likely to be quite large too, typically containing seven years worth of versions of tuples.

To reduce these costs, we propose to do real-time incremental auditing. This approach relies on an audit helper (AH) process that can run on the DBMS server platform (or on its own host, as long as its host and the WORM server keep their clocks within $r/2$ time units of each other). The helper reads $\mathcal{L}$ from WORM as fast as $\mathcal{L}$ is written, finds the new tuples on $\mathcal{L}$ and hashes them in exactly the same manner as an auditor would, and periodically writes the hashes to $\mathcal{H}$ in a special form discussed later. The auditor can only trust helper hashes that reach WORM within r time units after their transaction committed. As shown by experiments presented later, AH has a very small impact on transaction throughput, because the log pages it reads are generally already in the file system cache on the DBMS server or WORM server, its computational overhead is quite small, and it imposes only a tiny write overhead on the WORM server.

If a helper's hash of a new tuple reaches WORM more than r time units after that tuple is committed, then the auditor cannot use that late hash and must recompute it. To help the auditor to identify late hashes, the helper flushes $\mathcal{H}$ to WORM every $r/2$ time units. More precisely, every $r/2$ time units, the helper appends ($hv\ tid_1\ ptr_1\ \cdots\ tid_n\ ptr_n$) to $\mathcal{H}$ and flushes it to a new file on WORM, where $tid_1 \cdots tid_n$ are the IDs of the transactions whose COMMIT records the helper has scanned since the last flush and whose commit timestamps are within the last $r/2$ time units; $ptr_1 \cdots ptr_n$ point to the COMMIT records in $\mathcal{L}$ for $tid_1 \cdots tid_n$, respectively; and hv is the hash of the new tuples inserted by $tid_1 \cdots tid_n$ (and nothing else). After flushing, the helper continues to scan $\mathcal{L}$.

Mala cannot overwrite an existing tuple version by attacking AH, but she can insert a backdated tuple t as follows. First, she tampers with the database instance so that it includes t. Then she kills AH and replaces it by a tampered audit helper, TAH. Suppose that transaction T commits while TAH is running. TAH hashes the tuples inserted by T plus t, and writes the result out to $\mathcal{H}$. The auditor will not recognize that the hash value for T includes the backdated tuple t.

Mala's attack can easily be detected by hashing the tuples in $\mathcal{L}$ and comparing them to the hashes stored in $\mathcal{L}$, but the whole point of AH is to entirely avoid scanning $\mathcal{L}$. The chance of catching an attack by spot-checking a randomly selected $\mathcal{H}$ files is very low. Thus to counter this attack, we require AH to sign its $\mathcal{H}$ files using a key that Mala cannot easily obtain. For this purpose, AH may rely on an inexpensive TPM on the server where it runs, plus the ability to prove that it is a legitimate copy of an audit helper (i.e., certified code). The TPM can contain a key seed, known to the auditor, that is used to generate a secret key k for each time epoch using a one-way hash function. AH can use a cryptographic hash function h_k to compute $h_k(k, i)$, where i is the tuple hashes that AH is writing to $\mathcal{H}$. Then AH can append $h_k(k, i)$ to $\mathcal{H}$. The auditor knows the seed, which allows her to generate k quickly. As she scans $\mathcal{H}$, she looks for and verifies the signatures (keyed hashes) that she encounters. TPMs do *not* provide perfect security for keying material or for guaranteeing that only certified code is running on a platform, but they raise the bar sufficiently high to deter almost all system administrators.

If the helper crashes, the DBMS can continue running, and vice versa. When the DBMS comes back up, it will start a new transaction log file, and the helper must watch for this file to appear. When the helper comes back up, it goes to the current file for $\mathcal{L}$, i.e., the log file started after the most recent DBMS boot. The helper starts scanning from the current end of $\mathcal{L}$, but ignores tuples from logged pages whose PrevLSN is non-null, unless the helper already has started a data structure for that transaction. This way, the helper will only hash new tuples from transactions that made their first write after the helper restarted.

At audit time, the auditor must perform its usual sanity checks on the create and last write times of each $\mathcal{L}$ file, and perform sanity checks on the $\mathcal{H}$ files as well. Any $\mathcal{H}$ files written more than $r/2$ time units after the DBMS crashed, and before it came up again, must be discarded. Next, the auditor examines each $\mathcal{H}$ file to find its pointers to COMMIT records in $\mathcal{L}$. (If there is no $\mathcal{H}$ file for a particular interval, or the file fails this sanity check, then the auditor must perform traditional auditing for all transactions active during that period; we do not discuss that task further.) The auditor must also make sure that the COMMIT records that $\mathcal{H}$ points to are legitimate, i.e., not inserted in $\mathcal{L}$ by Mala. This involves a sanity check that the last write time (resp. create time) on the $\mathcal{L}$ file containing each COMMIT record is less than $r/2$ time units after (resp. before) the transaction's timestamp recorded in the COMMIT record. As its third sanity check, the auditor compares the last write time of the $\mathcal{H}$ file on the WORM server to the earliest commit time for any transaction ID listed in that file. If the difference is more than r time units, the auditor must redo the hash of the new tuples for those transactions. If the auditor only needs to compute the hashes for a few transactions, it can quickly find the relevant information for each of them by following the LastLSN pointers for a transaction backwards through $\mathcal{L}$, starting from its COMMIT record. After computing a hash value over the new tuples of all the transactions for which it could not use an $\mathcal{H}$ file, the auditor then adds the hash values from all the $\mathcal{H}$ files that it found to be trustworthy, to arrive at the final hash value for all of $\mathcal{L}$. The auditor goes on to compute the hashes over the current instance and previous snapshot, and compares them as under the TLOW approach.

One can also use an audit helper process to check that transactions read untampered data. We do not discuss the details here, but the essence is the same as for AH: move the auditor's tasks for hash-page-on-read [16] to the audit helper, and ensure that if the audit helper falls behind the DBMS, the next audit will detect the delay and will regenerate those hashes by scanning $\mathcal{L}$.

The other major cost of auditing is the time to scan and hash the current instance, which may become very large over the years. We adopt LDA's approach of migrating old versions of tuples to WORM, and using time-split B^+-trees to split nodes so that the index pages for historical tuples lie primarily on WORM [16]. Once the next audit has been completed, the pages on WORM can be excluded from subsequent audits, greatly reducing their cost. Another important advantage of migrating historical tuples to WORM is that the production database stays small and compact, so that transaction processing performance is not compromised by poor locality of current versions of tuples.

6 Basic Forensic Analysis

The TLOW architecture and AH allow an auditor or DBA to quickly determine whether the database instance has been tampered with. However, for forensic investigations, it is often important to pinpoint the particular tuples and times when the tampering occurred. We propose a solution to this problem.

When the audit fails, the forensic analysis process steps through each table of the previous snapshot, computing a new relation $Hashes$ that contains $(t.k, H(t))$ for each tuple t with key k and cryptographic hash $H(t)$. Then the forensic analysis process steps through $\mathcal{L}$, finding each new tuple t' and computing its hash $H(t')$. The forensic analysis finds the key of t' in $Hashes$ and changes the value $H(t)$ stored there to be $ADD\_HASH(H(t), H(t'))$. To make this fast, we can build a B^+-tree over the key attributes of $Hashes$ and use it to find each key as needed. Alternatively, we can scan the log and compute all the new $(t'.k, H(t'))$ values, sort them on k, and then do a zigzag join of them with $Hashes$, updating the content of $Hashes$ each time we find a matching tuple.

Finally, the forensic analysis considers each tuple in the current instance of the database, checking whether its hash is what is stored for its key k in $Hashes$, and marking the key k in $Hashes$ as having been matched with a key from the current instance. This process can

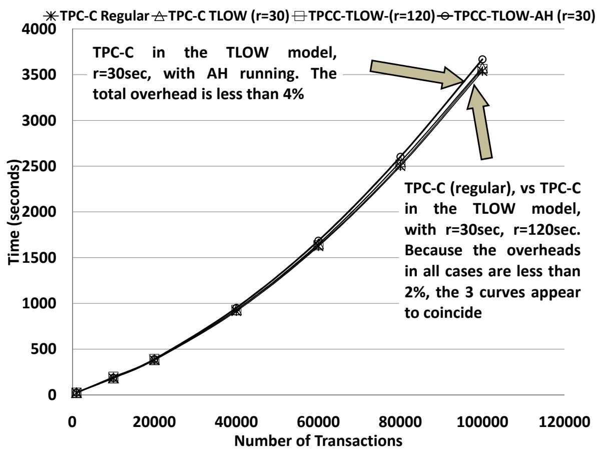


Figure 3: TPC-C run times with unmodified Berkeley DB, TLOW with regret interval 120 seconds, TLOW with regret interval 30 seconds, and TLOW with AH and a 30-second regret interval. As the overheads are less than 1% for TLOW-120, less than 2% for TLOW-30, and less than 4% with TLOW-30-AH, the curves almost coincide.

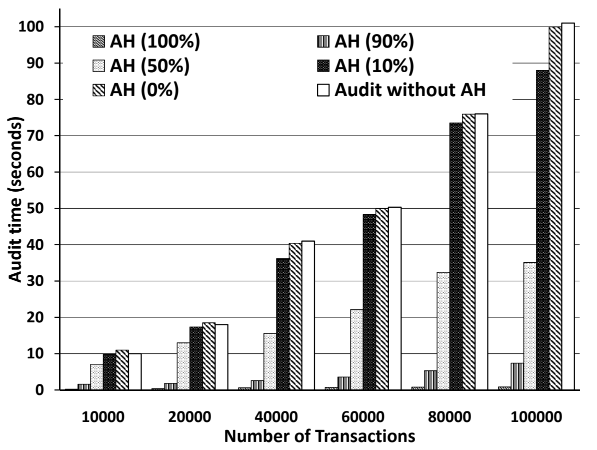


Figure 4: Audit time for the transaction log. Without any optimizations, the log file audit time is approximately 100 seconds for 100K TPC-C transactions (2.7% of transaction run time). With AH, the audit time drops to between 0.8 seconds (100% $\mathcal{H}$ files valid, 0.02% of transaction time) and 35 seconds (50% $\mathcal{H}$ files valid).

be sped up by using any available B$^+$-trees for the two relations. A mismatch in hash values indicates tampering, as does the absence of k in $Hashes$. At the end of the pass over the current instance, if any keys in $Hashes$ have not been matched, that also indicates tampering. If audit failures are common, then after repair and a successful re-audit, the auditor can sign $Hashes$ and store it on WORM for use during the next audit.

7 Empirical Evaluation

To evaluate TLOW performance, we used the Shore implementation of the industry standard TPC-C benchmark, ported to run on Berkeley DB version 4.7.25. The DBMS was hosted in a machine with a Pentium dual core 2.8 GHz processor, 512KB L2 cache, 4GB RAM, and a 1TB hard disk. We simulated the WORM server using a Pentium 2.8 GHz single core processor, 512KB L2 cache, with a portion of its local disks exported as an NFS volume. The DBMS mounted the WORM volume over NFS, and stored the logs there. We ran AH on the DBMS server. AH can be run on a separate server and still take advantage of the warm WORM server cache, but its overhead is so small that we left it on the DBMS server. All the machines ran on Linux with kernel version 2.6.11.

Our experiments ran 100,000 TPC-C transactions with a 512 MB DBMS cache and a 10 warehouse TPC-C configuration, resulting in a 2.5 GB database. We ensured that the file system caches on the WORM and DBMS servers were cold at the beginning of each run. For runs without support for term-immutability, we used the DBMS's default maximum log file size of 10MB. When this size is reached, the DBMS starts a new log file. For 100,000 TPC-C transactions in 10 warehouses, Berkeley DB created a total of 232 log files, of size 2.32 GB. On average, the log creation rate was approx. 38 MB/minute. For runs with support for term-immutability, every $r/2$ seconds we called a function supplied by Berkeley DB to flush and close the log file, and start a new log file.

We implemented the audit helper AH as a separate process that polls the log directory on the WORM server for files. When AH finds a new log file, it hashes the file's newly inserted tuples appropriately, and flushes the results to an $\mathcal{H}$ log file on WORM every $r/2$ seconds.

TLOW and AH TPC-C performance. We measured TPC-C performance with regret intervals r of 30 seconds and 2 minutes, with and without an audit helper AH. The resulting TPC-C run times are shown in Figure 3. With a 2 minute regret interval, we have less than 1% overhead in all cases up to 100,000 transactions. For a 30 second regret interval, the overhead is always less than 2%. In general, the faster the DBMS generates log files, the lower the overhead will be. The presence of AH added less than 1.5% overhead, and AH processed $\mathcal{L}$ files much faster than the DBMS generated them. As mentioned earlier, if policy-makers require detection of all state reversion attacks, the TPC-C overhead on our platform is approximately 10% [16]; we have not experimented with ways to potentially reduce this cost further. If policy-makers require probabilistic detection of state reversion attacks through frequent internal audits, this adds no overhead to the run times reported in Figure 3.

Audit tasks include the tuple completeness check, the check of the hashes of pages read by transactions (if policy-makers do not accept a probabilistic internal check for state-reversion attacks), and the integrity check on the current DB instance. The latter check should be performed at regular intervals in any case. The expensive steps of the tuple completeness check are hashing the new tuples in $\mathcal{L}$ and in the current DB instance. Figure 4 shows that the new tuples of 100K TPC-C transactions on $\mathcal{L}$ can be hashed in less than a second when $\mathcal{H}$ files are available for all transactions; if no $\mathcal{H}$ files are available, the hashing takes just over 100 seconds. As AH has no trouble keeping up with the DBMS, we expect that after a year, $\mathcal{H}$ files will be available for almost all committed transactions. This implies that the time to scan $\mathcal{H}$ files from one year of non-stop TPC-C on our platform will be just under two hours.

To generate the hash of the current instance, we propose to piggyback on the DB instance integrity check utility. This utility has to fetch and parse each page of the DB, so the additional cost of hashing each tuple and reporting the final hash is miniscule. On our platform, the page fetches are the major expense; the additional cost of parsing and hashing fetched pages after 100K transactions, given that the pages were already in the file system cache, was 8 seconds. Thus the time to parse and hash a year of TPC-C tuple versions is less than 20 hours on our platform, given that the pages were already being fetched from disk for another purpose, such as backup. As an instance containing 7 years of tuple versions will be very large, the WORM migration feature discussed in Section 3 should be used to move old versions to WORM, so that tuple versions migrated to WORM before the last audit do not need to be scanned and hashed. If old tuple versions are migrated to WORM once a month and omitted from subsequent informal audits, then the cost of parsing and hashing the remaining tuples will be quite modest.

We conclude that in practice, the tuple completeness check will be so affordable that the organization should be able to perform an informal audit whenever it runs a routine integrity check on the portion of the current database instance that is not on WORM, or even at every backup.

8 Related Work

Researchers have looked into several aspects of compliance for database data. We have already described LDA [16], which offers a different approach to the same problem that TLOW addresses. Another recent innovation is a framework for auditing changes to a database while respecting retention policies [14]. This approach focuses on policy based redaction and removal of sensitive information from the database and its history, and handling the uncertainties in answering audit queries from the resulting incomplete table and history. The TLOW approach for ensuring that database contents are term-immutable can be combined with this framework, so that one can support audit queries over sensitive information while guaranteeing that tampering with database contents or history, even by system administrators, can be detected.

Researchers have addressed the related problem of writing and enforcing tuple retention and shredding policies, expressed as restrictions on the operations that can be performed on views [3]. Their approach relies on the DBMS to enforce the policies. TLOW can augment this by protecting the database contents against tampering by adversaries who gain superuser access, or even insiders with incentives to tamper the data. For example, suppose that a skilled DBA opens the database file with a file editor or an uncertified copy of the DBMS, and removes or alters some of its content. TLOW can identify this tampering at the next audit.

Tamperproof database audit logs are another direction of research. In one scheme [20], the transactional data is cryptographically hashed by the DBMS, signed periodically by a trusted third-party notary, and then stored in the database. Later, a validator process verifies the current database state using the certified hashes. If tampering is detected, a forensic analyzer helps to identify when and where the database was tampered. One drawback of this approach is that newly added content can be tampered until the next notarization, without subsequent detection. As the notaries are trusted third parties, it is hard to shrink the notarization interval below, say, once a day. The TLOW approach shrinks this window of vulnerability to a minute or less, with minimal impact on transaction throughput.

Effective shredding of expired data is required by some regulations and is a corporate goal in many other situations. Researchers have proposed techniques for database scrubbing, i.e., removal of lingering traces of deleted tuples [21]. Such techniques can be used in conjunction with TLOW to provide more effective support for shredding of expired tuples.

Many researchers have tackled the security problems associated with outsourced database and file management [7, 8, 15, 18, 24], and cryptographic file systems [6, 10, 11, 17]. The high-level goals of outsourcing research are for the data owner to receive integrity guarantees for databases/files stored on untrusted servers, and correctness guarantees for DBMS/file system responses to user requests. Outsourcing research assumes that the data owner is trustworthy and will not tamper with the data, but the storage server is untrustworthy. In data retention scenarios, the WORM storage server is trusted, but we do not trust the data owner. The owner may have powerful incentives to tamper with the data, and may have administrator access to the database and its physical platform.

Data retention is just one important aspect of Sarbanes-Oxley compliance. Agrawal et al. [2] describe how databases can play an important role in helping companies comply with many other aspects of Sarbanes-Oxley, and present several open research problems related to this role.

9 Conclusion

In this paper, we proposed TLOW, a high-performance approach to supporting term-immutable databases for regulatory compliance. TLOW stores the current database instance on traditional storage and the transaction log on a low-cost WORM storage server; the transaction log is segmented in a manner that allows an auditor to detect a variety of attacks, including clock tampering. Our proof of correctness for TLOW sheds light on a variety of potential attacks, including one that causes trouble for a previously proposed approach.

We implemented a prototype of the TLOW architecture on top of Berkeley DB, by adding a process that forces it to switch log files twice during each regret interval, i.e., the maximum allowable period before a tuple must become term-immutable. Our experiments with TPC-C show that TLOW supports a regret interval as small as 2 minutes with less than 1% slowdown in transaction throughput. In other words, 2 minutes after a transaction commits a new tuple, the tuple becomes term-immutable, in the sense that we will be able to detect any subsequent attempts to tamper with it. This is a 5-fold performance improvement over the previous state of the art, and shows that TLOW is practical and efficient for OLTP applications that can employ a transaction-time approach, i.e., where one can keep all past versions of tuples.

Data tampering will be detected when the database is audited, which may take place once a year. Tampering is indicated by a mismatch between the current database instance, the contents of the transaction log, and the database instance that was in place during the previous audit. To detect tampering quickly, we use the incremental commutative hash function ADD_HASH that was used for that purpose in previous work. To make audits fast, we introduced an audit helper (AH) function that hashes new tuples on the transaction log as fast as they are written to disk, and saves the hashes on WORM. The auditor can use the hashes computed and saved by AH whenever the auditor determines that the tuple hashes reached WORM storage within one regret interval after the tuples were committed; otherwise, the auditor must hash those portions of the transaction log on its own. Our experiments show that AH has no significant impact on TPC-C performance, and speeds up audits enormously, reducing the cost of hashing the transaction log by a factor of 100. We also show how to reduce the cost of hashing the current database instance to just a few seconds, by piggybacking the task onto a periodic check of the integrity of the database contents. Finally, we show how to perform a tuple-level forensic analysis when tampering is detected.

References

[1] Windows kernel patch protection. `http://www.microsoft.com/whdc/driver/kernel/64bitpatch_FAQ.mspx`.

[2] R. Agrawal, C. Johnson, J. Kiernan, and F. Leymann. Taming compliance with Sarbanes-Oxley internal controls using database technology. In *Proc. of ICDE*, 2006.

[3] A. A. Ataullah, A. Aboulnaga, and F. W. Tompa. Records retention in relational database systems. In *Proc. of CIKM*, 2008.

[4] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Proc. of EUROCRYPT*, 1997.

[5] B. Chen and R. Morris. Certifying program execution with secure processors. In *Proc. of USENIX HotOS*, 2003.

[6] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *NDSS*. The Internet Society, 2003.

[7] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in database service provider model. In *Proc. of SIGMOD*, 2002.

[8] H. Hacigumus, S. Mehrotra, and B. Iyer. Providing database as a service. In *Proc. of ICDE*, 2002.

[9] C. S. Jensen and et al. The consensus glossary of temporal database concepts - February 1998 version. In *Temporal Databases*, pages 367–405, 1997.

[10] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proc. of USENIX FAST*, 2003.

[11] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. of OSDI*, 2004.

[12] D. Lomet, R. Barga, M. Mokbel, and G. Shegalov. Transaction time support inside a database engine. In *Proc. of ICDE*, 2006.

[13] D. Lomet and B. Salzberg. The performance of a multiversion access method. In *Proc. of SIGMOD*, 1990.

[14] W. Lu and G. Miklau. Auditing a database under retention restrictions. In *Proc. of ICDE*, 2009.

[15] G. Miklau and D. Suciu. Implementing a tamper-evident database system. In *Proc. of the Asian Computing Science Conference*, 2005.

[16] S. Mitra, M. Winslett, R. T. Snodgrass, S. Yaduvanshi, and S. Ambokar. An architecture for regulatory compliant databases. In *Proc. of ICDE*, 2009.

[17] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Trans. Inf. Syst. Secur.*, 2(2):159–176, 1999.

[18] R. Sion. Query execution assurance for outsourced databases. In *Proc. of VLDB*, 2005.

[19] R. Snodgrass. *Developing Time-Oriented Database Applications in SQL.* Morgan Kaufmann, 1999.

[20] R. T. Snodgrass, S. S. Yao, and C. S. Collberg. Tamper detection in audit logs. In *Proc. of VLDB*, 2004.

[21] P. Stahlberg, G. Miklau, and B. Levine. Threats to privacy in the forensic analysis of database systems. In *Proc. of SIGMOD*, 2007.

[22] M. Stonebraker. The Design of the POSTGRES Storage System. In *Proc. of VLDB*, 1987.

[23] G. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proc. of ICS*, 2003.

[24] M. Xie, H. Wang, J. Yin, and X. Meng. Integrity auditing of outsourced data. In *Proc. of VLDB*, 2007.

Appendix: Auditor Pseudocode

Audit ($\mathcal{T}, CrashList, DB_{old}, DB_{new}$)

```
1: Let $\mathcal{T}$ be the sequence of transaction log files, in create time order.
2: Let $CrashList$ be the trusted list of crash and recovery events.
3: Let $DB_{old}$ be the snapshot from the last audit, and let $DB_{new}$ be the current DB instance.
   {Extract the recovery log files $\mathcal{R}$, and no-crash version of transaction log files $\mathcal{L}$ from $\mathcal{T}$}
4: $(\mathcal{L}, \mathcal{R}) \leftarrow Extract\_Recovery\_Log(\mathcal{T}, CrashList)$
   {Perform global sanity checks on transaction log files}
5: if (GlobalSanityCheck($\mathcal{L}$, $\mathcal{R}$,$CrashList$) == AUDIT_FAIL) then
6:     return AUDIT_FAIL
7: end if
   {Is each individual log file okay?}
8: if (SanityCheck($\mathcal{L}, DB_{old}, DB_{new}$) == AUDIT_FAIL) then
9:     return AUDIT_FAIL
10: end if
11: $logHash \leftarrow$ HashTransactionLog($\mathcal{L}$)
12: $oldDBHash \leftarrow$ ComputeDbHash($DB_{old}$)
13: $newDBHash \leftarrow$ ComputeDbHash($DB_{new}$)
14: $computedHash \leftarrow ADD\_HASH(logHash, oldDBHash)$
15: if ($newDBHash == computedHash$) then
16:     return AUDIT_PASS
17: else
18:     return AUDIT_FAIL
19: end if
```

GlobalSanityCheck ($\mathcal{L}$, $\mathcal{R}$, $CrashList$)

```
1: if ($R$ contains anything other than recovery operations) then
2:     return AUDIT_FAIL
3: end if
4: for all ($c \in CrashList$) do
5:     {$c.Start$ is the time that the crash occurred, and $c.End$ is the time that the DBMS resumed normal operation}
6:     for all ($L \in \mathcal{L}$) do
7:         {Let $L.ct$ and $L.mt$ be the creation and last write times of $L$, respectively. Reject log files written or created while DBMS is down}
8:         if (($c.End > L.mt > c.Start$) OR ($c.Start < L.ct < c.End$)) then
9:             return AUDIT_FAIL
10:        end if
11:    end for
12: end for
13: if (the lifespans of any two log files in L overlap) then
14:     return AUDIT_FAIL
15: end if
16: return AUDIT_PASS
```

SanityCheck ($\mathcal{L}, DB_{old}, DB_{new}$)

```
1: for all ($L \in \mathcal{L}$) do
2:     if ($L$ has two COMMIT records for the same transaction) then
3:         return AUDIT_FAIL
4:     end if
5:     if ($L$ has a COMMIT record timestamp before the create time or after the last write time of $L$) then
```

```
6:      return AUDIT_FAIL
7:    end if
8:    if (COMMIT record timestamps in $L$ do not appear in
      chronological order) then
9:      return AUDIT_FAIL
10:   end if
11:   if ($lastModificationTime(L) > r/2 +$
      $createTime(L)$) then
12:     return AUDIT_FAIL
13:   end if
14: end for
15: if (The WORM and DBMS servers' clocks were more than
    r/2 time units apart at any point since the last audit) then
16:   return AUDIT_FAIL
17: end if
    {Is $DB_{old}$ okay?}
18: if ((The signature on $DB_{old}$ is not from the correct auditor)
    OR (The signature on $DB_{old}$ does not match its contents)
    ) then
19:   return AUDIT_FAIL
20: end if
    {Is the DB final state clean? Use the DBMS's native in-
    tegrity checker to verify that each data page is properly or-
    ganized, indexes and other metadata are set up properly,
    and crash recovery has been run if the last shutdown was
    due to a crash.}
21: if (the native DBMS integrity check on the database fails)
    then
22:   return AUDIT_FAIL
23: end if
24: return AUDIT_PASS
```

HashTransactionLog ($\mathcal{L}$)

```
1: {Initialize list of ongoing, committed and previously seen
   transactions}
2: $Txn \leftarrow \{\}; Seen \leftarrow \{\}; Committed \leftarrow \{\}$
3: $log\_hash \leftarrow 0$
4: for all ($L \in \mathcal{L}$) do
5:   {For each log file, scan records sequentially}
6:   for all ($record \in L$) do
7:     if (the record lists a tuple to be inserted) then
8:       $txnID \leftarrow get\_txn\_ID(record)$
9:       if (($txnID \in Committed$) OR
         $get\_timeStamp(record) < last\_audit\_time$))
         then
10:        continue
11:      end if
12:      if ($txnID \notin Seen$) then
13:        $Seen \leftarrow Seen \cup txnID$
14:        $Txn[txnID] \leftarrow$ new $TXN\_STRUCT()$
15:        $Txn[txnID].add(record)$
16:      end if
17:    end if
18:    if ($record$ is a COMMIT record) then
19:      $Committed \leftarrow Committed \cup txnID$
20:      $txnHash \leftarrow Hash\_TXN(Txn[txnID])$
21:      $log\_hash \leftarrow$
         $ADD\_HASH(log\_hash, txnHash)$
22:    end if
23:  end for
24: end for
25: return $log\_hash$
```