Threats to Privacy in the Forensic Analysis of Database Systems

Patrick Stahlberg, Gerome Miklau, and Brian Neil Levine
Department of Computer Science
University of Massachusetts, Amherst
{patrick | miklau | brian}@cs.umass.edu

ABSTRACT

The use of any modern computer system leaves unintended traces of expired data and remnants of users' past activities. In this paper, we investigate the unintended persistence of data stored in database systems. This data can be recovered by forensic analysis, and it poses a threat to privacy.

First, we show how data remnants are preserved in database table storage, the transaction log, indexes, and other system components. Our evaluation of several real database systems reveals that deleted data is not securely removed from database storage and that users have little control over the persistence of deleted data.

Second, we address the problem of unintended data retention by proposing a set of *system transparency* criteria: data retention should be avoided when possible, evident to users when it cannot be avoided, and bounded in time.

Third, we propose specific techniques for secure record deletion and log expunction that increase the transparency of database systems, making them more resistant to forensic analysis.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Relational Databases; H.2.7 [**Database Administration**]: Security, integrity, and protection; K.6.5 [**Security and Protection**]: Unauthorized access

General Terms

Security, Reliability

Keywords

Privacy, Forensics, Transparency

1. INTRODUCTION

Preserving a historical record of activities and data is critical for a wide range of applications. Historical data can be used to recover after system failure, to analyze past events after a breach, or to audit compliance with security policies. The *intentional* preservation of history can thus serve a good purpose, and inexpensive storage now makes it possible.

Conversely, in many scenarios, retaining a history of past data or operations can pose a serious threat to privacy and confidentiality. For example, in large institutions and enterprises, systems that retain data for too long risk unwanted disclosure, for example by security breach or by subpeona. Moreover, retention can violate privacy regulations like HIPAA [24], FERPA [16], or the E.U. privacy directive, each of which mandate the timely removal of data.

Modern computer systems *unintentionally* preserve history, and it can be surprisingly difficult to remove traces of the past from computer systems. Without precise control over data destruction, unwelcome remnants of past data can become a serious problem. For example, a wealth of sensitive data, including financial and medical records, were recovered from decommissioned hard drives [18]. Digital documents published on the Web have been found to include sensitive content believed to be deleted [15, 6]. Email was used in court cases against Enron employees and released to the public [22, 40], some of which was contained in "deleted items" folders [26]. The value of appropriately "forgetting history" to preserve privacy is increasingly being recognized [37].

In this paper, we examine the privacy and confidentiality threats in existing database systems resulting from the inadvertent preservation of historical data. Database systems make numerous redundant copies of sensitive data items in table storage, indexes, logs, materialized views, and temporary relations. The table storage manager makes copies of database records within allocated space, and it also pushes copies of data records into unallocated file system space. When data is deleted, it is not destroyed and often persists on disk. Data owners currently have little control over these operations. They cannot say with certainty where sensitive data may end up, whether it is destroyed after deletion, or how long it will persist.

These remnants of past data and activities are revealed through *forensic analysis*. Forensic analysis [10] is an emerging area of computer security focused on the collection and analysis of data recovered from computer systems. The goal is to validate hypotheses about past activities in a manner that is presentable in court or similar forums. When forensic analysis is performed by authorized investigators it can be a valuable tool, helping to hold individuals or systems *accountable* for malicious or mistaken actions. But when tools and methods of forensic analysis are used by an unauthorized party, it threatens privacy.

SIGMOD'07, June 11–14, 2007, Beijing, China.

The goal of our larger research program is to design database systems that allow users to appropriately balance the competing needs for privacy and accountability. In settings that require it, systems should support accountability by efficiently retaining a historical record of data and operations. At the other extreme, where privacy concerns take precedence, database systems should be memoryless and avoid unwanted history.

The present work is a first step in this program, and it focuses exclusively on privacy. We investigate the unintended retention of data in database systems, and we seek to build a database system that is resistant to unwanted forensic analysis. After a brief background on forensics and related work in Section 2, we present the following contributions:

- First, we formalize the problem of unintended recovery by proposing a set of desiderata to be satisfied by an ideal *forensically transparent system*. In brief, all data retained by the system should be accessible through a legitimate interface, and it should not be possible to recover hidden data through inspection of system state. (Section 3)
- Second, we investigate the forensic recoverability of data from database systems. We show how data remnants are preserved in database table storage, the transaction log, and indexes. We measure the persistence of deleted data in real systems under simulated query workloads. (Sections 4 and 5)
- Third, we propose changes to system internals that can significantly reduce unintended data retention in databases with little or no performance cost. In particular, we implement secure record deletion in MySQL, and we propose a method for the timely removal of data from the transaction log. (Section 6)

Several example scenarios can result in unintended data retention, all of which can benefit from our contributions. For example, as we stated above, businesses can unintentionally violate privacy regulations by leaving data in table or file storage. Adversaries that investigate databases recovered from lost or stolen computers can reveal sensitive information that was thought to be deleted. We also note that conversely, authorized investigators may find our results useful in recovering data from equipment subpoenaed or seized from a crime scene, or simply in situations where company policy has been violated.

In addition, while our primary focus is on conventional client-server databases, we note that database storage is increasingly *embedded* into a wide range of common applications for persistence. Embedded database libraries like BerkeleyDB [2] and SQLite [38] are used as the underlying storage mechanisms for email clients, web browsers, LDAP implementations, and Google Desktop, all of which store privacy-sensitive data. For example, message headers and time stamps for messages believed to be deleted can be found on disk in embedded databases[1]. As another example, Firefox 2.0 allows applications to store data that persists across sessions in an SQLite database. This storage is a sophisticated replacement for cookies, and will surely be a prime resource for forensic investigators to recover inadvertently retained deleted data.

[1]Apple's Mail.app stores all cached email headers in an SQLite database; see ∼`/Library/Mail/Envelope Index` on Mac OS X.

The vulnerability to forensic analysis for embedded database storage is particularly important because it impacts everyday users of desktop applications, and because embedded database storage is harder to protect from such investigation.

2. FORENSICS BACKGROUND AND RELATED WORK

Existing work in computer forensics has shown that in many operating systems and applications a deletion operation does not physically remove data. Researchers have studied the retention and recovery of expired data in file systems [9, 19, 17, 18], random access memory [11, 12, 19], and such applications as web browsers and document files [17]. Forensic tools like the Sleuth Toolkit [8] and EnCASE Forensic [14] are commonly used by investigators to recover data from computer systems. These tools are sometimes able to interpret common file types but, to our knowledge, none currently support forensic analysis of database files other than as hexidecimal dumps.

Military and intelligence agencies have set forth rigorous policies for the destruction of sensitive electronic data [29, 13]. A basic technique for the secure deletion of data is overwriting stored bytes with zeroes or random sequences. Guttman first observed that even overwritten bytes can sometimes be recovered from hard drives by magnetic analysis [23]. That possibility is increasingly unlikely with modern disks [3, 17].

Deletion through overwriting can be expensive, particularly for large files. A Linux file system with a secure deletion feature was proposed that performs asynchronous overwriting of deleted blocks to avoid the cost of synchronous overwriting upon deletion [3]. Privacy-conscious computer users can remove remnants stored on disk by using a variety of free or commercial tools to securely remove deleted files. Limitations of popular commercial tools have been described in a recent evaluation [20]. Importantly these tools securely remove deleted data from the file system; they very rarely address the persistence of data in applications.

Secure removal of data can also be accomplished by storing data in encrypted form and removing the decryption key (usually by overwriting). This technique was first used for efficient simultaneous removal of data from files and backup logs [5]. Perlman has recently proposed the design of a trusted service that would ensure timely expiration of email messages using key disposal [32]. A combination of overwriting and specialized cryptographic techniques were proposed for secure deletion in journaling file systems [33].

Attention to forensics in the database community has focused on enabling forensic investigation — for example by archiving data or auditing system behavior in a reliable way — as opposed to our goal here of minimizing data retention. Snodgrass et al. propose techniques for tamper detection and systematic analysis of audit logs [42, 31], and Goodrich et al. studied accurate detection of tampering in index structures [21]. The unintended retention of sensitive data in databases has not been studied, to our knowledge.

The role of encryption

While encryption has an important role in resisting forensic analysis, it does not provide an easy or immediate solution for database systems. Using an underlying encrypted

filesystem can avoid some forensic threats and can be effective for thwarting investigations of file systems. However, for the efficient removal of data in databases the granularity of data blocks that are encrypted (i.e., the association of keys to blocks of data) should match the granularity of data destruction. Such fine-grained encryption, perhaps associating keys with each record, implies severe performance, storage, and key management challenges. A further problem with encryption over an entire file system is that keys are often available to investigators; e.g., keys can be compromised, or users can be compelled by law to turn over keys. For databases operated by a large company, many users may have access to encryption keys or they will be stored in company files.

In Section 6, we show where in a database system the judicious use of encryption is most beneficial for resisting the threats considered here.

3. CRITERIA FOR SYSTEM TRANSPARENCY

The threats to privacy and confidentiality that we study in this paper all result from unintended retention of data in lower storage layers, where data is accessible through interfaces that are not controlled by the database. For example, a record deleted by a user and no longer accessible through the SQL interface can still be recovered from the filesystem. Anyone with access to these lower-layer interfaces can read data that was unintentionally retained. In this section we formalize this threat and introduce terminology used throughout the paper.

A database system can be regarded as providing a set of *services* that allow users and administrators to access stored records and perform operations on them in well-defined ways. For example, the SQL query processor is such a service, and so are the recovery and backup functions. All legitimate access to data in the database system is part of some service.

If a service needs a certain database record in order to function properly, the service creates a purpose for the record to exist. We call every record that has such a purpose an *active* record. A record that exists in the database is clearly active, but a copy of a deleted record in the transaction log is also active if it has a purpose (e.g. it may be required for concurrency control). Naturally, a properly functioning database system must retain all active records.

Services are comprised of *operations* that can change or create active records, or remove the purpose of active records. In the latter case, we say that the records become *expired*. For example, a record becomes expired when it is deleted, and when there is no combination of operations left that would ever use this record again.

An expired record is not needed for any aspect of the proper operation of the database system. We use the term *removal* to refer to the secure destruction of data. Examples of proper removal include overwriting the record or destroying the key if the record is encrypted. Mere restriction of access to a record, or placement on a free list, does not constitute removal. Throughout this paper, we consider a single basic overwriting operation to be sufficient for removal. As mentioned in Section 2, it may be possible to recover overwritten data from the physical surface of the disk, but in modern hard drives this is extremely unlikely.

Records that are expired but not removed, are recoverable from the hard disk. We call such data *slack data*, and it can appear in different forms. In an extreme but not uncommon case, a record can be fully recoverable. In other cases it might be only partially recoverable; for example when part of the record has been overwritten. Another factor that discriminates slack data is the location where it is stored. If the data is located in a file in use by the database system, we call it *database slack*, or DB-slack. If the data location is no longer allocated to a file, but it is still recoverable from the system, we call it *file system slack*, or FS-slack. Note that DB-slack and FS-slack are equivalent if the database uses raw disks or disk partitions as underlying storage medium.

Slack data can also appear as other *artifacts* if data structures like indexes, or unique identifiers used by the system, disclose expired metadata such as the length of records or the order of events that led to the current state of the database.

3.1 Threat model

Existing security threats make it impossible to ensure that users will be limited to the intended interface provided by the database system. We therefore consider an adversary with unrestricted access to storage on disk. This models the capabilities of a system administrator, a forensic investigator, a hacker who has gained privileges on the system, or an adversary who has breached physical security. We assume the attacker has access to the disk at some specific time t. Our goal is to prevent an adversary from retrieving from the system records that are *expired* as of time t.

3.2 Forensic transparency desiderata

We will see in Sections 4 and 5 that such an adversary is often capable of recovering parts of the expired data of current database systems. This means that the database system provides an unreliable view of the actual stored contents of the system. Users intend to remove data from the system, but it persists against their will and without their knowledge for an unknown period of time. Such a disconnect between the user interface and actual system behavior can have serious implications for privacy.

It is therefore desirable for a database to operate in a forensically transparent way. To determine the extent to which a database system is forensically transparent, we propose a set of desiderata. A database system is forensically transparent if it satisfies all three desiderata.

1. **Clarity:** The impact of each operation on the state of records (i.e active or expired) is clear to the user.
2. **Purposeful retention:** Only *active* records should be retained by the database.
3. **Complete removal:** Expired records must be *removed* by the system within a short, fixed time from when they become expired. In other words, there must be a small upper bound on the time that slack data exists in the database.

When these desiderata are satisfied, it will be easy for the user of a database system to determine and control which data are retained: it is clear which data are active, and the database guarantees that expired data is removed quickly. There is a small upper bound on the retention of expired data to allow for performance in the operation of the database system. Existing systems fail to meet these desiderata, as we show in the following sections.

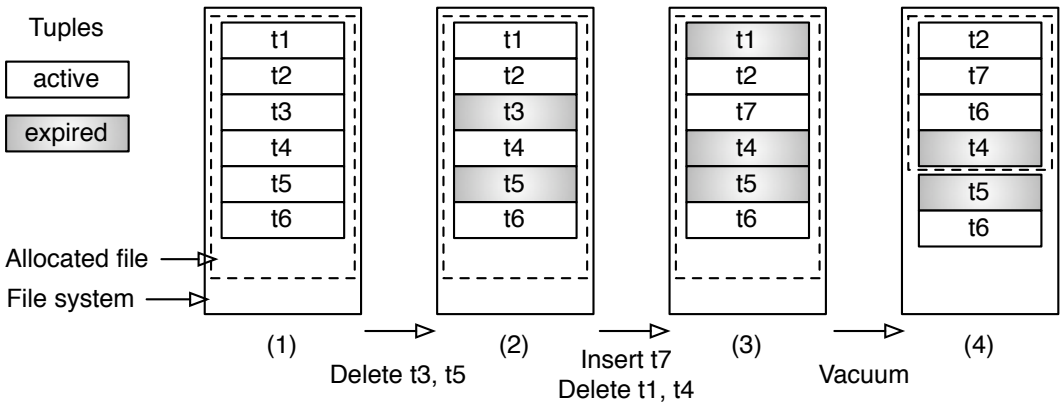


Figure 1: Illustration of table storage for four states of a database undergoing insertions, deletions, and table reorganization (or vacuum).

4. FORENSIC ANALYSIS OF DATABASE COMPONENTS

In this section, we detail why forensic recoverability of data is possible in table storage, indexes, and the transaction log. We explain the reasons for retention of expired data, how deletion is performed, and trace the lifetime of data items. The consequences of this behavior for real database systems are investigated experimentally in Section 5 with an emphasis on table storage.

4.1 Forensic analysis of table storage

Tables in a database system are stored in paged files, and many records usually share the storage space of one page. Different database systems may vary on how free space in table storage is recorded and allocated, whether a sort order is maintained, and more. However, many of the basic properties of table storage relevant to forensic analysis were consistent across the systems we studied (see Section 5).

In most cases, deletion of records is accomplished by setting a deletion bit — the data is not *removed* and is fully recoverable. The space occupied by the record is freed and may be reused by records inserted in the future. The reuse of freed space for a newly inserted record depends on at least two factors: whether the new record fits in the free space and whether a sort order is imposed on the table.

Because pages in table storage become fragmented over time, there is a table reorganization command (referred to here as *vacuum*) executed periodically by the database administrator. It improves storage performance, but it is often time-consuming. When vacuum executes, in addition to reorganizing records within and across pages, the size of the file used for table storage may be reduced, returning space to the file system. All systems we examined returned space to the file system without completely removing data. This means copies of database records are moved to unallocated file system space and remain recoverable. The following simplified example illustrates table storage and the main factors that contribute to data retention:

Example 4.1 Database table storage, undergoing a sequence of operations, illustrated in Figure 1.

1. State (1) shows six active records, occupying most of the space allocated to table storage.
2. After deletion of records t_3 and t_5, space is freed but the data is still fully recoverable, as shown in State (2).
3. Next, record t_7 is inserted, utilizing free space and overwriting the recoverable t_3 from above. In addition, two more deletions occur: t_1 and t_4, resulting in state (3).
4. In the next step, the vacuum procedure executes. It reorganizes the active records (t_2, t_7, t_6), and reduces the space allocated to the database file, as shown in (4). This leaves previously deleted record t_5 and a copy of active record t_6 in unallocated file system space. □

Updated records are not included in the example. When records have variable-length attributes, an update to a record may replace one or more attribute values with smaller attribute values. In this case, the table storage algorithm will perform the update in-place, overwriting attributes in the original record. This results in partially recoverable records, leaving the tail of prior data recoverable. If variable-length attributes are updated with larger values, an update in-place is not possible. The new record will be recorded in the next available free space, while the original record is logically deleted, resulting again in recoverable data.

The lifetime of data in table storage

Due to the behavior described above, data in table storage has a complex lifetime. Recalling the terms defined in Section 3, we can identify *active* and *expired* records in the database. An insertion creates an active record, a deletion changes a record from being active to being expired. For simplicity, we assume an update to any attribute in a record to be equivalent to a deletion followed by an insertion, creating one new active record and changing the state of another active record to being expired.

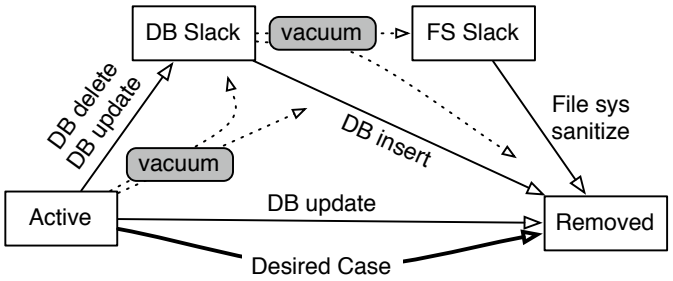


Figure 2: The flow of data during its lifetime. It begins in the *active* state. Before it is deleted and becomes *expired* it will often be retained as DB-slack and/or FS-slack.

After expiration, a tuple can be either *removed*, or it can continue to exist as *slack data*. In the latter case, we can further distinguish whether it is *DB-slack* or *FS-slack*.

Applying this terminology to record t_5 in Figure 1 we say t_5 is *active* in state (1). After deletion, it is *expired*, in states (2-4). Record t_5 exists as DB-slack in states (2) and (3). Then, as a result of the vacuum command, it is moved to FS-slack in state (4).

We illustrate the flow of data in table storage during its lifetime by the state diagram in Figure 2. It shows possible transitions of data from its *active* state. Ideally, in a forensically transparent system, *active* data would immediately become *removed* upon deletion, following the arrow along the bottom of the diagram. This rarely happens in the systems we studied. As mentioned above, some updates overwrite data, and in that case, data is *removed* as soon as it expires. But in most other cases, data follows a different path in the diagram. Deletion and updates that expand variable length fields both result in expired data that is preserved as DB-slack, shown by the upward arrow leaving the *active* state in Figure 2.

DB-slack may persist during the standard operation of a database system (we measure this extensively in Section 5). It may later become *removed* under two conditions. First, insertions applied to the database may overwrite DB-slack. Second, the vacuum procedure may reorganize records in the file, overwriting some DB-slack. The vacuum procedure, however, is complex. It may return allocated file space to the file system, and no database system we study removes data before doing so. Thus, DB-slack, instead of becoming *removed*, can become FS-slack data. In fact, we have discovered that in the course of reorganizing pages of database storage, *active* records are packed together by copying records within and across pages. This means that the vacuum procedure can result in retention of *active* data in unallocated spaces on the disk. If this data gets deleted at a later point in time, it will become DB-slack or FS-slack (as in Example 4.1), even if the allocated copy of the data is *removed*.

Finally, FS-slack can become *removed* if the block is allocated to a new process and overwritten, or as the result of active file system sanitization, if it is performed.

With access to the file system, the investigator or adversary will find all expired data in DB-slack and all expired data in FS-slack (in addition to the *active* data present in table storage). The distinction between DB-slack and FS-slack is important, however. DB-slack exists in files allocated to the database system. It therefore cannot be removed by file system sanitization techniques or by using a file system with secure deletion [3]. We are primarily focused on sources of retention in DB-slack, which require solutions in the database system. We are also interested in how and when databases create FS-slack, although we believe it should be removed through file system techniques.

Note finally that, while some database systems support storage on "raw" disk partitions, we focus on storage managed by the filesystem because it is the default behavior in all systems studied, and because three of our systems (PostreSQL, MySQL (MyISAM), and SQLite) do not support raw disks. Further, some have suggested that the complexities of administering raw disks and the increased use of networked storage services make raw disk use less advantageous for a vast majority of applications [1]. We do not expect raw disk use to appreciably change the results and techniques presented here.

4.2 Forensic analysis of the transaction log

The transaction log is an essential component of any database used to provide recovery from transaction and system failure. Write-ahead logging is the most common strategy [34]. Update records in the log include before- and after-images of modified data pages. For periods of time covered by the log, prior states of a database can be reconstructed, and a wealth of sensitive data will be retained.

Currently, the only bounds on data retention in the log result from resource constraints of the disk. Transaction logs are often implemented as circular files where records are written sequentially. As the file grows, it wraps around, overwriting old records. The amount of time data persists in the transaction log is difficult to predict, even for stable workloads. It will depend on the capacity of the log file, the rate of updates, the log space required per update, and the frequency of checkpointing. A large enterprise with heavy transaction processing could cycle its log file in a few days or once per day. In other settings, logs could easily contain months of historical data, much of which is expired but still recoverable.

To meet our transparency requirements, data should be completely deleted as soon as it is no longer needed for recovery. It is not hard to identify portions of the log that will never be used by the recovery manager and can be freely deleted without interfering with legitimate recovery practices. We propose techniques for efficient log expunction in Section 6.

4.3 Forensic analysis of indexes

The disk representation of B+tree indexes can reveal expired data and information about the history of operations that led to the current state of the database. There are two sources of data recovery from indexes.

First, logically deleted sort keys can sometimes persist in the internal nodes. For example, in MySQL and PostgreSQL we found that when entries in B+tree nodes are deleted, they are not overwritten, and that merging of B+tree nodes results in the logical deletion of B+tree nodes without their removal. This type of data retention is similar to DB-slack, the slack data in table storage. We return to this issue in Section 6 in the context of MySQL.

Database System	PostgreSQL	MySQL		IBM DB2	SQLite
Table format	default	MyISAM	InnoDB	default	default
DELETE physically overwrites	no	yes	no	no	no
DELETE creates free space	no	yes	yes	yes	yes
UPDATE always preserves past value	yes	no	no	no	no
UPDATE/short preserves residual	yes	no	yes	yes	yes
UPDATE/long preserves past data	yes	yes	yes	yes	yes
INSERT attempts to overwrite	no	yes	yes	yes	yes
UPDATE to NULL overwrites	no	yes	no	no	yes
VACUUM automatic	no	no	possible	no	possible

Table 1: Recovery properties of relational storage for three client-server databases (PostgreSQL, MySQL, IBM DB2) and one embedded database (SQLite).

Second, B+trees have standard deterministic procedures for insertion and deletion. It is therefore possible to infer, from the structure of a B+tree, partial information about the sequence of insertions and deletions that led to the current state of the database. For example, if a sequence of items are inserted into a B+tree in ascending order, the resulting tree will be different than the tree formed by the same items inserted in descending order. The degree to which data structures reveal information about their past states, as well as the design of so-called "history-independent" data structures, has been analyzed before [30, 27] (although not specifically for B+trees in databases). Despite the fact that B+trees are not history-independent, the shape of a B+tree will almost never give exact information about past operations. Instead, a forensic analyst would only be able to determine that certain sequences of operations were impossible, leaving many equally likely possibilities. Further, the degree of certainty about past operations diminishes as the fan-out of the B+tree increases. (In the extreme, if the fan-out is large enough so that all keys are stored in the root node, then no history is revealed, since keys are stored in sorted order.) B+trees in databases commonly have high fan-out, so this will tend to result in less precise inference.

4.4 Other forensic targets

Temporary tables

As the database system executes queries, temporary relations may be constructed automatically and written to disk to speed later query execution. SQL queries that include ORDER BY or GROUP BY clauses require sorting, which will be performed by constructing a temporary relation containing the attributes being sorted. As a representative example, in MySQL, each temporary relation is written as a separate file to a specified temporary directory. No explicit deletion is performed until the database server process is terminated. At this point the files will be deleted by the filesystem, but will remain as FS-slack. This is a notable case in which the workload of *select* queries may determine forensic recoverability, distinct from the workload of *delete, update*, and *insert* queries studied in Section 5.

Record identifiers

Many systems introduce tuple identifier fields into the schema of tables. These identifiers are often generated automatically using sequential numbering. They can be retrieved and analyzed by users to trace the order of insertions and deletions in the database — information not available through the intended interface.

5. FORENSIC RECOVERY EXPERIMENTS FOR TABLE STORAGE

In this section, we examine the operation of four database systems to measure the amount of data that is recoverable through forensic analysis of *table storage*. Our study focuses on PostgreSQL (version 7.4.8), MySQL (version 5.0.24a), IBM DB2 (version 8.1.0), and the SQLite embedded database (version 3.2.2). We study two table storage formats for the MySQL database (called MyISAM and InnoDB[2]), resulting in a total of five test cases.

Our findings show considerable retention of data in both DB-slack and FS-slack, interesting differences between systems, and key factors that impact data retention such as clustering and the frequency of vacuum. In addition to the absolute quantity of data, we also study the lifetime of data in DB-slack.

5.1 Forensic properties of different systems

Table 1 compares the deletion, update, insert, and vacuum behavior of the five systems we examined. As the table shows, we found that InnoDB (MySQL), DB2, and SQLite share three important behaviors: data is deleted logically, and not *removed*; previous values of a record are in many cases completely overwritten during an update; and deleted tuples are freed so that subsequent insertions may overwrite the data.

Freeing tuples to enable later overwriting has important consequences for forensic recovery. DB2 frees tuples immediately upon deletion. InnoDB (MySQL) and PostgreSQL use multi-version concurrency control, which forces them to retain deleted records for some time to provide a consistent view to running transactions. In InnoDB, a continuously-running *purge thread* frees tuples after they are *expired*, because they are no longer needed. PostgreSQL never frees records until vacuum is performed. This is not a requirement of multi-version concurrency control, but instead a legacy feature of the storage manager. PostgreSQL originally supported versioning, in which all past states of the database are preserved [43]. The versioning features have since been deprecated, but there is still no active thread to free deleted tuples. During vacuum some expired tuples are overwritten, space in table storage is freed, and some data is moved to FS-slack. MyISAM (MySQL) is the sole storage format that removes data on deletion. However, once vacuum behavior is taken into account, we will see that MyISAM moves considerable data into FS-slack.

[2]MyISAM is the default format, and it is recognized as fast for simpler applications. InnoDB is an advanced table format offering ACID compliance.

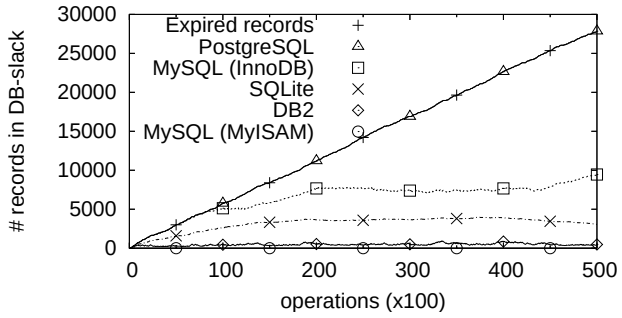


Figure 3: DB-slack data of the different database systems (default configuration and without vacuum). The *Expired* and *PostgreSQL* lines have the same values.

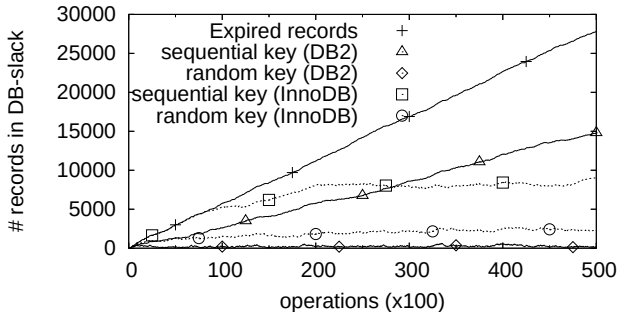


Figure 4: The impact of clustering and sort key distribution on the retention of data in DB-slack for DB2 and InnoDB (MySQL).

5.2 Workload experiments

We repeated several experiments on each of the five systems using Linux and running the ext2 [7] file system. Each experiment is initialized with a database of 12,500 variable-length records. The workload consists of 50,000 database modifications, in random order. Of these, 45% are insertions, 35% are deletions, and 20% are updates to existing records. Updates are equally likely to increase or decrease the size of the modified record. After every 100 operations, we measured the number of database records found in DB-slack and FS-slack using a set of custom tools. These programs bypass the SQL interface of the database system by opening the table storage files of the hosting filesystem image and searching for database records in the binary data. In actuality, tuples in the DB-slack or FS-slack may be fully or partrially recoverable. Our results conservatively report only fully recoverable tuples.

In the following, we are sometimes forced by space limitations to present a subset of results for a particular issue. We never omit results that are inconsistent with what we have chosen to present.

Default behavior of five storage managers

Figure 3 shows the number of tuples recovered from DB-slack when our randomized workload is run using five different systems in their default configuration. The figure also includes the total number of expired records, which increases linearly as deletes and updates are executed. It represents the maximum amount of data that could be potentially recovered. We see that PostgreSQL keeps 100% of the expired records in the DB-slack: its trend line is superimposed on that of the expired records. At the other extreme, MyISAM (MySQL) shows no DB-slack. The recovery rates for InnoDB (MySQL), SQLite, and DB2 are between these extremes, with the recovery rate for DB2 fairly low at about 400–500 tuples throughout the workload.

These results assess forensic recoverability for a small-scale synthetic workload in the default configuration of the databases. We found that scaling the database in size did not significantly change these results. This is because slack is created as a result of operations on records and is not influenced by the number of pages. The rates of deletion and update also impact data retention in table storage. However, the effects of changes in the deletion rate were dominated by other factors like the vacuum operation, or changes in sort-key distribution and clustering, as explained next.

Impact of clustered table storage

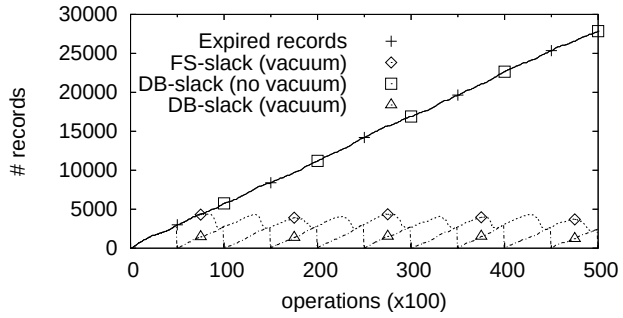
Clustering records based on a sort key is a common feature of database storage. Because clustering imposes a constraint on the placement of new records in table storage, it has a significant impact on data retention. For example, if a table is stored using a clustered B+tree and inserted records tend to increase with respect to the sort key, then records must always be added to pages at the end of the file. Records that are deleted from earlier pages in the file will enter DB-slack and are unlikely to be overwritten, unless counteractive measures like the execution of *vacuum* are taken.

It is not uncommon for insertions to be monotonically increasing with respect to the clustered sort key of a B+tree index. This may occur because the database designer chooses automatically increasing identifiers for their tables, including time and date or a unique event number. In fact, InnoDB (MySQL) builds a clustered index by default. If no primary key is defined on the table, then InnoDB clusters records using an internal identifier that is strictly increasing. Thus inserts are sequential with respect to the clustered order.

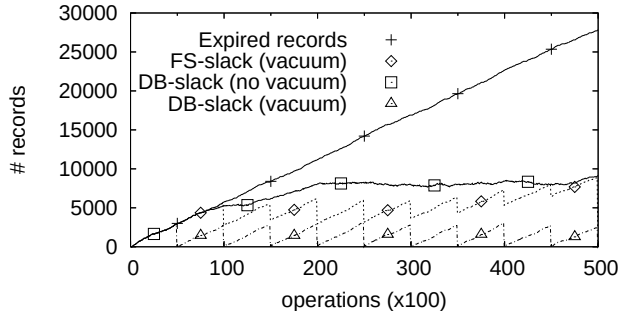
It is these effects that explain the substantial difference in recoverability between InnoDB and DB2 in Figure 3: DB2 by default does not cluster records, whereas InnoDB by default clusters inserted records using a sequential index. We compare these two systems in Figure 4 for workloads of sequential inserts/updates and randomly distributed inserts/updates (deletes are random in both cases). Sequential inserts have a dramatic effect in both systems: the amount of DB-slack is more than doubled for InnoDB and is even greater for DB2 where there was very little before. Although the random and sequential inserts are characteristics that could occur in real workloads, many applications will generate workloads that fall between these two extremes.

Vacuum and file system slack

All the systems we studied offer a vacuum command intended to periodically reorganize table storage. The vacuum operation has two important effects on data retention. First, it reorganizes tuples, overwriting many, and it will tend to reduce DB-slack. Second, during reorganization most vacuum procedures release pages to the filesystem that contain expired or active tuples; this process generates FS-slack.

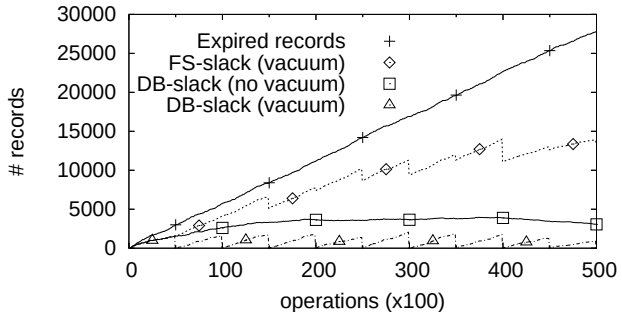


(a) PostgreSQL

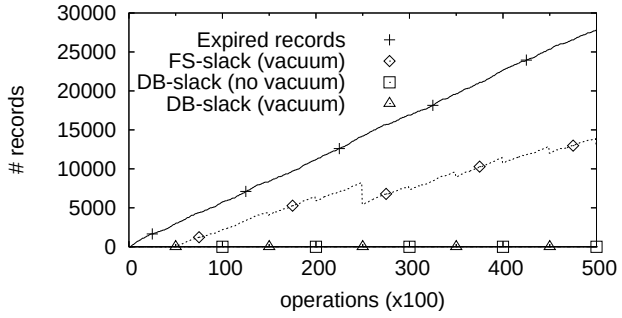


(b) InnoDB (MySQL)

Figure 5: The impact of vacuum on DB-slack and FS-slack for PostgreSQL and InnoDB (MySQL).



(a) SQLite



(b) MyISAM (MySQL)

Figure 6: The impact of vacuum on DB-slack and FS-slack for SQLite and MyISAM (MySQL).

Figures 5 and 6 show the amount of DB-slack with and without vacuum for four of the systems. The vacuum operation is called every 5,000 operations. The impact of vacuum on DB-slack is largely similar across systems. After each vacuum, the DB-slack is removed completely, and as the workload continues, new DB-slack accumulates. DB2 displayed this behavior as well.

Figures 5 and 6 also show the creation of FS-slack as a result of vacuum. Page reorganization during vacuum is generally not done in-place. Instead it is often more efficient to scan the existing fragmented pages and write entirely new pages. This implementation of vacuum often leads to FS-slack, when the old pages are released to the filesystem. For example, SQLite and MyISAM (MySQL) are examples of systems that generate substantial filesystem slack through the implementation of their vacuum procedures. SQLite performs vacuum by rebuilding the table in newly allocated storage space. After the operation, freed pages are returned back to the filesystem creating FS-slack consisting of both active and expired tuples. Both the MyISAM and SQLite formats return free space back to the file system.

Vacuum in DB2 reduced DB-slack but, unlike the other systems, did not create FS-slack as a result. We suspect that deallocated pages that result from vacuum remain in the DB2 tablespace and are not released to the filesystem. This means the data may remain in DB-slack, but if so, it is not in a form captured by our forensic recovery tools. Without access to the source code it is difficult to verify this hypothesis.

Lifetime of recoverable data

In addition to the *quantity* of data recovered, the *duration* of data retention is also an aspect of the privacy threat posed by forensic analysis. We measured the period of time each tuple spends in DB-slack before being removed by vacuum or overwritten by an insertion. The results of the experiments are shown in Figure 7, which plots the complementary CDF of the age of records in DB-slack at the end of the 50,000-step workload for InnoDB (MySQL). A point (x, y) in this graph represents the percentage of tuples y that survive at least x operations. About 40% of the tuples in DB-slack survive 25,000 operations when vacuum is not used. With vacuum, the graph drops more steeply, showing that the records in DB-slack are younger on average. The oldest records survived for about 25,000 operations using vacuum. Inspection of the InnoDB source code suggests that DB-slack is always overwritten after this number of operations because of the size of the allocated tablespace, required table size in our experiments, and the reuse of freed pages after sequential vacuums. Overall, analysis of the DB-slack duration for the other database systems showed similar results.

These results suggest that database slack could persist for a significant period of time. In these results lifetime is measured in the number of operations. Translating this into hours, weeks or months would depend on the frequency of operations in the system, and would vary greatly by application.

The lifetime of data moved to FS-slack is harder to measure and harder to predict. This data exists in unallocated file system space. Its lifetime depends on the file system policy for allocating space to new processes, and on the overall usage pattern of the file system. Accurate measurement of

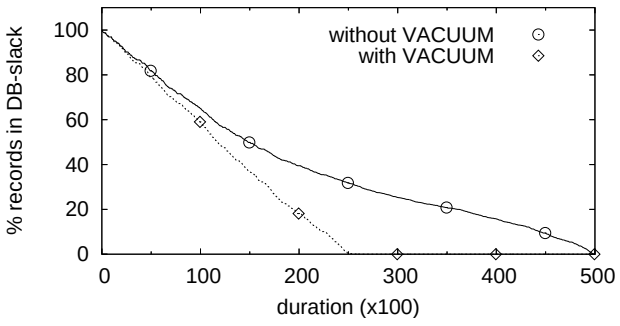


Figure 7: DB-slack time distributions for InnoDB (MySQL).

its lifetime would require modeling file system use of all processes running along with the database, and is beyond the scope of this paper.

Conclusions

Sections 4 and 5 show clearly that the investigated database systems violate our desiderata for forensic transparency (see Section 3) by retaining expired data. While the amount of data retained and the lifetime of retention can vary considerably, our results show that there are common configurations that lead to significant retention of expired data. At a high level, we have seen that DB-slack acts like an unmanaged store of expired data, inaccessible from the intended interface. In most cases, that store appears to have a constant capacity that is determined by configuration parameters and clustering constraints. FS-slack acts as an additional unmanaged store, but is often larger.

Our experiments are not exhaustive. We have attempted to present the most important factors impacting forensic retention, using a simple workload that permits effective analysis of the results. Due to the complicated interaction of effects of the workload, clustering, frequency of vacuum, and system configuration, the actual retention of data for a particular database application would require targeted analysis.

6. MAKING DATABASE SYSTEMS TRANSPARENT

In this section, we present specific techniques for minimizing data retention and increasing forensic transparency. We focus on what we consider to be the two most important threats to privacy in database systems: slack data in table storage and the uncontrolled persistence of data in the transaction log.

We begin by comparing different methods for destroying data and then present, in Section 6.1, our implementation of a modified InnoDB storage manager that removes expired and unnecessary data remnants securely from DB-slack. We chose to modify the InnoDB storage manager because it seemed to us closest to "industrial strength" of the available open-source implementations (in fact it is owned by Oracle, and dually licensed). In addition, the forensic properties of InnoDB described in the previous section seemed generally representative of other systems and had more DB-slack than others, excluding PostgreSQL. Our performance results show these modifications do not appreciably affect running time, and additional experiments show quantitatively the removal of DB-slack. In Section 6.2, we propose an efficient method for removing expired data from the transaction log.

Note that our focus in this section is on the removal of DB-slack. Methods for removing file system slack have been studied by others [17, 13, 3, 20, 5].

Methods for destroying data

Recall from Section 2 that there are two basic strategies for destroying data stored on disk. The first is to overwrite data, which can be time consuming for large blocks of storage. The second stores data in an encrypted form, which enables quick destruction of data by overwriting the key [5].

However, in this latter approach, during the lifetime of the data, every read requires decryption and every write or update requires encryption. Each technique is most suitable for different database components.

For removal of data from table storage, we believe overwriting is the best approach, and it is the technique we implemented in MySQL, as described below. An encryption-based approach is likely to introduce severe performance and management penalties when applied to table storage. Since secure deletion takes place with the granularity of an individual record, distinct keys must be associated with each record; otherwise old data could be recovered while portions of the table still exist. The resulting number of keys used in the system is a storage and management burden, and scans of table pages could be severely slowed down by repeated decryptions.

Conversely, encryption is a desirable approach for removal of data from the transaction log since the log records are written once, requiring only a single encryption operation. Decryption is usually only necessary on abort or system failure, and repeated decryption is very unlikely.

6.1 Implementing transparent table storage for MySQL

Our investigation of the InnoDB source code revealed that the three causes of DB-slack are the *deletion, B+tree,* and *vacuum* functions.

- InnoDB deletes tuples without *removing* the data; tuples are simply marked as deleted.
- InnoDB stores all table data in a B+Tree data structure. The functions that modify the B+Tree often leave copies of database records in unused parts of the B+Tree. For example, when a page is split, half the records are copied to the new page and the old storage space is marked as available to the database (but not the filesystem) without being overwritten. This process allows active tuples to migrate into the DB-slack, and even if the tuples are later securely deleted, these now-expired copies will not be overwritten.
- The vacuum function also rearranges the B+Tree, causing the creation of DB-slack when stale copies of data are not overwritten. Additionally, vacuum can release storage to the file system, which can cause FS-slack.

The contribution of each of the three functions to DB-slack is dependent on the workload of a database and the frequency of calls to vacuum. Securing table storage therefore requires addressing each of these issues. We investigate here only secure versions of deletion and B+Tree operations because they are critical to system performance. We leave securing vacuum for future work.

Securing deletion

Because InnoDB uses multi-version concurrency control, tuples cannot simply be overwritten after a deletion. The system first marks tuples scheduled for deletion. When the last transaction whose snapshot still includes that tuple has ended, the tuple is *expired* and put on a free list by the so-called *purge thread*. At that point, it is possible that it will be overwritten if another tuple is inserted or updated or if the page is reorganized.

We modified the purge thread so that tuples are overwritten as they are put on the free list. Because the free list is part of the page organization, this approach incurs virtually no additional disk I/O operations, only a call to `memset()`. It also does not impact the amount of data that must be written to log records. The drawback is that the purge thread does not typically execute immediately after a deleted tuple becomes *expired*. Instead, the purge thread might start to lag behind if the system is under high load. There are configuration options in MySQL to put an upper bound on the lag of the purge thread, but imposing a tight upper bound comes at the price of a possible performance degradation. Nevertheless, in most settings, we expect this lag would be quite small—seconds or minutes, but not more.

Securing B-Tree operations

Securing the B+tree implementation required changes to the code for insert, delete, and update. For each of these functions, we modified any operations that copied, purged, or reorganized the B+tree and used overwriting to remove obsolete data. Because our modifications operate on pages that are already loaded in memory and are updated by other mechanisms in the code, we incurred no additional I/O operations. We found that the modified code ran at virtually the same speed as the unmodified code, as we discuss below.

Impact on transparency

To measure the decrease in DB-slack produced by our modified InnoDB, we re-ran our experiments from Section 5. The results are shown in Figure 8. The "original" line is the same as the MySQL (InnoDB) result in Figure 3, although the scale is changed to offer a better view of the "secure" trend line. With our modifications, there is virtually no DB-slack data left. Remaining slack (barely visible in the figure) is the result of either the purge thread lagging slightly behind or because the database system's buffer manager had not yet flushed the changed page to disk.

Taking our transparency criteria from Section 3.2 into consideration, we conclude that for table storage, our modified version of InnoDB indeed satisfies our Desiderata 2 and 3 for fully recoverable records.

Performance impact

To evaluate the performance impact of our modifications to InnoDB, we measured the running time of an intensive workload of 100,000 INSERTS and 100,000 DELETES, executed as a stored procedure. At the time of the experiment, no other jobs were running on our Linux system. We repeated this experiment twenty times each for the unmodified and modified versions of InnoDB. There were no substantive differences in the execution times. The average running time of the the unmodified InnoDB was 51.11 seconds with a standard deviation of 0.06 seconds. The average running time of our secure version of InnoDB was 51.07 seconds with a standard deviation of 0.09 seconds, and its minimum (50.91 seconds) and maximum (51.26 seconds) running times were within one standard deviation of the unmodified version.

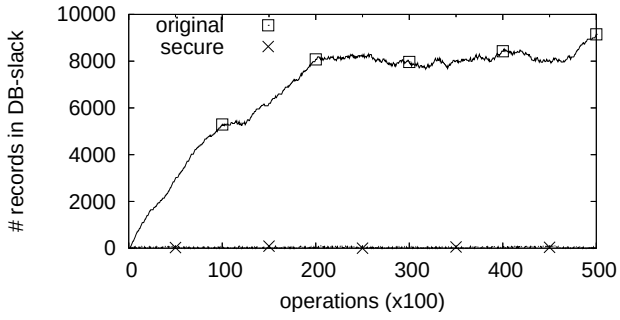


Figure 8: A comparison of tuples retained in DB-slack in InnoDB (MySQL) before and after the implementation of our secure deletion techniques.

6.2 Efficient log expunction

The age and quantity of data retained in the transaction log is bounded only by physical constraints of the disk, as we noted in Section 4. Because retention is highly dependent on the workload, it is difficult to predict and may vary widely over time. Moreover, as transactions run and the log grows, older parts of the log will never be used for recovery and cease to serve any legitimate purpose In the remainder of this section, we describe techniques for the efficient expunction of data from the transaction log that meet the transparency desiderata that we described in Section 3.2. We focus on write-ahead logging and the ARIES [28] recovery algorithm.

Candidates for expunction

The transaction log is written sequentially, with new records written to the log tail. As transactions are processed by the system, parts of the log cease to serve any purpose for recovery. The log can protect against up to three modes of failure, and the recovery configuration determines the point at which a log record can be safely removed. The transaction log is commonly used to recover from transaction failure and system failure. Transaction failure occurs when a transaction halts due to an unavailable resource, bad input, or a constraint violation, among other reasons. A system failure involves the loss of volatile memory. The log may also be used to recover from media failure, in which a hardware malfunction causes loss of data on stable storage.

The status of records in the log changes from possibly useful to certainly useless upon *commit*, *checkpoint*, or *fuzzy dump* actions. When a transaction commits, it will never be undone; and at that point, *before-images* corresponding to the transaction stored in log records are no longer needed. When a (fuzzy) checkpoint is taken, updates prior to the penultimate checkpoint will never be used to recover from transaction or system failure. If the log is also being used for media recovery then when fuzzy dumps are performed, the log records prior to the dump are no longer needed for recovery from media failure.

A naive strategy for log expunction is to physically overwrite useless log data. This approach is expensive, as it requires a trailing process to continuously read and write pages of the log, deleting data as it becomes *expired*. Instead, we propose to encrypt the log and perform expunction efficiently by removing the keys used for encryption. This encryption scheme requires no additional I/O for logging or recovery.

To simplify our presentation, we first describe an encryption scheme designed to remove whole log records once they are no longer needed for recovery from transaction or system failure. Expunction of records occurs as a consequence of the checkpoint operation. Similar techniques can be adapted to finer-grained log expunction; e.g., to remove before-images on commit. We discuss supporting recovery from media failure at the end of this section. A system implementing these techniques would have a guarantee — determined by the checkpointing frequency — that data is securely removed in a timely manner.

Using encryption for efficient log expunction

Log records are the basic unit of data written to the log, and they are typically identified by unique, sequential identifiers called *log sequence numbers* (LSN). In our scheme, the content of each log record is encrypted under a different cryptographic key. For a log record with LSN l, we denote the cryptographic key used to encrypt the record by K_l. By removing the key K_l, we effectively expunge log record l, but without having to read or write the page that stores the record (assuming no weakness exists in the cryptographic algorithm).

Log record keys are constructed as an ordered sequence using a *hash chain* [25, 36]. On initiation of the log, a random seed value s is chosen. The key of the first log record, K_1, is set to $H(s)$, where $H(\cdot)$ is a cryptographic hash function (e.g., SHA-256 [39]). The key for the next log record is computed by applying $H(\cdot)$ to the current key. In general, we have $K_n = H(K_{n-1})$. A key K_n can be efficiently computed by repeated hashing from any previous key as $H^{n-i}(K_i)$, where $i < n$. However, assuming H is pre-image resistant, it is computationally infeasible to compute any earlier key from a later key. For example, K_{n-1} cannot be feasibly computed from K_n.

At any point in time, we keep in stable storage exactly *one* cryptographic key. This key, denoted $K_{current}$, is sufficient to allow decryption of all needed log records. Log record expunction is accomplished by a simple operation called *key update*, performed at the end of checkpointing. To perform key update we simply update $K_{current}$ by overwriting it with a new key corresponding to a subsequent LSN. For example, if key update takes $K_{current} = K_n$ and replaces it with K_{n+i}, then each log record with LSN between n and $n+i-1$ is effectively removed. The records will be present in the log in encrypted form, but their keys will be lost. Note that $K_{current}$ can be stored on the same page as the log records, because it is not secret and gets overwritten as soon as it gets invalid.

To ensure that recovery is possible, we must guarantee two conditions. First, $K_{current}$ must be available after system failure. We guarantee this by storing $K_{current}$ in the *master log record*, which always contains the LSN of the latest begin-checkpoint record. It is already the case that the master record is written to a special place on disk only after the end-checkpoint record is written to disk, ensuring that the fuzzy checkpoint is complete [34, 41]. Second, we must ensure that the key for any needed log record is computable during recovery. The value of *current* (i.e., the index of $K_{current}$) must be less than or equal to the earliest LSN of any log record that could be read during the recovery procedure. The key update procedure guarantees this by choosing the new value for $K_{current}$ equal to the LSN of the end-checkpoint record of the penultimate checkpoint [4].

The cost of log encryption

The transaction manager is a performance-critical component of any database system. Our proposed expunction scheme performs timely removal of log records *without any special I/O operations*. The scheme imposes computational overhead that we believe will be small on an absolute scale and is likely to be hidden by higher cost operations common to transaction management. The implementation and performance evaluation of our scheme has not been completed, but in the remainder of the section, we explain the expected costs of our modifications to transaction processing algorithms.

To append a new log record to the tail of the log in memory we compute a new key by hashing and then encrypt the contents of the log record. A stream cipher is an ideal choice for the encryption function. Given a key, a stream cipher generates a pseudo-random stream of bytes. Encryption is accomplished by XOR of the stream with the plain text. Decryption is accomplished by XOR with the cipher text. RC4 [35] is a popular and extremely fast stream cipher, with a small memory footprint, that encrypts at 120 MB/s on our experimental system. In addition, the pseudo-random stream can be computed and cached prior to encryption, and does not impose a size overhead on the encryption of small log records due to padding that would be necessary if we used a block cipher.

The checkpoint operation proceeds normally in our scheme, encrypting the records it normally writes to the log, and writing only 16 extra bytes (the value of $K_{current}$) along with the master log record. Since the checkpointing operation is fairly expensive, this is likely to have a negligible impact on performance.

During recovery, the current key is read from the master log record. All keys, up to the key for the last log record in the log, can be efficiently computed by repeated hashing. (SHA-256 can be repeatedly computed on a 256 bit key about 700,000 times per second.) The recovery manager then proceeds normally with the analysis phase by decrypting log records. Recovery also writes log records, and therefore will incur the cost of encryption in our scheme. Again, with modern processing speeds, we believe the computational overhead imposed by key computation and decryption will be dwarfed by the I/O cost of recovery.

If the log is used for recovery from media failure then the key update procedure described above will result in the loss of necessary records. This limitation can be easily addressed by writing a copy of $K_{current}$ along with backup records (usually stored on a different disk to ensure independent failures). In this case, the transaction log is still protected from forensic investigation. Finally, we note that standard techniques can be used to reset the hash chain seed to improve both the performance and security of this scheme.

7. CONCLUSIONS

We have demonstrated that database systems fail to remove data securely after deletion, contain remnants of past operations and data in allocated storage, and make numerous redundant copies of data items that can persist in the file system. As a result, database systems provide a false view of stored data, which threatens privacy and makes adherence to information handling policies impossible. We have described a set of transparency principles and modifications to MySQL internals that improve privacy substantially with minor performance impact. Our future work will continue to explore the performance trade-offs of transparency in database systems and explore balancing the competing needs for accountability and privacy.

7.1 Acknowledgements

We thank Phil Bernstein for helpful comments on this work. This work was funded in part by NSF CAREER Award 0643681 and by NSA grant NSA H98230-05-1-124.

8. REFERENCES

[1] A. Ailamaki, S. Krishnamurthy, S. Papadimitriou, and B. Schroeder. *"PostgreSQL", Chapter 26 of Database System Concepts*. McGraw-Hill, 5th edition, 2006.

[2] Berkeley db xml. Available at `www.sleepycat.com`.

[3] S. Bauer and N. B. Priyantha. Secure data deletion for linux file systems. In *Procedings of the 10th USENIX Security Symposium*, pages 153–164, 2001.

[4] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 1997.

[5] D. Boneh and R. J. Lipton. A revocable backup system. In *USENIX Security Symposium*, pages 91–96, 1996.

[6] S. Byers. Scalable Exploitation of, and Responses to Information Leakage Through Hidden Data in Published Documents, April 2003.

[7] R. Card, T. Tso, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proc. Dutch International Symposium on Linux*, 2004.

[8] B. Carrier. Sleuth toolkit / Autopsy forensic browser. Available at `www.sleuthkit.org`.

[9] B. Carrier. *File System Forensic Analysis*. Addison-Wesley Professional, 2005.

[10] E. Casey. *Digital Evidence and Computer Crime*. Elsevier, 2nd edition, 2004.

[11] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proc. USENIX Security Symposium*, August 2004.

[12] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *Proc. USENIX Security Symposium*, August 2005.

[13] National Industrial Security Program Operating Manual DoD 5220.22-M. `www.dss.mil/isec/nispom_0195.pdf`, Jan 1995.

[14] Encase forensic. Available at `www.guidancesoftware.com`.

[15] R. Edmonds. Justice department hid parts of report criticizing diversity effort. Associated Press/USA Today, October 2003.

[16] U.S. Family Educational Rights and Privacy Act (FERPA). `www.ed.gov/offices/OII/fpco/ferpa`.

[17] S. L. Garfinkel. *Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable*. PhD thesis, M.I.T., 2005.

[18] S. L. Garfinkel and A. Shelat. Remembrance of data passed: A study of disk sanitization practices. *IEEE Security and Privacy*, Jan/Feb 2003.

[19] T. Garfinkel, B. Pfaff, J. Chow, and M. Rosenblum. Data Lifetime is a Systems Problem. In *Proc. ACM SIGOPS European Workshop*, September 2004.

[20] M. Geiger and L. Cranor. Scrubbing stubborn data: An evaluation of counter-forensic privacy tools. *IEEE Security and Privacy Magazine*, 4(5):16–25, 2006.

[21] M. Goodrich, M. Atallah, and R. Tamassia. Indexing information for data forensics. In *Applied Cryptography and Network Security Conference (ACNS)*, pages 206–221, 2005.

[22] T. Grieve. The decline and fall of the enron empire. Salon Magazine, October 2003.

[23] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Proc. USENIX Security Symposium*, July 1996.

[24] U.S. health insurance portability and accountability act (HIPAA). `www.hhs.gov/ocr/hipaa`.

[25] N. M. Haller. The S/Key One-Time Password System. In *Proc. ISOC Symposium on Network and Distributed System Security*, Feb. 1994.

[26] B. Klimt and Y. Yang. Introducing the Enron Corpus. In *Proc. Conference on Email and Anti-Spam (CEAS)*, July 2004.

[27] D. Micciancio. Oblivious data structures: applications to cryptography. In *Symposium on Theory of Computing*, pages 456–464, 1997.

[28] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[29] Magnetic storage device procedures. The National Security Agency Central Security Service (NSA/CSS) Policy Manual.

[30] M. Naor and V. Teague. Anti-persistence: History Independent Data Structures. In *Proc. Symposium Theory of Computing*, May 2001.

[31] K. Pavlou and R. T. Snodgrass. Forensic analysis of database tampering. In *Conference on Management of Data (SIGMOD)*, pages 109–120, 2006.

[32] R. Perlman. The ephemerizer: Making data disappear. Technical Report TR-2005-140, Sun Microsystems, 2005.

[33] Z. Peterson, R. Burns, J. Herring, A. Stubblefield, and A. Rubin. Secure Deletion for a Versioning File System. In *Proc. File And Storage Technologies (FAST)*, pages 143–154, December 2005.

[34] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2000.

[35] R. L. Rivest. The RC4 encryption algorithm, Mar 1992.

[36] R. L. Rivest and A. Shamir. Payword and micromint: Two simple micropayment schemes. In *Proceedings of the International Workshop on Security Protocols*, pages 69–87, London, UK, 1997. Springer-Verlag.

[37] J. M. Rosenbaum. In defense of the delete key. *The Green Bag*, 3, 2000.

[38] Sqlite. Available at `www.sqlite.org`.

[39] Secure hash standard. *Federal Information Processing Standards Publication (FIPS PUB)*, 180(1), April 1995.

[40] J. Shetty and J. Adibi. The enron email dataset database schema and brief statistical report. Technical report, Information Sciences Institute, 2004.

[41] A. Silberchatz, H. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 5th edition, 2006.

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

[43] M. Stonebraker and L. A. Rowe. The design of postgres. In *SIGMOD Conference*, pages 340–355, 1986.