Securing history: Privacy and accountability in database systems

Gerome Miklau
miklau@cs.umass.edu

Brian Levine
brian@cs.umass.edu

Patrick Stahlberg
patrick@cs.umass.edu

University of Massachusetts
140 Governors Drive
Amherst, MA 01003

ABSTRACT

Databases that preserve a historical record of activities and data offer the important benefit of system *accountability*: past events can be analyzed to detect breaches and maintain data quality. But the retention of history can also pose a threat to privacy. System designers need to carefully balance the need for privacy and accountability by controlling how and when data is retained by the system and who will be able to recover and analyze it. This paper describes the technical challenges faced in enhancing database systems so that they can securely manage history. These include: first, assessing the unintended retention of data in existing database systems that can threaten privacy; second, redesigning system components to avoid this unintended retention; and third, developing new system features to support accountability when it is desired.

1. INTRODUCTION

Because errors and malicious behavior can never be perfectly avoided, many applications that manage sensitive data preserve a historical record of activities and data. This provides *accountability* because past events can be analyzed to detect breaches, maintain data quality, and audit compliance with security policies. For example, when managing medical information, accountability is crucial. If false information is discovered in a patient's medical record, it is important to find out who is responsible for the error, when it occurred, and how the erroneous data may have been used. Inexpensive storage makes the preservation of all past database states feasible, and many have argued that versioning in databases should be the rule and not the exception [48, 49, 2, 29].

Yet there are settings where retaining a history of past data or operations poses a serious threat to privacy and confidentiality. For example, in large institutions and enterprises, systems that retain data for too long risk unwanted disclosure (either by security breach or forced by subpeona) and may violate privacy regulations that mandate the timely removal of data [32, 25, 23]. Similarly, without precise control over data destruction, unwelcome remnants of past data can become a serious problem. For example, researchers recovered a wealth of sensitive data after inspecting a sample of decommissioned hard drives [27]. Digital documents have been found to include sensitive content believed to be deleted [24, 14]. Email was used in court cases against Enron employees and released to the public [30, 47], some contained in "deleted items" folders [33].

Systems that "forget history" can preserve privacy, and they have real value in certain settings. Rosen [44] reports anecdotal evidence of this claim when he describes an antiquated pharmacy in Washington, D.C. It keeps no computer records, and has a booming business supplying antidepressants, Viagra, and other sensitive medications to prominent political figures. In addition, a Minnesota judge has argued that the use of deleted but recoverable digital data should be outlawed [45].

The fact is, privacy and accountability are both legitimate goals. But they are often at odds, and system designers need to carefully manage the balance between them. The central issues are how and when data is retained by the system and who will be able to recover and analyze it. This paper argues for enhancements to database systems that allow users to **securely manage history**, balancing the needs for privacy and accountability. In settings that require it, databases should be configurable as "memoryless" systems that protect privacy by resisting unauthorized attempts to trace activities or recover deleted data. In other settings, databases should support accountability by efficiently retaining history, enabling its analysis, and controlling access to it. Unfortunately, as we describe below, existing databases are ill-equipped for balancing privacy and accountability.

Threats to privacy

Existing database systems threaten user privacy and the confidentiality of data by *inadvertently* retaining historical traces of data and operations. A functioning database system makes 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 also pushes copies of data records into general file system space. When

3^{rd} *Biennial Conference on Innovative Data Systems Research (CIDR)* January 7-10, 2007, Asilomar, California, USA.

data is deleted, it is not destroyed and may persist on disk. Data owners currently have little or no control over these operations, and cannot say with certainty where sensitive data may end up, whether it is destroyed after deletion, or how long it will persist.

The emerging field of computer forensics is concerned with the analysis of systems in order to validate hypotheses about past activities. A forensic analysis of a database system can reproduce partial histories from the unintended traces stored by the system. Typically, the forensic investigator is a law enforcement professional who has seized a computer and gained unrestricted access to the disk image. But forensic analysis can be performed by a range of potential adversaries: hackers, privileged insiders, or anyone who has gained physical access to hardware through theft or loss. While data encryption can help protect against unauthorized access, it may not deter insiders and is often an unenforced policy [54]. Individuals concerned about personal privacy must consider threats that may result from forensic analysis.

Insufficient accountability

The unintended retention of historical information is pervasive in databases but does more to weaken privacy than strengthen accountability. First, there is no way to guarantee that needed information will be available, since data retention is incidental (see Section 2). Second, there is a basic asymmetry between privacy and accountability. A single sensitive data value retained and recovered in an inappropriate context can violate privacy; e.g., a social security number or medical diagnosis. But individual retained values are not sufficient for accountability, which often requires fairly complete evidence of past events.

Existing database systems include a number of components designed to intentionally retain history. Transaction logs record all changes to the database, backups are retained, and many systems include operational logs that record queries issued, access control operations, and system monitoring facts, among other items. In addition, researchers have proposed a range of versioning and archiving mechanisms for databases that retain complete histories and can offer query capabilities over past states of the database [40, 51, 50, 11, 34, 35].

But these technologies have some severe limitations. Transaction logs and backups do not provide an efficient means of querying past states of a database. Archiving and temporal databases are not widely used in practice, and if versioning is required it is often implemented in middleware. Furthermore, in all these systems, the mechanisms for controlling access to logs and historical records maintained for accountability are extremely limited. Historical data and logs are wholly outside traditional database access control mechanisms. Overall, existing systems lack efficient and configurable means for retaining desired historical information, analyzing that information to provide accountability, and controlling access and retention of that information for privacy purposes.

Contributions

The remainder of the paper describes our current efforts in engineering database systems capable of managing history securely.

We begin by describing the threats to privacy that result from the forensic analysis of database systems. Our initial work [37] has investigated four popular database systems and one embedded database library — Postgres, MySQL, IBM DB2, and SQLite, respectively — demonstrating the somewhat surprising persistence of data in table storage. Expired tuples can often be recovered long after they have been deleted. In addition, data values have a complex lifetime — not only within files allocated to the database, but also in file system storage outside of database control. We briefly summarize in Section 2 the main findings from our preliminary work [37].

This forensic analysis shows that the standard interface to the database (i.e., SQL) does not reliably represent the actual stored contents of the database. Such a disconnect between the interface and actual system state is a serious concern for data owners because it gives a false view of data retention. To address this problem we describe in Section 3 a novel system design goal called *transparency*. 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. In cases where data is intentionally retained beyond its active lifetime, its persistence in the system should be evident to users and accurately configurable. We describe the technical challenges in modifying system components (table storage, indexes, and logging) for transparency. These challenges include implementing secure deletion and making data structures history-independent.

Finally, in Section 4, we describe key challenges of intentionally building accountability features into database systems. The main goals are to collect appropriate data through persistence mechanisms, to permit analysis of the data through efficient query processing, and importantly, to protect the data by controlling its periods of retention and the parties that can access it. To enhance the protection of history, we also describe a historical *redaction* operation which can be used to remove all past versions of sensitive items from the database.

We describe work related to each of these topics within the relevant sections.

2. FORENSIC ANALYSIS OF DATABASES

Computer forensics is an emerging field [18] which has studied the recovery of data from file systems [17, 28, 26, 27], and the unintended retention of data by applications like web browsers and document files [26]. Forensic tools like the Sleuth Toolkit [16] and EnCASE Forensic [22] 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 provide support for analysis of database files.

Forensic analysts typically have unrestricted access to storage on disk. We consider as our threat model adversaries

with such access, as this models the capabilities of system administrators, a hacker who has gained privileges on the system, or an adversary who has breached physical security. We also note that database storage is increasingly *embedded* into a wide range of common applications for persistence. For example, embedded database libraries like BerkeleyDB [5] and SQLite [46] are used as the underlying storage mechanisms for email clients, web browsers, LDAP implementations, and Google Desktop. For example, Apple Mail.app uses an SQLite database file to support searches on subject, recipient, and sender (stored as `~/Library/Mail/Envelope Index`). Recently, Mozilla has adopted SQLite as a unified storage model for all its applications. In Firefox 2.0, remote sites can store data that persists across sessions in an SQLite database as a sophisticated replacement of cookies. The forensic analysis of such embedded storage is particularly interesting because it impacts everyday users of desktop applications, and because embedded database storage is harder to protect from such investigation.

Forensic analysis can be applied to various components of a database system, and it reveals not only data currently active in the database, but also previously deleted data and historical information about operations performed on the system. Record storage in databases contains data that has been logically deleted, but not destroyed. Indexes also contain such deleted values, and in addition may reveal, through their structure, clues about the history of operations that led to their current state. Naturally, the transaction log contains a wealth of forensic information since it often includes before- and after-images of each database update. Other sources of forensic information include temporary relations (often written to disk for large sort operations), the database catalog, and even hidden tuple identifiers that may reveal the order of creation of tuples in the database. The goal here is to understand the magnitude of data retention, and to measure (or bound) the expected lifetime of data.

We also note that encrypted storage is often not sufficient to address these threats. As a practical matter encrypted storage is not widely used. In databases, encryption often introduces unacceptable performance costs. In addition, forensic investigators or adversaries may recover cryptographic keys because they are shared by many employees, easily subpoenaed, or stored on disk and recovered.

2.1 Forensic analysis of table storage

In initial work [37], we have studied the unintended retention of data in real database systems including PostgreSQL, MySQL, IBM DB2, and a popular embedded database, SQLite.

Our first observation is that, in all systems studied, *record deletion is insecure*. Deletion of records is accomplished by setting a deletion bit — the data is not overwritten and is fully recoverable. This recoverable data is expired (it cannot be retrieved in query results) and we call it **database slack**, abbreviated *DB-slack*.

The lifetime of database slack in record storage is difficult to predict. After deletion, 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 whether the new record fits in the free space (since attributes of the record may be variable-length) and whether a sort order is imposed on the table by clustering constraints. In addition, because pages in record storage become fragmented over time, there is a table reorganization command (referred to here as *vacuum*), which is executed periodically by the database administrator. When vacuum executes, records are copied within and across pages, and the size of the file used for table storage may be reduced, thus returning space to the file system.

Vacuum has the potential to reduce recoverable data by overwriting *DB-slack* data. But we have found the dominant effect of vacuum is actually to increase forensically recoverable data. This is a consequence of our second finding, which is that *the vacuum operation is insecure*. All systems we examined returned space to the file system without overwriting data. This means copies of database records are moved to unallocated file system space and remain recoverable. Data that persists in the filesystem after being removed from database files is called **file system slack**, abbreviated *FS-slack*.

The distinction between *DB-slack* and *FS-slack* is important. *DB-slack* exists in files allocated to the database system. It therefore cannot be removed by sanitization procedures implemented in the file system, and it cannot be solved by using a file system with secure deletion [6]. We discuss the removal of slack data in Section 3.

Tracing tuple lifetime in table storage

Understanding the lifetime of data values in a database system is important for providing privacy guarantees. The state diagram in Figure 1 illustrates the complex flow of data in a database system during its lifetime. Data begins in the *Active* state, upon insertion into the database. Ideally, *Active* data would immediately become *Lost* upon deletion, following the transition along the bottom of the diagram. This does happen with some updates to tuples that immediately overwrite data. But in most other cases, data follows a different path in the diagram. Deletions, as well as updates that expand variable length fields, result in expired data that is preserved as *DB-slack*, shown by the upward arrow leaving the *Active* state in Figure 1.

Once data is preserved as *DB-slack*, it can later become *Lost* 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 may also return allocated file space to the file system, and in doing so *DB-slack*, instead of becoming *Lost*, can become *FS-slack* data. In fact, we have found that vacuum can push copies of *Active* records becoming *DB-slack* or *FS-slack*. Thus, the states in Figure 1 are not mutually exclusive — data items can be *Active* and in *FS-slack* simultaneously. Finally, *FS-slack* can become *Lost* from normal operation in which it is allocated to a new process (and overwritten with zeros) or as the result of file system sanitization, if it is performed.

Workload experiments

To understand the impact of these findings, we monitored *DB-slack* and *FS-slack* in real systems under a simulated workload of tuple insertions, deletions, and updates. We

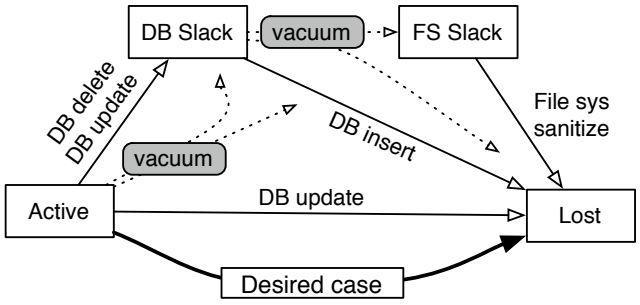

Figure 1: A state diagram describing the flow of data during its lifetime. Data begins in an *Active* state. Before it is removed and becomes *Lost* it will often be retained as *DB-slack* and/or *FS-slack*.

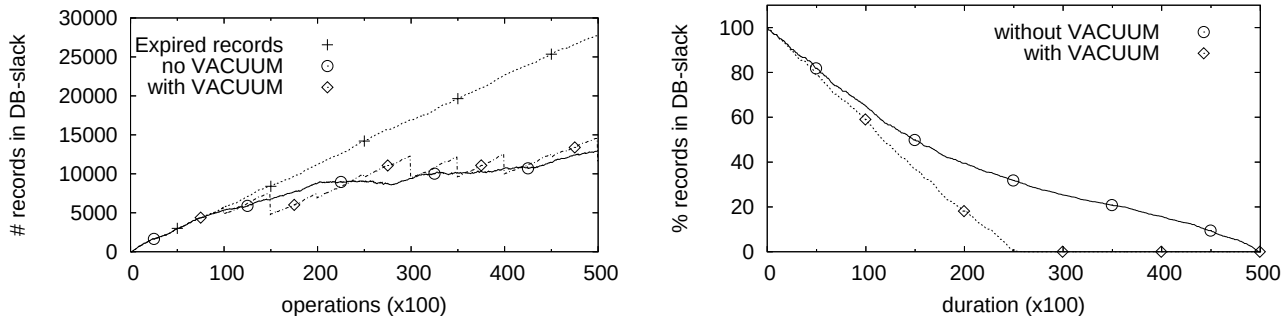

Figure 2: Measures of recoverable data, for the MySQL system using the InnoDB table format, under a synthetic workload of update, deletes, and inserts: (a) *DB-slack* with and without vacuum. (b) cummulative measure of the lifetime of tuples in *DB-slack*.

built preliminary forensic recovery tools to parse database files and recover *DB-slack* and used file system forensic techniques to recover *FS-slack*.

Although all systems shared the same basic properties outlined above, we found some interesting differences using simulated workloads. PostgreSQL had the highest retention of data in table storage. Tuples are never overwritten, except during vacuum, which is a consequence of legacy persistence features in the PostgreSQL storage manager (see Section 4.1). Both MySQL (using the MyISAM table format) and DB2 had generally low retention of data in table storage at default settings. However, in both cases we found that the existence of clustered indexes can dramatically increase slack data retained in table storage. This is because clustering imposes a constraint on the reuse of freed space in record storage, reducing the likelihood that data in *DB-slack* is overwritten. In addition, DB2 has two configurable parameters[1] intended to improve the performance inserts and updates by leaving more space free on pages and leaving some entirely free pages during VACUUM. These are effectively "knobs" for the database system that enhance performance at the cost of unintended data retention.

As one sample of our experimental results, we describe the effect of vacuum on recoverable data in MySQL. Using the InnoDB table format, we found that vacuum increased data in *DB-slack* because table reorganization was performed by simply copying all active tuples to a new portion of the tablespace. Figure 2(a) demonstrates this last result, showing the growth of *DB-slack* in MySQL (InnoDB). The Expired Records line in the graph reflects the maximal recoverable data, while the other two lines show the number of expired tuples recoverable with and without vacuum. Although vacuum increases the number of tuples recoverable from *DB-slack*, it does tend to decrease the expected lifetime of tuples, as shown in Figure 2(b).

[1]These parameters are called FREEPAGE and PCTFREE.

2.2 Forensic analysis of indexes

There are two sources of forensic recovery from B+tree indexes. First, deletion in B+trees is logical — deleted sort keys are not overwritten and can persist in the internal nodes (much like table storage above). 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.

As an example, consider two tables with indexes which both undergo insertions of the same 100 elements. Suppose the insertions for the first table are ascending by sort key, but for the second table are descending by sort key. Although the contained sort keys are the same, the structure of the resulting B+trees will be quite different, which is an indication that history is preserved by B+tree operations. In

general, inspection of the B+tree structure may allow an investigator to infer that the record with key k_1 was inserted into the table before the record with k_2. This is information about records in the database that is outside the data model, and cannot be collected through the query interface to the database. There are cases where the order of events is a crucial aspect to forensic analysis or auditing.

The degree to which data structures reveal information about their past states has been considered before [38, 36], although not specifically for B+trees in databases. The precise analysis of the information that can be gathered from the structure of B+trees is an interesting open problem. However, the practical impact of this analysis may be small. First, we hypothesize that in relational databases, the information gathered would be subsumed by information that could be collected from studying OIDs assigned to tuples, or other artifacts in the database. Second, disclosures from B+tree structures are likely to shrink as the order (the number of keys per internal node) increases, and B+trees are typically high order.

Analyzing history from index structures may be most relevant to embedded database tables, like BerkeleyDB [5] indexes. These indexed tables are often used to store application data, and are commonly replicated and shipped to untrusted environments where the presence of data remnants and history is a serious threat. We briefly discuss some techniques to address this disclosure in the next section.

2.3 Forensic analysis of the transaction log

Write-ahead logging is the most common logging strategy in existing systems [43]. After each update to the database, a log record will include before- and after-images of modified data pages. Thus, for periods of time covered by the log, all prior states of a database can be reconstructed, and a wealth of data can be recovered.

While collecting data from the log is straightforward, it is more difficult to derive a bound on the length of the retention period. Transaction logs are often implemented as circular files, and log records are written sequentially. As the file grows, it wraps around, overwriting old records. The amount of time data persists in the transaction log depends on the space allocated to the log file, the rate of update operations, the log space required per update, and the frequency of checkpointing. An enterprise could cycle its log in a few days, effectively bounding the amount of retained data in the system. But other applications may generate long histories that persist nearly indefinitely. In the next section, we discuss techniques for removing log records that are no longer needed for abort or recovery.

3. DESIGNING PRIVATE SYSTEMS

The problem exposed by forensic analysis is that the standard interface to the database (i.e., SQL) does not reliably represent the actual stored contents of the database. Deleted tuples are omitted from query results but remain in database storage. Tuples do not have an "age" or order of creation in the intended data model, yet such an order can be recovered from the physical representation. Such a disconnect between the interface and actual system state is a serious concern for data owners because it gives a false view of data retention.

To address this problem we propose that systems be *transparent*, so that they represent faithfully the data retained. Ideally, data deleted by users should be destroyed, including all copies of data. If data is retained beyond deletion for a legitimate purpose, users should have clear and accurate bounds on the lifetime of data, and they should be able to tune these bounds whenever possible using system parameters. Transparency allows users to know if their system satisfies *privacy* policies.

Note that our definition of a transparent system is not at odds with various legitimate reasons to intentionally retain data after deletion. These include: versioning databases, which preserve past states of a database that can be queried; database recovery mechanisms, which retain deleted data in the transaction log to provide atomicity and durability; and backups, taken for auditing and to recover from media failure. First of all, when backups and archiving are performed, it is important to guarantee that *only* the intended valid data is preserved in the backup, not unexpected remnants of past states. Second, if handled appropriately, intentional data retention can be exposed through a legitimate interface and configured. In fact, a straightforward way of achieving transparency goals is to preserve historical versions of the database and simply expose them faithfully through the query interface. (This is the focus of Section 4.)

There will always be cases, however, where versioning is not desired. In this case, the database must remove deleted data securely, provide indexes that are history-independent, and enable transparency in applications built using databases. We describe these technical challenges next.

3.1 Secure deletion in databases

The security community has studied secure deletion for backup logs [9], traditional and versioning file systems [6, 42], and for an email storage manager supporting timely message expiration [41]. There are two basic techniques for the physical removal of data: physical destruction by overwriting and encryption followed by key disposal. The security of overwriting for data removal was first investigated by Guttman [31]. He argued that overwritten data could be recovered from disk storage, though this is increasingly unlikely given the density of modern disks [6, 26]. Nevertheless, overwriting each deleted byte with zeroes may introduce a substantial performance penalty for large blocks of data. A clever alternative, first proposed by Boneh and Lipton [9], stores data in encrypted form and disposes of the encryption key (by overwriting). While overwriting keys is efficient, the granularity of the encryption must match the granularity of deletion. For example, in table storage, secure deletion of tuples would require one key per tuple. Key management overhead, as well as the penalty of performing decryption/encryption with each read/write is likely to make this an undesirable strategy.

Database versus filesystem solutions

In addition to *how* secure deletion may be performed, we must consider the part of the system responsible for implementing deletion. Figure 3 summarizes a range of possi-

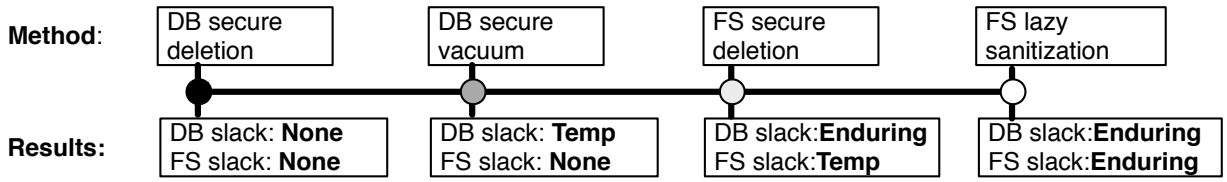

Figure 3: A spectrum of solutions to forensic recoverability of data: from database to file system.

ble solutions for *where* to implement secure deletion. The choice involves tradeoffs in performance, security, and the complexity of engineering the solution. At one end of the spectrum, secure deletion and update can be implemented by the database. In this case, *DB-slack* and *FS-slack* are completely eliminated because data is destroyed immediately. To get an upper bound on the performance impact of secure deletion, we tested a naive secure deletion consisting of an update (overwriting each attribute of a record with zeros) followed by deletion. We used MySQL (MyISAM), with random deletions on an indexed table of fixed-length attributes. Simulating secure deletion decreased the execution rate of deletions by about half, from 13,175 per second to 6,995 per second. This naive user-level implementation is far from optimal, and it suggests that reasonable performance could be achieved by integrating secure deletion into the database system. Also, when deletion operations are part of a larger workload of read and write queries, even the witnessed overhead may not be unacceptable.

If temporary *DB-slack* is tolerable, secure deletion performed lazily by the database will offer better performance. Although a few applications may want a dedicated lazy deletion procedure, it makes sense to combine secure deletion with the vacuum procedure. Such a secure vacuum procedure would reorganize records on the page, destroy any expired data in *DB-slack*, and remove slack data before space is returned to the file system, thus avoiding *FS-slack* as well.

Modifications to the file system offer better performance, but sacrifice security. Bauer et al. found synchronous secure deletion in file systems unacceptably slow, but implemented a research prototype of the ext2 [15] filesystem which performed asynchronous secure deletion by overwriting. Using this filesystem would eliminate *FS-slack* after a short period of time — but would not remove *DB-slack*. An advantage of file system improvements is generality: it avoids having to modify each database system (and each application that stores structured data) to obey forensic transparency requirements.

Unfortunately we are not aware of any file systems in common use that offer secure deletion. Therefore, the final solution is lazy sanitization of the file system, at the discretion of the user or database administrator. A number of tools exist for disk sanitization, that can be configured to periodically remove data from unallocated file system space. These tools often lack privileges to perform secure deletion in a thorough manner. Most try to fill the free space on the disk with a large file, overwriting all bytes. This technique may leave file metadata remnants recoverable.

Given these considerations, our view is that the database storage manager should support secure deletion. We believe that secure deletion can be implemented using overwriting without an unreasonable performance penalty, especially if a small bounded time lag is permitted.

Expunction from the transaction log

It is not hard to identify portions of the transaction log that will never be used by the transaction manager for recovery or abort and can freely be deleted. This is typically any log entry recorded prior to the penultimate checkpoint. In some settings, the timely removal of this data may be desirable. Encryption with key disposal, while probably inappropriate for table storage, is a promising approach for secure deletion in the transaction log. Encryption is particularly appropriate in this case because log records are written once, requiring only a single encryption operation. If system failure or transaction abort does not occur, decryption may never be necessary. Keys can be stored in the transaction table and deleted in the course of standard transaction table maintenance. We hypothesize that the penalty for encryption/decryption of log records during normal transaction processing would be small, and that secure deletion would be feasible.

3.2 History-independent indexes

As mentioned above, database indexes can reveal information about past history through their structure and through their internal representation. Formal definitions of so-called "anti-persistence" have been developed in [36, 38]. An "oblivious" 2-3 tree is proposed in [36], whose shape reveals nothing about past operations applied to the tree. "History-independent" data structures have been designed [38] that avoid disclosures resulting from the shape and from the memory representation of the data structure.

One technique for a perfectly secure B+tree index is to enforce a canonical form after each update. This will be prohibitively expensive. Another technique is to randomize operations or interpose local rebalancing, which can obscure disclosures. Designing B+tree indexes that bound historical information disclosed without sacrificing performance is an open challenge.

3.3 Advice for the practitioner

In the short-term, a costly and time-consuming redesign of database and file system internals is not a feasible solution to the threats to privacy present in database systems. We therefore suggest some simple steps that will reduce unintended data retention in current systems.

- The transaction log should be a circular file. The maximum space allocated to the log file is a configuration parameter that should be set in conjunction with the checkpointing rate to insure that the log will cycle with an acceptable frequency.
- For highly sensitive attributes (e.g., credit card and social security numbers), it may be worth simulating secure deletion at the user level, as noted in Sec. 3.1. Replacing expired data values with NULL should be avoided, since in many systems (PostgreSQL, MySQL (InnoDB), and DB2) this leaves data recoverable.
- In most systems, vacuum will remove retained data from *DB-slack* (MySQL (InnoDB) is the exception) and generate *FS-slack*. Vacuum should be followed immediately by file system sanitization.

3.4 Promoting transparency in DB applications

Insecure deletion, and the unintended data retention it causes, is a systemic problem. It has been studied in filesystems [6], volatile memory [20, 19], and now database storage. The fact is, whenever space is allocated for management by a higher level component of the system, deletion may be performed insecurely. In particular, this pattern may be repeated whenever a database schema includes BLOB or CLOB data types. In that case the database allocates a large block of bytes, and it cannot securely delete the data stored within the block when it expires without an explicit instruction from the application. BLOB/CLOBs are commonly used to store complex objects, multimedia data, and XML data, and retention of this data can pose a serious threat. The secure deletion techniques described above should be implemented and exposed at the user level to allow application designers to remove slack data securely.

4. DESIGNING ACCOUNTABLE SYSTEMS

In the previous section, we discussed the challenge of designing database systems that reliably represent to the user the actual state of stored data, to align system behavior with a desired privacy policy. In this section, we address the complementary problem of designing accountable database systems. Such systems require two key capabilities. First, complete historical data must be efficiently collected and integrated to permit accountability inquiries. Second, historical data must be *protected*, which requires extending current access control policies, and supporting redaction and expunction operations, as described below. Before discussing these challenges, we reviewing the capabilities of existing systems.

4.1 Existing capabilities

Conventional database systems are designed for efficient access to the current state of the database (often called a *snapshot*). Using the transaction log, some past states of the database can be recovered. For example, many systems support "point in time recovery", which will typically redo transactions starting from a backup version, but may also undo from a current version [39]. In either case, reinstating a past version by repeating transactions is an expensive operation that must be completed before any query on the past state can be executed. If we wish to analyze, for example, the evolution of a single tuple through states of the database, these features are inadequate. Traditional databases also support operational logging, which may preserve records of the administrator's actions, queries submitted, remote connections, etc.

Preserving history in databases has received considerable attention from the research community and goes by many names: persistent databases, archiving databases [11], versioning databases [53], transaction-time databases [34, 35]. The Postgres system was originally designed to support versioning, in which expired versions of tuples are retained and time stamped based on the commit time of the modifying transaction. A deletion never destroys data, and an update to a data item creates a new version. It is possible to pose queries "as-of" any past point in time. Full versioning support has not been widely implemented in commercial systems. Versioning capabilities were eliminated from the PostgreSQL system as of version 6.3 (released in March 1998), apparently for performance reasons [1]. Techniques from temporal databases are also closely related [40], and a number of data models and query languages have been proposed. Some research prototypes have attempted to build temporal or transaction-time databases on top of existing systems like MySQL [51], BerkeleyDB [50], and SQL Server [35]. TSQL [52] is a well-known extension to SQL which provides temporal capabilities.

Though there is a large body of existing work in this area, a number of challenges remain for providing accountability. We do not focus on the challenges of building a versioning database — we assume that such a system is available, perhaps modeled after recent work [51, 35].

4.2 Integrating and querying historical data

Accountability inquiries often involve the analysis of sequences or time series of values in a database, as well as the users who performed operations [18]. To support such inquiries we propose to integrate base data with the history of user operations involved in creating, updating or selecting data records. Currently, some of this data is recorded only in transaction or operational logs and is not typically part of conventional or versioned data stores. We propose to have an integrated data model with no logical distinction between base data and operational history. Physically, the accountability metadata may be stored separately from base data, but it should be managed by the database system.

This could be thought of as a versioning database system that also maintains *accountability provenance*. Database provenance is concerned with tracing and recording the origin of data and its movement through databases. There has been substantial work on data provenance [21, 12, 13], including two recent research prototypes [3, 10]. However our focus is on maintaining security provenance of a single database, as opposed to managing provenance in data warehouses or integrated databases that have typically been studied. Accountability inquiries require an emphasis on *who* took what actions and *when* they were taken.

Managing accountability data will allow investigators to analyze patterns of access and activity by users. But merely posing "as-of" queries, which produce consistent answers

as of a past moment in time, will not be sufficient. New language features will be needed to express accountability queries. A number of query languages have been developed in the context of temporal databases [40]. They enable comparison across versions, but may need to be extended for integrated accountability data.

4.3 Protecting history

Protecting history means defining and enforcing access rights over historical retained data. Protection of history is particularly important since historical data is, by definition, stored for long periods of time. Access control in traditional databases regulates only the current snapshot. When applied to versioned tuples (typically represented in relational tables with timestamp fields) SQL access control is insufficient because it does not support row-level policies. While temporal access control models [8, 4, 7] have been proposed, they emphasize access rights that change over time, which is different than specifying stable access rights over time-varying datasets. The design of an access control model to support descriptive policies over historical data appears to be an open problem. Access control rules must be capable of stating expressive conditions about time, database operations, as well as data objects. In addition, audit and transaction logs are not protected by classical database authorization mechanisms. Most systems apply only coarse-grained access controls to logs, implemented by the file system, in which the privilege to access a log entails complete access.

Redaction and expunction in a versioning database

Even in a versioning system where explicit deletion does not occur, there may be sensitive values that should be removed. For example, an enterprise may wish to retain a history of all past states of a customer's account, but may at some point wish to remove all records of that customer's credit card number. We believe that such an action should be a basic capability of database systems that retain history, and should be implemented as a *redaction* operation. Redaction requires efficient identification of all stored copies of a data value, which must be securely deleted and replaced with special values to indicate that data has been removed. This may be contrasted with an *expunction* operation that would remove the records along with all evidence of their existence. Note that while updates in a typical versioning database change the current version only, these privileged operations modify history. Thus there are challenges to query consistency in this model: if historical redaction is possible, queries over past states of the database must produce answers containing references to redacted values that are still meaningful to users.

The desired access control model will allow the accurate description of policies over this extended data model and will accommodate these novel privileges. Enforcement mechanisms for such policies are likely to rely on both *active* access control enforcement (in which a conventional access control monitor regulates access) as well as so-called *passive* access control enforcement (in which cryptography is used to regulate access). Passive access control seems particularly appropriate to the efficient enforcement of redaction and expunction.

5. CONCLUSION

We described the design criteria and technical challenges in building a database system whose "historical memory" can be safely and accurately managed. In settings where privacy is critical, databases should delete expired data in a timely manner and remove activity histories from storage. In settings where accountability is a priority, a database system should allow users to efficiently monitor systems and safely protect the sensitive data that results from monitoring. We believe managing history securely will grow to be a critical feature as systems are increasingly capable of retaining all past states and as security concerns require controlled accountability.

6. REFERENCES

[1] PostgreSQL release notes, v 6.3. Available at: `www.postgresql.org/docs/current/static/release-6-3.html`, March 1998.

[2] Serge Abiteboul, Rakesh Agrawal, Phil Bernstein, and et. al. The lowell database research self-assessment. *Commun. ACM*, 48(5):111–118, 2005.

[3] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha U. Nabar, Tomoe Sugihara, and Jennifer Widom. Trio: A system for data, uncertainty, and lineage. In *VLDB*, pages 1151–1154, 2006.

[4] Vijayalakshmi Atluri and Avigdor Gal. An authorization model for temporal and derived data: securing information portals. *ACM Trans. Inf. Syst. Secur.*, 5(1):62–94, 2002.

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

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

[7] Elisa Bertino, Claudio Bettini, Elena Ferrari, and Pierangela Samarati. A temporal access control mechanism for database systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):67–80, 1996.

[8] Elisa Bertino, Claudio Bettini, and Pierangela Samarati. A temporal authorization model. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 126–135, New York, NY, USA, 1994. ACM Press.

[9] Dan Boneh and Richard J. Lipton. A revocable backup system. In *USENIX Security Symposium*, pages 91–96, 1996.

[10] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *ACM SIGMOD Conference on Management of Data*, pages 539–550, New York, NY, USA, 2006. ACM Press.

[11] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang Chiew Tan. Archiving scientific data. In *SIGMOD Conference*, 2002.

[12] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[13] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. On propagation of deletions and annotations through views. In *PODS '02*, pages 150–158, 2002.

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

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

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

[17] Brian Carrier. *File System Forensic Analysis*. Addison-Wesley Professional, 2005.

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

[19] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proc. USENIX Security Symposium*, August 2004.

[20] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *Proc. USENIX Security Symposium*, August 2005.

[21] Yingwei Cui and Jennifer Widom. Practical lineage tracing in data warehouses. In *International Conference on Data Engineering*, pages 367–378, 2000.

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

[23] European union directive on privacy and electronic communications. `register.consilium.eu.int/pdf/en/02/st03/03636en2.pdf`.

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

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

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

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

[28] Tal Garfinkel, Ben Pfaff, Jim Chow, and Mendel Rosenblum. Data Lifetime is a Systems Problem. In *Proc. ACM SIGOPS European Workshop*, September 2004.

[29] Jim Gray. Database operating systems: Storage and transactions. Invited Talk, SIGMOD, 2006.

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

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

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

[33] Bryan Klimt and Yiming Yang. Introducing the Enron Corpus. In *in Proc. Conference on Email and Anti-Spam (CEAS)*, July 2004.

[34] David Lomet, Roger Barga, Mohamed Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. Immortal db: Transaction time support for sql server. In *SIGMOD Conference*, 2005.

[35] David Lomet, Roger Barga, Mohamed Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. Transaction time support inside a database engine. In *International Conference on Data Engineering*, 2006.

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

[37] Gerome Miklau, Patrick Stahlberg, and Brian Levine. Threats to privacy in the forensic analysis of database systems. Technical report, University of Massachusetts Amherst, 2006 *Submitted for Publication*.

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

[39] Oracle flashback technology. Available at `www.oracle.com/technology/deploy/availability/htdocs/Flashback_Overview.htm`, 2006.

[40] Gultekin Özsoyoglu and Richard T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Trans. Knowl. Data Eng.*, 7(4):513–532, 1995.

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

[42] 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.

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

[44] Jeffrey Rosen. *The Unwanted Gaze: The destruction of privacy in America*. Random House, 2000.

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

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

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

[48] Avi Silberschatz, Michael Stonebraker, and Jeff Ullman. Database systems: achievements and opportunities. *Commun. ACM*, 34(10):110–120, 1991.

[49] Avi Silberschatz, Mike Stonebraker, and Jeff Ullman. Database research: achievements and opportunities into the 1st century. *SIGMOD Rec.*, 25(1):52–63, 1996.

[50] Richard T. Snodgrass and Christian S. Collberg. The τ-BerkeleyDB temporal subsystem. Available at `www.cs.arizona.edu/tau/tbdb/`.

[51] Richard T. Snodgrass and Christian S. Collberg. The τ-MySQL transaction time support. Available at `www.cs.arizona.edu/tau/tmysql`.

[52] Richard Thomas Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Norwell, MA, USA, 1995.

[53] Michael Stonebraker and Lawrence A. Rowe. The design of postgres. In *SIGMOD Conference*, pages 340–355, 1986.

[54] Hope Yen. VA worker had OK to use data at home. In *Boston Globe*, June 29 2006.