Carving Database Storage to Detect and Trace Security Breaches

James Wagner[a], Alexander Rasin[a], Boris Glavic[b], Karen Heart[a], Jacob Furst[a], Lucas Bressan[a], Jonathan Grier[c]

[a] *DePaul University, Chicago, IL, USA*
[b] *Illinois Institute of Technology, Chicago, IL, USA*
[c] *Grier Forensics, USA*

Abstract

Database Management Systems (DBMS) are routinely used to store and process sensitive enterprise data. However, it is not possible to secure data by relying on the access control and security mechanisms (e.g., audit logs) of such systems alone - users may abuse their privileges (no matter whether granted or gained illegally) or circumvent security mechanisms to maliciously alter and access data. Thus, in addition to taking preventive measures, the major goal of database security is to 1) detect breaches and 2) to gather evidence about attacks for devising counter measures. We present an approach that evaluates the integrity of a live database, identifying and reporting evidence for log tampering. Our approach is based on forensic analysis of database storage and detection of inconsistencies between database logs and physical storage state (disk and RAM). We apply our approach to multiple DBMS to demonstrate its effectiveness in discovering malicious operations and providing detailed information about the data that was illegally accessed/modified.

Keywords: database forensics, file carving, memory analysis

1. Introduction

Database Management Systems (DBMSes) are commonly used to store sensitive data and, accordingly, significant effort has been invested into securing DBMSes with access control policies. However, once a user has gained elevated privileges in the DBMS (either legitimately or through an attack), the security scheme put into effect can be bypassed, and therefore, can no longer assure that data is protected according to policy. A well-known fact from security research and practice is that it is virtually impossible to create security measures that are unbreakable. For example, access control restrictions 1) may be incomplete, allowing users to execute commands that they should not be able to execute and 2) users may illegally gain privileges by using security holes in DB or OS code or through other means, e.g., social engineering. Thus, in addition to deploying *preventive* measures such as access control, it is necessary to be able to 1) *detect security breaches* when they occur in a timely fashion and 2) in event of a detected attack *collect evidence* about the attack in order to devise counter-measures and assess the extent of the damage, e.g., what information was leaked or perturbed. This information can then be used to prepare for legal action or to learn how to prevent future attacks of the same sort.

When malicious operations occur, whether by an insider or by an outside attacker that breached security, an audit log containing a history of SQL queries may provide the most critical evidence for investigators [1]. The audit log can be used to determine whether data has been compromised and what records may have been accessed. DBMSes offer built-in logging functionality but can not necessarily guarantee that these logs are accurate and have not been tampered with. Notably, federal regulations, such as the Sarbanes-Oxley Act [2] and the Health Insurance Portability and Accountability Act [3], require maintaining an audit trail, yet the privileged user can skirt these regulations by manipulating the logs. In such cases, companies maintaining these systems are, technically, in violation of these regulations. Hence, assurance that security controls have been put into place properly cannot rest merely on the existence of logging capabilities or the representations of a trusted DBA. Internal controls are needed in order to assure log integrity.

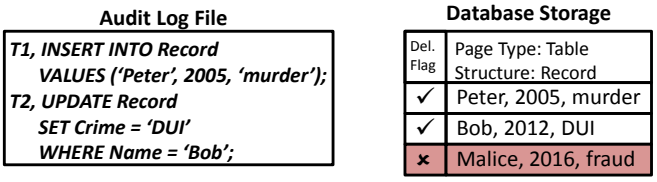

Figure 1: Illustrates that the active records for Peter and Bob can be explained by audit log events, whereas the deleted record Malice can not be explained by any audit log events.

Example 1. *Malice is the database administrator for a government agency that keeps criminal records for citizens (an example instance is shown in Figure 1). Malice recently got convicted of fraud and decided to abuse her privileges and delete her criminal record by running* `DELETE FROM Record WHERE name = 'Malice'`. *However, she*

Email addresses: jwagne32@depaul.edu (James Wagner), arasin@depaul.edu (Alexander Rasin), bglavic@iit.edu (Boris Glavic), kheart@depaul.edu (Karen Heart), jfurst@depaul.edu (Jacob Furst), lucasbressan3@gmail.com (Lucas Bressan), jdgrier@grierforensics.com (Jonathan Grier)

is aware that database operations are subjected to regular audits to detect tampering with the highly sensitive data stored by the agency. To cover her tracks, Malice deactivates the audit log before running the `DELETE` *operation and afterwards activates the log again. Thus, there is no log trace of her illegal manipulation in the database. However, database storage on disk will still contain evidence of the deleted row (until several storage artifacts caused by the deleted are physically overwritten). Our approach detects traces of deleted and outdated record versions and matches them against the audit log to detect such attacks and provide evidence for how the database was manipulated. Using our approach, we would detect the deleted row and since it does not correspond to any operation in the audit log we would flag it as a potential evidence of tampering.*

In Section 3 we showcase, for several databases, how an attacker like Malice can ensure that her operations are not being included in the audit log. Given that it is possible for a privileged attacker to erase log evidence and avoid detection, the challenge is to detect such tampering and collect additional evidence about the nature of the malicious operations (e.g., recover rows deleted by a malicious operation). It may not be immediately clear that this recovery of evidence is possible at all. However, any operation leaves footprints in database storage on disk (writes) or in RAM (both reads and writes). For instance, DBMSes mark a deleted row rather than overwrite it. Thus, if we recover such evidence directly from storage then, at least for some amount of time until the deleted value is overwritten by future inserts, we would be able to detect that there exists a discrepancy between the content of the audit log and database storage.

Given that evidence of operations exists in database storage, the next logical question to ask is whether Malice can remove this evidence by modifying database files directly. While a user with sufficient OS privileges may be able to modify database files, it is extremely challenging to tamper with database storage directly without causing failures (e.g., DBMS crashes). Direct manipulation of DBMS files will uncover the tampering attempt because: 1) in addition to the actual record data on a page, the database system maintains additional references to that record (e.g., in index structures and page headers). Deleting a record from a page without modifying auxiliary structures accordingly will leave the database in an inconsistent state and will lead to crashes; 2) databases have built-in mechanisms to detect errors in storage, e.g., checksums of disk pages. A tampering attempt has to correctly account for all of these mechanisms; 3) incorrect storage for a value can corrupt a database file. To directly modify a value, an attacker needs to know how the DBMS stores datatypes.

Because it is not only hard but, at times, next to impossible to spoof database storage, it follows that database storage can provide us with valuable evidence of attacks. We use an existing forensic tool called `DICE` [4] to reconstruct database storage. However, we are still left with the problem of matching recovered artifacts to queries in audit log – doing so requires a thorough analysis of how database storage behaves. Our approach automatically detects potential attacks by matching extracted storage entries and reporting any artifacts that cannot be explained by logged operations (summarized in Figure 2). Our method is designed to be both *general* (i.e., applicable to any relational database) and *independent* (i.e., entirely outside of DBMS control). Our system `DBDetective` inspects database storage and RAM snapshots and compares what it finds to the audit log; the analysis of this data is then done out of core without affecting database operations. `DBDetective` can operate on a single snapshot from disk or RAM (i.e., multiple snapshots are not required), but additional snapshots provide extra evidence and improve detection quality. Data that has changed between two snapshots need be matched only against audit log entries of commands that were executed during the time span between these snapshots. Thus, more frequent snapshots increase the detection accuracy because it is less likely to match a row against an incorrect operation and the probability that deleted rows are still present is higher. Moreover, frequency of snapshots increase the performance of detection because a smaller number of recovered rows have to be matched against a smaller number of operations. We can reduce storage requirements by only storing deltas between snapshots in the same fashion as incremental backups are used to avoid the storage overhead of full backups.

Our focus is on identifying the likelihood of database tampering, as well as pointing out specific inconsistencies found in database storage. Determining the identity of the party responsible for database tampering is beyond the scope of this paper. Due to the volatile nature of database storage, it is not possible to guarantee that all attacks will be discovered. We will discuss how false negatives or positives can occur for particular types of tampering in Sections 4 and 5. It may sound unsatisfactory that we are not able to detect all attacks. However, these types of attack bypass the database audit log and thus have no chance of being detected natively.

In this paper, we demonstrate techniques to detect and identify database operations that were masked by the perpetrator through use of our system `DBDetective`. For each of the major DBMSes we evaluated, we assumed that the DBMS has enabled an audit log to capture SQL commands that are relevant to an investigation. We further assumed an attacker who found a way to prevent logging of executed malicious commands by: a) deactivating audit policies and temporarily suspending logging or b) altering the existing audit log (both discussed in Section 3).

By applying forensic analysis techniques to database storage or buffer cache and matching evidence uncovered by these techniques against the audit log, we can:

- Detect multiple types of database access and manipulation that do not appear in the DBMS audit log.
- Classify unattributed record modifications as an obfuscated `INSERT`, `DELETE`, or `UPDATE` command.

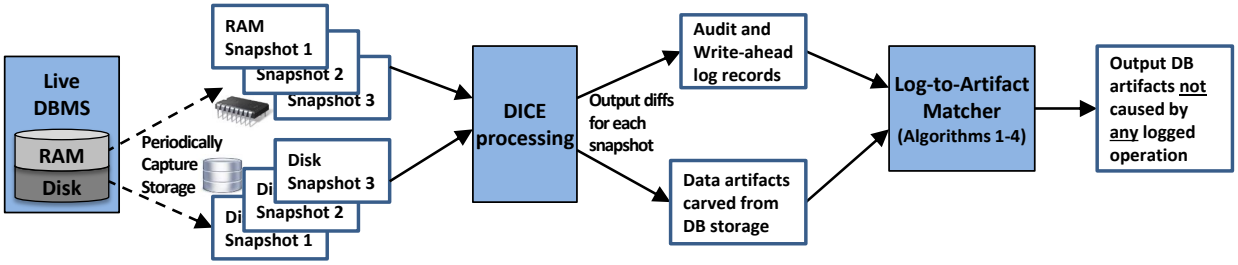

Figure 2: Architecture of the DBDetective.

- Detect cached data from (read-only) SELECT queries that cannot be derived from activity in the audit log.

The rest of the paper is organized as follows: Section 2 covers related work. Section 3 discusses DBMS logging mechanisms and how operations can be hidden from logs by an attacker. Section 4 details how table modifications that are missing from the log files can be identified in storage. Section 5 discusses how read-only (SELECT) queries hidden from the logs can be detected based on memory snapshots. We evaluate our system in Section 6.

2. Related Work

2.1. Database Forensics

Database page carving [4] is a method for reconstructing the contents of a relational database without relying on file system or DBMS metadata. Database carving is similar to traditional file carving [5, 6] in that data, including deleted data, can be reconstructed from images or RAM snapshots without the use of a live system. The work in [7] presented a comparative study of the page structure of multiple DBMSes. Subsequent work [8] described how long forensic evidence resides within a database even after being deleted and defragmented. While a multitude of built-in and 3^{rd} party recovery tools (e.g., [9, 10, 11]) can extract database storage, none of these tools are helpful for independent audit purposes because they only recover "active" data. A forensic database tool (just like a forensic file system tool) should also reconstruct unallocated pieces of data, including deleted rows, auxiliary structures (indexes) or buffer cache space.

For our storage analysis we rely on DICE tool created by Wagner et al. [8]. DICE accepts a snapshot of disk or RAM and produces text output listing all reconstructed content. We contacted Wagner et al. to acquire a copy for the experimental evaluation presented here.

2.2. Database Auditing and Security

Peha used one-way hash functions to verify an audit log and detect tampering [12]. They relied on an external, trusted notary to keep track of every transaction. Snodgrass et al. also used a one-way hash function to validate audit logs [13]. Alternatively, their hash function uses the record itself and a last modification timestamp, avoiding the external notary. Pavlou et al. expanded this work by determining when audit log tampering occurred [14]. While this mechanism ensures an accurate audit log with high probability by sending the secure hashes to a notarization service, it is ultimately useless if logging has been disabled by a privileged user. Our approach detects log tampering even if logs files have been disabled.

Sinha et al. used hash chains to verify log integrity in an offline environment [15]. In this situation, communication with a central server is not required to ensure log authenticity. Crosby et al. proposed a data structure called a history tree to reduce the log size produced by hash chains in an offline environment [16]. Rather than detecting log tampering, Schneier and Kelsey made log files impossible to read and impossible to modify [17]. Under this framework, an attacker does not know if his activity has been logged, or which log entries are related to his activity.

An event log can be generated using triggers, and the idea of a SELECT trigger was explored for the purpose of logging [18]. This would allow all table access to be logged – but a malicious user could also utilize triggers to remove traces of her activity or simply bypass a SELECT trigger by creating a temporary view to access the data.

ManageEngine's EventLog Analyzer [19] provides audit log reports and alerts for Oracle and SQL Server based on actions, such as user activity, record modification, schema alteration, and read-only queries. However, the Eventlog Analyzer creates these reports based on native DBMS logs. Like other forensic tools, this tool is vulnerable to a privileged user who has the ability to alter or disable logs.

Network-based monitoring methods have received significant attention in audit logging research because they can provide independence and generality by residing outside of the DBMS. IBM Security Guardium Express Activity Monitor for Databases [20] monitors incoming packets for suspicious activity. If malicious activity is suspected, this tool can block database access for that command or user. Liu et al. [21] monitored DBAs and other users with privileged access. Their method identifies and logs network packets containing SQL statements.

The benefit of monitoring activity over the network and, therefore, beyond the reach of DBA's, is the level of independence achieved by these tools. On the other hand, relying on network activity ignores local connections to the DBMS and requires intimate understanding of SQL commands (i.e., an obfuscated command could fool the system). By contrast, our approach detects both local and

network activity because SQL is ultimately run against the database instance affecting database storage state.

3. Reliability of Database Logs

An attacker can alter two types of logs to interfere with an investigation: write-ahead logs (WAL) and audit logs (event history records). WALs record database modifications at a low level in order to support ACID guarantees, providing a history of recent table modifications. Audit logs record configured user database actions, including SQL operations and other user activity.

WALs. WALs cannot normally be disabled or easily modified, and require a special-purpose tool to be read (e.g., Oracle LogMiner or PostgreSQL pg_xlogdump). Some DBMSes allow WALs to be disabled for specific operations, such as bulk load or structure rebuild. Thus inserting records without leaving a log trace can be done through this feature. Since the WAL file format is not human-readable, and requires specific tools for parsing, this would seem to protect it from tampering. However, DBMSes (including Oracle, MySQL, PostgreSQL, and SQL Server) allow the administrator to switch to a new WAL file and delete old WAL files. Therefore, executing a WAL switch and deleting the original WAL can effectively allow a user to perform transactions without leaving a WAL record. For example, an administrator could switch from log file A to log file B, perform the malicious SQL operation(s), switch back to log file A (or a new log file C), and delete log file B. For example, to implement this operation in Oracle:

1) `ALTER DATABASE ADD LOGFILE ('path/logB.rdo')`
2) `ALTER SYSTEM SWITCH LOGFILE`
3) Run the malicious SQL operation(s)
4) `ALTER SYSTEM SWITCH LOGFILE`
5) `ALTER DATABASE DROP LOGFILE MEMBER 'path/logB.rdo'`

Audit Logs. Audit logs store executed SQL commands based on logging policies that are configured by database administrators. Therefore, an administrator can disable logging or modify individual log records as they see fit. For example, records in the Oracle `sys.aud$` table can be modified with SQL commands, and records in the PostgreSQL `pg_audit` log and MySQL general query log are stored as human-readable text files. Table 1 summarizes how to modify the audit log for three major DBMSes.

DBMS	Command
Oracle	SQL commands against `sys.aud$`
PostgreSQL	Edit files in the `pg_log` directory
MySQL	Edit the `general_log_file`

Table 1: Commands to edit the audit log.

4. Detecting Hidden Record Modifications

When a table record is inserted or modified, a cascade of storage changes occurs in the database. In addition to the affected record's data itself, page metadata may be updated (e.g., a delete mark is set) and page(s) of an index storing the record may change (e.g., to reflect the deletion of a record). Each of the accessed pages would be brought into RAM if it is not already cached. Row identifiers and structure identifiers can be used to tie all of these changes together. Furthermore, DBAs can also disable logging for bulk modifications (for performance considerations); this privilege can be exploited to hide malicious modifications. In this section, we describe how we detect inconsistencies between modified records and logged commands.

4.1. Deleted Records

Deleted records are not physically erased but rather marked as "deleted" in the page; the storage occupied by the deleted row becomes unallocated space, and eventually will be overwritten by a new row. Unlike audit log records, these alterations to database storage cannot be bypassed or controlled – thus if a reconstructed deleted record does not match the `WHERE`-clause condition of any delete statement in the audit log, then a log record is missing.

`DICE` returns the status of each row as either "deleted" or "active." Reconstructed deleted rows and the audit log are used in Algorithm 1 to determine if a deleted row can be matched with at least one `DELETE` command. Here we use $cond(d)$ to denote the condition of delete d. The conditions of delete operations may overlap, potentially creating false-negative matches (i.e., a delete's condition may match a row that was *already* deleted by another `DELETE`). However, we are interested in identifying deleted rows in storage that do not match any delete operation in the log. A false-negative match presents a problem if it hides a missing match with a delete that the attacker attempted to hide. Only if *all* reconstructed deleted rows that the attacker attempted to hide have false-negative matches will the attack go unnoticed, because a single unaccounted for deleted record is sufficient to detect suspicious activity.

Algorithm 1 Accounting for Deleted Records in Log Files

```
1: Deletes ← DELETE statements in audit log
2: DelRows ← deleted records reconstructed by DICE
3: Unaccounted ← DeletedRows
4: for each d ∈ Deletes do
5:     for each r ∈ DelRows do
6:         if r ⊨ cond(d) then
7:             Unaccounted ← Unaccounted − {r}
8: return Unaccounted
```

Figure 3 gives an example for detecting unaccounted deleted rows. `DICE` reconstructed three deleted rows from the Customer table: (1,Christine,Chicago), (3,Christopher, Seattle), and (4,Thomas,Austin). The log file contains two operations: `DELETE FROM Customer WHERE City = 'Chicago'` (T1) and `DELETE FROM Customer WHERE Name LIKE 'Chris%'` (T2). In Algorithm 1, DeletedRows was set to the three reconstructed deleted rows. Algorithm 1 returned (4,Thomas, Austin), indicating that this deleted record could not be

attributed to any of the deletes. We cannot decide which operation caused deletion of (1,Christine,Chicago) row (T1 or T2), but that is not necessary for our purpose of finding that record #4 is an unattributed delete.

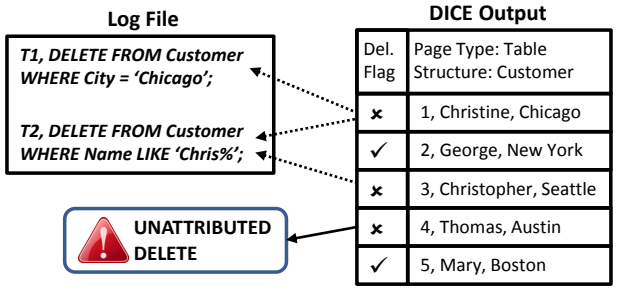

Figure 3: Detecting unattributed deleted records.

4.2. Inserted Records

New inserted rows are either appended to the end of the last page (or a new page if the last page is full) of a table or overwrite free space created by previously deleted rows. A new row has to be smaller than or equal to the old deleted row to overwrite its previous storage location; some databases (Oracle and PostgreSQL) explicitly delay the overwriting unallocated page space. When an inserted row is smaller than the deleted row, only a part of the deleted row is overwritten leaving traces of the old row behind. If an "active" new table row does not match any of the insert operations from the audit log, then this row is a sign of suspicious activity. These "active" records are used in Algorithm 2 to determine if a reconstructed row can be attributed to an insert from the audit log.

Algorithm 2 Accounting for Inserted Data in Log Files

```
1: InsertedRows ← rows created by INSERT log entries
2: Rows ← reconstructed active rows
3: Unaccounted ← Rows
4: for each r ∈ Rows do
5:     if r ∈ InsertedRows then
6:         Unaccounted ← Unaccounted − {r}
7: return Unaccounted
```

Figure 4 shows an example for detecting an INSERT operation that does not match any commands in the audit log. The log contains six operations. As rows are inserted from T1 to T4, they are appended to the end of the table. At T5, (3,Lamp) was deleted followed by an insert of (5,Bookcase) at T6. Since row (5,Bookcase) is larger than the deleted row (3,Lamp), it is appended to the end of the table. DICE reconstructed five active records, including (0,Dog) and (2,Monkey). Rows was initialized to the five reconstructed active rows for Algorithm 2. Algorithm 2 thus returned (0,Dog) and (2,Monkey) because these records could not be matched to logged inserts (only the latter is an INSERT as we will see in Section 4.3). The character p found with (0,Dog) was not part of the record, indicating that this record overwrote a previously deleted row. Since (0,Dog) is one character smaller than (3,Lamp) and the last character from (3,Lamp) was found, it was likely that (0,Dog) overwrote the deleted record (3,Lamp). We describe how to confirm this in Section 4.4.

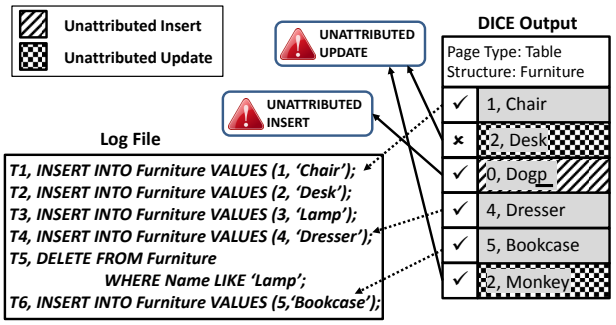

Figure 4: Detecting unattributed inserted and updated records.

4.3. Updated Records

An UPDATE operation is essentially a DELETE operation followed by an INSERT operation. To account for updated rows, we use unmarked deleted rows returned by Algorithm 1 and unmarked inserted rows returned by Algorithm 2 as the input for Algorithm 3. If a deleted row can be matched to the WHERE clause of an update, then this deleted row operation is marked as present in the log. Next, if an unmarked inserted row can be matched to the value from the SET clause, and the inserted row matches all values in the deleted row except for the SET clause value, then this inserted row operation is present in the log. Currently, our implementation is limited to simple SET clause expressions of the form A = c for an attribute A and constant c. In the algorithm, we use $cond(u)$ for an update u to denote the update's WHERE clause condition and $set(u)$ to denote the its set clause. Furthermore, we use APPLY(r,s) to denote evaluating SET-clause s over row r.

Algorithm 3 Accounting for Updated Data in Log Files

```
1: Deleted ← unmarked deleted rows from Alg. 1
2: Inserted ← unmarked inserted rows from Alg. 2
3: Updates ← set of updates from the audit log
4: for all r_del ∈ Deleted do
5:     if ∃u ∈ Updates : r_del ⊨ cond(u) then
6:         Deleted ← Deleted − {r_del}
7:         for all r_ins ∈ Inserted do
8:             if APPLY(r_del, set(u)) = r_ins then
9:                 Inserted ← Inserted − {r_ins}
10: return Deleted, Inserted
```

Figure 4 also shows an example of how we detect an UPDATE operation not present in the log. Algorithm 1 returned the row (2,Desk), and Algorithm 2 returned the row (0,Dog) and (2,Monkey). Using these sets of records, Algorithm 3 returned (2,Desk) as the list of deleted records, and (0,Dog) and (2,Monkey) as the list of inserted records. Additionally, Algorithm 3 recognized the shared value, 2,

for the first column in (2,Desk) and (2,Monkey). While this does not confirm an `UPDATE` operation by itself, it is reasonable to conclude that (2,Desk) was updated to (2,Monkey).

4.4. Indexes

In some cases, records from table pages are insufficient to draw reliable conclusions about record modification. For example, in Figure 4 we did not have enough information to confirm that (3,Lamp) was overwritten by (0,Dog). Reconstructed index pages provide additional information because deleted index values have a significantly longer lifetime compared to deleted records themselves [8]. Using the pointer associated with deleted (but still recoverable) index entry allows us to determine values previously stored at a particular location within a page.

Figure 5 demonstrates how old index values supply evidence of a deleted record that was overwritten by new values. The index stores the furniture table ID and a pointer to the row address. Using index pointers, we can be certain that the overwritten row once stored record with ID of 3. This allows us to extrapolate a partial deleted record, (3, ?), that we can include in Algorithms 1 and 3. If a secondary index on the second column (furniture name) is available, we could also extrapolate Lamp from the index.

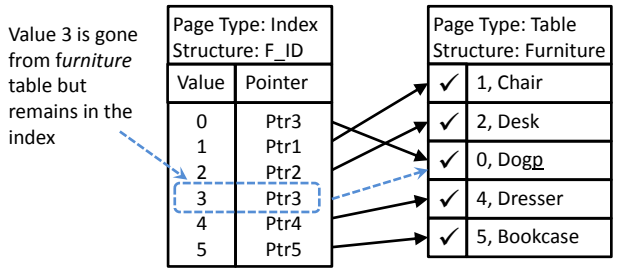

Figure 5: Matching index values to table records.

5. Detecting Inconsistencies for Read-Only Queries

DBMSes use a component called buffer manager to cache pages from disk into memory. Data is read into the buffer pool in units of pages, that can be reconstructed by `DICE`. In this section, we describe how artifacts carved from the buffer pool can be matched to read-only queries in the audit log. A database query may use one of two possible way of accessing a table: a full table scan (FTS) or an index scan (IS). An FTS reads all table pages, while an IS uses an index structure (e.g., B-Tree) to retrieve a list of pointers referencing particular table pages (or rows) to be read based on a search key. All accessed index pages and some of the table pages (depending on access type) are placed in the buffer pool by the DBMS.

5.1. Full Table Scan

When a query uses an FTS, only a small part of a large table will be cached. A small table (relative to the buffer pool size) may be cached in its entirety. Every database stores a unique page identifier within the page header which allows us to efficiently match cached pages to their counterpart on disk. The particular number of pages cached by a FTS can be derived from the size of the table, although it is not always proportional (e.g., a larger table may result in fewer cached pages). Thus, after FTS is executed, typically pages from the physical end of table storage would be in the cache (i.e., a few pages including the one where new inserts would be appended). In Section 6.3 we analyze caching behaviour for multiple DBMSes.

Figure 6 provides an example of an FTS over the Employee table. We can identify pages that belong to Employee by the structure identifier 131, which is stored in the page header. `DICE` can return just the page structure identifiers (without parsing page content) at a much faster speed. Both Q2 and Q4 access Employee via an FTS. Each time the Employee table is scanned, the same four pages (identifiers: 97, 98, 99, and 100) from the table are loaded into the buffer pool. Therefore, when four pages with the page identifiers 97, 98, 99, and 100 and a structure identifier of 131 are found in memory, a FTS on the Employee table can be assumed.

5.2. Index Access

DMBSes use IS to optimize performance for queries that access data based on the key attributes of an index. Caching of index pages identifies what attribute was queried (a query posed conditions over this attribute) and provides a rough estimate of what value range was selected for an indexed attribute (since values stored in index pages are ordered). Cached index pages are more precise in determining what the query accessed because cached table pages contain the entire table row (regardless of which columns were accessed), but index pages contain only the relevant columns. A sequence of index pages in the buffer pool that does not correspond to any logged query can present evidence of hidden access. Algorithm 4 describes how to use the minimum and maximum values of index pages to determine if a cached index page can be attributed to logged query. Again, $cond(q)$ denotes the conditions used by query q (OR'ed together).

Algorithm 4 Accounting for Index-Access Queries

```
1: IndexPages ← set of all cached index pages
2: Queries ← queries from the audit log
3: for each i ∈ IndexPages do
4:     if ∃q ∈ Queries : ∃r ∈ i : r ⊨ cond(q) then
5:         IndexPages ← IndexPages − {i}
6: return IndexPages
```

Figure 6 shows examples of index accesses on the Customer table. The Customer table's structure identifier is 124, and the secondary index on the C_City column has a structure identifier of 126. Q1 filters on the city Dallas, and it caches the index page with identifier 2. This page has a minimum value of Chicago and a maximum value of Detroit. Q3 filter on the city Jackson, and it caches the

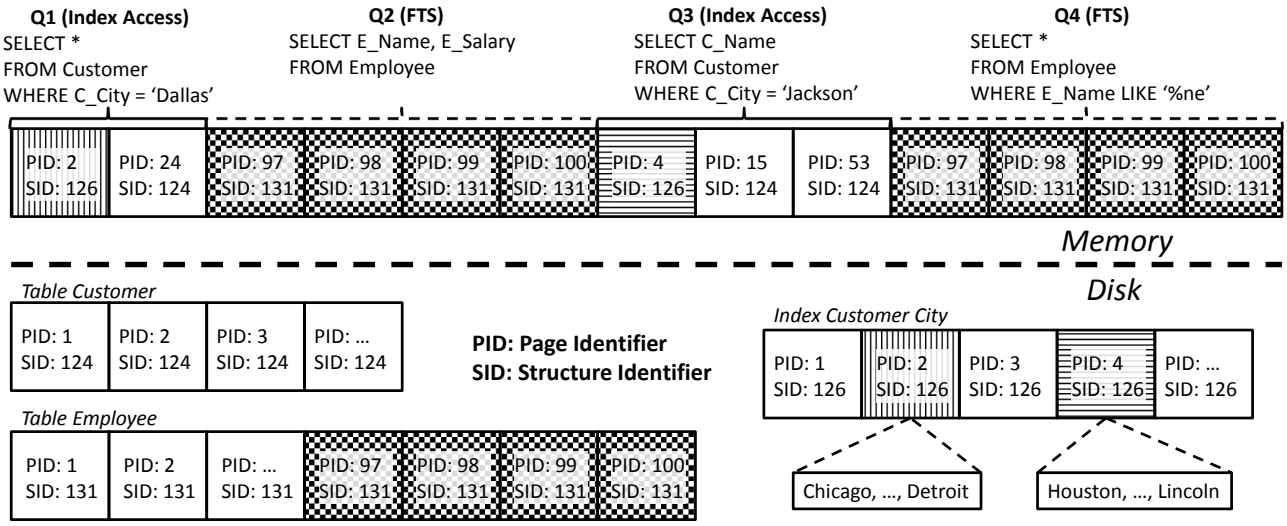

Figure 6: Read-only query matching between disk and buffer cache.

index page with the page identifier of 4. This page has a minimum value of Houston and a maximum value of Lincoln. If a query in the audit log filters on a values within the minimum and maximum range of values for an index page, then that page can be attributed to that query.

5.3. Data Lifetime in Memory

As new data is read into cache, old data is evicted (using a buffer replacement strategy such as LRU) providing us with an approximate timeline of recent accesses. A malicious user can not directly control the buffer pool; regardless of one's permission level, there are no direct APIs to control buffer pool behavior. Assuming that the attacker cannot do something as conspicuous as powering down the computer, the only available command is to flush the cache (only available in Oracle, SQL Server and MySQL). Interestingly, flushing buffer cache will *mark* pages as unallocated instead of physically evicting any data from RAM.

6. Experiments

Our experiments use three databases (Oracle, PostgreSQL, and MySQL) that we consider representative (both open- and closed-source, all three very widely used) due to space limitations. We have used data and queries from TPCC [22, 23] and SSBM [24] benchmarks. These benchmarks were used because they were designed to measure the performance of DBMSes.

Our experiments were carried out on servers with an Intel X3470 2.93 GHz processor and 8GB of RAM running Windows Server 2008 R2 Enterprise SP1 or CentOS 6.5. Windows OS memory snapshots were generated using a tool called User Mode Process Dumper (version 8.1). We used regular SATA magnetic drives for storage and processing. Linux OS memory snapshots were generated by reading the process' memory under `/proc/$pid/mem`.

6.1. Experiment 1: DBDetective Performance Evaluation

The objective of this experiment is to explore the costs associated with using `DBDetective` and the estimated reaction time to detect tampering. In Part-A of this experiment, we provide cost estimates to perform memory snapshots. In Part-B, we test the carving process performance against database files. In Part-C, we test the carving speed against memory snapshots.

Part A. To estimate the cost to perform memory snapshots, we copied a 2.5GB snapshot from an Oracle database process to a magnetic disk. This operation took approximately 31 sec. In practice, the snapshot cost is dominated by the cost of writing the result to disk but and can be sped up significantly by shipping data to a remote machine or using a faster drive (e.g., PCIe). As long as snapshots are taken as often as the entire buffer pool is replaced by query activity, we expect to detect most activity.

Part B. To obtain a performance estimate for file carving, we ran `DICE` tool against five Oracle database files ranging in size from 1MB to 3GB. All Oracle files contained 8KB database pages. We observed that `DICE` parsed the files at an average rate of 1.1 MB/s and continued to scale linearly with respect to the file size (using SATA magnetic disk).

Part C. Finally, we tested the performance of the carving tool against memory snapshots of Oracle buffer cache. We collected a 2.5GB snapshot taken from the Oracle database process and an 8GB snapshot of the entire RAM content. Each of the snapshot required detecting and parsing the contents of roughly 80,000 pages (600MB). The 2.5GB snapshot was carved at a rate of 4.2 MB/s, and the 8GB snapshot was carved at a rate of 13.2 MB/s. We can thus conclude that the runtime of page parsing depends solely on the number of database pages rather than raw file size.

6.2. Record Modification Detection

6.2.1. Experiment 2: Deleted Record Detection

The objective of this experiment is to identify deleted rows from storage that could not be matched to commands in the log files. We also evaluate the situation where a row deleted by a malicious query was overwritten or was attributed to a non-malicious query (a false-negative match).

Part A. For this experiment we use MySQL. By default, MySQL creates an index-organized table (IOT) when a primary key is declared for a table. MySQL uses the primary key as the row identifier, and all rows are physically ordered within index (leaf) pages by the row identifier. If no primary key is declared, MySQL will synthesize a unique row identifier for each row. MySQL stores the row identifier as the pointer in the index value-pointer pairs.

We initially started with the Item table (100K records) from the TPCC benchmark. We created a primary key on the I_ID column, a secondary index on the I_Name column, and a secondary index on the I_IM_ID column. Next, we issued two delete commands:
(Delete 1) `DELETE FROM Item WHERE I_Name LIKE 'w2G%'`
(Delete 2) `DELETE FROM Item WHERE I_IM_ID = 8563`.

Delete 1 represents malicious activity, and was therefore removed from the log. Delete 1 deleted records with the I_ID values of 92328 and 95136. Delete 2 is in the log and was responsible for deletion of 10 records. We used `DICE` to reconstruct deleted rows from Item in storage: and 12 deleted rows were reconstructed.

Algorithm 1 returned one record with the I_ID value of 92328. 11 of the deleted records were matched with the logged Delete 2 command: the 10 records it deleted and the record with I_ID 95136. Even though the 11^{th} record was caused by Delete 1, it resulted in false-negative match to Delete 2 because it happened to have a I_IM_ID value of 8563. However, false-negatives are only problematic if they prevent all maliciously deleted records to be detected.

Part B. Realistically, investigators may not be able to perform forensic analysis at the most opportune time. We next consider what determination can be made if the trace of the maliciously deleted record has been overwritten.

To instrument an overwrite of a deleted record in an IOT, a record with the same primary key value had to be inserted. We inserted the record (92328,100,DBCarver1,0.0, This is a trick1). The original deleted record with the I_ID value of 92328 was permanently overwritten. However, the secondary indexes on I_Name and I_IM_ID columns retain traces of this record until something causes an index rebuild. The pointers stored with index values are the row identifiers (or primary key) for table records.

We found that the row identifier 92328 had two entries in the I_Name index: the value for the current (new) record, w2GSyVRavpUbCr2bEzqOb for the old record, and two entries in the I_IM_ID index: the value for the current record and 4763 for the overwritten record. This allowed us to extrapolate a partial deleted record as an input to Algorithm 1: (92328,4763,w2GSyVRavpUbCr2bEzqOb,?,?). Since Algorithm 1 could not match the partial record to any of the logged commands, it also provides evidence of the missing log record.

6.2.2. Experiment 3: Updated Record Detection

The objective of this experiment is to identify the by-product of an `UPDATE` operation in persistent storage that can not be matched to commands in the log. Similar to Experiment 6.2.1-B, we evaluated records that were overwritten by an in-place `UPDATE`.

Part A. We again use MySQL and the Item table with 100K rows and indexes defined as in previous experiments. Records in Item include (53732, 1004, Name_Val_{53732}, 14.55, Data_Val_{53732}) where Name_Val_{53732} is Us65fCVCCfrOMDT6bpDDE and Data_Val_{53732} is mpDSxHpz0ftrSI2aP0rXpZhdYSakGcqrSqeI6a6p2cE4Q. All of `INSERT` commands creating the table were logged. Next, we issued an update,
`UPDATE Item SET I_Name = 'DBCarver' WHERE I_ID = 53732`
to simulate malicious activity, and removed this operation from the log. We then passed the database files containing the Item table and the I_Name secondary index to `DICE`.

Algorithm 2 returned the record (53732, 1004, DBCarver, 14.55, Data_Val_{53732}) since it does not match any logged `INSERT` command. `DICE` did not return deleted rows because when the row was updated, the new version of the row physically overwrote the old version. Two pieces of evidence help classify the row 53732 as an overwrite of a deleted row: table pages and the pages for the index on I_Name. In the table page, the new row used less storage space than the old overwritten row. Therefore, part of the old row was still present – 13 characters from the last column were reconstructed:
~~mpDSxHpz0ftrSI2aP0rXpZhdYSakGcqr~~SqeI6a6p2cE4Q
These 13 characters could be distinguished from new row because new row metadata specifies where the current row ends. This behavior is illustrated in Figure 4. In the secondary index page, the pointer (or row identifier) 53732 had two entries, both with the new value (DBCarver) and the old value (Name_Val_{53732}). Since the value DBCarver was present in the current active record, we could assert that DBCarver overwrote Name_Val_{53732}. This allowed us to extrapolate a partial pre-update record, (53732, ?, Name_Val_{53732}, ?, ?) despite the fact that it was destroyed.

Part B. Having detected unmatched active record (53732, 1004, DBCarver, 14.55, Data_Val_{53732}), and a partially reconstructed deleted record, (53732, ?, Name_Val_{53732}, ?, ?), we can link them as evidence of an update in Algorithm 3. First, we use Algorithm 1, which returned our partially deleted record as not a non-match. We next added our partially deleted record 53732 to Deleted and our active record to Inserted in Algorithm 3.

Algorithm 3 returned (53732, 1004, DBCarver, 14.55, Data_Val$_{53732}$) as an active record and (53732, ?, Name_Val$_{53732}$, ?, ?) as a deleted record. Since they share the 53732 primary key value, it is reasonable to conclude that these records should match with an UPDATE command, rather than both a DELETE and INSERT. Technically, this behavior could be caused by a hidden combination of DELETE and INSERT – either way, we uncovered a maliciously hidden modification. We can also determine that the third column was changed from Name_Val$_{53732}$ to DBCarver.

6.2.3. Experiment 4: Modified Record Detection

We now explore the objectives of Experiments 1 and 2 in an Oracle setting. In Part A of this experiment, we identify the by-products of DELETE and UPDATE commands in storage that do not match any logged operations. In Part B, we simulated a scenario in which deleted records are overwritten. We then determined what malicious DELETE and INSERT commands could still be detected. In Part C, we used available indexes and results from Part B to match UPDATE operations.

Unlike MySQL, Oracle does not create an IOT by default when a primary key is declared (IOTs must be created explicitly). Instead, a regular B-Tree index is created on the primary key. Without IOT, unique row identifier are not stored with each row. Instead, Oracle uses physical row identifiers consisting of a structure identifier, page identifier, and row's position within the page.

Part A. We use the TPCC NewOrders (NO_O_ID, NO_D_ID, NO_W_ID) table with 9K rows. We declared a primary key on the NO_O_ID column and a secondary index on the NO_D_ID column. Next, we issued the following queries to simulate malicious activity:

(Command 1)
```
DELETE FROM New_Orders
    WHERE NO_O_ID = 2500 AND NO_D_ID = 1
```
(Command 2)
```
UPDATE New_Orders SET NO_D_ID = 777
    WHERE NO_O_ID = 2700 AND NO_D_ID = 1.
```

We removed both Command 1 and Command 2 from the log. We then passed the database files containing the NewOrders table and both indexes to DICE.

We reconstructed the deleted record (2500, 1, 1) caused by Command 1. A copy of the indexed values for this record were reconstructed from the primary and secondary index. DICE also reconstructed the active record (2700, 777, 1) – Command 2 caused an in-place update and overwrote the old version, (2700, 1, 1). However, the old NO_D_ID value is still present in the index, and could be mapped back to the overwritten row.

Part B. To continue this experiment, we simulated normal database activity to observe what causes commands from 6.2.3-A to be no longer immediately detectable. This was done by repeatedly deleting 10 records using the NO_O_ID column, and inserting 20 records. We passed the database files containing the NewOrders table and indexes to DICE after each set of operations. We passed the carved output to Algorithms 1 and 2 after each set of operations.

After the first and second sequence of 30 commands, Algorithm 1 returned (2500, 1, 1), and Algorithm 2 returned (2700, 777, 1). This meant that we had detected a DELETE command and an INSERT command missing from the log file. After the third set of commands, Algorithm 1 did not return any records because (2500, 1, 1) was overwritten by an inserted record, and Algorithm 2 returned (2700, 777, 1). Now, only an INSERT command was only detected as missing from the log file.

Part C. While we detected missing operations during our simulation, we wanted to see if indexes can serve as an extra source of evidence of malicious activity. The unidentified DELETE command was no longer detected after the third set of database activity commands, and the unidentified INSERT command could have actually been an in-place update that we demonstrated in Experiment 6.2.2.

The third set of database activity commands overwrote the deleted record of interest, seemingly avoiding detection. However, we found multiple values for the pointer to this record in both the primary key index and the secondary index. We then reconstructed a partial deleted record using the index values that weren't found in the current record: (2500, 1, ?). Algorithm 1 did not associate this partial record with any DELETE command in the log file since all of the DELETE commands were on the primary key. Therefore, we had found evidence of a DELETE operation not recorded in the log files.

Throughout all of the database activity, we detected that the record (2700, 777, 1) was part of an INSERT command removed from the log files. However, more conclusions could be derived from the index values.We found the one value for the pointer in the primary key index, but we found two values for the same pointer in the secondary index. This indicated that the record was likely updated by a previous command. Given the one value in the primary key index and the two values in the secondary index, we could reconstruct the partial deleted record: (2700, 1, ?). Finally, Algorithm 3 identified the commonality, 2700, between the unattributed active record, (2700, 777, 1), and the partial deleted record, (2700, 1, ?). Based on this result, it was reasonable to assume that the record with the NO_O_ID value of 2700 was involved in a hidden UPDATE command.

6.3. Read-Only Query Detection

6.3.1. Experiment 5: Full Table Scan Detection

Part A. The objective of this experiment is to demonstrate full table scan (FTS) detection. FTSes leave a consistent pattern of pages in the buffer cache for each table they access which can be detected in RAM.

We used a PostgreSQL DBMS with 8KB pages and a buffer cache of 128MB (or 16,000 pages). We evaluated FTS for two tables: the Item table (1284 pages) from the TPCC benchmark and the LineOrder table (77K pages)

from the SSBM. To do this, we ran three queries that all used an FTS. The first query accessed Item, and the second and third queries accessed LineOrder.

In Snapshot 1, we observed 32 pages from the Item table. These 32 pages that DICE reconstructed represented the 32 highest page identifiers for Item table (i.e., the last 32 pages in the physical database file), just as described in Figure 6. We verified that this is the case by inputting the Item database file into DICE. We did not observe any other cached table pages or cached index pages related to the Item table in the buffer cache. In Snapshot 2, DICE reconstructed the same 32 pages from Item and an additional 32 pages from LineOrder. The by-product from scanning Item was still detectable in memory, although it is unallocated space from DBMS's perspective. Similar to the Item FTS, the 32 pages cached for LineOrder had the highest page identifiers from the database file where LineOrder was stored. For Snapshot 3, DICE returned 32 pages from Item and 64 pages from LineOrder. The Item pages were the same pages from Snapshots 1 and 2. The new set of 32 pages from LineOrder had the exact same page identifiers, found at a different location in the memory snapshot. Each FTS access demonstrated a consistent caching pattern in PostgreSQL, 32 pages for every table, producing a new set of pages at a location in memory adjacent to the previous pattern thereby creating a crude timeline of queries in buffer cache. Note that other DBMSes exhibit their own (consistent) caching pattern for an FTS. For example, the exact number of pages cached for a table in Oracle is not constant, but relies on a predictable pattern for each table.

Part B. To demonstrate that FTS caching depends on buffer cache size, we increased buffer cache to 256MB in PostgreSQL and performed the same sequence of queries. As a result, we observed that the FTS(Item) query switched to caching the whole table (all 1284 pages). However, the FTS(LineOrder) query cached 32 pages each in the exact same pattern as before. In general, DBMSes use an internal heuristic threshold to decide when a whole table is "small enough" to be fully read into the buffer cache.

6.3.2. Experiment 6: Index Access Detection

The objective of this experiment is to demonstrate index access detection. When a table is accessed using an index, both the index pages and table pages are cached in memory. The ordered values stored in the index pages (leaves and intermediate nodes) provide a rough estimate of the range of values accessed by a query.

For this experiment, we used a PostgreSQL DBMS with 8KB pages and a buffer cache of 128MB (or 16,000 pages). We created the Item table with a secondary index on the I_NAME column. Next, we issued two queries that used an index access for the Item table:

(Query 1) `SELECT * FROM Item WHERE I_Name BETWEEN 'aa' AND 'ab'`

(Query 2) `SELECT * FROM Item WHERE I_Name BETWEEN 'ba' AND 'bb'`.

Query 1 selected 105 rows (0.08 selectivity) and Query 2 selected 109 rows (0.08 selectivity). After each query, we captured a cache snapshot that we passed to DICE.

DICE reconstructed 102 table pages and 2 leaf index pages from the memory snapshot after Query 1. Since Query 1 used a secondary index (the table is not organized on this column), almost every accessed row cached a new table page. DICE reconstructed 94 new table pages and 2 new index leaf pages from the memory snapshot after Query 2, while the pages cached by Query 1 remained in memory. Similar to Query 1, Query 2 cached a page for almost every row selected. Since the indexes stored ordered values, they provided an estimate of how the table was accessed. Table 2 summarizes the detailed breakdown of index page contents returned by DICE. Table 2 shows that a value range between 'a6j3' and 'AaBD' must have been read to cache index page 1, a value between 'AaBd' and 'ac5U' was accessed to cache index page 2, a value between 'b76G' and 'bAGT' must have been to read to cache index page 3, and a value between 'BaGW' and 'bcDi' was accessed to cache index page 4. These index value ranges matched to Query 1 and Query 2 in Algorithm 4.

Snapshot	Index Page	Min Val	Max Val
1	1	a6j3 ...	AaBD ...
1	2	AaBD ...	ac5U ...
2	3	b76G ...	bAGT ...
2	4	BaGW ...	bcDi ...

Table 2: Index page contents found in memory.

7. Conclusions and Future Work

Audit logs and other build-in DBMS security mechanisms are designed to detect or prevent malicious operations executed by an attacker. An inherent weakness of such mechanisms is that attackers with sufficient privileges can bypass them to hide their tracks. We present and thoroughly evaluate DBDetective, an approach for detecting database operations that were hidden by an attacker by removing them from the audit log and collecting evidence about what data was accessed and modified by an attacker. Our approach relies on forensic inspection of database storage and correlates this information with entries from an audit log to uncover evidence of malicious operations. Importantly, database storage is nearly impossible to spoof and, thus, is a much more reliable source of tampering evidence than, e.g., audit logs.

Given that storage snapshots provide incomplete information, we will explore probabilistic matching that determines the likelihood of a storage artifact being caused by the operations in the audit log, exploit additional constraints based on temporal ordering of operations, simulate partial histories of SQL commands from an audit log for more precise matching, and dynamically adapt the frequency of taking snapshots based on detected anomalies.

References

[1] R. T. Mercuri, On auditing audit trails, Communications of the ACM 46 (1) (2003) 17–20.

[2] S.-O. Act, Sarbanes-oxley act, Washington DC.

[3] A. Act, Health insurance portability and accountability act of 1996, Public law 104 (1996) 191.

[4] J. Wagner, A. Rasin, T. Malik, K. Heart, H. Jehle, J. Grier, Database forensic analysis with DBCarver, CIDR, 2017.

[5] S. L. Garfinkel, Carving contiguous and fragmented files with fast object validation, digital investigation 4 (2007) 2–12.

[6] G. G. Richard III, V. Roussev, Scalpel: A frugal, high performance file carver., in: DFRWS, 2005.

[7] J. Wagner, A. Rasin, J. Grier, Database forensic analysis through internal structure carving, Digital Investigation 14 (2015) S106–S115.

[8] J. Wagner, A. Rasin, J. Grier, Database image content explorer: Carving data that does not officially exist, Digital Investigation 18 (2016) S97–S107.

[9] OfficeRecovery, Recovery for mysql, `http://www.officerecovery.com/mysql/`.

[10] Percona, Percona data recovery tool for innodb, `https://launchpad.net/percona-data-recovery-tool-for-innodb`.

[11] S. Phoenix, Db2 recovery software, `http://www.stellarinfo.com/database-recovery/db2-recovery.php`.

[12] J. M. Peha, Electronic commerce with verifiable audit trails, in: Proceedings of ISOC, 1999.

[13] R. T. Snodgrass, S. S. Yao, C. Collberg, Tamper detection in audit logs, in: Proceedings of the Thirtieth international conference on Very large data bases-Volume 30, VLDB Endowment, 2004, pp. 504–515.

[14] K. E. Pavlou, R. T. Snodgrass, Forensic analysis of database tampering, ACM Transactions on Database Systems (TODS) 33 (4) (2008) 30.

[15] A. Sinha, L. Jia, P. England, J. R. Lorch, Continuous tamper-proof logging using tpm 2.0, in: International Conference on Trust and Trustworthy Computing, Springer, 2014, pp. 19–36.

[16] S. A. Crosby, D. S. Wallach, Efficient data structures for tamper-evident logging., in: USENIX Security Symposium, 2009, pp. 317–334.

[17] B. Schneier, J. Kelsey, Secure audit logs to support computer forensics, ACM Transactions on Information and System Security (TISSEC) 2 (2) (1999) 159–176.

[18] D. Fabbri, R. Ramamurthy, R. Kaushik, Select triggers for data auditing, in: Data Engineering (ICDE), 2013 IEEE 29th International Conference on, IEEE, 2013, pp. 1141–1152.

[19] Eventlog analyzer, `https://www.manageengine.com/products/eventlog/`.

[20] Ibm security guardium express activity monitor for databases, `http://www-03.ibm.com/software/products/en/ibm-security-guardium-express-activity-monitor-for-databases` (2017).

[21] L. Liu, Q. Huang, A framework for database auditing, in: Computer Sciences and Convergence Information Technology, 2009. ICCIT'09. Fourth International Conference on, IEEE, 2009, pp. 982–986.

[22] W. Kohler, A. Shah, F. Raab, Overview of tpc benchmark c: The order-entry benchmark, Transaction Processing Performance Council, Technical Report.

[23] F. Raab, Tpc benchmark c, standard specification revision 3.0, Tech. rep., Technical report. Transaction Processing Performance Council, 15 Feb (1995).

[24] P. O.Neil, E. O.Neil, X. Chen, S. Revilak, The star schema benchmark and augmented fact table indexing, in: Performance evaluation and benchmarking, Springer, 2009, pp. 237–252.