Detecting Database File Tampering through Page Carving

James Wagner, Alexander Rasin, Karen Heart,
Tanu Malik, Jacob Furst
DePaul University
Chicago, Illinois
[jwagne32,arasin,kheart,tmalik1,jfurst]@depaul.edu

Jonathan Grier
Grier Forensics
Pikesville, Maryland
jdgrier@grierforensics.com

ABSTRACT

Database Management Systems (DBMSes) secure data against regular users through defensive mechanisms such as access control, and against privileged users with detection mechanisms such as audit logging. Interestingly, these security mechanisms are built into the DBMS and are thus only useful for monitoring or stopping operations that are executed through the DBMS API. Any access that involves directly modifying database files (at file system level) would, by definition, bypass any and all security layers built into the DBMS itself.

In this paper, we propose and evaluate an approach that detects direct modifications to database files that have already bypassed the DBMS and its internal security mechanisms. Our approach applies forensic analysis to first validate database indexes and then compares index state with data in the DBMS tables. We show that indexes are much more difficult to modify and can be further fortified with hashing. Our approach supports most relational DBMSes by leveraging index structures that are already built into the system to detect database storage tampering that would currently remain undetectable.

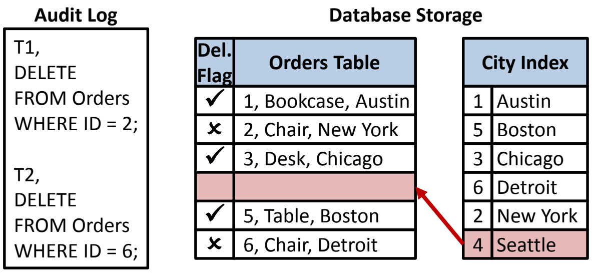
1 INTRODUCTION

DBMSes use a combination of defense and detection mechanisms to secure access to data. Defense mechanisms, such as access control, determine the data granularity and system access granted to different database users; defense mechanisms, such as audit logging, monitor all database activity. Regardless of the defense mechanisms, security breaches are still a legitimate concern – sometimes due to unintentional granting of extra access control and sometimes due to outright hacking, such as SQL injection. Security breaches are typically detected through analysis of audit logs. However, audit log analysis is unreliable to detect a breach that originated from privileged users.

Privileged users, by definition, already have the ability to control and modify access permissions. Therefore, audit logs fundamentally cannot be trusted to detect suspicious activity. Additionally, privileged users commonly have access to database files. Consider a system administrator who maliciously, acting as the root, edits a DBMS data file in a Hex editor or through a programming language, such as Python. The DBMS, unaware of external file write activity taking place outside its own programmatic access, cannot log it, and thus the tampering attack remains undetected.

Current DBMSes do not provide tools against insider threats – in general, a built-in security mechanism is vulnerable to insider attacks. While a DBMS will not be able to detect direct storage changes, file-level modifications potentially create inconsistencies within the auxiliary data structures maintained by a DBMS. Forensics tools that examine file contents can be used to detect such inconsistencies, and determine if insider threats have taken place. Recently we proposed the first database forensic tool, `DBCarver`, that can be used to detect deleted data from database pages [31]. However, database forensic tools such as `DBCarver` merely extract forensic artifacts but do not search for inconsistencies within the data structures maintained by a DBMS.

In this paper, we propose a system, `DBStorageAuditor`, that detects database file tampering by identifying inconsistencies in storage through a direct inspection of internal database structures. `DBStorageAuditor` utilizes existing database forensic techniques and expands them to extract additional necessary storage artifacts. These artifacts are then used to detect inconsistencies within indexes and between indexes and tables. The underlying premise of our approach is that all relational databases follow patterns in storage over which the privileged user has little or no control. We inspect these storage patterns to detect unusual activity. We motivate `DBStorageAuditor` through an example:



```
T1,
DELETE
FROM Orders
WHERE ID = 2;

T2,
DELETE
FROM Orders
WHERE ID = 6;
```

Del. Flag	Orders Table
✓	1, Bookcase, Austin
✗	2, Chair, New York
✓	3, Desk, Chicago
✓	5, Table, Boston
✗	6, Chair, Detroit

City Index	
1	Austin
5	Boston
3	Chicago
6	Detroit
2	New York
4	Seattle

Figure 1: Example attack through DBMS files.

Example 1. *Malice is the system administrator for a shipping company, FriendlyShipping. Malice is bribed by a competing company to interfere with the orders going to Seattle. Malice* **does not** *have access to the DBMS, but she* **does** *have access to the server where the database files reside.*

Malice writes a Python script that will open and directly modify the database file containing the `Orders` *table. The script then opens the database file, finds all records containing the string 'Seattle', and explicitly overwrites entire records with the* `NULL` *ASCII character (decimal value 0).*

Figure 1 illustrates the result of Malice's script actions. Since the record was erased without the DBMS (API has never seen that command) all DBMS security was bypassed, and the operation was never recorded in the log file. When FriendlyShipping investigates the missing Seattle orders, the audit log can only explain deleted orders for `(2, Chair, New York)` *and* `(6, Chair, Detroit)`*. The audit logs contain no trace of the Seattle order being deleted because it was not deleted but rather wiped out externally.*

To simplify in the above example, we have omitted some details of database file tampering, which we expand on later in Section 5. Barring those details in Example 1, the value in the `City` index still exists in index storage even though the entire record is erased. Therefore, an inconsistency can be identified by mapping back the index value to the empty gap in table storage. The empty gap in table storage exists because a database only marks a record when it is deleted, and only overwrites the record with data from a newly inserted record. However, making the mapping from the index value to the associated record must be based on the behavioral rules of database storage, such as page and record layout. We use database forensic tools to understand database layout, and using that layout, perform the necessary mapping.

It is not impossible for a scrupulous system administrator to (i) tamper with the index and create a cascade of inconsistencies throughout the index structure, or (ii) for an attacker who has privileges to modify database files to acquire privileges to suspend or kill logging mechanisms at the operating system level if necessary, or (iii) for a knowledgeable adversary to easily avoid corrupting storage and keep checksum values consistent. However, in spite of increased level of threat, we repeatedly show that accurate knowledge about data layout can be used to gather evidence and prove if any malicious activity has taken place.

Previously we developed an approach to detect malicious activity when DBMS logging is disabled [28]. In this approach we analyzed unlogged activity (executed through a proper DBMS API) but strictly assumed that database files were not exposed to tampering. In this paper, we address the tampering vulnerability where the database files are physically altered. Developing an auditing system for DBMSes is part of our larger goal to open up the database system and its storage to users, for performance and forensics investigation.

The rest of the paper is organized as follows: Section 2 covers related work. Section 3 discusses concepts of database storage used throughout the paper. Section 4 defines the adversary we seek to defend against. Section 5 details how to perform database file tampering. Section 6 provides an overview of DBStorageAuditor. Section 7 describes how we utilize database forensics. Section 8 addresses index tampering. Section 9 proposes a method to organize carved index output making our system scalable. Section 10 discusses how to detect file tampering using inconsistencies between carved index data and table data. Section 11 provides a thorough evaluation of our system.

2 RELATED WORK

This paper focuses on the detection of database file tampering. Therefore, we discuss work related to protecting DBMSes against privileged users as well as work that detects regular (non-DBMS) file tampering. We outline why existing file tampering and anti-forensic methods are inapplicable to database files.

2.1 Database Auditing and Security

Database audit log files are of great interest for database security because they can be used to determine whether data was compromised and what records were accessed. Methods to verify log integrity have been proposed to detect log file tampering [18, 25]. Pavlou et al. expanded upon this work to determine the time of log tampering [17]. Sinha et al. used hash chains to verify log integrity in an offline environment without requiring communication with a central server [24]. Crosby et al. proposed a data structure, history tree, to reduce the log size produced by hash chains in an offline environment [2]. Rather than detecting log tampering, Schneider and Kelsey developed an approach to make log files impossible to parse and alter [23]. An event log can be generated using triggers, and the idea of a SELECT trigger was explored for the purpose of logging [3]. ManageEngine's EventLog Analyzer provides audit log reports and alerts for Oracle and SQL Server based on actions, such as user activity, record modification, schema alterations, and read-only queries [13]. We previously described a method to detect inconsistencies between storage and log files, allowing tampering detection when logging was disabled (i.e., when an operation was excluded from the log) [28]. All of this work assumes that database storage can not be altered directly – an action which bypasses logging mechanisms.

Network-based monitoring methods have received attention in audit log research because they provide independence and generality by residing outside of the DBMS. IBM Security Guardium Express Activity Monitor for Databases [9] monitors incoming packets for suspicious activity. Liu et al. [12] monitored DBAs and other privileged users by identifying and logging 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 DBMS connections and requires intimate understanding of SQL commands (i.e., an obfuscated command can fool the system).

2.2 Database Forensics

Stahlberg demonstrated the retention of deleted data and proposed techniques to erase data for a MySQL DBMS [26]. While this work was only ever implemented for MySQL, it validates our threat model by imposing custom DBMS file modifications.

Database page carving [31] is a method for reconstructing the contents of a relational database without relying on the file system or DBMS. Page carving is inspired by traditional file carving [6, 21], which reconstructs data (active and deleted) from disk images or RAM snapshots without the need for a live system. The work in [29] presented a comparative study of the page structure for multiple DBMSes. Subsequent work in [30] described how long forensic evidence resides within a database even after being deleted or reorganized. While a multitude of built-in and third party recovery tools (e.g., [15, 19, 20]) aim to extract database storage, none of these tools are helpful for forensic analysis because they can only recover "active" data. Forensic tools, such as Sleuth Kit [1] and EnCASE Forensic [4], are commonly used by digital investigators to reconstruct file system data, but they are not capable of parsing database files. A database forensic 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.

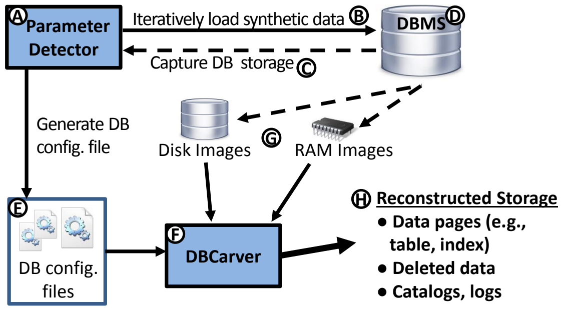


Figure 2: DBCarver architecture.

Our storage analysis relies on DBCarver tool described in [31], which was revised to process additional artifacts for this paper. Figure 2 provides an overview of DBCarver, which consists of two main components: the parameter collector(A) and the carver(F).

The parameter detector loads synthetic data into a DBMS(B), captures storage(C), deconstructs pages from storage, and describes the page layout with a set of parameters which are stored in a configuration file(E) – a text file that captures page-level layout information for that particular DBMS. These configuration files are used by the carver(F) to reconstruct DBMS content from disk images, RAM snapshots, or any other input file(G). The carver returns storage artifacts(H), such as user records, metadata describing user data, deleted data, and system catalogs.

2.3 File Tampering and Anti-Forensics

One-way hash functions have been used to detect file tampering at the file system level [7, 11]. However, we expect database files to be regularly modified by legitimate operations. Distinguishing a malicious tampering operation and a legitimate SQL operation would be nearly impossible at the file system level without knowledge of metadata in DBMS storage. Authenticating cached data on untrusted publishers has been explored by Martel [14] and Tamassia [27]. Their threat model defends against an untrusted publisher that provides cached results working with a trusted DBMS and, while our work addresses an untrusted DBMS.

Anti-forensics is defined as a method that seeks to interfere with a forensic process [8]; file tampering threat model we address in this paper exhibits anti-forensics behavioral properties. Two traditional anti-forensics techniques are data wiping and data hiding [5, 10]: 1) data wiping explicitly overwrites data to delete it rather than mark it as deleted, 2) data hiding seeks to hide the message itself. We are not aware of any existing literature that addresses anti-forensics within DBMSes [22]; we consider adding or erasing data through file tampering (that bypasses DBMS itself) to be the equivalent of anti-forensics for DBMSes.

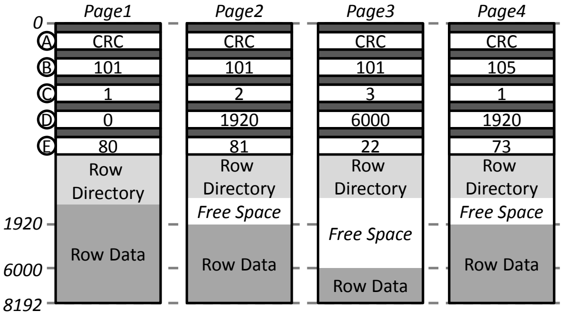


Figure 3: Example page headers.

3 BACKGROUND

The security threats we consider in this paper affect the lowest level of database storage (details of which are hidden from the users by design). In this section, we briefly generalize storage of the RDBMS row-store pages and define terminology used throughout this paper. The concepts formulated in this section apply to (but are not limited to) IBM DB2, SQL Server, Oracle, PostgreSQL, MySQL, Apache Derby, MariaDB, and Firebird.

3.1 Page Layout

When DBMS data is accessed or modified through an API, the DBMS implements data changes within pages and maintains a variety of additional metadata. While each DBMS employs its own storage engine, there are many conceptual commonalities between DBMSes in how data is stored and maintained. Every DBMS uses fixed-size pages with three main structures: header, row directory, and row data.

A DBMS page header stores metadata describing user records stored in the page. The metadata of interest (to this paper) are the checksum, object identifier, page identifier, free space pointer, and record count. Figure 3 demonstrates an example of how this metadata could be positioned in an 8K page. The checksum(A) detects data corruption within a page; whenever a page is modified, the checksum is updated. The object identifier(B) represents the database object to which the page belongs (the object name is stored in a separate system table). In Figure 3, Pages 1-3 have the object identifier 101, and Page 4 has the object identifier 105. The page identifier(C) is unique to each page for either an object, a file, or across all files. In Figure 3, the page identifier is unique for each object because the value 1 occurs for both objects 101 and 105. The free space pointer(D) references unallocated space within the page where a new record can be added. If the page is full, the free space pointer is `NULL` (decimal value 0). In Figure 3, Page1 is full since it has a `NULL` free space pointer, while Pages 2, 3, and 4 point to unallocated space. The record count(E) refers to the number of *active* records in a page. If a record is a deleted, the record count will be decremented by one, and if a record is added to a page, it will be incremented by one. In Figure 3, Page1 has 80 active records, and Page2 has 81 active records.

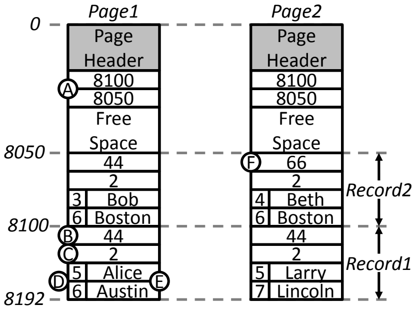


Figure 4: Example row directory and row data layouts.

The row directory stores pointers to each page record (row) – when a record is added to a page, a pointer is added to the row directory. Figure 4 shows one example of how the row directory (A) could be positioned; the row directory in this example has two pointers referencing records within the row data.

The row data stores the user data along with additional metadata. Figure 4 shows an example of how the row data may be structured (with some minor DBMS-specific variations). Each record stores the user data values (E), a row delimiter that separates the records (B), the number of columns for the record (C), and the size of each string (D).

Deleted Data. When a record is deleted, a DBMS either overwrites the row directory pointer for that record or marks the record itself in the row data – it is important to note that the record entry is not erased. Figure 4 shows an example of when the row metadata is marked for a deleted record (F). Deleted records become unallocated space, and DBMS settings and operations dictate when records are (eventually) overwritten by new data.

Index Pages. Index value-pointer pairs are stored in pages, which are similar to table pages including a header, row directory, and row data. The only significant difference between table and index pages is the layout of records – index pages store value-pointer pairs in the row data. Furthermore, in practice, index values are *not* marked as unallocated space when a corresponding table record is deleted. Stale index values persist in storage, typically until the B-Tree is explicitly rebuilt by the user and long after the table record was overwritten.

4 THREAT MODEL

In this section, we define the attack vectors, different possible adversary types, and the privileges we expect them to wield. We consider two types of privileged users: database administrator (DBA) and system administrator (SA). A DBA can issue privileged SQL commands against the DBMS including disabling logs or granting privileges to users. However, a DBA would not have administrative access to the server OS. The SA has administrative access to the server OS including the ability to suspend processes and read/write access to all files, but no access to privileged SQL commands in the DBMS. The SA can still have a regular DB user account without affecting our assumptions.

Since a DBA can bypass DBMS defense mechanisms, detection mechanisms are best suited to identify anomalous behavior. An audit log containing a history of SQL commands is accepted as one of the best detection mechanisms for a DBMS. In Section 2, we discussed prior work designed to prevent audit log tampering and detect malicious behavior in the event that logging was disabled. In this paper, we focus on a detection mechanism for a user often ignored in DBMS security, the SA.

The SA can bypass all DBMS security defense and detection mechanisms by reading and editing a database file with a tool other than the DBMS. For example, a SA could use Python to open a file and change the value 'Hank' to 'Walt.' In Section 5 we discuss additional steps that must be considered to successfully perform such an operation, but it can ultimately be achieved. Since this operation occurs outside of the DBMS, it bypasses all DBMS access control, and it will not be included any of the DBMS log files. Furthermore, one can assume that the SA would have the ability to suspend any logging mechanism in the server OS. Although changes to a file will also be recorded in the file system journal, the SA has the ability to turn off journaling to the file system by using tune2fs on Unix or the FSCTL_DELETE_USN_JOURNAL control code on NTFS (Windows). However, the file system must be shutdown first in order to prevent possible corruption. Therefore, the SA may have to effect a shutdown of the DBMS before making changes to the database files. The shutting down and restarting of the database instance and the system will generate events that are logged; however, as mentioned earlier, the SA can turn off system logging easily. Moreover, the SA could revise the DBMS log in order to hide evidence of the shutdown and restart. Hence, it would be somewhat involved but not difficult for a SA to cover his/her tracks when tampering with a DBMS file.

5 FILE TAMPERING

The threats to data we consider in this paper occur at the OS level outside of DBMS control. In this section, we formulate the threat and introduce concepts and categories of tampering.

A DBMS allows users and administrators to access and modify data through an API. Access control guarantees that users will be limited to data they are privileged to access. In this section, we discuss how an adversary can perform file tampering. To limit the scope of this paper, we assume that file tampering involves user data and not metadata (changing metadata can easily damage the DBMS but that will not alter any of its records). We define user data as records created by the user or copies of record values that may reside in auxiliary structures (e.g., indexes). File tampering actions that we discuss in this section ultimately produce one of two results in storage: 1) **Extraneous data** is a record or a value that has been added through file tampering or 2) **Erased data** is a record that has been explicitly overwritten (rather than marked deleted by a command as described in Section 3).

Three things must be considered when performing database file tampering: 1) page checksum, 2) write lock on files, and 3) dirty pages. In Section 3, we discussed the functionality and placing of the page checksum. Figure 5 shows three different page alterations, in all of which the checksum is (also) updated. Some DBMS processes hold write locks on the database files. Therefore, tampering would require that the attacker release or otherwise bypass OS file write locks. DBMSes do not immediately write pages back to disk after they are modified in the buffer cache. That is significant because a maliciously altered page on disk can be overwritten when a dirty page is flushed to disk – or, alternatively, a dirty page could be altered directly in RAM instead (bypassing file locks that way).

Write-Locks. The file locking system API, through the `fcntl` system call in Unix, is set up so that a process can prevent writes to (as well as reads from) a file that it has locked successfully. An attacker can potentially cause the process holding the lock, in this case the DBMS, to release the lock. Otherwise, a sophisticated attacker with root privileges can release the lock without involvement of the process by using kernel code. Once the lock is released, the attacker would lock the file, tamper with its content, and then release the lock. The DBMS would not receive any signal or other indication of the tampering and could continue to use the file as if it were locked after the attacker releases the lock. While the attacker holds the lock, however, DBMS access to the file would be suspended. In order to prevent the DBMS from discovering this condition, the attacker could suspend the DBMS process temporarily until the tampering has been completed. An attacker with root privileges could also mark memory used by the DBMS as shared and tamper directly with memory.

Data Encryption. Different levels of encryption can be employed to protect database files, but they can ultimately be bypassed by an adversary with SA privileges. It is reasonable to assume that the SA would have the ability to decrypt any data that has been encrypted at the OS level. The SA would most likely not have the privileges to decrypt any internal database encryption. However, individual (value or record based encryption) is still subject to tampering since the metadata describing the encrypted values is still readable. Furthermore, column-level encryption values are decrypted when they are read into memory making it possible to map the decrypted values in memory back to the encrypted values in persistent storage.

5.1 Value Modification

The first category of file tampering action we consider is value modification. Value modification is logically similar to a SQL `UPDATE` command; this type of tampering results in extraneous data. Storage space and value encoding (see Section 3) are the main considerations when modifying a value.

If a modified value requires the same storage space as the original entry, no metadata needs to be updated. If the newly modified value requires less storage than the original, then metadata needs to be modified, and other values in the record may need to be shifted. For example, many DBMSes explicitly store string sizes on page – e.g., changing 'Hank' to 'Gus' requires metadata value with the size of the string to be changed from 4 to 3. Furthermore, if the modified value is not the last column in the record, all other columns must be shifted by one byte. Only the columns in the modified record need to be shifted; other records in the page can remain as-is, leaving a gap (1 byte in our example). Shifting all other records in the page to close the gap would require all of the corresponding row directory addresses and relevant index pointers to be updated. If a value is modified to a value that requires more storage space, the old version of the record must be erased and the new version of the record must be appended to the table. These operations are discussed in the remainder of this section. Shifting the following records to accommodate a large value modification is not practical – unless the modified value happens to be in the last record on the page (and there is free space at the end of the page).

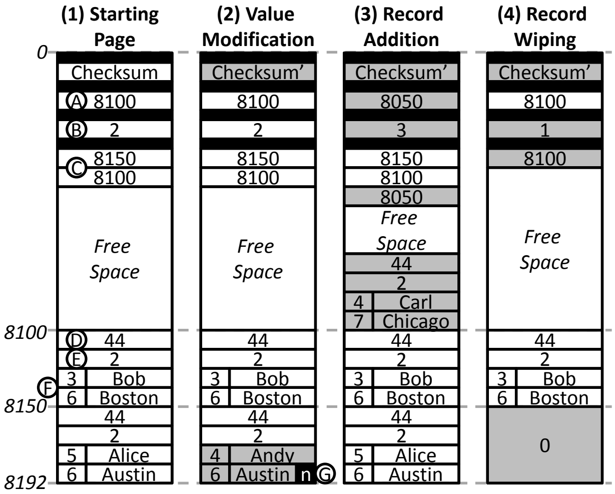


Figure 5: Database file tampering examples.

Figure 5.2 shows an example of a value changed to a smaller size. Since 'Andy' is one byte smaller than 'Alice', the column size must be changed from 5 to 4. Furthermore, the name is not the last column so next column ('Austin') is shifted by one byte, which overwrites the 'e' at the end of 'Alice' and leaves an unused 'n' character from 'Austin'.

5.2 Record Addition

The next file tampering action we consider is new record addition, which is logically similar to a SQL `INSERT` command. This type of file tampering results in extraneous data generated within the DBMS. When adding a record to a file, metadata in the row data, row directory, and page header must be considered along with the correct value encodings.

When a record is appended to an existing page, the structure of the record must match the proper active record structure for that DBMS. Section 3 discusses metadata that a DBMS uses to store records. For the DBMS to recognize a newly added record, a pointer must be appended to the page row directory. Finally, the free space pointer must be updated and the active record count (if used by the DBMS in question) must be incremented.

Figure 5.3 shows an example of the record ('Carl', 'Chicago') added to the page. Along with the values themselves, additional metadata is included in the row data. The size of each column, 4 and 7 bytes, is included, the column count, 2, and the row delimiter, 44. Next, a pointer, 8050, is added to the row directory, and the record count is updated to 3. Finally, the free space address is updated since the record was added to free space of the page.

5.3 Record Wiping

The final tampering action category we discuss is record wiping. Record wiping is logically similar to a SQL `DELETE` command, except that it fully erases the record. A proper SQL `DELETE` command will merely mark a record as deleted; record wiping explicitly overwrites the record to destroy the data, even from a forensic recovery tool. Record wiping erases data with no forensic trace as there is no indication that a record existed in a place where it was overwritten. Wiping a record from a file is essentially the reverse operation of adding a record to a file: the metadata in the row data, row directory, and page header must all be altered.

When a record is overwritten in a page, the entire record (including the metadata) is overwritten with the `NULL` ASCII character (a decimal value of 0). Next, the row directory pointer must also be overwritten in the same way. Finally, the free space pointer must be updated and the active record count (if used by the DBMS) must be decremented.

Figure 5.4 shows an example of the record ('Alice', 'Austin') erased from the page. Every byte used for the values and their metadata (column sizes, column count, and row delimiter) is overwritten with the decimal value 0. The row directory address for that row is erased and the row directory is defragmented. Finally, the record count is updated to 1.

Record Removal. Rather than explicitly overwriting a record, the record metadata could also be marked to mimic a SQL `DELETE`. We define such changes as a record removal (versus record wiping). We do not address record removal in this paper because such unlogged action can be detected by our previous work in [28] by comparing and flagging inconsistencies between DBMS storage forensic artifacts and the audit logs.

6 APPROACH OVERVIEW

Our goal in this paper is to eliminate a major security vulnerability stemming from file tampering; our solution is envisioned as a component of a comprehensive auditing system that employs database forensics. We have previously built a tool that detects malicious activity when database logging was disabled [28] by comparing forensic artifacts and database logs. That approach relied on forensic artifacts left by SQL commands and assumed no OS level file tampering. `DBStorageAuditor` finds inconsistencies that were done by direct file modification. Future work, such as recovering a time line of events or user attribution, would involve expanding upon the current components to the system.

The remainder of the paper describes our system to detect database file tampering, `DBStorageAuditor`, followed by an experimental evaluation in Section 11. Figure 6 provides an overview of `DBStorageAuditor`, which consists of four components: forensic extraction(A), index integrity verification(B), carved index sorting(C), and tampering detection(D).

The forensic processing component is based on the forensic tool `DBCarver` [31] described in Section 2. `DBCarver` retrieves from storage all table records (including deleted records), record metadata, index value-pointer pairs, and several additional storage artifacts. We discuss new functionality that was added to `DBCarver` for this paper in Section 7 (e.g., a page checksum extraction and comparison, a generalized approach to pointer deconstruction for several RDBMSes).

We first verify the integrity of indexes (discussed in Section 8) because indexes are used later to detect tampering of table data, so it is critical to verify index structure integrity. To achieve that, we evaluate the B-Tree in storage, consider corrupt data that

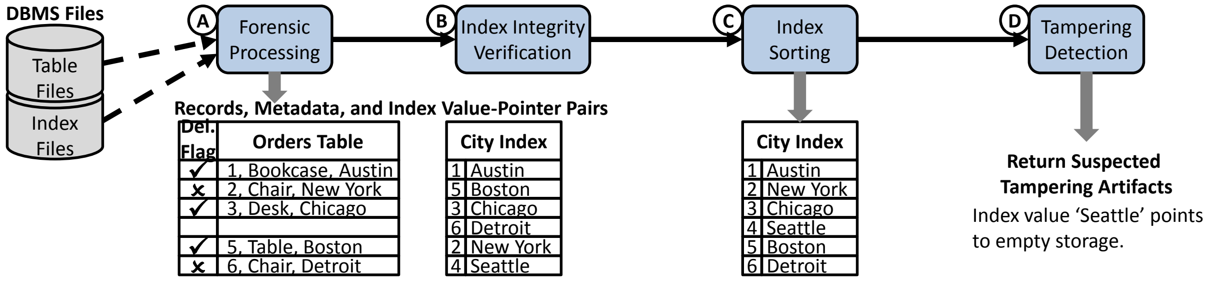


Figure 6: Architecture of the DBStorageAuditor.

matches B-Tree organization, and check for traces of an index rebuild (e.g., *REORG*, *VACUUM* – depending on a DBMS).

We cannot assume that index artifacts can be fully stored in RAM while matching index values to table records. Therefore, the carved index sorting component discussed in Section 9 preprocesses index artifacts to make DBStorageAuditor approach scalable. We approximately sort the index values based on their pointers which correspond to the physical location of records in a file and improves the runtime the matching process.

Finally the tampering detection component discussed in Section 10 detects cases of extraneous and erased data in storage. If a record and its artifacts can not be reconciled with index value-point pairs, such entries are flagged and returned to the user as suspected file tampering.

7 FORENSIC ANALYSIS

Our proposed analysis relies on an expanded version of DBCarver [31] to extract database storage artifacts that can not be queried using the DBMS. These artifacts include record metadata, deleted records, and index value-pointer pairs. In this section, we discuss the addition of a checksum comparison and generalized pointer deconstruction to DBCarver.

7.1 Checksum Comparison

In Section 3, we defined the checksum stored in the page header. Whenever data or metadata in a page is updated, either legitimately or through data tampering, the checksum must be updated accordingly. If the checksum is incorrect, the DBMS will recognize the page as corrupt. This will result in warnings as well as data loss ranging from page to the table or the entire database instance. Therefore, we can assert that if a checksum did not change between time T1 (previous inspection) and T2 (current inspection), then the page has not been modified and the records have not been exposed to tampering.

We implemented a dictionary of checksums taken from the DBMS pages that are to be evaluated by DBCarver (it is possible to inspect any subset of the DBMS for tampering signs – focusing only on data-sensitive tables). Our dictionary stores the checksum values, where the object identifier and page identifier (Section 3) were the key and the checksum was the value. The checksum dictionary should be stored off-site so it is not at risk of tampering.

If the checksum has changed for a given page, the entire page must be inspected and validated by DBCarver. If the checksum did not change for a page, only page metadata was necessary to reconstruct. The metadata is needed to avoid false-positives in Algorithm 2. Some DBMSes (e.g., Oracle, MySQL) allow the page checksum to be disabled. If the checksum is disabled or believed to have been disabled at some point, then a checksum comparison is unreliable and all data must be carved and inspected.

7.2 Index Carving and Pointer Deconstruction

DBStorageAuditor uses index value-pointer pairs to identify inconsistencies in DBMS storage. Therefore, the value-pointer pairs must be inspected. DBMSes do not allow indexes to be queried directly (i.e., indexes can not appear in the FROM clause) which is why we use DBCarver to retrieve index contents. However, the pointer parsing by DBCarver was limited and specific to each DBMS; we developed a generalized approach to pointer deconstruction allowing DBStorageAuditor to be compatible with any investigated RDBMS.

We performed an analysis of pointers for 7 commonly used RDBMSes. Table 1 lists these RDBMSes and summarizes our conclusions. We found that all of these DBMSes, except for MySQL, stored a PageID and a Slot#. By default, MySQL creates an indexed organized table (IOT) so the pointer deconstruction process is slightly different. We address index pointers for IOTs later in this section. The PageID refers to page identifier that is stored in table page header (Section 3). The Slot# refers to a records position within a page. SQLServer and Oracle both store a FileID, which refers to file in which the page is located. The DBMSes that do not include a FileID in the pointer, use a file-per-object storage architecture (i.e., each table and index are stored in different files). The FileID for these pointers is the ObjectID or it can be mapped back to the ObjectID if the object name is the file name. Thus, an index pointer can be deconstructed into a FileID, PageID, and Slot# to map a value back to a table record location. Index pointers are typically the same as the internal DBMS row identifier pseudo-column.

DBMS Version	FileID	PageID	Slot#
SQLServer	Yes	Yes	Yes
Oracle	Yes	Yes	Yes
ApacheDerby	No	Yes	Yes
PostgreSQL	No	Yes	Yes
Firebird	No	Yes	Yes
DB2	No	Yes	Yes
MySQL	No	Yes*	No

*The pointer references the second level of an IOT.

Table 1: Pointer Deconstruction.

Figure 7 demonstrates how index values are mapped back to the table records through our generalized pointer deconstruction. For each index value(A), the pointer stores a PageID(B) and Slot#(C). The pointer PageID(B) corresponds to the page identifier(D) in the table page header. The pointer Slot#(C) corresponds to the row directory address(E) in the table page. For example, the pointer for 'Austin' stores PageID = 8 and Slot# = 12. To find the record, the table page with identifier = 8 is found and the 12th row directory address is used to locate the record *(68, 'Alice', 'Austin')* within the page.

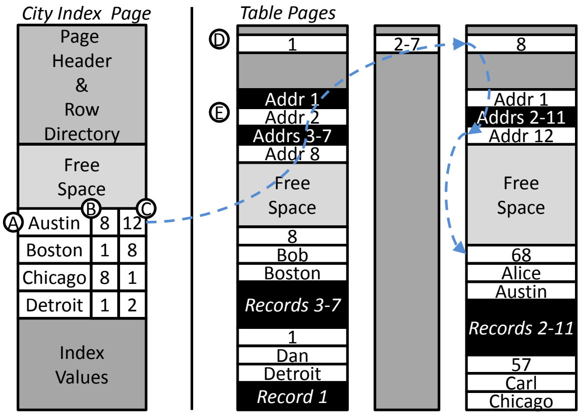


Figure 7: An example of mapping index values to a record.

Index Organized Tables. While MySQL was the only evaluated DBMS that created IOTs by default, IOTs are commonly used in other DBMSes under different names (e.g., IOT in Oracle, Included Columns in SQL Server) so we incorporated their pointer deconstruction. The pointer for a secondary index built on an IOT is made of a PageID that references a page one level above the IOT B-Tree leaf page, and the primary key value. The PageID for the IOT leaf page can then be retrieved from the pointer stored in the second level of the B-Tree. After performing this additional IOT B-Tree access, we can associate every secondary index value with a PageID and a primary key value, where the PageID references an IOT leaf page and the primary key value replaces the Slot#. Figure 8 illustrates how a secondary index value can be mapped back to an IOT record. We have the same index on `City` and the same records from Figure 7. However, the records are now stored in an IOT, and we now have a B-Tree page one level above the IOT leaf pages. The `City` index values(A) now store the PageID for IOT B-Tree page(B) and the primary key values(C) as the pointer. The IOT B-Tree page stores primary key values(F) and leaf PageIDs(G) as the pointer. For example, the pointer for 'Austin' stores PageID 20 and the primary key 68. This directs us to the IOT B-Tree page with PageID 20 and the value-pointer pair `(57, 8)`. The IOT B-Tree pointer tells us 'Austin' is in the leaf page with PageID 8 and the primary key value 68.

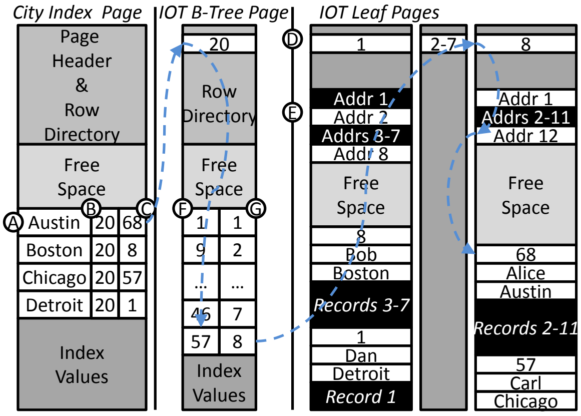


Figure 8: Mapping index values to an IOT record example.

8 VERIFYING INDEX INTEGRITY

It is plausible for an adversary to tamper with the relevant index values in an attempt to conceal evidence of file tampering. In this section, we address several types of index tampering, and how to detect such activity.

8.1 B-Tree Integrity Verification

If the attacker changes a value, adds a record, or wipes a record from a table, he may also perform a complimentary operation in the index. For example, 'Dan' was changed to 'Jane' in a table record could also be similarly modified in the index leaf node.

Interestingly, this type of activity creates inconsistencies in the index B-Tree that do not arise in the table. We consider the case where an index value is changed in-place and the case where index value was erased (and possibly reinserted into the correct position in the B-Tree). If the index value was changed in-place, it would appear out-of-order in the leaf of the B-Tree. If the index value was erased, it creates an uncharacteristic blank space between values within the leaf page, which *never* occurs naturally.

8.2 The Neighboring Value Problem

An index value may sometimes be altered without violating the correct ordering of the B-Tree. For example, in Figure 9 'Dan' is changed to 'Dog' preserving a correct value ordering of the `Name` index. This example shows how a table and an index can be altered without producing an inconsistency.

We build a function-based index that stores the hash value of column(s) to thwart tampering that involves neighboring range values. The values in hash-index will have a different ordering than the values in the secondary index so a neighboring value can occur in one, but not both. Figure 9 shows an example of how a hash index can be used to detect index tampering the involves neighboring values. In both the table and the `Name` index, the value was changed to 'Dog.' Changing the value in the `Name` index preserved the correct ordering. However, changing the value in the hash-index would result in an incorrect ordering since the values are organized differently. Function-based indexes are supported by many major DBMSes (e.g., IBM DB2, Oracle, and PostgreSQL); a computed column can be used for DBMSes that do not support function-based indexes (e.g., MySQL and Microsoft SQL Server).

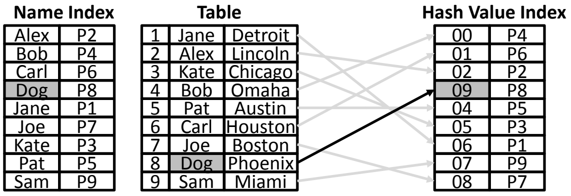


Figure 9: Preventing the neighboring value problem.

8.3 SQL Index Rebuild

Although we assume that the attacker does not have privileges to rebuild an index through SQL, the index may nevertheless be rebuilt as part of routine maintenance. If an index is rebuilt post tampering, the reconstruction of the index will eliminate any inconsistencies (extraneous or erased data) between the table and the index because indexes will be built anew using the *current* table state. However, when an object is rebuilt, a new object is created and artifacts (discarded pages) from the old object are left behind in storage. Many of the pages from the old index are likely to be overwritten, but some pages are going to persist in storage following the rebuild [30].

Pages left behind from an index rebuild can serve as separate evidence to detect tampering. The old index version (or the parts recovered) can be treated as a separate index (I_{t-1}) from the newly rebuilt version (I_t). While the old index version does not contain a complete set of values due to having been partially overwritten, it can still be used to detect tampering. This would be applicable when auditing is not performed at regular intervals relative to the frequency of index rebuilds.

8.4 Manual Index Rebuild

In order to deceive DBStorageAuditor, an attacker would have to completely rewrite the entire index (or at least several different pages in it). While such operation is possible, performing it successfully poses several major challenges. We emphasize that typical security solutions are designed to greatly increase the level of difficulty to perform an attack, rather than create an absolute defense.

Section 5 discussed cached dirty page problem when physically modifying a page. Moreover, dirty index pages can introduce additional complications. First, a given index page is more likely to have a dirty version cached compared to a table page. An index page is not only modified when the indexed column is updated, but the index pointer must also be updated if an update causes a record to be written to a new location. Furthermore, index pages store significantly more values than table pages, increasing their chance to be modified. Second, as the index changes, the database may reorganize the B-Tree structure (e.g., page split). As parts of the index are rebuilt, pages are likely to be written to new locations in a file. We note that the physical order of a B-Tree does not reflect the logical order of the B-Tree. Third, the attacker may have to discover the physical location of other connected index pages (i.e., just finding the page with needed value is insufficient, several parts of the B-Tree would need to reconstructed). Index leaf pages point to the next logical page in the B-Tree and sometimes to the previous page as well. This means that if a logically adjacent page is rebuilt and written to a new location, then a modified index page would need to reflect that change. Therefore, the attacker would need to be aware of all internal B-Tree structure changes to guarantee a successful manual index rebuild. Finally, if a function-based index storing a hash value exists, we assume that an SA would not have knowledge of this function. Therefore, inconsistencies would still arise in any attempts to manually rewrite the index.

9 INDEX SORTING

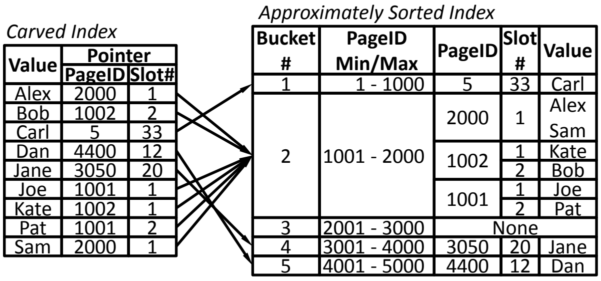
When tables and indexes are carved, the data is extracted based on the physical location within the files. Therefore, the relationship between the ordering of the carved table records compared to the index values is random, with a possible exception of a clustered index (it is common for a clustered index to be manually updated, such as PostgreSQL with VACUUM command). Assuming that the index can not be fully loaded into into RAM, expensive random seeks must be performed to map index values to table records. In this section we propose a method to reorder the index to make the process of matching index values and table records scalable.

As demonstrated in Section 7, index pointers correspond to the physical position of the table records. Therefore, sorting the index values by the pointers produces the same ordering for index values and table records. Carved table records and index values are then read sequentially, similar to a merge join process.

For an index that is too large to fit into memory, sorting the index pointers can be a costly operation. If we assume that N table pages will be read into memory when detecting table tampering (Section 10), then index values need to be sorted across every N pages, but values do not need to be sorted within N pages. We call each set of index values that belong in N table pages a **bucket**. We perform approximate sorting by re-ordering index values across buckets but not within buckets.

For each index bucket, we record the minimum and maximum table page identifier. If an index value is in the range of page identifiers for a bucket, the page identifier, slot number, and index value are stored in that bucket. When table pages are read for table tampering detection, the relevant bucket(s) are read into memory using the table page identifier and the index bucket minimum and maximum values.

Figure 10 shows an example of an index that is approximately sorted on the pointer. For each value in the index, there is a pointer that contains a PageID and a Slot#. We first create a set of buckets where each bucket contains 1000 PageIDs. We read the carved index data, and assign a value to the appropriate bucket using the pointer. For example, the first and second index values ‘Alex’ and ‘Bob’ belong in bucket #2 because their PageIDs, 2000 and 1002 are between the minimum and maximum PageID range for the bucket. We then store the PageID, Slot#, and Value in the bucket. ‘Carl’ has a PageID 5 so that value belongs in bucket #1. Bucket #2 demonstrates that PageIDs do not need to be sorted within the bucket. Furthermore, we see that PageID 2000 in bucket #2 has two values. This can occur as a result of legitimate SQL operations that create stale index values.



Carved Index

Value	Pointer	
	PageID	Slot#
Alex	2000	1
Bob	1002	2
Carl	5	33
Dan	4400	12
Jane	3050	20
Joe	1001	1
Kate	1002	1
Pat	1001	2
Sam	2000	1

Approximately Sorted Index

Bucket #	PageID Min/Max	PageID	Slot #	Value
1	1 - 1000	5	33	Carl
2	1001 - 2000	2000	1	Alex Sam
		1002	1	Kate
			2	Bob
		1001	1	Joe
			2	Pat
3	2001 - 3000	None		
4	3001 - 4000	3050	20	Jane
5	4001 - 5000	4400	12	Dan

Figure 10: An approximately sorted index example.

Our current implementation does not use the FileID even when it is stored in the pointer. We assume that each table is stored in a single file, and that the user has directed DBStorageAuditor to the relevant table and index files. DBMS-specific system tables would allow us to connect FileID to the information on target table and index files.

Index Organized Tables. Approximately sorting secondary indexes for index organized tables (IOT) is a slightly different process. When an IOT is used, the secondary index pointer is made up of a PageID that references a second level B-Tree page and the primary key value instead of a PageID that references the table and a Slot#. To sort the secondary index values, the second level BTree pages from the primary key is used to retrieve the table PageIDs for each value. Furthermore, the primary key value is now used in place of the Slot#.

The cost of approximate sorting is dependent on the amount of available memory. A bucket must fit into memory. Fewer buckets results in quicker bucket assignment for values, but buckets will be larger requiring more memory. In Section 11.2 we provide costs of approximately sorting an index.

10 DETECTING TABLE TAMPERING

In Section 5 we discussed how database files, specifically tables, are vulnerable to tampering. We propose using the validated indexes (Section 8) to verify the integrity of table records in storage. Earlier in this paper, we classified data tampering that involves changing a value or adding records as extraneous data, and we classified data tampering that involves wiping records as erased data. In this section we present and discuss algorithms to detect both extraneous and erased data.

10.1 Extraneous Data Detection

Extraneous data is a record or a value that has been added to a table through file tampering. Since extraneous data is not added using the DBMS, it is not reflected in the indexes. Therefore, if a record does not have *any* corresponding index pointer, then the entire record is suspected of having been added through file tampering. Any table with a primary key can be tested because an index is automatically created for a primary key constraint. Similarly, if a table value does not match an index value with the corresponding pointer, then the value is assumed to have been modified through file tampering. This validation test does require that an index exist on the column(s). We use the carved data from Section 7 and an approximately sorted index (Section 9) that was not been subject to tampering (Section 8).

Algorithm 1 describes how to detect extraneous data. First, we read N table pages at a time for evaluation; we then scan the approximately sorted index buckets for the relevant table page identifiers and read the index pages from the relevant bucket(s). For every record in the N table pages, we find the corresponding index pointer. If an index pointer does not exist, this record is added to a list of likely extraneous data. If an index pointer does exist for a record, the indexed column is compared to the index value(s) for that pointer (there may be more than one index value per pointer for legitimate reasons). If the table value is not in the set of index values, then this value is added to a list of likely extraneous data. This is evidence of a value that has been changed. After all table pages have been read and all records evaluated, the resulting extraneous data list is returned to the user.

Algorithm 1 Extraneous Data Detection

```
$Table \leftarrow$ carved table data: PageIDs, Slot #s, and Records.
$N \leftarrow$ the number of table pages to be read.
$SortedIndex \leftarrow$ the approximately sorted index (Section 9).
$Flag \leftarrow$ an empty list to store extraneous data.
for each $NPages \in Table$ do
    $MinPID \leftarrow$ the minimum page ID from $NPages$.
    $MaxPID \leftarrow$ the maximum page ID from $NPages$.
    $Indexes \leftarrow$ an empty list to store index pages.
    for each $Bucket \in SortedIndex$ do
        if $(MinPID \in Bucket) \vee (MaxPID \in Bucket) \vee (MinPID < Bucket \wedge MaxPID > Bucket)$ then
            $Indexes.append(Bucket)$
    for each $Rec \in NPages$ do
        $RecPtr \leftarrow Rec.PageID.Slot\#$
        if $RecPtr \in Indexes.PageID.Slot\#$ then
            if $Rec.Val \notin Indexes.PageID.Slot\#.Vals$ then
                $Flag.append(['ModVal', RecPtr, Rec, Val])$
        else
            $Flag.append(['HiddenRecord', RecPtr, Rec])$
return $Flag$
```

10.2 Erased Data Detection

Erased data is data explicitly wiped from table storage through file tampering. Deleted records are likely to be overwritten by new records over time as the DBMS runs. However, a deleted record will never be overwritten by something that is not another record of the same structure. Therefore, if an index value points to an area in storage that does not contain a proper record (including metadata), then record wiping is suspected. We are not concerned with matching the specific index value since this is done in Algorithm 1, but rather that a pointer must reference an area in storage that resembles a record.

Algorithm 2 describes how to detect erased data. First, we read each bucket from the approximately sorted index. When a bucket is read, the table pages with the relevant page identifiers are also read. We iterate through each index value in the bucket. If the pointer for an index value does not match any record in the table pages, then the index value is appended to a list of erased data. After all index buckets have been evaluated, the list of erased data is returned to the user.

Algorithm 2 Erased Data Detection

```
$Table \leftarrow$ carved table data: PageIDs, Slot #s, and Records.
$SortedIndex \leftarrow$ the approximately sorted index (Section 9).
$Flag \leftarrow$ an empty list to store erased data.
for each $Bucket \in SortedIndex$ do
    $NPages \leftarrow$ pages from $Table$ where $PageID \in Bucket$
    for each $IndexValue \in Bucket$ do
        $Ptr \leftarrow IndexValue.PageID.Slot\#$
        if $Ptr \notin NPages$ then
            $Value \leftarrow$ the index value
            $Flag.append(['ErasedRecord', Ptr, Value])$
return $Flag$
```

Adjacent Deleted Records. It is possible that multiple deleted records can exist adjacent to one another in a page. When this happens it is also possible the a single record could overwrite all of one record and part of another. For example, (1, 'Ed') and (2, 'Tom') are deleted records that are next to each other in storage. The inserted record (3, 'Karen') could overwrite all of (1, 'Ed') and part of (2, 'Tom'). This presents a problem because any old index value for (2, 'Tom') would now point to the middle of the inserted record, rather than to a full record. In this scenario, Algorithm 2 would return a false-positive for the index value from (2, 'Tom'). These false-positives can be eliminated by comparing these results to audit log entries. For example, if a delete command in the log could explain (2, 'Tom'), then this could be declared as not malicious. This functionality is not currently supported by `DBStorageAuditor`, and it would be explored in future work to achieve a more complete auditing system.

11 EXPERIMENTS

In this section, we present a set of experiments that evaluate the performance, accuracy, and limitations of `DBStorageAuditor`. Table 2 summarizes the experiments in this section.

MySQL 5.7, PostgreSQL 9.6, and Oracle 11g R2 DBMSes were used in these experiments. We believe these three RDBMSes are a good representative selection from the commonly used RDBMSes. Not only are they widely used commercial and open-source DBMSes, but they also represent the spectrum of different storage decisions across about ten DBMSes we have studied. For example, PostgreSQL does not support IOTs, Oracle offers an option to create IOTs, and MySQL automatically uses IOTs. The default page sizes for each DBMS were used: 8K for Oracle and PostgreSQL and 16K for MySQL. Data from the Star Schema Benchmark (SSBM) [16] was used to populate our DBMS instances. Table 3

#1	Forensic analysis (Sec 7) cost evaluation. DB files were carved at a rate of 1.2 MB/s. A checksum comparison can improve carving costs.
#2	Approximate sorting (Sec 9) cost evaluation. Fewer buckets improves runtime, but requires more memory.
#3	Algms 1 and 2 (Sec 10) cost evaluation. Both algorithms increase linearly with table size.
#4	`DBStorageAuditor` detection evaluation. Algm 1 detects an added record, Algm 1 detects a modified value only for an indexed column, and Algm 2 reconstructs erased data that was indexed.
#5	`DBStorageAuditor` detection limitations after an index rebuild (Sec 8). `DBStorageAuditor` can use the old version of an index depending on the DBMS.

Table 2: Summary of experiments.

can be used to reference table sizes used throughout this section. DBMS instances ran 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.

Table	Scale	DB File Size(MB)	Values(M)
Lineorder	1	600	6
Lineorder	4	2400	24
Lineorder	14	8300	84
Supplier	1	<1	2K

Table 3: SSBM table sizes used through the experiments.

The different DBMS storage-altering operations that we are seeking to detect are discussed in Section 10. When modifying files, we re-calculated and updated the page checksum value for the PostgreSQL pages; in MySQL and Oracle we disabled the page checksum validation. Before modifying files, we first shutdown the DBMS instance.

11.1 Forensic Processing

The objective of this experiment is to evaluate the computational cost associated with the forensic processing component of `DBStorageAuditor` discussed in Section 7. In Part-A, we provide `DBCarver` runtimes against database files of various sizes from MySQL, Oracle, and PostgreSQL DBMSes. In Part-B, we repeat the same evaluation, further including a checksum re-computation.

Part-A. We created a series of database files for each DBMS to pass to `DBCarver`. We created three `LINEORDER` tables: Scale 1, 4, and 14. Each table was stored in a separate file. The PostgreSQL files were carved at an average rate of 1.0 MB/s, the MySQL files were carved at a rate of 1.2 MB/s, and the Oracle files were carved at a rate of 1.5 MB/s.

Part-B. We used the PostgreSQL `LINEORDER` Scale 4 table from Part-A to evaluate the checksum comparison we added to `DBCarver`. We modified pages that induced a checksum change for 1%, 5%, 10%, and 100% of the pages in the database file. The carving rate for each percent modification was 100% → 1MB/s, 10% → 9 MB/s, 5% → 18 MB/s, and 1% → 58 MB/s. The cost of forensic pre-processing is thus proportional to the number of modified pages rather than the total size of the DBMS storage.

11.2 Index Sorting

The objective of this experiment is to evaluate the costs associated with approximately sorting the index values on the pointers. The output produced by the forensic analysis is similar for all DBMSes so this component of `DBStorageAuditor` is not tested for DBMS-specific features. In Part-A, we vary the size of bucket; in Part-B, we vary the size of the indexes.

Part-A. To evaluate approximate sorting with respect to bucket size, we used the carved output from PostgreSQL database files containing a `LINEORDER` Scale 4 table, a secondary index on `LO_Revenue`, and a secondary index on `LO_Orderdate`. Table 4 summarizes the performance results. As the number buckets decreases the time to sort the data decreases. However, a bucket must fit into memory, so increasing of bucket sizes is limited by available RAM.

Bucket Size (Pages)	Bucket Count	Orderdate (sec)	Revenue (sec)
5,000	63	1366	1380
10,000	32	1121	1131
50,000	7	932	945
100,000	4	909	926
200,000	2	903	918

Table 4: Index sorting costs with varying bucket sizes.

Part-B. To evaluate approximate sorting with respect to the size of an index, we used the carved output from PostgreSQL database files containing `LINEORDER` Scale 1, 4, and 14 tables and a secondary index on `LO_Revenue` for each table. Table 5 summarizes the results. If the bucket size is increased proportionally for the table size, the approximate sorting cost increases linearly.

Bucket Size	Index sorting time (sec)		
(Pages)	Scale 1	Scale 4	Scale 14
10,000	239	1131	6193
50,000	231	945	3797
100,000	n/a*	926	3486
200,000	n/a*	918	3357

*Bucket size is larger than the table.

Table 5: Approximate sorting costs for varying table sizes.

11.3 Tampering Detection Costs

The objective of this experiment is to evaluate the costs associated with of Algorithms 1 and 2. For this experiment we used the `LINEORDER` Scale 4 table. We used one index on the `LO_Revenue` and multiple indexes on the `LO_Revenue` and `LO_Orderdate`. We approximately sorted the index using buckets with 50K pages.

Part-A: Algorithm 1. To evaluate the costs associated with Algorithm 1, we used the output from two different secondary indexes (`LO_Revenue` and `LO_Orderdate`) on `LINEORDER` Scale 4 and one secondary index (`LO_Revenue`) on `LINEORDER` Scale 14. Table 6 summarizes the runtime results. The runtime for Algorithm 1 was the same for `LO_Revenue` and `LO_Orderdate` on `LINEORDER` Scale 4, and the cost increased linearly for `LO_Revenue` on `LINEORDER` Scale 14.

Table	Index	Part-A (sec)	Part-B (sec)
Scale 4	LO_Revenue	966	503
Scale 4	LO_Orderdate	961	476
Scale 14	LO_Revenue	3482	1773

Table 6: Algorithm 1 and 2 runtimes.

Part-B: Algorithm 2. We used the same tables in indexes from Part-A of this experiment to evaluate the costs associated with Algorithm 2. Table 6 summarizes the runtime results. Similar to Algorithm 1, the cost for Algorithm 2 was nearly the same for `LO_Revenue` and `LO_Orderdate` on `LINEORDER` Scale 4, and the cost increased linearly for `LO_Revenue` on `LINEORDER` Scale 14.

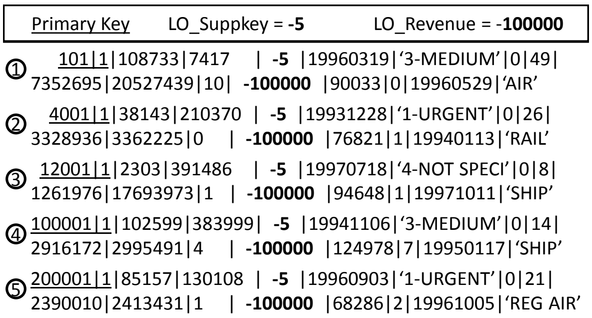
11.4 Detection Capabilities

The objective of this experiment is to demonstrate the file tampering activity that DBStorageAuditor is capable of detecting. For each part in this experiment, we simulate one defined type of malicious activity and explain how it was detected. We manually add records to the database file (Part-A), change values in the database file (Part-B), and erase records from the database file (Part-C). We present results only for PostgreSQL because we our results for Oracle and MySQL were very similar.

Setup. We created a LINEORDER Scale 4 table for a PostgreSQL DBMS. An index existed on the primary key (LO_Orderkey, LO_Linenumber) and we created a secondary index for LO_Revenue and LO_Orderdate.

We also created a function-based index on LO_Revenue that used the 32-bit version of the MurmurHash2 hash function.

Part-A. We manually added 5 records (shown in Figure 11) to the file containing the LINEORDER table. We added a record to five different pages (with PageIDs 11, 12, 13, 14, and 15). Existing primary key values were included in each of the five records. For each of these records, all of the data was the same as the existing records with the same primary key except we used LO_Suppkey -5 and LO_Revenue -100000.



```
 Primary Key     LO_Suppkey = -5          LO_Revenue = -100000

①    101|1|108733|7417    |  -5  |19960319|'3-MEDIUM'|0|49|
 7352695|20527439|10|  -100000  |90033|0|19960529|'AIR'
②   4001|1|38143|210370  |  -5  |19931228|'1-URGENT'|0|26|
 3328936|3362225|0    |  -100000  |76821|1|19940113|'RAIL'
③  12001|1|2303|391486   |  -5  |19970718|'4-NOT SPECI'|0|8|
 1261976|17693973|1   |  -100000  |94648|1|19971011|'SHIP'
④ 100001|1|102599|383999|  -5  |19941106|'3-MEDIUM'|0|14|
 2916172|2995491|4    |  -100000  |124978|7|19950117|'SHIP'
⑤ 200001|1|85157|130108  |  -5  |19960903|'1-URGENT'|0|21|
 2390010|2413431|1    |  -100000  |68286|2|19961005|'REG AIR'
```

Figure 11: Records added to the LINEORDER file.

The addition of these five records produced several interesting outcomes. First, these records bypassed the primary key constraint since they contained primary key values that previously existed in the table. The DBMS only checks constraints when executing API-based load commands, and it does not retroactively check the table for constraint violations. Adding the record to the file bypasses all official channels and is thus never checked for constraint violations. Second, these records also bypassed referential integrity since the LINEORDER table references the SUPPLIER table, and LO_Suppkey -5 did not exist in the SUPPLIER. Similar to the primary key violation, the constraint violation was never caught by the DBMS. Finally, table access for the same query could produce different results because the indexes were not updated after we added these five records. For example, the two versions of the following query returns different results:

- Query 1 → 34600180980
  ```
  SELECT SUM(LO_Revenue) FROM Lineorder
  WHERE LO_Orderdate = 19960319;
  ```
- Query 2 → 34600180980 - 100000
  ```
  set enable_seqscan=true;
  SELECT SUM(LO_Revenue) FROM Lineorder
  WHERE LO_Orderdate = 19960319;
  ```

Query 1 uses the LO_Orderdate index to access the table while Query 2 uses a full table scan. Record #1 from Figure 11 was included in Query 2, but it was not included in Query 1.

Algorithm 1 successfully detected the fact that five new records do not have corresponding pointers in the primary key index, the two secondary indexes, and in the function-based index. Problem was flagged by a False value for the line 14 If condition resulting in the malicious records being added to the list of invalid data at line 18. Each existing index serves as an additional validation to detect table tampering – and the function-based makes sure that small incremental changes are not possible.

Part-B. Next, we changed LO_Revenue for all 41 records where the LO_Custkey 4321 and LO_Orderdate between 19930101 and 19931231. To simulate a neighboring value problem (a small change that does not violate index ordering), we changed the record with LO_Custkey 4321 and LO_Revenue 3271986 to 3271987 in both the table and the LO_Revenue index. For all other records we subtracted 100000 from LO_Revenue in the table.

Algorithm 1 reported that 40 records had an inconsistent value based on the LO_Revenue index and 41 records had an inconsistent value based on the function-based index on LO_Revenue. The difference of the one additional record was due to the neighboring value attack which regular index may fail to detect. These values were detected by a False value for the line 15 If condition resulting in the malicious values being added to the list of invalid data at line 16. We can conclude that the primary key and LO_Orderdate columns were not tampered with for these and all records since they were not included in the invalid data. However, we can not make any conclusion if any other of the non-indexed columns for these or any records were tampered.

Part-C. Next, we erased all 3085 records with the LO_Suppkey 123 from the file. For data erasure, we explicitly overwrote the records and their metadata with the NULL ASCII character.

Algorithm 2 returned that primary key index, the two secondary indexes, and the function-based index each had 3085 values that did not point to a valid record structure. These were detected by a True value for the line 8 condition in Algorithm 2, resulting in malicious data being added to the list of invalid data at line 10. By combining the values for each pointer we reconstructed partial records containing the index columns to explain the missing data. However, the data for the non-indexed columns was unable to be reconstructed since it was not indexed.

11.5 Long-Term Detection

The objective of this experiment is to evaluate the artifacts produced by an index rebuild that can used by DBStorageAuditor. We evaluate a different DBMS for each part of this experiment: Oracle in Part-A, MySQL in Part-B, and PostgreSQL in Part-C.

We performed the following steps for each DBMS. After each step, we copied the database file for analysis. Table 7 summarizes the results.

- T_0: Started with the Supplier Scale1 (2K records) table and a secondary index on S_Name.
- T_1: Erased/wiped all 829 records where S_Region equaled 'ASIA' or 'EUROPE'.
- T_2: Rebuilt the index. Each DBMS used a different index rebuild command:
  - **Oracle:** `ALTER INDEX Supp_Name REBUILD ONLINE`
  - **MySQL:** `DROP` and `CREATE` commands
  - **PSQL:** `REINDEX TABLE Supplier`

Part-A: Oracle. The index contained 1 root page and 9 leaf pages after creation at T_0. No changes were made to the index after the table records were erased at T_1. After the index rebuild at T_2, the new index contained 1 root page and 5 leaf pages. All of

DBMS	T_0(pgs)	T_1	T_2
Oracle	1 root, 9 leaf	no change	All index pages from the old index remained in DB storage.
MySQL	1 root, 5 leaf	no change	2 leaf pages from the old index remained in DB storage.
PSQL	1 root , 10 leaf	no change	None of the old index remained in DB storage.

Table 7: Index rebuild summary.

the pages from the original version at T_0 remained in the database file. The DBMS assigned a new ObjectID to the new version of the index so index pages between versions were easily distinguished. Since the entire version of index was found, it could be used by DBStorageAuditor. The old version of the index still contained pointers to the erased records, whereas the new version only contained pointers to active records in the table.

Part-B: MySQL. The index contained 1 root page and 5 leaf pages after creation at T_0. No changes were made to the index after the table records were erased at T_1. After the index rebuild at T_2, the new index contained 1 root page and 3 leaf pages. 2 out of the 5 leaf pages from the original index remained in database storage. This demonstrates that the DBMS immediately reclaimed the pages from the dropped index. Since the new index version used less storage space, 2 pages from the old version remained in the file. In this scenario, a B-Tree could not be fully validated with only 2 leaf pages making them less useful as evidence for DBStorageAuditor. It is likely that copies of the index could be carved from a disk image due to activity such as writes that do not occur in place and paging files. DBStorageAuditor does not currently reconstruct entire B-Tree indexes from disk images. Future work will seek to reconstruct objects from disk images, which requires multiple versions of pages to be considered.

Part-C: PostgreSQL. The index contained 1 root page and 10 leaf pages after creation at T_0. No changes were made to the index after the table records were erased at T_1. After the index rebuild at T_2, the new index contained 1 root page and 6 leaf pages. The new version of the index was assigned a new ObjectID and a separate file. All pages belonging to the old version of the index were disassociated with its file, and this storage was reclaimed by the file system. In this scenario, DBStorageAuditor can no longer detect that the records were erased. As discussed in Part-B, it is likely that the index could be carved from a disk image. This will be explored in future work since a logical timeline would need to be recreated to account for multiple page versions.

12 CONCLUSION

Database file tampering can be used to perform malicious operations while bypassing database security mechanisms (logging and access control) and constraints. We presented and evaluated DBStorageAuditor component that detects database file tampering. Our approach relies on a forensic inspection of database storage and identifies inconsistencies between tables and indexes.

Future work plans to expand upon this paper and work from [28] to create a complete database auditing framework. This future work would include creating a timeline of events and user attribution of storage artifacts. Our auditing framework relies on inherent characteristics of database storage that users, including privileged users, are incapable of controlling.

ACKNOWLEDGMENTS

This work was partially funded by the US National Science Foundation Grant CNF-1656268.

REFERENCES

[1] Brian Carrier. 2011. The sleuth kit. http://www.sleuthkit.org (2011).

[2] Scott A. Crosby and Dan S. Wallach. 2009. Efficient Data Structures for Tamper-evident Logging. In Proceedings of the 18th Conference on USENIX Security Symposium (SSYM'09). USENIX Association, Berkeley, CA, USA, 317–334. http://dl.acm.org/citation.cfm?id=1855768.1855788

[3] Daniel Fabbri, Ravi Ramamurthy, and Raghav Kaushik. 2013. SELECT triggers for data auditing. In Data Engineering (ICDE), 2013 IEEE 29th International Conference on. IEEE, 1141–1152.

[4] Lee Garber. 2001. Encase: A case study in computer-forensic technology. IEEE Computer Magazine January (2001).

[5] Simson Garfinkel. 2007. Anti-forensics: Techniques, detection and countermeasures. In 2nd International Conference on i-Warfare and Security, Vol. 20087. Citeseer, 77–84.

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

[7] Michael T Goodrich, Mikhail J Atallah, and Roberto Tamassia. 2005. Indexing information for data forensics. In ACNS, Vol. 5. Springer, Citeseer, 206–221.

[8] Ryan Harris. 2006. Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem. digital investigation 3 (2006), 44–49.

[9] IBM. 2017. 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).

[10] Gary C Kessler. 2007. Anti-forensics and the digital investigator. In Australian Digital Forensics Conference. Citeseer, 1.

[11] Gene H Kim and Eugene H Spafford. 1994. The design and implementation of tripwire: A file system integrity checker. In Proceedings of the 2nd ACM Conference on Computer and Communications Security. ACM, 18–29.

[12] Lianzhong Liu and Qiang Huang. 2009. A framework for database auditing. In Computer Sciences and Convergence Information Technology, 2009. ICCIT'09. Fourth International Conference on. IEEE, 982–986.

[13] ManageEngine. [n. d.]. EventLog Analyzer. https://www.manageengine.com/products/eventlog/. ([n. d.]).

[14] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G Stubblebine. 2004. A general model for authenticated data structures. Algorithmica 39, 1 (2004), 21–41.

[15] OfficeRecovery. [n. d.]. Recovery for MySQL. http://www.officerecovery.com/mysql/. ([n. d.]).

[16] Patrick O.Neil, Elizabeth O.Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In Performance evaluation and benchmarking. Springer, 237–252.

[17] Kyriacos E Pavlou and Richard T Snodgrass. 2008. Forensic analysis of database tampering. ACM Transactions on Database Systems (TODS) 33, 4 (2008), 30.

[18] Jon M Peha. 1999. Electronic commerce with verifiable audit trails. In Proceedings of ISOC. Citeseer.

[19] Percona. [n. d.]. Percona Data Recovery Tool for InnoDB. https://launchpad.net/percona-data-recovery-tool-for-innodb. ([n. d.]).

[20] Stellar Phoenix. [n. d.]. DB2 Recovery Software. http://www.stellarinfo.com/database-recovery/db2-recovery.php. ([n. d.]).

[21] Golden G Richard III and Vassil Roussev. 2005. Scalpel: A Frugal, High Performance File Carver.. In DFRWS. Citeseer.

[22] Christian S. J. Peron and Michael Legary. 2017. Digital Anti-Forensics: Emerging trends in data transformation techniques. (09 2017).

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

[24] Arunesh Sinha, Limin Jia, Paul England, and Jacob R Lorch. 2014. Continuous tamper-proof logging using TPM 2.0. In International Conference on Trust and Trustworthy Computing. Springer, Springer, 19–36.

[25] Richard T Snodgrass, Shilong Stanley Yao, and Christian Collberg. 2004. Tamper detection in audit logs. In Proceedings of the Thirtieth international conference on Very large data bases-Volume 30. VLDB Endowment, 504–515.

[26] Patrick Stahlberg, Gerome Miklau, and Brian Neil Levine. 2007. Threats to privacy in the forensic analysis of database systems. In Proceedings of the 2007 ACM SIGMOD international conference on Management of data. ACM, Citeseer, 91–102.

[27] Roberto Tamassia. 2003. Authenticated data structures. In ESA, Vol. 2832. Springer, Springer, 2–5.

[28] James Wagner, Alexander Rasin, Boris Glavic, Karen Heart, Jacob Furst, Lucas Bressan, and Jonathan Grier. 2017. Carving database storage to detect and trace security breaches. Digital Investigation 22 (2017), S127–S136.

[29] James Wagner, Alexander Rasin, and Jonathan Grier. 2015. Database forensic analysis through internal structure carving. Digital Investigation 14 (2015), S106–S115.

[30] James Wagner, Alexander Rasin, and Jonathan Grier. 2016. Database image content explorer: Carving data that does not officially exist. Digital Investigation 18 (2016), S97–S107.

[31] James Wagner, Alexander Rasin, Tanu Malik, Karen Heart, Hugo Jehle, and Jonathan Grier. 2017. Database forensic analysis with DBCarver. In CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research.