# Auditing DBMSes through Forensic Analysis

James Wagner
Supervised by Dr. Alexander Rasin
School of Computing, DePaul University
Chicago, IL USA
jwagne32@depaul.edu

## ABSTRACT

The pervasive use of databases for the storage of critical and sensitive information in many organizations has led to an increase in the rate at which databases are exploited in computer crimes. While there are several techniques and tools available for database forensics, they usually assume apriori database preparation, such as relying on tamper-detection software to already be in place or use of detailed logging. However, investigators need forensic tools and techniques that work on poorly-configured databases and make no assumptions about the extent of damage in a database.

In this paper, we present our database forensics methods, which are capable of examining database content from a storage image (disk or RAM) without using logs or any system metadata. We describe how these methods can be used to detect security breaches in a compromised environment where the security threat arose from a privileged user (or someone who has obtained such privileges).

## 1. INTRODUCTION

Cyber-crime (e.g., data exfiltration or computer fraud) is a significant concern in today's society. A well-known fact from security research and practice is that unbreakable security measures are virtually impossible to create. For example, 1) incomplete access control restrictions allows users to execute commands beyond their intended roles, and 2) users may illegally obtain privileges by exploiting security holes in a Database Management System (DBMS) or OS code or through other means (e.g., social engineering). Thus, in addition to deploying *preventive* measures (e.g., access control), it is necessary to 1) *detect security breaches* in a timely fashion, and 2) *collect evidence* about attacks to devise counter-measures and assess the extent of the damage (e.g., what data was leaked or perturbed). This evidence can provide preparation for legal action or valuable information to prevent future attacks.

DBMSes are targeted by criminals because they serve as repositories of data. Therefore, investigators must have the capacity to examine and forensically interpret contents of a DBMS. Currently, an audit log with SQL query history is a critical (and perhaps only) source of evidence for investigators [5] when a malicious operation is suspected. In field conditions, a DBMS may not provide the necessary logging granularity (unavailable or disabled). Moreover, the storage itself might be corrupt or contain multiple DBMSes.

Digital forensics provides tools for independent analysis with minimal assumptions about the environment. A particularly important and well-recognized technique is file carving [9], which extracts files (but not DBMS files) from a disk image, including deleted or corrupted files. Traditional file carving techniques interpret files (e.g., JPEG, PDF) individually and rely on file headers. DBMS files, on the other hand, do not maintain a file header and are never independent (e.g., table contents are stored separately from table name and logical structure information). Even if DBMS files could be carved, they cannot be meaningfully imported into a different DBMS and must be parsed to retrieve their content. To accomplish that task, DBMSes need their own set of digital forensics rules and tools.

Even in an environment with ideal log settings, a DBMS cannot necessarily guarantee log accuracy or immunity from tampering. For example, log tampering is a concern when a breach originated from a privileged user such as an administrator (DBA or an attacker who obtained DBA privileges). Tamper-proof logging mechanisms were proposed in related work [7, 10], but these only prevent logs from modifications and do not account for attacks that skirt logging (e.g., logging was disabled). Knowing that even privileged users have almost no control of how lowest level storage behaves, an analysis of forensic artifacts provides a unique approach to identify tampering in an untrusted environment.

The goal of this work is to 1) develop DBMS forensic methods, and 2) use these methods to detect and describe security breaches in compromised environments. Table 1 summarizes the remainder of this paper; future work is bolded.

| § | Summary |
|---|---|
| **2** | We describe our page-level DB forensics methods:<br>• Page carving is our DB forensic method. [12, 13].<br>• `DBCarver` [15] is our page carving implementation.<br>**• A framework to generalize `DBCarver` output that supports application development.**<br>**• DB anti-forensics protects against data theft.** |
| **3** | Forensic-based attack detection:<br>• `DBDetective` [14] detects activity that occurred when logging was disabled by a DBA.<br>• `DBStorageAuditor` [16] detects DBMS direct file tampering (without SQL) by a SysAdmin.<br>**• We will address DBMS log backdating.**<br>**• We will quantify the accuracy of our systems. A reproducible analysis will support our evidence.** |

**Table 1: Summary of the remaining paper.**

*Proceedings of the VLDB 2018 Ph.D. Workshop,* August 27, 2018. Rio de Janeiro, Brazil.

## 2. DATABASE FORENSICS

Unlike traditional files (e.g., PDF), DBMS files do not contain headers that allow for file identification. At the same time, all row-store DBMSes use fixed-size pages to store user data, auxiliary data (e.g., indexes and materialized views), and the system catalog. DBMS data is accessed and cached in page units. Pages maintain a consistent structure, whereas individual record structure varies throughout DBMS storage, which is why we approach database forensics at the page level. In this section, we describe page carving including our implementation (`DBCarver`), future work support application development from `DBCarver` output, and anti-forensics techniques that can sanitize and hide data in DBMS storage.

### 2.1 Page Carving

Database page carving is a method we previously introduced for the reconstruction of relational DBMSes without relying on file system or the DBMS. Page carving is similar to traditional file carving [9] in that data, including deleted data, can be reconstructed from images or RAM snapshots without the use of a live system. Forensic tools, such as Sleuth Kit [1] and EnCASE Forensic [2], are commonly used by investigators to reconstruct file system data but are incapable of parsing DBMS files. None of the third party recovery tools (e.g., [6, 8]) are helpful for independent audit purposes because (at best) they only recover "active" data from current tables. A database forensic tool (just like a forensic file system tool) should also reconstruct unallocated pieces of data including deleted rows, auxiliary structures (indexes, MVs), or buffer cache space.

While each DBMS uses its own page layout, a great deal of overlap between page layouts allowed us to generalize storage for most row-store DBMSes. In [12] we presented a comparative page structure study for IBM DB2, Oracle, MS SQL Server, PostgreSQL, MySQL, SQLite, Firebird, and Apache Derby. In this work, we also described a parameter set to define page layout for the purpose of reconstruction.

*Deleted Data.* When data is deleted, the DBMS initially marks it as deleted, rather than explicitly overwriting it. This data becomes unallocated (free listed) storage – in [13] we described the expected lifetime of forensic evidence within database storage following deletion and defragmentation. We described three categories of deleted data: records, pages, and values. A record is the minimum deletion unit and can be attributed to a `DELETE`, an old version of an `UPDATE`, or an aborted transaction. A deleted record is identified by its delete marking during page reconstruction. Dropped or rebuilt objects create deleted pages, which are identified by carving system catalog tables. Values from deleted records are found in auxiliary objects – e.g., indexes; they are identified by mapping pointers back to records (only records but *not* index values are deleted). We presented generalized pointer deconstruction and pointer-record mapping in [16].

Figure 1 visualizes an example of deleted records for several DBMSes. In all three pages, Row2-(*Customer2, Jane*) is deleted while Row1-(*Customer1, Joe*) and Row2-(*Customer3, Jim*) are active. Page#1 shows a case when the row delimiter is marked, such as in MySQL or Oracle. Page#2 shows when the raw data delimiter is marked in PostgreSQL. Page#3 shows when the row identifier is marked in SQLite. Figure 1 omits DB2 and SQL Server example because they only alter the row directory on deletion.

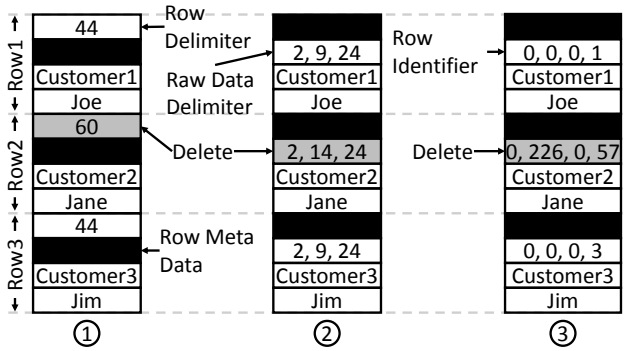


**Figure 1: Deleted row examples: 1-MySQL/Oracle, 2-PostgreSQL and 3-SQLite**

*Column-Store and NoSQL DBMSes.* Currently, our page carving only supports row-store DBMSes. Column-store and NoSQL DBMSes do not use the same pages structure as row-store DBMSes. Future work will expand our database forensic methods to column-store and NoSQL DBMSes.

### 2.2 DBCarver

We previously presented our implementation of page carving called `DBCarver` [15]. Figure 2 provides an overview of `DBCarver` architecture, which consists of two main components: the parameter collector (A) and the carver (F).

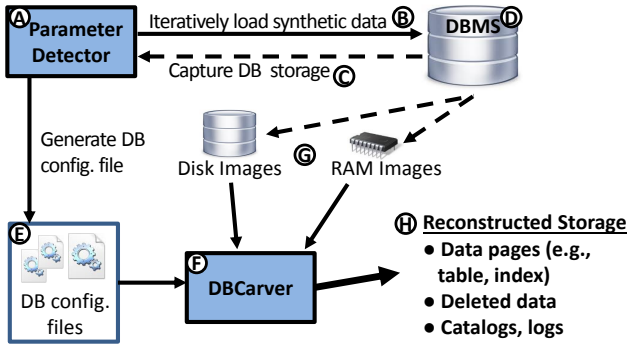


**Figure 2: DBCarver architecture.**

The parameter detector loads synthetic data into a DBMS (B), captures storage (C), finds pages in storage, and captures page layout parameters in a configuration file (E) – a text file describing page-level layout for that particular DBMS. Parameters include those described in [12], and have since been expanded to support other metadata. `DBCarver` automatically generates parameters values for new DBMSes, or new DBMS versions. While most DBMSes retain the same page layout across versions, we observed different parameter values between PostgreSQL versions 7.3 and 8.4.

The carver (F) uses the configuration files to reconstruct any database 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 Database Forensic Querying

Even though `DBCarver` provides a transparent view of DBMS storage, the output lacks composability needed for application development. Applications that use `DBCarver` output include the work in [16, 17, 15]. Currently, specialized output is generated for applications that use `DBCarver`

output. Furthermore, analyzing DBCarver output often requires an in-depth understanding of DBMS storage internals, which is unreasonable to expect from most users.

To introduce composability for application development with database forensic output, we propose a framework that has two goals: 1) defines a standard set of fields that describe forensically extracted user data, system catalogs, auxiliary objects, and other metadata, and 2) an user-friendly Python library that interprets this set of fields. This module will remove the need for specialized knowledge of database storage in development of applications based on DBCarver output.

We propose building a unified and standard output that automates the initial forensic analysis. To do this, we will store metadata from the DBCarver output in a series of JSON objects that maintain a consistent structure for all row-store relational DBMSes, while the original DBMS snapshot returned by DBCarver will be loaded back into our internal DBMS. The JSON objects will contain categories based on the work of Garfinkel et al. [4], and designed to include all available information from DBCarver output.

Working with the forensic output may require users to have an in-depth understanding of DBMS storage, which is unreasonable because each DBMS uses a highly customized storage engine. Such a requirement may prevent users in other domains from developing applications. We propose building a Python module to ease the interaction with the JSON objects and the reconstructed data stored in a DBMS. This module will contain methods to access individual properties from the JSON files. Furthermore, this module will allow for connections to be made between the metadata stored in the JSON files and the database snapshot stored in a relational DBMS.

## 2.4 Anti-Forensics

Anti-forensics (AF) is the field of interfering with forensic techniques [3]. We note that digital forensic tools can be used by either investigators or criminals, to both protect data and to interfere with a criminal investigation. In this section, we discuss future work that uses AF to protect data.

Two of the most representative AF techniques we consider are data wiping and steganography. A corporation can use data wiping to erase the already-deleted customer information to prevent data theft. Steganography is a data hiding technique – e.g., a means to discretely blow a whistle on company's wrongdoing. Most prior work in database AF has been highly DBMS-specific; e.g., Stahlberg erased deleted MySQL data by modifying the purge thread in source code [11]. We propose a more generalized sanitization method for all (including closed-source) DBMSes. We distinguish four categories of deleted DBMS data to wipe in order to prevent unintended data exposure: records, auxiliary data (e.g., indexes), system catalog, and unallocated pages. To effectively erase this data, the data itself must be overwritten and page metadata (e.g., checksums and pointers) must be updated accordingly. We further propose a steganography strategy that additively alters the database state through database file modification. This approach bypasses all constraints and logging mechanisms since the operation is performed without the DBMS. For example, domain constraints can be violated, NULL can be added to a primary key column, and foreign key constraints can be violated – making it unlikely that the hidden row is found through regular queries.

# 3. DATABASE SECURITY

Privileged users (e.g., DBA), by definition, have the ability to control and modify access permissions. Therefore, audit logs alone are fundamentally unsuitable for the detection of malicious, privileged users. DBMSes do not provide many tools to defend against insider threats. Interestingly, DBAs have little to no control over how data is stored at the lowest level. Thus, malicious activity will still create inconsistencies within storage artifacts. In this section, we consider attack vectors that are detectable using database forensics methods from Section 2. All of these solutions assume that some level of logging was enabled and is available.

## 3.1 DBDetective

Audit logs are a critical piece of evidence for investigators – and existing research has explored tamper-proof logs. However, DBAs can disable logging for legitimate operations (e.g., bulk loads). Therefore, we consider an attack where logging was disabled, malicious activity was performed, and logging was re-enabled. We proposed DBDetective in our previous work [14] to detect activity missing from the logs.

To detect unlogged activity, DBDetective compares the disk images and/or RAM snapshots output from DBCarver against the audit logs. We classify two categories of hidden activity: record modifications and read-only queries (i.e., SQL SELECT). When a record is inserted or modified the record itself changes, page metadata may be updated (e.g., a delete mark is set) and index page(s) are likely to change. We flag any artifacts that cannot be explained by a log entry as suspicious, as shown in Figure 3.

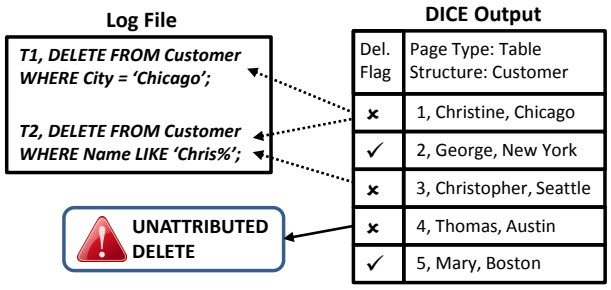


**Figure 3: Detecting unattributed deleted records.**

Figure 3 is an example of unaccounted, deleted row detection. DBCarver reconstructed 3 deleted rows from *Customer*: *(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*). After comparing the deleted records to the log file operations, DBDetective returned *(4,Thomas,Austin)*, indicating a deleted record that could not be attributed to any of the logged deletes. Here, we cannot conclude whether *T1* or *T2* caused the deletion of *(1,Christine,Chicago)*, but that is not necessary to identify record #4 as an unattributed delete.

When a SELECT query reads a table or a materialized view from disk, it ultimately uses one of two access patterns: a full table scan or an index access. Both of these query access types produce a consistent, repeatable caching pattern. Using metadata from the pages in the buffer cache, we identify caching patterns and match them to the logged commands.

## 3.2 DBStorageAuditor

Privileged OS users commonly have access to database files. Consider a SysAdmin who, acting as the root, maliciously edits a DBMS file in a Hex editor or through Python. The DBMS is unaware of external file write activity taking place outside its own programmatic access and thus cannot log it. Such an attack is a 'black-hat' application of anti-forensics discussed in Section 2.4. In our previous work [16], we proposed `DBStorageAuditor` to detect database file tampering.

To detect database file tampering, `DBStorageAuditor` [16] uses indexes to verify the integrity of table data. We first verify the integrity of the indexes by checking for tampering-based inconsistencies within the B-Tree structure. Once the index integrity is verified, we deconstruct the index pointers and match them to table records using the table page metadata; we generalized the deconstruction of index pointers for all major DBMSes. We organize the index pointers based on physical location to keep our matching approach scalable. Finally, any extraneous data or erased data found through index and table comparison is flagged as suspicious.

## 3.3 Event Timeline Analysis

Privileged users with access to the DBMS server have the capability to change server information, specifically the global clock. This quietly affects the veracity of DBMS audit logs. Consider a system administrator who changes the server global clock to an earlier date, performs malicious activity, and resets the global clock. Such an attack backdates activity without altering the log files, and disguises when the actual execution time of the malicious activity. As future work, we will detect such attempts to backdate log entries.

In such an environment, any global or logical clock can not be assumed to be reliable. Therefore, to create a timeline of events, we believe it is necessary to use storage metadata, which even a privileged user cannot modify. The internal RowID pseudo-column is of particular interest to construct a timeline. RowID is used by indexes and reflects the physical location of a record including its PageID. Whenever a page is modified, we can store the PageID to know when data was modified. Thus, the order of the PageIDs must be consistent with the order of the log events. We will propose tamper-proof techniques to store the PageID.

## 3.4 Quantitative Analysis and Reproducibility

As future work, we will determine the detection accuracy for each attack described in this section. For each detection type, we will compute a confidence rating based on a variety of environment variables (e.g., buffer cache size, volume of operations, and DBMS storage engine). For example, given a low volume of `DELETE` operations in Oracle, `DBDetective` would detect attacks with higher accuracy because Oracle controls storage with a percent page utilization. This engine setting prevents deleted records from being overwritten until a page contains a significant quantity of deleted data.

To verify the presence of malicious operations, a repeatable analysis analysis must be guaranteed. We will develop algorithms to collect the minimal subset of storage artifacts needed to reproduce our results. These collected storage artifacts must be sufficient to verify the security breach independent of our analysis. For example, such functionality is needed to present evidence in court.

# 4. CONCLUSION

In this work, we presented page carving and our page carving implementation, `DBCarver`. Future work will expand this method to include support for column-store and NoSQL DBMSes, offer meta-querying functionality, and incorporate anti-forensic methods to further protect data. We also presented methods that use page carving to detect security breaches in untrusted environments. `DBDetective` considered an attack where logging was disabled, `DBStorageAuditor` addressed DBMS file tampering, and future work will address tampering of the system global clock to backdate logs.

# 5. ACKNOWLEDGMENTS

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

# 6. REFERENCES

[1] B. Carrier. The sleuth kit. TSK. http://www.sleuthkit.org/sleuthkit, 2011.

[2] L. Garber. Encase: A case study in computer-forensic technology. IEEE Computer Magazine January, 2001.

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

[4] S. L. Garfinkel. Automating disk forensic processing with sleuthkit, xml, and python. SADFE, 2009.

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

[6] OfficeRecovery. Recovery for mysql. http://www.officerecovery.com/.

[7] J. M. Peha. Electronic commerce with verifiable audit trails. In Proceedings of ISOC. Citeseer, 1999.

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

[9] G. G. Richard III and V. Roussev. Scalpel: A frugal, high performance file carver. In DFRWS, 2005.

[10] R. T. Snodgrass et al. Tamper detection in audit logs. In Proceedings of the Thirtieth international conference on Very large data bases-Volume 30, pages 504–515. VLDB Endowment, 2004.

[11] P. Stahlberg, G. Miklau, and B. N. Levine. Threats to privacy in the forensic analysis of database systems. In Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pages 91–102. ACM, Citeseer, 2007.

[12] J. Wagner et al. Database forensic analysis through internal structure carving. In DFRWS, 2015.

[13] J. Wagner et al. Database image content explorer: Carving data that does not officially exist. In DFRWS, 2016.

[14] J. Wagner et al. Carving database storage to detect and trace security breaches. In DFRWS, 2017.

[15] J. Wagner et al. Database forensic analysis with dbcarver. In CIDR, 2017.

[16] J. Wagner et al. Detecting database file tampering through page carving. In EDBT, 2018.

[17] J. Wagner, A. Rasin, D. H. T. That, and T. Malik. Pli: Augmenting live databases with custom clustered indexes. In SSDBM, page 36. ACM, 2017.