Database Forensic Analysis with DBCarver

James Wagner, Alexander Rasin, Tanu Malik, Karen Heart, Hugo Jehle
School of Computing
DePaul University, Chicago, IL 60604
{jwagne32, arasin, tanu, kheart}@depaul.edu, jehlehugo@gmail.com

Jonathan Grier
Grier Forensics
Pikesville, MD 21208
jdgrier@grierforensics.com

ABSTRACT

The increasing use of databases in the storage of critical and sensitive information in many organizations has lead 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 mostly assume apriori database preparation, such as relying on tamper-detection software to be in place or use of detailed logging. Investigators, alternatively, 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 `DBCarver`, a tool for reconstructing database content from a database image without using any log or system metadata. The tool uses page carving to reconstruct both query-able data and non-queryable data (deleted data). We describe how the two kinds of data can be combined to enable a variety of forensic analysis questions hitherto unavailable to forensic investigators. We show the generality and efficiency of our tool across several databases through a set of robust experiments.

CCS Concepts

•Security and privacy → Information accountability and usage control; Database activity monitoring;

Keywords

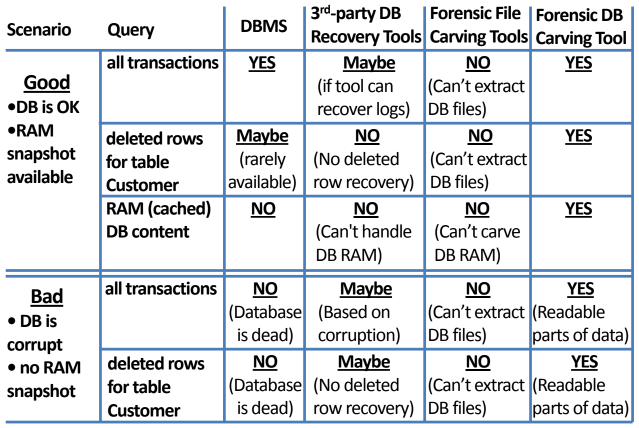
Database forensics; page carving; digital forensics; data recovery

1. INTRODUCTION

Cyber-crime (e.g., data exfiltration or computer fraud) is an increasingly significant concern in today's society. Federal regulations require companies to find evidence for the purposes of federal investigation (e.g., Sarbanes-Oxley Act [3]), and to disclose to customers what information was compromised after a security breach (e.g., Health Insurance Portability and Accountability Act [2]). Because most cyber-crime involves databases in some manner, investigators must have the capacity to examine and interpret the contents of database management systems (DBMSes). Many databases incorporate sophisticated security and logging components. However, investigators often do their work in field conditions – the database may not provide the necessary logging granularity (unavailable or disabled by default). Moreover, the storage image (disk and/or memory) itself might be corrupt or contain multiple (unknown) DBMSes.

Where built-in database logging is unable to address investigator needs, additional forensic tools are necessary. Digital forensics has addressed such field conditions especially in the context of file systems and memory content. A particularly important and well-recognized technique is *file carving*, which extracts, somewhat reliably, files from a disk image, even if the file was deleted or corrupted. There are, however, no corresponding carving tools or techniques available for database analysis.

In this paper, we focus on the need for *database carving* techniques (the database equivalent of file carving) for database forensic investigations. Databases use an internal storage model that handles data (e.g., tables), auxiliary data (e.g., indexes) and metadata (e.g., transaction logs). All relational databases store structures in pages of fixed size through a similar storage model (similar across relational databases and thus generalizable). File carvers are unable to recover or interpret contents of database files because file carvers are built for certain file types (e.g., JPEG) and do not understand the inherent complexity of database storage. Database carving can leverage storage principles that



Scenario	Query	DBMS	3rd-party DB Recovery Tools	Forensic File Carving Tools	Forensic DB Carving Tool
Good •DB is OK •RAM snapshot available	all transactions	YES	Maybe (if tool can recover logs)	NO (Can't extract DB files)	YES
	deleted rows for table Customer	Maybe (rarely available)	NO (No deleted row recovery)	NO (Can't extract DB files)	YES
	RAM (cached) DB content	NO	NO (Can't handle DB RAM)	NO (Can't carve DB RAM)	YES
Bad • DB is corrupt • no RAM snapshot	all transactions	NO (Database is dead)	Maybe (Based on corruption)	NO (Can't extract DB files)	YES (Readable parts of data)
	deleted rows for table Customer	NO (Database is dead)	Maybe (No deleted row recovery)	NO (Can't extract DB files)	YES (Readable parts of data)

Figure 1: State-of-the-art tools for database forensic analysis.

8th Biennial Conference on Innovative Data Systems Research (CIDR '17) January 8-11, 2017, Chaminade, California, USA.

are typically shared among DBMSes to generally define and *reconstruct* pages; hence, page carving can be accomplished without having to reverse-engineer DBMS software. Furthermore, while forensic memory analysis is distinct from file carving, buffer cache (RAM) is also an integral part of DBMS storage management. Unlike file carving tools, database carving must also support RAM carving for completeness. In practice, a DBMS does not provide users with ready access to all of its internal storage, such as deleted rows or in-memory content. In forensic investigations the database itself could be damaged and be unable to provide any useful information. Essentially, database carving targets the reconstruction of the data that was maintained by the database rather than attempting to recover the original database itself.

We further motivate `DBCarver` by an overview of what current tools can provide for forensic analysis in a database. Because investigators may have to deal with a corrupt database image, we consider two scenarios: "good" (database is ok) and "bad" (database is damaged). As basic examples of forensic questions that can be asked, we use three simple queries ("find all transactions", "find all deleted rows" and "find contents of memory"). Figure 1 summarizes what the DBMS itself, 3^{rd} party tools, file carving and database carving tools can answer under different circumstances.

1.1 Our Contributions

In this paper, we present a guide for using database carving for forensic analysis based on the digital investigation process described by the National Institute of Justice (NIJ) [1] and Carrier 2005[6]. We describe a database forensic procedure that conforms to the rules of digital forensics:

- We describe how "page-carving" in `DBCarver` can be used to reconstruct active and deleted database content. (Section 3)
- We describe SQL analysis on reconstructed active and deleted data from disk-image and memory snapshots to answer forensic questions regarding the evidence (Section 4).
- We evaluate the resource-consumption in `DBCarver`, the amount of meaningful data it can reconstruct from a corrupted database, and the quality of the reconstructed data (Section 5).

Section 2 summarizes related work in database forensics, and we conclude in Section 6, also describing future work.

2. RELATED WORK

A compromised database is one in which some of the metadata/data or DBMS software is modified by the attacker to give erroneous results while the database is still operational. Pavlou and Snodgrass [10] have proposed methods for detection of database tampering and data hiding by using cryptographically strong one-way hashing functions. Similarly Stahlberg et. al [14] have investigated a threat model and methods for ensuring privacy of data. However, very little work is done in the context of a damaged/destroyed database and collection of forensic evidence by *reconstruction* of data using database artifacts.

Adedayo 2012 [4] introduced an algorithm for record reconstruction using relational algebra logs and inverse relational algebra. Their heuristic algorithms assume not only the presence of audit logs but also requires other database logs to be configured with special settings that might be difficult to enforce in all situations. While this work is useful and complementary, in this paper we propose methods for database reconstruction for forensic analysis without any assumptions about available logging, especially audit logs.

In fact, our method is more similar to file carving [7, 13], which reconstructs files in the absence of file metadata and accompanying operating system and file system software. We assume the same forensic requirements as in file carving, namely absence of system catalog metadata and unavailability of DBMS software, and describe how carving can be achieved generally within the context of relational databases. In our previous paper, [15] we have described how long forensic evidence may reside within a database, even after being deleted. In this paper, we delve deeper into the process of page carving and describe a database agnostic mechanism to carve database storage at the page level, as well as show how forensic analysis can be conducted by an investigator.

Database carving can provide useful data for provenance auditing [8], and creation of virtualized database packages [11], which use provenance-mechanisms underneath and are useful for sharing and establishing reproducibility of database applications [12]. In particular, provenance of transactional or deleted data is still a work-in-progress in that provenance systems must support a multi-version semi-ring model [5], which is currently known for simple delete operations and not for delete operations with nested subqueries. Our technique can reconstruct deleted data, regardless of the queries that deleted the data.

3. PAGE CARVING IN DBCARVER

3.1 Page Carving Requirements

We assume a forensic framework for examination of digital evidence as established by the National Institute of Justice [1] and also described in detail by Carrier in Foundations of Digital Investigations [6]. This framework identifies three primary tasks that are typically performed by a forensic investigator in case of a suspicious incident, namely (i) evidence acquisition, (ii) evidence reconstruction, and (iii) evidence analysis. In acquisition, the primary task is to preserve all forms of digital evidence. In this paper, we assume evidence acquisition corresponds to preserving disk images of involved systems. A forensic investigator, depending on the investigation, may also preserve memory by taking snapshots of the process memory. Snapshots of the database process memory can be especially useful for forensic analysis because dirty data can be examined for malicious activity.

Once potential evidence is acquired and preserved, the investigator must reconstruct data from the preserved disk image to determine and analyze potential evidence. To do so, the investigator must follow, as specified in [1, 6], two strict requirements. First, forensic reconstruction or analysis must not write to the acquired disk image as it may potentially change embedded evidence. In the case of database forensics, this implies that a disk image must not be restarted within the context of the original operating or database system because this action might compromise the image. Second, reconstruction must not rely on global system metadata as system metadata may, too, have been compromised or damaged during the incident. In the case of database forensics, this implies not relying on any file inodes or system catalogs for reconstruction. Because most OS and DBMSes need system metadata when restarting from a disk image, the lack of metadata prevents the use of such systems. Thus, for all practical purposes forensic reconstruction and analysis as specified in [1, 6] assumes the lack of availability of system software in which the data was originally resident and any global system metadata.

3.2 DBCarver Overview

The `DBCarver` tool reconstructs data from a relational database that is resident on a disk image for the purpose of a forensic investigation. It reconstructs by interpreting, aka "carving", each individual page, while satisfying reconstruction requirements. Carving each page independently is a practical approach because pages are the smallest unit of persistent storage. Figure 2 summarizes the overall architecture of `DBCarver`. `DBCarver` consists of two main components: the parameter detector(A) and the carver(F).

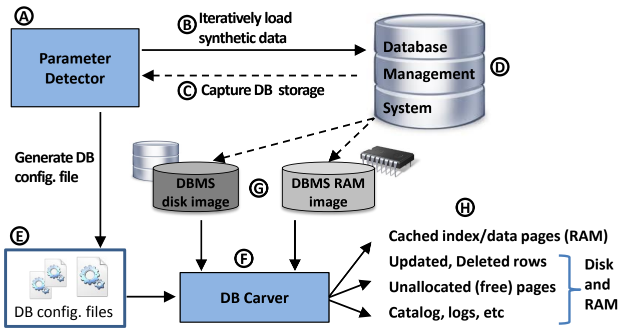


Figure 2: Architecture of `DBCarver`.

The parameter detector calibrates `DBCarver` for the identification and reconstruction of different database pages. To do this, the parameter detector loads synthetic data(B) into a working version of the particular DBMS(D), and it captures underlying storage(C). The parameter detector then learns the layout of the database pages and describes this layout with a set of parameters, that are written to a configuration file(E). A configuration file only needs to be generated once for each specific DBMS version, and it is likely that a configuration file will work for multiple DBMS versions as page layout rarely changed between versions.

The carver(F) then uses these configuration files(E) to identify and reconstruct pages from any type of file(G) passed to it, such as disk images, RAM snapshots, or individual files. The carver searches the input files for database page headers. For each page header found the carver reconstructs the page, and outputs the records(H), along with additional metadata(H) from the pages. This output includes records from tables, value-pointer pairs from indexes, system tables, and deleted data. `DBCarver` has been tested against ten different databases along with several versions for each: DB2, SQL Server, Oracle, PostgreSQL, MySQL, SQLite, Apache Derby, Firebird, Maria DB, and Greenplum.

3.3 Parameter Collector

The parameter detector runs against a DBMS on a trusted machine, and is not intended to operate on a suspect machine. It deconstructs storage, and describes database page structure with a set of parameters that are used later by the carver for page reconstruction. In this section, we discuss how the parameter detector operates, and describe some of the more significant parameters created by `DBCarver` – we do not describe the entire set of parameters due to space limitations.

With the exception of modest user intervention, the parameter collector has been automated. Prior to running the parameter collector, the user is required to provide a configuration file containing several database settings: page size, a directory where database file(s) are to be stored, database connection information, and user credentials with sufficient privileges to create tables/load data. The user may also be required to create a new wrapper class for the DBMS, which must accept user credentials, database connection information, and a SQL file as arguments, and runs the SQL file commands against the database. Additionally, the user may be required to change the SQL schema file for the synthetic tables. This last requirement may occur because there are inconsistencies in data type definitions across DBMSes.

In order to learn details about database storage by the DBMS, the parameter collector automatically loads our own set of synthetically generated data and SSBM [9] data and performs snapshots as the database is being populated. During this process, we perform individual `INSERT` commands rather than bulk load tools. We observed that bulk load tools do not always preserve an insertion order, which is an assumption made by the parameter collector when learning storage layout. Once snapshots are acquired, the parameter collector deconstructs the database storage and outputs the parameters to a file.

For all page types and all RDBMSes, we observed three common page components that we used to categorize the parameters: the page header, the row directory, and the row data. The page header stores characteristics shared by all pages. The row directory maintains pointers to records within the page. The row data contains the raw data itself along with additional metadata.

Page Header.

The page header primarily contains metadata that provides general page information and details about a page's relationship with a database. Figure 3 displays two example page headers from different databases containing four types of metadata: *general page identifier* (A), *unique page identifier* (B), *object identifier* (C), and *record count* (D). The *general page identifier* is a sequence of (typically 2 to 4) bytes shared by all database pages, and it is used for database page identification by the carver. The *unique page identifier* is typically a 32-bit or 64-bit number that is unique for each page within a file or across the entire database. The *object identifier* is usually a 32-bit number that is unique for each object (e.g., table or index) across the database. The *record count* is a 16-bit number that represents the number of *active* records within the page, and it is updated when a record in the page is modified.

The page header parameters are determined by comparing many pages (on the order of 10^5) belonging to various objects, objects types, and database files. Table 1 lists and describes the parameters the parameter collector returned in order to determine how this page header metadata were stored. The **general page identifiers**, *(162, 0, 0)* and *(32, 8, 32)*, for each example were recorded along with their

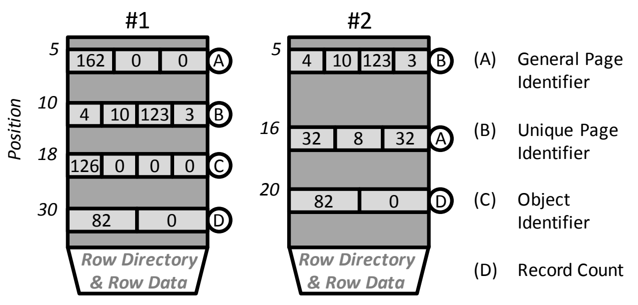


Figure 3: Two example page headers belonging to different databases.

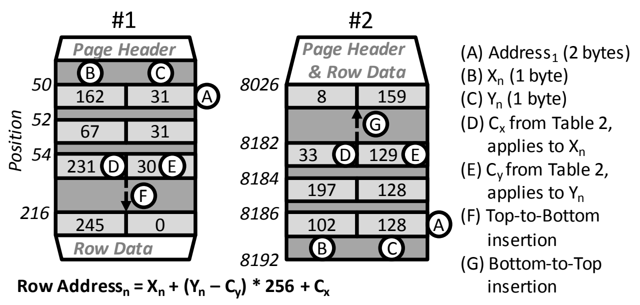

#1
#2
Page Header
Position
50
52
54
216
162 31
67 31
231 30
245 0
Row Data
Page Header & Row Data
8026
8182
8184
8186
8192
8 159
33 129
197 128
102 128
Row Address$_n$ = X$_n$ + (Y$_n$ – C$_y$) * 256 + C$_x$
(A) Address$_1$ (2 bytes)
(B) X$_n$ (1 byte)
(C) Y$_n$ (1 byte)
(D) C$_x$ from Table 2, applies to X$_n$
(E) C$_y$ from Table 2, applies to Y$_n$
(F) Top-to-Bottom insertion
(G) Bottom-to-Top insertion

Figure 4: Two example row directories belonging to different databases.

positions from the top of the page (or the **general page identifier position**), 5 and 16. Both examples stored a *unique page identifier*. The **unique page identifier size**, 4 bytes, and the **unique page identifier positions**, 10 and 5, were recorded. Example #1 in Figure 3 contains an *object identifier*, but example #2 in Figure 3 does not. In example Figure 3-#1, the **object identifier size**, 4 bytes, and the **object identifier position**, 18, were recorded. A `NULL` value was record for both of these parameters in example Figure 3-#2. Both examples contain a *record count*. The **record count size**, 2 bytes, and the **record count positions**, 30 and 20, were recorded for each example.

Parameter	**Figure 3 Value**	
	3-#1	**3-#2**
General Page Identifier	(162, 0, 0)	(32, 8, 32)
General Page Identifier Position	5	16
Unique Page Identifier Position	10	5
Unique Page Identifier Size	4 bytes	
Object Identifier Position	18	NULL
Object Identifier Size	4 bytes	NULL
Record Count Position	30	20
Record Count Size	2 bytes	

Table 1: Page header parameters used to reconstruct Figure 3.

Row Directory.

The row directory maintains a set of addresses referencing the records within a page. The row directory can be positioned either between the page header and the row data or at the end of the page following both the page header and the row data. A row directory may store an address for each record (dense) or an address per multiple records (sparse). Furthermore, the row directory addresses may be used to mark row status (deleted or active). Figure 4 displays two example row directories for different databases. Both examples store an address as a 16-bit, little endian number (B & C). The decoding constants C_x (D) and C_y (E) are used when the explicit addresses are not stored. These values are the same for all addresses and all pages for a DBMS. Example 4-#1 was positioned between the page header and the row data. The first address (A) began at position 50 and addresses are appended from top-to-bottom (F). Example 4-#2 was positioned after the page header and the row data. The first address (A) began at position 8186 and addresses are appended from bottom-to-top (G).

The row directory parameters were determined by searching within a page for a set of candidate addresses and validating this set with many pages. While the row directory is similar for an object type (e.g., table, index, system table), differences may exist across object types; consequently, this process is repeated for different object types. Table 2 lists and describes the parameters the parameter detector used to deconstruct each row directory example. In both examples, the position of the first address was recorded as the **Row Directory Position**, 50 and 8186. The **Address Size** in both examples was 2 bytes, and both examples used **Little Endian**. Example #1 in Figure 4 appends addresses from **Top-to-Bottom**, and example #2 in Figure 4 instead appends rows from **Bottom-to-Top**. Figure 4-#2 required decoding constants to calculate the explicit addresses. In the Figure 4-#2 parameter file, -2 was recorded for C_x and 128 was recorded for C_y. Figure 4-#1 stored the explicit addresses; hence, 0 was recorded for both decoding constant parameters.

Row Data.

The row data stores the actual raw data itself along with metadata that describes the raw data. The layout of the row data is similar across objects of a similar type. For example, the row data for table pages contains data inserted by the user, but the row data for index pages contains value-pointer pairs. Furthermore, the metadata in the row data may describe the status of raw data (active or deleted). Figure 5 visualizes three example row data for different databases. Example #1 in Figure 5 used a *row delimiter* (A) in order to separate rows. This position is typically where a row directory points within a row. Examples #1, #2 and #3 in Figure 5 all store a *column count* (B), which is an explicit numbers of columns stored in each row. Example #2 in Figure 5 uses a *row identifier* (E), which is a segment of an internal database pseudocolumn. This pseudocolumn is referred to as 'ROWID' in Oracle and 'CTID' in PostgreSQL. Examples #1 and #2 in Figure 5 store the column sizes. Figure 5-#1 stores the *column sizes within the raw data* (C), and Figure 5-#2 stores the column sizes in the row header (F) before the raw data began. Alternatively, Figure 5-#3 used a column directory (G) to store column addresses rather than column sizes. Figures 5-#1 and 5-#2 use column sizes and, thus, store raw numbers with strings (D); Figure 5-#3 uses a column directory and, therefore, stores raw numbers separately from raw strings (H) in the column directory.

The row data parameters were determined by locating

Parameter	Description	Figure 4 Value	
		4-#1	4-#2
Row Directory Position	The position of the first address.	50	8186
Little Endian	Little endian is used to store addresses.	True	
Top-to-Bottom Insertion	Addresses are appended in ascending order.	True	False
Address Size	The number of bytes used to store each address.	2 bytes	
C_x	A decoding constant for X_n when the explicit address is not stored.	0	-2
C_y	A decoding constant for Y_n when the explicit address is not stored.	0	128

Table 2: Row directory parameters used to reconstruct Figure 4.

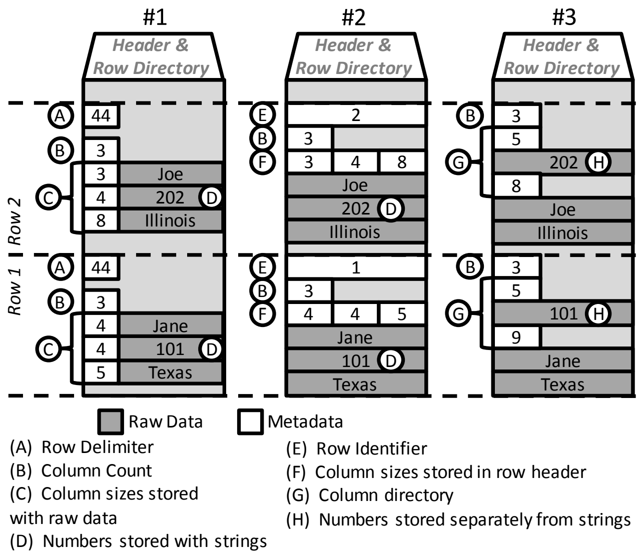


Figure 5: Three example row data layouts.

Parameter	Figure 5 Value		
	5-#1	5-#2	5-#3
Row Delimiter	44	NULL	
Row Identifier Position	NULL	0	NULL
Column Count Position	1	4	0
Column Sizes in Raw Data	True	False	
Column Sizes Position	NULL	5	NULL
Column Directory Position	NULL		1

Table 3: Row data parameters used to reconstruct Figure 5.

known synthetic data and comparing the metadata for many rows (on the order of 10^6) for dozens of objects. These parameters were then confirmed using the SSBM data. This process was repeated for each object type. Table 3 lists and describes the detected parameters that were used to characterize each row data layout. Example 5-#1 in Table 3 was the only one that uses a *row delimiter*, thus the **row delimiter** parameter value 44 was recorded. Only example 5-#2 stored a *row identifier*, consequently the **row identifier position** within the row, 0, was recorded. Examples 5-#1, 5-#2, and 5-#3 in Table 3 all stored a column count; accordingly, their **column count positions** (1, 4, and 0) were stored. The **column sizes in raw data** Boolean parameter signaled that the column sizes should be read in the raw data, such as in example 5-#1. The position of the column sizes in the row header in example 5-#2 was recorded with **column sizes position**, 5. Finally, the *column directory* in example 5-#3 was recorded using **column directory position**, 1.

3.4 Carver

The carver is the read-only component of `DBCarver` that accepts any type of storage from a suspect machine and any number of parameter files generated by the parameter collector as input, parses the storage contents for the relevant databases, and returns all discovered database content. The carver is a command line tool that requires two arguments: the name of a directory that contains the input image files and the name of a directory where the output should be written. No other user intervention is necessary. Figure 6 summarizes the database content that `DBCarver` can carve and make available for forensic investigation. When the input is a disk image, the page carving process from `DBCarver` results in two kinds of information: (i) the original database content, which is queryable by the user, reconstructed as database tables, indexes, materialized views, system catalogs, and log files; (ii) the non-queryable data that is embedded with the reconstructed data objects, such as data that was deleted from a table or materialized view or system catalog or unallocated pages, i.e. zombie data. The latter data can be extracted by `DBCarver` only, it cannot queried from the database and log files. When the input is a RAM snapshot, the result is database buffer cache pages (as distinguished from other memory pages), which may correspond to intermediate results or log buffer pages.

The carver begins by searching the input files for the **general page identifier** from Table 1. When a **general page identifier** is found, the carver reconstructs each of the three page components: page header, row directory, and row data. Because the **general page identifier** is typically a sequence of a few bytes, false positives are likely to occur. The carver verifies each page component using a number of assumptions, which eliminates false positives. Some of these assumptions include: the identifiers in the page header must be greater than 0, the row directory must have at least on address, and the row data must contain at least one row.

Page Header.

The parameter values in Table 1 were used to reconstruct the page header metadata in both Figure 3 examples. Table 4 summarizes the reconstructed metadata. In example 3.1, the carver moved to position 10 and read four bytes to re-

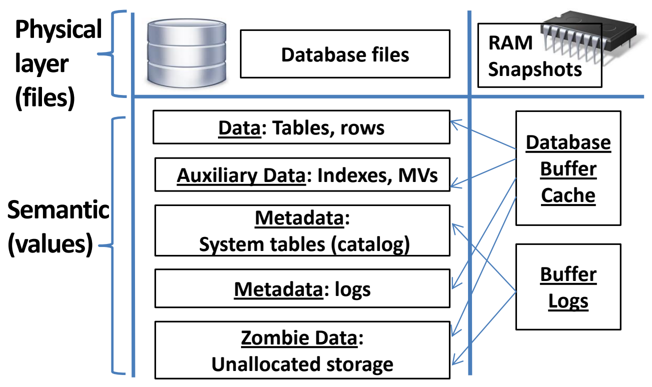


Figure 6: Forensically relevant content in a database: with the exception of indices, every category can include both active and deleted values.

construct the *unique page identifier* as a 32-bit little endian number, 58395140. The carver then read four bytes at position 18 to reconstruct the the *object identifier*, 126. Finally, the carver moved to position 30 to reconstruct the *record count*, 82. This process was repeated for example 3.2 except an *object identifier* was not able to be reconstructed because the **object identifier position** and **object identifier size** were `NULL`.

Meta Data	Figure 3 Value	
	3.1	**3.2**
Unique Page Identifier	58395140/(4, 10, 123, 3)	
Object Identifier	126	NULL
Record Count	82	

Table 4: Reconstructed page header meta data values from Figure 3.

Row Directory.

The parameter values in Table 2 were used to reconstruct the row directory in both Figure 4 examples. Table 5 summarizes the reconstructed row directory addresses. The parser used **row directory position** to move to the beginning of the row directory. Each address was reconstructed using the equation: $RowAddress_n = X_n + (Y_n - C_y) * 256 + C_x$, where C_x and C_y are decoding constants stored as parameters, and X_n and Y_n are the least-significant and most-significant bytes of the 16-bit number. After the first address has been reconstructed, the parser moves on the remaining address using **Address Size** and **Top-to-Bottom Insertion**. The carver makes some assumptions to validate an address, such as that the address cannot be larger than the page size and an address must be located somewhere within the row data of the page.

Row Data.

The parameter values in Table 3 were used to reconstruct the row data in the three examples from Figure 5. Table 6 summarizes the reconstructed row data and row meta data. The carver reconstructed the *column count* by moving to the **column count position** within the row and reading the respective byte. The carver reconstructed the *row identifier*,

Address	Figure 4 Value	
	4.1	**4.2**
$Address_1$	8098	100
$Address_2$	8003	195
$Address_3$	7911	287
$Address_n$	245	7942

Table 5: Reconstructed row directory address from Figure 4.

for example 5.2, by moving to the **column count position** within the row and reading the value. Finally, the carver reconstructed each column of raw data by first determining the column size using either the **column sizes in raw data** or the **column sizes position** and then reading column data at the **column directory position**.

Data/Meta Data	Figure 5 Value		
	5.1	**5.2**	**5.3**
Column Count	3		
Row_1 Row Identifier	NULL	1	NULL
Row_1 Raw Data	Jane, 101, Texas		
Row_2 Row Identifier	NULL	2	NULL
Row_2 Raw Data	Joe, 202, Illinois		

Table 6: Reconstructed data and meta data from Figure 5.

Meta-Columns.

While the reconstructed data can tell us what was present in database tables, page carving must explicitly expose the internal data and metadata in order to enable forensic queries about that data. Table 7 summarizes a few internal columns that are a part of each reconstructed table and materialized view and that enable detailed forensic analysis. In order to enable such questions, we add a few meta-columns to all reconstructed tables.

Meta-Column	Description
Object Identifier	A unique identifier for each object across the database
Page Identifier	A unique identifier for each page for joining DB and RAM pages
Row Offset	Unique identifier of a row within a page.
Row Status	Distinguishes active rows from deleted rows.

Table 7: Metadata used to describe the reconstructed data.

4. DATABASE FORENSICS ANALYSIS

After data has been extracted from the storage, it must be analyzed to determine its significance. By connecting reconstructed metadata and data, investigators can ask simple questions that validate whether system metadata is consistent with the data (i.e., no column type or number of columns were altered). More interesting forensic analysis can be performed using recovered deleted data and by combining both active, deleted, and memory data. We present several types of scenarios that a forensic investigator may wish to explore and present queries that can be answered with the help of carved data. We term the scenarios "meta-queries"' because such queries are not executed on the original active database but on the reconstructed data.

Scenario 1: Reconstruction of Deleted Data.

An analyst may need to determine what values were potentially deleted in a database. In particular, identifying deleted rows would be of interest if we assume that the audit logs are missing. For example, a logged query,

```
DELETE FROM Customer
WHERE Name = ReturnNameFunction(),
```

does not reveal anything about the specific records that were deleted. With database carving analysis however, that records that were deleted could be identified readily by running the following query:

```
SELECT * FROM CarvCustomer
WHERE RowStatus = 'DELETED'.
```

Notably, database carving can only determine whether rows were deleted and not the reasons for or mechanism by which the deletion occurred.

Scenario 2: Detecting Updated Data.

Similar to the deleted values, we may want to find all of the most recent updates, carved from a database RAM snapshot. For example, consider the problem of searching for all recent product price changes in RAM. In order to form this query, we would need to join disk and memory storage, returning the rows for which price is different:

```
SELECT *
FROM CarvRAMProduct AS Mem, CarvDiskProduct AS Disk
WHERE Mem.PID = Disk.PID
AND Mem.Price <> Disk.Price.
```

Scenario 3: Tampering of Database Schema.

If we suspect that someone may have tampered with the database by making changes to a database schema (e.g,. remove a constraint, drop a table) we can query the carved system tables to find schema changes. For example:

```
SELECT * FROM CarvSysConstraints
WHERE RowStatus = 'DELETED'.
```

Scenario 4: Identifying Missing Records in a Corrupted Database.

Forensic analysis may be performed in the face of database damage or corruption. For example, the perpetrator may delete database files to impede the investigation. If the files in question were not yet overwritten, then `DBCarver` will successfully reconstruct all of the database content. Once the database file is partially overwritten though, we can carve all surviving pages and explore auxiliary structures to identify missing records. For example, when searching for customer records of a partially overwritten table, we could use the query:

```
SELECT * FROM CarvCustomer,
```

to find remaining customer records and the following query to determine how many customers are missing from the output of the first query:

```
SELECT COUNT(SSN) FROM CarvCustIndex
WHERE SSN NOT IN (SELECT SSN FROM CarvCustomer),
```

(because `UNIQUE` constraint will automatically create an index).

5. EXPERIMENTS

Our current implementation of `DBCarver` applies to *ten different* RDBMSes under both Windows and Linux OS. We present experiments using four representative databases (Oracle, PostgreSQL, MySQL, and SQL Server). In this section, we used data from the SSBM [9] benchmark.

Our experiments were carried out using an Intel X3470 2.93 GHz processor with 8GB of RAM; Windows experiments run Windows Server 2008 R2 Enterprise SP1 and Linux experiments use CentOS 6.5. Windows operating system RAM images were generated using Windows Memory Reader. Linux memory images were generated by reading the process' memory under `/proc/$pid/mem`. `DBCarver` read either the database files or the raw hard drive image because the file system structure is not needed.

5.1 Experiment 1. System Table Carving

The objective of this experiment is to demonstrate the reconstruction of system tables with `DBCarver`. In Part A, we retrieve the set of column names that belong to tables in a PostgreSQL DBMS, using them to reconstruct the schema. In Part B, we associate the name of a view with its SQL text in an Oracle DBMS.

Part A.

For a PostgreSQL database, we created the *CUSTOMER* table (8 columns) and the *SUPPLIER* table (7 columns) from the SSBM benchmark. We then passed all of database system files related to this instance to `DBCarver`.

Our analysis focuses on two tables used by PostgreSQL. Specifically, PostgreSQL stores information about each object in the *PG_CLASS* table and information about each column in the *PG_ATTRIBUTE* table. From the `DBCarver` output, we performed a grep search to locate the records for the *CUSTOMER* and the *SUPPLIER* tables in the reconstructed *PG_CLASS* table. In order to abbreviate the output, we reported only the *Object Name* and *Object Identifier* for each tuple: *('customer', 16680)* and *('supplier', 16683)*. In the reconstructed *PG_ATTRIBUTE* table, we found 14 records with the *Table Object Identifier* of '16680' and 13 records with the *Table Object Identifier* of '16683'. We then used the *Object Identifier* column from both *PG_CLASS* and *PG_ATTRIBUTE* to reconstruct the schema. For both the *CUSTOMER* and the *SUPPLIER* tables, 6 records from *PG_ATTRIBUTE* were observed to have been created by the system (i.e., they were not created by us). This means we connected 6 system-related pseudo-columns for each table in addition to the columns we declared. We also note that the *Object Identifier* we used to join the two system tables corresponds to the database file name for each table. This identifier is not stored in the page header for PostgreSQL. Alternatively, a database like Oracle or DB2 does not use a file-per-object storage architecture and stores the *Object Identifier* in the page header.

Part B.

For an Oracle database, we initially started with the *CUSTOMER* table, and we created the view *MyQuery*,

```
CREATE View MyQuery AS
SELECT C_Name, C_Address
FROM CUSTOMER.
```

Next, we passed the default system database file, 'SYSTEM01.DBF', used by Oracle to `DBCarver`.

Oracle stores information about each object in the *obj$* table and information about views in the *view$* table. In the *obj$* table reconstructed by `DBCarver`, we found a record with an *Object Name* value 'MYQUERY', *Object Identifier* value '109661', and *Object Type* of 'view'. Again, in order to abbreviate the output, we reported only the *Object Name* and *Object Identifier* for this record: *(109661, 'MYQUERY')*. In the *view$* table reconstructed by `DBCarver`, we found a record with the *Object Identifier* value '109661' along with the following SQL text:

```
SELECT C_Name, C_Address
FROM CUSTOMER.
```

We then joined both tables using the *Object Identifier* attribute in order to associate 'MYQUERY' with the SQL text.

5.2 Experiment 2. Baseline Carving Performance

The objective of this experiment is to demonstrate that the runtime for `DBCarver` is linearly proportional to the number of pages in a database file. To show this, we evaluate the performance of `DBCarver` against Oracle database files.

We loaded the five tables from the SSBM benchmark (Scale 4) into an Oracle database. Each table was loaded into a separate file. Finally, we passed all five files to `DBCarver`.

Table 8 summarizes the file sizes, pages, and rows returned by `DBCarver`, in addition to the `DBCarver` runtimes. `DBCarver` carved the Oracle database files at a rate of 1.3 MB/s.

Table	DB File (MB)	Pages	Rows	Time (MB/s)
DWDate	1	32	2556	1.7
Supplier	1	94	8K	1.4
Customer	15	1611	120K	1.3
Part	63	7223	600K	1.3
Lineorder	2443	289K	24M	1.3

Table 8: Cost to carve data files using `DBCarver`.

5.3 Experiment 3. Selective Carving Performance

The objective of this experiment is to demonstrate that the runtime for `DBCarver` is dependent on the quantity of reconstructed data, not the file size. To show this, we evaluate the performance of `DBCarver` against a RAM image. In Part A, we evaluate the performance when `DBCarver` returns all active and deleted records. In Part B, we evaluate the performance when `DBCarver` returns only deleted records.

Part A.

We captured an 8GB RAM image containing an Oracle buffer cache. The Oracle database used 8kB pages. We then passed the 8GB RAM image to `DBCarver`, and configured the output to include all active and deleted records.

`DBCarver` returned roughly 65,000 pages (500MB) from the 8GB image at an average rate of **14.3 MB/s**. Given that Experiment 2 demonstrated that database files are carved at the rate of 1.3 Mb/s, the increase in carving rate for RAM images indicates that the carving cost is dependent on output size (the database pages) rather than the input size.

Part B.

We passed the same 8GB RAM image from Part A to `DBCarver`. However, this time we configured the output to include only the deleted records.

`DBCarver` was still accessed 65,000 page headers and row directories but did not reconstruct active records. 1956 deleted records were reconstructed at an average rate of **34.1 MB/s**. Therefore, if the investigator knows what he or she needs (e.g., specific rows or particular schema) carving performance can be greatly optimized.

5.4 Experiment 4. Evidence Source Corruption

The objective of this experiment is to investigate the accuracy of `DBCarver` against corrupted data sources. We simulate corruption by performing random overwrites of a disk image containing SQL Server database files.

We initially loaded the five table from the SSBM benchmark (Scale 1) into an SQL Server database. Next, we created an image of persistent storage. We then simulated corruption by randomly overwriting 1kB segments in the image. We applied four levels of corruption: 1%, 2%, 5%, and 10% worth of image overwrites, where each level was independent of others. Finally, we passed each damaged image to `DBCarver`.

Table 9 summarizes the data reconstructed by `DBCarver` for each level of corruption. The `JOIN` results represent all complete records that could be reconstructed with full denormalization. We have two explanations of why an overall percentage of rows reconstructed was slightly lower than the percent damaged. First, when the page header or row directory is damaged, the page cannot be carved. Therefore, `DBCarver`cannot reconstruct the remaining page contents. Second, if a row directory address is damaged, `DBCarver` cannot reconstruct that row.

5.5 Experiment 5. Database Storage Convergence

The objective of this experiment is to trace the storage changes step-by-step within an Oracle DBMS. In Part A, we follow the side-effects of a `DELETE` command, which is summarized in Table 10. In Part B, we follow the side-effects of an `UPDATE` command, which is summarized in Table 11. For each command, we describe the changes related to the table, index, and materialized view(MV) in both the database files and the buffer cache.

Part A.

- **T_0:** We initially created an *EMPLOYEE* table, which included an *SSN* column. We also created a primary key index on the *SSN* column and included this column in an MV.
- **T_1:** Next, we issued a `DELETE` command:
  `DELETE FROM Employee WHERE SSN = '222'.`
  In order to delete this record, the DBMS accessed data that caused the system to cache the table page and the index page(s) used in lookup. The record was marked deleted (denoted by ✗) in the buffer cache table page, but '222' cannot be marked deleted in the index page. The record was not marked as deleted in the database file on disk because the page had not yet been flushed from memory.

	File Percent Damage				
Table	**0%**	**1%**	**2%**	**5%**	**10%**
DWDate – # of reconstructed rows (%)	2556 (100%)	2459 (96%)	2384 (93%)	2130 (83%)	2147 (84%)
Supplier – # of reconstructed rows (%)	2000 (100%)	1987 (99%)	2000 (100%)	1740 (87%)	1680 (84%)
Customer – # of reconstructed rows (%)	120K (100%)	118K (98%)	115K (96%)	108K (90%)	96K (80%)
Part – # of reconstructed rows (%)	200K (100%)	195K (97%)	189K (94%)	174K (87%)	146K (73%)
Lineorder – # of reconstructed rows (%)	6M (100%)	5.8M (97%)	5.7M (95%)	5.2M (87%)	4.5M (75%)
Full `JOIN` – # of reconstructed rows (%)	6M (100%)	5.3M (88%)	4.9M (81%)	2.9M (49%)	1.9M (31%)

Table 9: Data reconstructed from a corrupted file.

- **T_2:** We then refreshed the MV. This caused the MV page to be cached, and the record with SSN value '222' was marked deleted in the buffer cache MV page. The record for the MV had not been yet been marked as deleted in the database file.
- **T_3:** We then flushed the buffer cache with the following privileged command:
  `ALTER SYSTEM FLUSH BUFFER CACHE.`
  This command forced the modified pages to be written to the database files. All pages in the buffer cache became unallocated space but were not immediately overwritten, thereby permitting them to be carved. `DBCarver` reconstructed the deleted '222' record from the table page and the MV page, and the value from the index page.
- **T_4:** Next, we issued many queries to fill the buffer cache. All of the free-listed pages were overwritten and could no longer be reconstructed from memory. '222' was no longer found in memory.
- **T_5:** Finally, we issued explicit rebuild commands for the table, index, and MV. This action caused the deleted record to be overwritten in the table and MV, and the value to be overwritten in the index. Such a rebuild can also be caused by a sufficiently high volume of modifications.

	Hard Drive			**Buffer Cache (RAM)**		
	Table	**Index**	**MV**	**Table**	**Index**	**MV**
T_0	222	222	222			
T_1	222	222	222	✗222	222	
T_2	222	222	222	✗222	222	✗222
T_3	✗222	222	✗222	✗222	222	✗222
T_4	✗222	222	✗222			
T_5						

Table 10: A timeline for the `DELETE` convergence.

Part B.

- **T_0:** We continued with the setup from Part A. Times T_1 through T_3 are split into two options: T_n-A represents an update-in-place and T_n-B corresponds to `DELETE`+`INSERT` alternative. The latter is a possibility because some databases (e.g,. PostgreSQL) avoid in-place updates in general and most databases cannot update in-place when new value is larger than the old value.
- **T_1:** First, we issued an `UPDATE` command:
  `UPDATE Employee SET SSN = 'JAY' WHERE SSN = '888'.`
  The update required both the table and index pages to be read into the buffer cache. In the T_1-A buffer cache, '888' was overwritten by 'JAY' in the table page, and the index stores both '888' (without deletion mark) and 'JAY'. Alternatively, at T_1-B, the table page was cached with '888' marked as deleted while a new row with 'JAY' was inserted into another table page in memory. The index at time T_1-B still maintains both values, the same as at T_1-A. No changes occurred on disk at this point.
- **T_2:** Next, we refreshed the MV. At T_2-A, '888' was overwritten by 'JAY' in the buffer cache MV page. At T_1-B, the MV page was cached with '888' marked as deleted while a new row with 'JAY' was inserted into another MV page in memory.
- **T_3:** We then issued the privileged flush buffer cache command. In both T_3-A and T_3-B, the cached page versions are written to disk, overwriting the old page versions.
- **T_4:**Finally, T_4 represents the final convergence state to which T_3-A and T_3-B arrive *after* all three structures have been rebuilt. Achieving final convergence requires explicit rebuild commands or a long duration of normal operations.

	Hard Drive			**Buffer Cache (RAM)**		
	Table	**Index**	**MV**	**Table**	**Index**	**MV**
T_0	888	888	888			
T_1-A	888	888	888	JAY	888 JAY	
T_2-A	888	888	888	JAY	888 JAY	JAY
T_3-A	JAY	888 JAY	JAY			
T_1-B	888	888	888	✗888 JAY	888 JAY	
T_2-B	888	888	888	✗888 JAY	888 JAY	✗888 JAY
T_3-B	✗888 JAY	888 JAY	✗888 JAY			
T_4	JAY	JAY	JAY			

Table 11: A timeline for the `UPDATE` convergence.

5.6 Experiment 6. Connecting Multiple Forensic Evidence

The objective of this experiment is to use analytic queries to connect and query two sources of forensic evidence. We connect a RAM and a disk image from the same machine.

We started with the *CUSTOMER* table from the SSBM benchmark and a MySQL database. Next, we issued a DELETE command to simulate malicious activity:

```
DELETE FROM Customer WHERE C_Nation = 'Canada'.
```

In order to investigate database activity, we collected a RAM image and a disk image, and we passed each image to DBCarver. DBCarver returned a *CUSTOMER* table from each image. We named the two tables *CustomerDisk* and *CustomerRAM* and loaded the reconstructed rows into a clean database. Along with columns belonging to the original table *CUSTOMER* table, we included our meta-columns: object identifier, page identifier, row offset, and row status.

Because we suspected a malicious delete, we had two forensic questions: 1) what records recently deleted, and 2) what values were the likely target of the deletion. The following query returned 4854 rows where the row status was 'active' on disk but 'deleted' in memory (or recently modified):

```
SELECT *
FROM CustomerDisk AS Disk, CustomerRAM AS RAM
WHERE Disk.ObjectIdentifier = RAM.ObjectIdentifier
AND Disk.PageIdentifier = RAM.PageIdentifier
AND Disk.RowOffset = RAM.RowOffset
AND Disk.RowStatus = 'ACTIVE'
AND RAM.RowStatus = 'DELETED'.
```

Next, we were able to issue queries to search for similarities between the deleted records. For example, the following query told us that all of the recently modified records had the *C_Nation* value 'Canada' in common:

```
SELECT *
FROM CustomerRAM
GROUP BY C_Nation
HAVING COUNT(*) =  (Count of recently modified rows).
```

6. CONCLUSION AND FUTURE WORK

In this work, we have presented and evaluated DBCarver, a novel forensic tool that permits reconstruction of database storage. The advantage of our carving approach is that it generalizes to many relational databases and does not rely on DBMS. Reconstructed database content can then be queried to help with forensic analysis of the database. Our analysis can be applied even if the database logs are missing or the database itself is corrupt.

We intend to investigate how database carving can augment and generalize provenance auditing solutions. Database carving will contribute to reproducibility of database applications and performing comparative studies across different database engines. Database storage analysis can provide new ways for monitoring user access and detecting log tampering. Finally, we plan to combine database carving and other database forensic and provenance techniques to develop comprehensive forensic analysis that can answer evidential queries while preserving the evidence from a read-only data source.

Acknowledgments

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

7. REFERENCES

[1] Forensic examination of digital evidence: A guide for law enforcement. https://www.ncjrs.gov/pdffiles1/nij/199408.pdf.

[2] A. Act. Health insurance portability and accountability act of 1996. Public law, 104:191, 1996.

[3] S.-O. Act. Sarbanes-oxley act. Washington DC, 2002.

[4] O. M. Adedayo and M. S. Olivier. On the completeness of reconstructed data for database forensics. In International Conference on Digital Forensics and Cyber Crime, pages 220–238. Springer, 2012.

[5] B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Reenactment for read-committed snapshot isolation. In Proceedings of the 25th ACM International Conference on Information and Knowledge Management (CIKM), 2016.

[6] B. Carrier. File system forensic analysis. Addison-Wesley Professional, 2005.

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

[8] B. Glavic. Perm: Efficient Provenance Support for Relational Databases. PhD thesis, University of Zurich, 2010.

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

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

[11] Q. Pham, T. Malik, B. Glavic, and I. Foster. Ldv: Light-weight database virtualization. In 2015 IEEE 31st International Conference on Data Engineering, pages 1179–1190. IEEE, 2015.

[12] Q. Pham, S. Thaler, T. Malik, I. Foster, and B. Glavic. Sharing and reproducing database applications. Proceedings of the VLDB Endowment, 8(12):1988–1991, 2015.

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

[14] 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, 2007.

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