Digital Investigation 29 (2019) S42–S50

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

DFRWS 2019 USA — Proceedings of the Nineteenth Annual DFRWS USA

DB3F & DF-Toolkit: The Database Forensic File Format and the Database Forensic Toolkit

James Wagner [a, *], Alexander Rasin [a], Karen Heart [a], Rebecca Jacob [a], Jonathan Grier [b]

[a] DePaul University, Chicago, IL, USA
[b] Grier Forensics, USA

ARTICLE INFO

Article history:

ABSTRACT

The majority of sensitive and personal user data is stored in different Database Management Systems (DBMS). For Example, Oracle is frequently used to store corporate data, MySQL serves as the back-end storage for most webstores, and SQLite stores personal data such as SMS messages on a phone or browser bookmarks. Each DBMS manages its own storage (within the operating system), thus databases require their own set of forensic tools. While database carving solutions have been built by multiple research groups, forensic investigators today still lack the tools necessary to analyze DBMS forensic artifacts. The unique nature of database storage and the resulting forensic artifacts require established standards for artifact storage and viewing mechanisms in order for such advanced analysis tools to be developed.

In this paper, we present 1) a standard storage format, Database Forensic File Format (DB3F), for database forensic tools output that follows the guidelines established by other (file system) forensic tools, and 2) a view and search toolkit, Database Forensic Toolkit (DF-Toolkit), that enables the analysis of data stored in our database forensic format. Using our prototype implementation, we demonstrate that our toolkit follows the state-of-the-art design used by current forensic tools and offers easy-to-interpret database artifact search capabilities.

1. Introduction

Database Management Systems (DBMS) are often used to manage data in both corporate and personal settings. For Example, a lightweight DBMS, such as SQLite, commonly manages personal data stored on mobile phones or web browsers. Whereas, a DBMS that supports more robust access control and storage management, such as Oracle, PostgreSQL, MySQL, or Microsoft SQL Server, is better suited to manage corporate data. Given the widespread use of DBMSes, their contents are frequently relevant to forensic investigations.

DBMSes manage their own storage (both on disk and in RAM) independently from the operating system. As a consequence, the operating system is oblivious to DBMS storage activity such as record modifications or caching policies. Therefore, an investigation that only uses digital forensics tools to reconstruct storage contents from the operating system is inadequate when a DBMS is involved.

Database forensic carving tools have been proposed (Guidance Software, 2018a; Drinkwater; Wagner et al., 2015, 2016, 2017a; Percona, 2018), but incorporating their output into an investigation remains difficult to impossible. The storage disparity between DBMSes and operating systems may well in fact be the main culprit for the stunted growth and limited applications of database forensics. We identified two major pieces currently missing from the field of database forensics that have prevented its involvement in forensic investigations: 1) a standardized storage format, and 2) a toolkit to view and search database forensic artifacts.

Standard Storage Format. A standard database forensic storage format would abstract the specifics of DBMS storage engines for users unfamiliar with DBMS internals and guide the development of database carving tools. All DBMSes use their own storage engine. A standard storage format would allow users to view and search database forensic artifacts, generate reports, and develop advanced analytic tools without knowledge of storage engine specifics for any given DBMS. A uniform output for database carving tools would

* Corresponding author.
E-mail addresses: jwagne32@depaul.edu (J. Wagner), arasin@depaul.edu (A. Rasin), kheart@depaul.edu (K. Heart), rebecca.ann.jacob@gmail.com (R. Jacob), jdgrier@grierforensics.com (J. Grier).

https://doi.org/10.1016/j.diin.2019.04.010

also allow these tools to be compared and tested against each other.

View and Search Toolkit. A toolkit to view and search reconstructed DBMS artifacts would allow investigators to easily interpret the artifacts. While database data is stored and queried through records in tables, the records alone do not accurately represent the forensic state of a database since this data is accompanied by a variety of metadata (e.g., byte offset of the record). Investigators need a way to view how the metadata and table records are interconnected.

In this paper, we describe a comprehensive framework to represent and search database forensic artifacts. A preliminary version of this framework was implemented for this paper, which includes a format specification document and an evidence querying application. Section 2 considers the work related to our framework, and Section 3 defines our framework requirements. Next, Sections 4 and 5 present the two main contributions of this paper, which are the following:

1. We define a storage format and data abstraction for database forensic artifacts called the Database Forensic File Format (DB3F). Section 4 provides a detailed description of DB3F. The DB3F definition can be downloaded from our research group website: http://dbgroup.cdm.depaul.edu/DF-Toolkit.html.
2. We describe a toolkit called the Database Forensic Toolkit (DF-Toolkit) to view and search data stored in DB3F. Along with a description of DF-Toolkit, Section 5 presents a user interface that implements DF-Toolkit. This user interface can be downloaded from our research group website: http://dbgroup.cdm.depaul.edu/DF-Toolkit.html.

Fig. 1 displays how DB3F and DF-Toolkit are used in database forensic analysis. Database carving tools return output in DB3F. DB3F files are filtered and searched using DF-Toolkit, which stores filtered results in DB3F. DB3F files are then either directly reported to the end user or passed to further advanced analytic applications.

The introduction of a standardized intermediate format and a comprehensive toolkit for database forensics benefits the community in two important ways. First, it streamlines the addition of new tools on either side of the flow chart in Fig. 1. With the introduction of a new database carving tool (e.g., Tool D), users would benefit from all available advanced applications that support DB3F. Similarly, any newly developed advanced application can trivially process output from any carving tool that supports DB3F output. This intermediary approach is conceptually similar to Low Level Virtual Machine (LLVM) (Lattner and Adve, 2004), a collection of reusable compiler technologies that defines a set of common language-independent primitives. The second benefit is the explicit documentation and built-in reproducibility of the analyses process and outcomes, bringing a scientific approach to digital forensics. Garfinkel (Garfinkel et al., 2009) emphasized the lack of scientific rigor and reproducibility within the field; although in (Garfinkel et al., 2009) he focused on developing standard corpora, a standard storage format as well as a querying and viewing mechanism is also necessary to achieve these goals. Rather than building custom analytic tools (e.g., (Wagner et al., 2017b)), DF-Toolkit's approach will offer a well-documented querying mechanism based on defined standard fields in DB3F. Any query report can be easily reproduced by another party or re-tested via a different database carver.

This paper serves as the foundation for a vision of a complete system with full support for database forensics and integration with other forensic tools. Section 6 discusses planned improvements for future developments to our framework, including advanced analytic applications.

2. Related work

This section presents work related to both DB3F and DF-Toolkit. To help formulate our storage format, we took into consideration metadata usage by many forensic tools, the capabilities of database carving tools, and forensic languages used outside of database forensics. To help design our view and search toolkit, we consider the evidence tree structure used by many forensic tools and current data filtering approaches.

2.1. Storage format

Metadata Standards. File system metadata is widely used in digital forensics to navigate file system information and reconstruct event timelines. Popular tools, such as The Sleuth Kit (Carrier, 2019a), FTK (Access Data, 2019), and EnCase (Guidance Software, 2018b) use body files to represent this metadata. Thus, our database forensic storage format was designed to include not only the records that could be accessed through a live system, but also the DBMS metadata, which users may not always have access to through the DBMS API.

Database Carving Tools. Several database carving tools exist, but they lack a unified output to store their results. These tools examine and reconstruct database forensic artifacts at the page level. Pages (typically 4 K or 8 K) are the minimum read/write unit for all row-store relational DBMSes. Page configuration is typically described in documentation by DBMS vendors (e.g., Oracle (Oracle Corporation), Microsoft SQL Server (Microsoft), IBM DB2 (IBM), PostgreSQL (Group), MySQL (MySQL), and SQLite (SQLite)). Drinkwater was one of the earliest to describe a database carving approach for SQLite DBMSes (Drinkwater). Guidance Software's SQLite Parser implements much of what Drinkwater discussed; they reconstruct both allocated and unallocated SQLite data (Guidance Software, 2018a). SQLite Parser returns the results in the form of a new SQLite instance (i.e., a single database file). Wagner et al. proposed a generalized method to learn and reconstruct DBMS storage through page carving (Wagner et al., 2015, 2017a). They proved this method

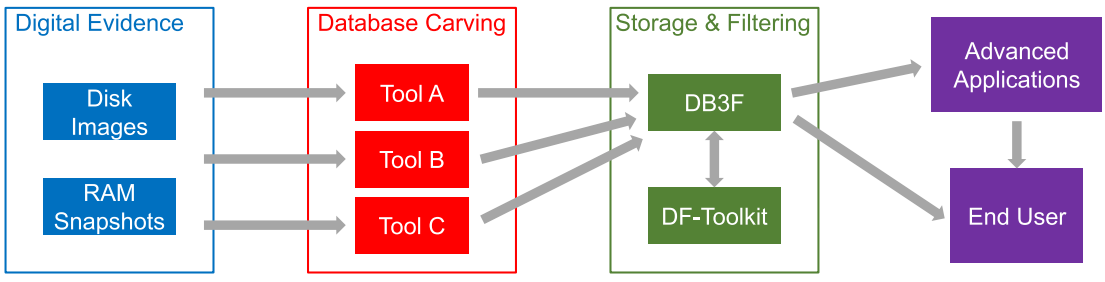

Fig. 1. The role of DB3F and DF-Toolkit in database forensics.

worked for most row-store relational DBMSes, including Apache Derby, Firebird, IBM DB2, Microsoft SQL Server, MySQL, Oracle, PostgreSQL, and SQLite. Their tool, DBCarver, returned much of the metadata along with the allocated and unallocated user records in a series of CSV files. Percona's Recovery Tool for InnoDB recovers MySQL DBMS files (Percona, 2018), but we do not consider it a tool for forensic investigations. Once MySQL files are recovered, they are imported into a live MySQL instance. Therefore, none of the unallocated data or metadata is presented to the user. One of the main goals in this paper is to define a unified storage format for the allocated data and unallocated data returned by the work of Drinkwater and Guidance Software, and the allocated data, unallocated data, and metadata returned by the work of Wagner et al. To evaluate DB3F and DF-Toolkit for this paper, we used our previously developed page carving tool, DBCarver (Wagner et al., 2017a). As DBCarver does not support DB3F output, we converted its output (CSV files) into DB3F.

Structured Forensic Languages. File names and file system properties are represented in formats such as JSON or XML with digital forensic tools. Some examples include Mandiant's Indicators of Compromise in Malware Forensics (Lock and Kliarsky), The MITRE Corporation's Making Security Measurable Project (Martin, 2008), and DFXML by Garfinkel et al. (Garfinkel, 2009, 2012). For this project we used JSON to represent database forensic artifacts. JSON can readily be migrated to XML if needed using most programming languages.

2.2. View and search model

Evidence Tree. Most forensic tools (e.g., FTK, The Sleuth Kit/Autopsy, and Encase) that offer an interface to traverse and view artifacts use a tree structure to present these forensic artifacts. Database forensic artifacts are inherently different from typical forensic objects; therefore, objects such as files cannot serve as tree nodes. For Example, a database table can span across multiple files (as in PostgreSQL) or a database file can contain multiple database tables and indexes (as in Oracle). In this paper, we present a new evidence tree that was inspired by existing tools, but designed to represent database forensic artifacts.

Filtering. SQL is a powerful tool that can enhance searching forensic artifacts. Instead of iterating over a series of files, forensic filtering tools can integrate SQL (i.e., relational) database capabilities. FTK (Access Data, 2019) and The Sleuth Kit (Carrier, 2019a) store case information in SQL databases, and we believe our framework should take the same approach. The main challenge with this, which we address in this paper, is that to properly use SQL, the data must be first stored in a properly defined relational schema. Some of the forensic SQLite tools (e.g., Guidance Software's SQLite Parser) return results as a SQLite DBMS file, which can be natively filtered using SQL. However, it does not include forensically relevant metadata defined in (Wagner et al., 2015), which we believe should be incorporated. Therefore, simply recreating the DBMS is insufficient as it provides only data and not metadata. The following examples illustrate this problem with two simple questions a database filtering framework should be capable of answering.

Example 1. *"Return all deleted records and their offsets".* A recreated DBMS does not store metadata that describes deletion status of a record or its offset within a disk image. To answer this query, at least two additional columns (deletion flag and position offset) must be added to every table reconstructed in the DBMS. It is immediately apparent that such a model is not extensible, as additional metadata columns will be needed to support answers for other types of forensic queries. Furthermore, by adding meta-columns, distinguishing the meta-columns from the original ("real") data columns could become a challenge for users.

Example 2. *"Find all records containing the string 'MaliciousText'".* This query poses even more challenges than the previous example. The user must search all columns across all tables. Such operation is not well-supported by SQL, as SQL language has no capability to apply a filter condition "for all columns". To illustrate this problem, assume we know there is just one table, Employee. The following query would have to be written for every table:

```
SELECT *
FROM Employee
WHERE FirstName LIKE '%MaliciousText%'
OR LastName LIKE '%MaliciousText%'
OR Department LIKE '%MaliciousText%'
OR JobTitle LIKE '%MaliciousText%';
```

We discuss our solution for this problem in Section 5.1.

3. Design requirements

The requirements identified for this work were based on the overall goals and challenges in digital forensics discussed by Garfinkel (2010) and the requirements defined by other digital forensic frameworks, including Autopsy (Carrier, 2019b), DFXML (Garfinkel, 2009, 2012), and FROST (Dykstra and Sherman, 2013). This section describes some of the key requirements we considered for the design DB3F and DF-Toolkit.

3.1. DB3F requirements

Storage Engine Agnostic. One of the major goals of DB3F is to abstract DBMS storage engine specifics. This abstraction must generalize to all row-store DBMSes and not lose any forensic artifacts. One Example of an artifact that may be interpreted differently depending on the specific DBMS is the storage of the DBMS-internal object identifier metadata. An object identifier is a unique identifier for each object in the DBMS; it maps back to a system table for the object's plaintext name (e.g., Employee). Most DBMSes store the object identifier in the page header. Alternatively, PostgreSQL stores the object identifier with each individual record (even though it is redundant, as a single database page can only contain data belonging to one object). The function of the object identifier remains the same despite where it stored. Therefore, DB3F should remove the need to know the specifics of how such metadata is stored.

Simple to Generate and Ingest. DB3F should be generated by all database carving tools and used by any searching or advanced analytic tools. Therefore, the DB3F should be easy to generate, and parsing data from DB3F should be trivial.

Open and Extensible. DB3F should be publicly available and open sourced. Fields should be easy to add to the public standard. Additionally, given the potentially wide variety of organizations and databases that may use DB3F, custom field addition should be allowed — new custom fields should be easy to introduce. For Example, the standard operating procedure for one organization may require chain of custody information that is currently not a field in the DB3F file header. In such cases, it should be easy for an organization to introduce this information into DB3F.

Scalable. The amount of database forensic artifacts that may be discovered and will require processing is unpredictable (and projected to continuously increase). An investigation may involve a small (KBs), lightweight DBMS from a personal device, or it may involve a large (PBs), data warehouse stored on multiple servers.

Moreover, an investigation may involve multiple computers (e.g., a network of IoT devices), each with their own individual DBMS. Therefore, the amount of carved data stored in DB3F should not impact the system capabilities.

3.2. DF-toolkit requirements

Visibility. Forensic tools return a wide variety of data and metadata to users. These artifacts should be organized and presented to users in a manner such that the data can be traversed. This is traditionally done using a representative tree structure where the root nodes are the critical data structures (e.g., disk partitions), the next level nodes are used to store the data objects (e.g., stand-alone files), and all other node levels are used to store information about the carved data objects.

Display Data Objects. Given that the user can view a logical organization of the forensic artifacts in an evidence tree, the user would most certainly want to view the data objects and their content. Such viewing should be allowed through a user interface.

Object Filtering. When a user is presented with a large number of data objects, she may desire to filter these to a relevant subset. For Example, a user may only be interested in JPEG files so a corresponding filtering condition (`filetype = 'JPEG'`) may be applied. In DBMSes, a user may want to filter objects based on the metadata, such as object type (e.g., table, index, materialized view), number of columns, or object size.

KeywordSearches. Keyword searches are commonly used in forensic investigations to find relevant evidence. String matches and regular expressions should be supported for filtering records (e.g., find all mentions of 'Bob').

Reporting. Reports need to be generated to help analysts make conclusions and present their findings. Furthermore, this reporting should allow for comparison and validation of database forensic carving tool output.

4. The Database Forensic File Format

This section presents our storage format for database forensics, DB3F. This is the format that should be used by different database carving tools to output their results.

4.1. File Layout

When a database carving tool is passed a file, a carver tool analyzes it for the presence of one or more different DBMSes. Since each DBMS is a self-contained system, data from different DBMSes should not be mixed within the same carver output file. Each DBMS is stored as a separate output file.

Multiple DBMSes may exist on the same piece of evidence. However, it is acceptable for multiple carver output files to be associated with a single DBMS. For Example, a series of DBMS files (from a single file system or multiple nodes) belonging to the same DBMS may be passed to the carver as opposed to a single disk image. Moreover, the RAM snapshot(s) will be a separate evidence file for any given DBMS. Therefore, this condition is required if one wants to compare the data from a disk image and a RAM snapshot.

Example 3. *File Layout.* `DiskImage01.img` is passed to a database carving tool. The carving tool analyzes the evidence for data belonging to PostgreSQL and SQLite DBMSes. This results in two output DB3F files (one for each DBMS): `PostgreSQL.json` and `SQLite.json`.

4.2. DB3F files

Each DB3F file stores a series of JSON objects. The first line in a DB3F files contains a JSON object that serves as a header. Every other line in the DB3F contains a JSON object that represents a database page.

Representing the entire carved DBMS with a single JSON object has scalability problems because the amount of data in a DBMS can be arbitrarily large. Therefore, one JSON object per DBMS page allows us to achieve the scalability requirement (see Section 3). The physical order of DBMS pages is irrelevant because each page object stores its offset within the disk image.

4.3. DB3F header

The DB3F file header JSON object contains high-level metadata about the artifacts in the file and how they were collected. The list below describes the fields, which should be returned by the database carving tool, stored in the header. Additionally, Fig. 2 displays a DB3F file header with Example data. Since we cannot anticipate all of the header information each specific organization may require, JSON fields can easily be added.

context (array): namespace information.

name (string): the organization name used to identify custom header information.

uri (string): unique identifier for an organization.

evidence_file (string): the disk image or RAM snapshot from where the forensic artifacts originated.

forensic_tool (string): the database carving tool used to generate the forensic artifacts.

carving_time (string): the system global time when the carver finished generating the DB3F file.

dbms (string): the DBMS vendor and its version.

page_size (number): the page size used by the DBMS. Page size is assumed to be constant across an entire DBMS. It is theoretically possible to use more than one page size in a DBMS. However, we assume the database carving process will extract different page sizes as belonging to different DBMSes.

4.4. DB3F database pages

Each line following the DB3F header contains a single JSON object that represent a database page. Each page stores 1) page header fields and 2) an array of JSON objects that represent records. Fig. 3 displays an Example of how DB3F represents a PostgreSQL DBMS page storing Star Schema Benchmark data (Neil et al., 2009). The fields in this figure are defined in this section.

Page Header. The page header stores information that is general to all records within the page. The page header fields are the following:

```
{
   "@context": {
      "name": "DePaul Database Group",
      "uri": "http://dbgroup.cdm.depaul.
          edu"
   },
   "evidence_file": "DiskImage01.img",
   "forensic_tool": "Anonymous Tool",
   "carving_time": "2019-01-19 22:45:32",
   "dbms": "PostgreSQL 8.4",
   "page_size": 8192
}
```

Fig. 2. An Example of DB3F header.

```
{
   "offset": 3743744,
   "page_id": "0",
   "object_id": "1113440",
   "page_type": "Table",
   "schema": ["Nbr","Str","Str","Str","Str","
       Str","Str"],
   "records": [
      {
         "offset": 382,
         "allocated": true,
         "row_id": "71"
         "values": [
            "71",
            "Supplier#000000071",
            "31CSQET",
            "ARGENTINA5",
            "ARGENTINA",
            "AMERICA",
            "11-710-812-5403"
         ]
      }, {
         "offset": 486,
         "allocated": true,
         "row_id": "70"
         "values": [
            "70",
            "Supplier#000000070",
            "jd4djZv0cc5KdnA0q9oO",
            "FRANCE    0",
            "FRANCE",
            "EUROPE",
            "16-828-107-2832"
         ]
      }, {
         "offset": 589,
         "allocated": true,
         "row_id": "69",
         "values": [
               "69",
               "Supplier#000000069",
               ...
         ]
      }, ...
   ]
}
```


Fig. 3. An Example DB3F page JSON object.

offset (number): the page address within the evidence.

page_id (string): a unique identifier the DBMS assigns to a page.

object_id(string): an identifier that the DBMS uses to map to the plaintext name of each object (e.g., table or index).

page_type (string): the type of object to which the page belongs.

schema (array): the data types for the record columns within the page.

records (array): a list of JSON objects for each record.

Record. A JSON object exists for each record in the page. The record fields are the following:

offset (number): the record address within the page.

allocated (boolean): True indicates the record is allocated, while False indicates the record is deleted (i.e., unallocated storage).

row_id (string): an internal DBMS pseudo-column.

values (array): the individual record values.

The fields defined in this section is not an exhaustive list. We anticipate that new fields will be added to the DB3F standard as the tool use grows and organizations will want to add their own custom fields.

Discussion: Datatype Support. While the Example data in Fig. 3 illustrates only strings and numbers, DB3F supports all DBMS datatypes. Each datatype is described in the page header schema field, and the value is stored among the values field for a record. Users may be concerned about storing values that do not fit into a single page, such as Binary Large Objects (BLOBs) and large text fields. To store BLOBs, DBMSes do not directly store the binary data within the page, but rather store a reference to a file containing the binary data. For example, a DBMS would store a reference to a JPEG file in a page rather than the binary JPEG data. DB3F would similarly store a reference to a file, with the actual binary file (e.g., JPEG) stored in a separate dedicated location. It is possible for a text value to span across more than one page. In this instance, each DB3F page object describes the text stored in an individual page, allowing the long text value to be reconstructed independently. Additionally, in some case text field will store pointers to the remainder of the text located in different pages. In such cases, DB3F will store whatever information is provided by the database page carving tool. Additional analysis is required to rebuild the entire text value – a DBMS pointer can be reconstructed using the metadata already stored in DB3F fields. The work in (Wagneret al., 2018) discusses DBMS pointer reconstruction in more detail.

4.5. Evaluation

To verify the reliability of DB3F, we used three DBMSes: PostgreSQL, Oracle, and SQLite. We loaded Star Schema Benchmark data (Neil et al., 2009) at Scale 1 (600 MB) into each DBMS, used DBCarver to carve the DBMS files, and converted the CSV file output into DB3F. We converted the artifacts carved from Oracle and PostgreSQL DBMS files into DB3F without any problems. However, since SQLite does not store an object identifier in the pages, this metadata could not be included in DB3F directly. As an alternative, we used the table schema (i.e., the string with column datatypes) to represent the object identifier. This decision was made because all records for the same table will have the same number of columns and the same datatype for each column. However, we note that more than one table can have the same schema; thus, our decision merged tables with identical columns in SQLite. Table 1 summarizes the sizes of the DBMS files passed to DBCarver and our generated DB3F files for the 600 MB Scale 1 SSBM data used. The DB3F storage overhead allows for human readability. However, DB3F can be compressed to scale for analysis of larger forensic datasets.

5. The Database Forensic Toolkit

This section presents our toolkit, DF-Toolkit, to view and filter DBMS forensic artifacts stored in DB3F. First, we describe the evidence tree structure that serves as a core concept behind this toolkit. This tree structure allows users to traverse and view metadata and data stored in DB3F files. Next, we discuss how this tree allows carved database metadata and data to be searched and filtered by the user. Finally, our solution to reporting filtered metadata and data in DB3F is described.

Throughout this section we refer to Figs. 4 and 5. As a proof of

Table 1
File size comparison of DB3F file to the DBMS file for a 600 MB CSV file of raw data.

DBMS	DBMS(MB)	DB3F(MB)
Oracle	625	1329
PostgreSQL	648	1298
SQLite	445	1308

concept, Fig. 4 displays our implemented user interface to display the evidence tree. Fig. 5 contains the relational schema used to store the evidence tree nodes in a SQL database for searching and filtering results. These tables are populated when a tree is first viewed; they can be cached or rebuilt by DF-Toolkit as necessary.

5.1. The evidence tree

The evidence tree presented in this section follows the same principles as many popular digital forensic tools (e.g., The Sleuth Kit, FTK, EnCase). Similar to these tools, we classify three main node levels in the tree: root, object, and object description. Alternatively in this paper, the tree nodes are defined to accurately represent database forensic artifacts.

Root. The root node serves as a critical storage structure from which all other data can be reached. For Example, a disk partition may be a root in commonly used forensic tools. Since DBMSes manage their own storage, a disk partition does not represent a major storage structure in a DBMS. For example, a DBMS may store files across multiple disk partitions. When this is done, system tables and user tables would likely be stored on different partitions. Furthermore, a single table could be stored on multiple disk partitions. Therefore, a DBMS sample (i.e., the complete or partial DBMS storage content taken from a storage medium) makes an appropriate storage structure for a root. A database root node is not expected to contain an entire DBMS. It is likely that the piece of evidence is a disk partition, RAM snapshot, or contains a corrupt (e.g., partially overwritten) DBMS. Therefore, by "DBMS sample", we mean all of the data associated with a DBMS for a given piece of evidence.

In Fig. 4, there are two images that represent evidence, Image01.img and Image02.img. Image01.img contains two root nodes (i.e., DBMS samples), PostgreSQL and MySQL. Since DB3F requires that a carver tool store DBMS samples in separate output files, each root node always corresponds to a single DB3F file.

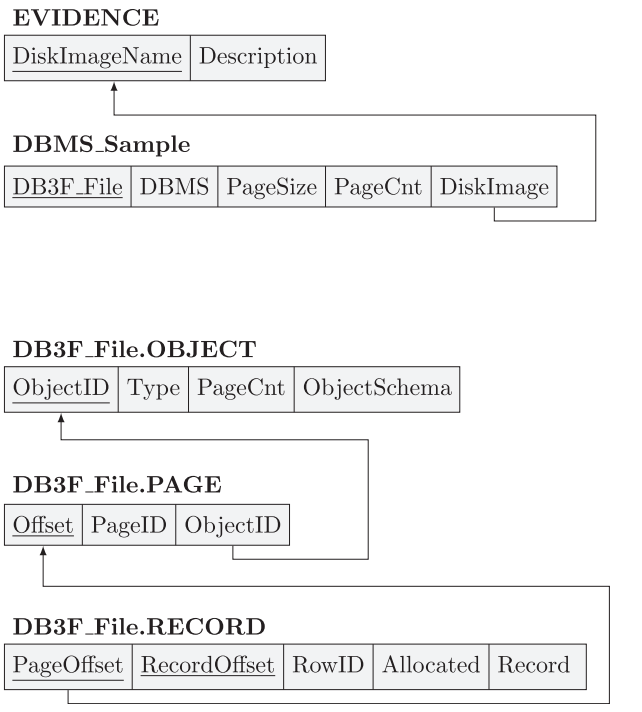

Fig. 5. The relational schema used to store the evidence tree data in a SQL database.

In Fig. 5, the DBMS_Sample table stores a record for each root node. DB3F_File is a reference to the DB3F file. This also serves as a

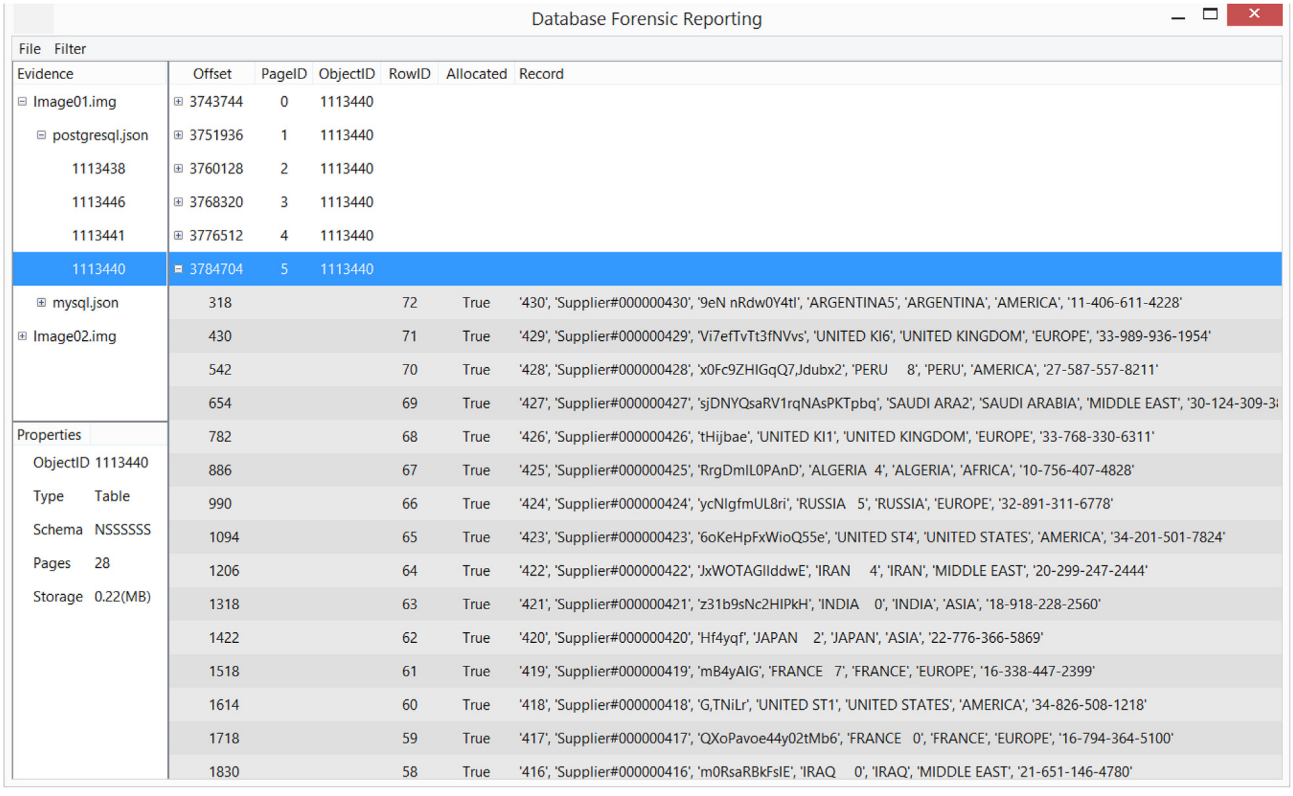

Fig. 4. DF-Toolkit evidence tree implemented with a user interface.

unique identifier (i.e., primary key) for each DBMS sample record. DBMS is the DBMS vendor name and version. PageSize is the page size used by the DBMS sample. PageCnt refers to the number of pages associated with the DBMS sample. Therefore, the total DBMS sample storage size can be calculated using PageSize × PageCnt. DiskImage is a reference to the evidence (e.g., disk image, RAM snapshot) associated with this DBMS sample. This column also references (i.e., a foreign key) the Evidence table. For every entry in the DBMS_Sample table, a new schema is created containing an Object table, Page table, and Record table.

Data Objects. The next level in the tree are the data objects for which the root is examined. For Example, a stand-alone file (e.g., PDF, Word document) may be a data object in commonly used forensic tools. DBMS files can contain multiple DBMS objects (e.g., tables), and a DBMS object can span across multiple DBMS files. Artifacts belonging to each DBMS object should be associated with each other. Therefore, DBMS files themselves should not be treated as the data objects like traditional stand-alone files. A more suitable candidate for the data object node are the DBMS objects (e.g., customer table, employee table, customer name index). DBMS metadata and data can be associated with DBMS objects by using the object identifier metadata stored within DBMS pages (discussed in Section 3.1). Additionally, viewing the DBMS files themselves does not provide the user with much useful information since they are not stand-alone.

In Fig. 4, the PostgreSQL root node has four data objects: 1113438, 1113446, 1113441, and 1113440. Statistics and metadata describing the selected object, 1113440, is displayed in the bottom left-hand box. This object is a table with 28 pages (not all displayed) and seven columns (one number and six strings), beginning under the heading, “Record”.

In Fig. 5, the Objects table stores information about each object. ObjectID is the object identifier used by the DBMS, which also serves as the primary key. Type represents the type of DBMS object (e.g., table, index, or materialized view). PageCnt stores the number of pages associated with the object. ObjectSchema represents the data types for each column in the table.

Object Information. Two more tree levels are used to recursively store information about each object at the page level and the record level. Storing information about each DBMS page allows for statistics to be quickly collected for an object (or a fast stochastic analysis), and removes data redundancy at the record level.

In Fig. 4, the pages associated with the selected object, 1113440, are displayed in the right-hand side box. We know there are a total of 28 pages, which are not all displayed in the figure, based on the object information in the bottom left-hand box.

In Fig. 5, the Page table stores information about each page. Offset refers to the byte address of the page with the evidence file. This also serves as the primary key. PageID is metadata used by the DBMS to uniquely identify pages. Note, that we do not use this as the primary key because multiple copies of a page may exist (e.g., one from the DBMS file and one from a paging file on the same disk image). ObjectID is metadata used by the DBMS to identify objects, and this column also references the Object table.

Information about each record within a page is the last node level in our evidence tree. In Fig. 4, the records associated with the selected page, offset = 3784704, are displayed in the right-hand side box. In Fig. 5, the Record table stores information about each records. PageOffset refers to the byte address of the page within the evidence file. This column also references the Page table. RecordOffset refers to the byte address of a record within the page. PageOffset and RecordOffset together serve as the primary key. RowID is metadata, which is a DBMS internal pseudo-column. Allocated identifies the record as ‘active’ or ‘deleted’ (i.e., unallocated). Record is a string that combines all record values. Each value within the string has single quotes around it, and all values are separated by a comma.

We stop recursively constructing the tree at the record level. That is, the leaf level of the evidence tree is a database record (e.g., a single row in Fig. 4) rather than a field (e.g., ‘ARGENTINA’ in Fig. 4). Logically, another tree level could be added for individual values. For our current version of DF-Toolkit, this step is not needed for plaintext searches. We believe that extending the evidence tree to include individual fields of the database table should be explored in the future to support more advanced analysis; however, the proper execution of such a feature will introduce significant implementation challenges. Continuing to represent data with a proper relational schema (as in Fig. 5) does not scale well when individual values are considered because each value must now be stored as an entry in the Value table – for Example, representing the first row from Fig. 4 at individual value level as shown in Table 2. Therefore, to search for an individual value, an entry from the Value table would need to `JOIN` with the Record table.

Another possible approach would be to create a new table for each DBMS object from each DBMS. The data would be ingested from a CSV file generated from the DB3F file. This approach would be similar to Guidance Software's SQLite Parser (discussed in Section 2). While we envision this to be a more viable solution, an incomplete DBMS from evidence such as a RAM snapshot or corrupt DBMS poses an implementation challenge; table columns would be ambiguously defined creating issues when querying data. For Example, column names would need be created as Column1, Column2, etc. We do not consider the presence of a complete DBMS to be a safe assumption for DF-Toolkit purposes.

5.2. Data display filters

Data filtering is performed at the DBMS level; tables (or objects) for each DBMS schema are considered. The following is the basic query need to properly connect a DBMS schema before applying filtering conditions, where DB3F_File is the root node:

```
SELECT *
FROM DB3F_File.Object O,
DB3F_File.Page P,
DB3F_File.Record R
WHERE O.ObjectID = P.ObjectID
AND P.Offset = R.PageOffset
```

This query returns all rows from the `Objects`, `Page`, and `Record` tables for a given DBMS so that the data can be put back into DB3F (this is further explained in Section 5.3). Beyond this query, only a basic understanding of SQL is needed to perform custom filtering.

Objects. Users can filter objects by simply adding `WHERE` clause conditions to the query above. Objects can be filtered based on the following metadata fields: ObjectID, Object Type, Object Page Count, and Object Schema. For Example, if the user was only concerned with the object with seven columns (one number and six

Table 2
Sample representation of carved rows on per-value basis.

Offset	RowID	Alloc.	Pos.	Value
318	72	True	1	‘430’
318	72	True	2	‘Supplier#000000430’
318	72	True	3	‘9eN nRdw0Y4tl’
318	72	True	4	‘ARGENTINA5’
318	72	True	5	‘ARGENTINA’
318	72	True	6	‘AMERICA’
318	72	True	7	‘11-406-611-4228’

strings), the following condition would be added:

```
AND O.Schema = 'NSSSSSS'
```

Pages. Users can also filter pages with `WHERE` clause conditions. Pages can be filtered based on the following metadata fields: Page Offset, PageID, and Page ObjectID.

Records. Finally, users can filter record with `WHERE` clause conditions. Records can be filtered based on the following metdata fields: Record PageOffset, Record Offset, Record RowID, Record Allocated/Deallocated, and the data stored in the record. Most importantly, users would want to apply keyword searches to the data stored in records. All of the values for a carved record are stored as a single string making this feature easy to support. Since SQL supports string matches, wildcards, and regular expressions, keyword searches can be applied by adding another `WHERE` clause condition(s). For Example, to search for all records containing a phone number (in the format of the data from Fig. 4):

```
AND R.Record REGEXP '\d{2}-\d{3}-\d{3}-\d{4}'
```

Fig. 6 displays an Example interface to apply filtering within our user interface. The `JOIN` conditions are previously written, simplifying user interaction. The user then adds the two example conditions presented for object filtering and keyword searches.

5.3. Report generation

After filtering is applied, the results are returned as DB3F. Storing the report back into DB3F allows the data to be viewed within the evidence tree, available for further filtering, and to future advanced analytic tools. We note that DF-Toolkit was able to find every relevant carved artifact in its search (providing a search accuracy of 100%). Report accuracy is thus dependent only on the accuracy of carving provided by the database carving tool(s).

6. Conclusion and future work

This paper presented a new storage format for database forensic artifacts called the Database Forensic File Format (DB3F), and a toolkit to view and search data stored in DB3F called the Database Forensic Toolkit (DF-Toolkit). Additionally, a user interface was presented to provide a display of DF-Toolkit. To adhere to the DFRWS double-blinded peer review process, links to the implementation were not included. Both DB3F and DF-Toolkit will be made publicly available through our research group website with the camera-ready version of this paper.

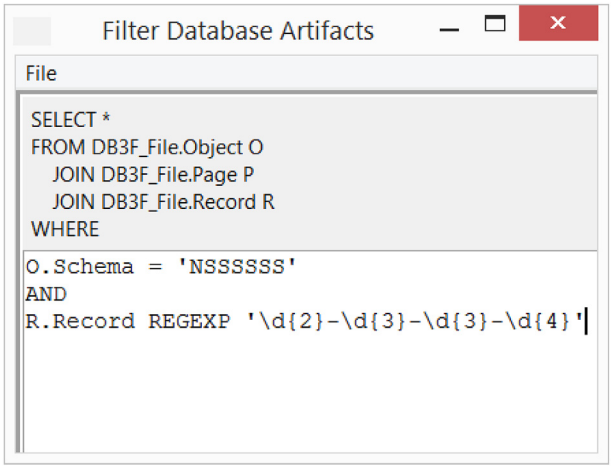

Fig. 6. DF-Toolkit filtering implemented with a user interface.

We envision that DB3F and DF-Toolkit will serve as the groundwork for a complete forensic and security analysis system. Future work for this system is discussed below, which includes: incorporating DBMS system data carved from the evidence, carver tool integration, multi-evidence analysis, and non-page data integration.

6.1. System catalog information

While the metadata presented to users through DF-Toolkit is accurate, some DBMS forensic artifacts may become difficult to interpret for users, especially as the amount of data increases. For Example, the object identifiers (e.g., '1113440') alone do not mean as much as the plaintext object name (e.g., 'Supplier') to an investigator exploring evidence. Our top priority for future work is to automate the detection and association of DBMS system catalog information, which is stored in DBMS system tables, to replace such metadata with more readable plaintext. We do see two main challenges with this work. First, the system catalog may not always be present (e.g., corruption of data on disk or when using a RAM snapshot). Therefore, DF-Toolkit would need to accurately communicate to a forensic analyst why such metadata is not available. Second, each DBMS has its own system table schema. Therefore, detection and association of this information requires tailored functions for each DBMS vendor.

6.2. Carver tool integration

For this paper, we generated DB3F files from carved output stored in CSV files. This step would be tedious for users, and we believe it should be streamlined. Ideally, we would like to work with the current and future creators of database carving tools (Section 2) to return their results in DB3F. Making DB3F publicly available will help to catalyze this effort.

6.3. Advanced analysis

This paper presented straightforward filter and search examples for single pieces of evidence. However, we envision a more complete toolkit to access and interpret database forensic artifacts. This mostly comes in the form of a database forensic API, which would be a DBMS complement to Garfinkel's Fiwalk (Garfinkel, 2009). The primary uses for such work include multi-evidence analysis and integration with non-DBMS page data and other forensic tools.

Multi-Evidence. An investigation may involves multiple pieces of evidence when a series of disk images or RAM snapshots was collected, a DBMS was distributed across multiple nodes, or multiple devices contained individual DBMSes. In these cases, metadata and data can be compared to recreate event timelines. Most IoT devices typically store information locally on a lightweight DBMS (e.g., SQLite), send information to a server that uses a more robust DBMS (e.g., MySQL), or both. For Example, the Amazon Alexa and Samsung Galaxy images from the DFRWS IoT Challenge 2018–2019 (DFRWS, 2018) each contain a SQLite DBMS. Assuming that these devices had some form of interaction, connecting data and metadata from both devices would help to create an event timeline.

Integration of Non-DBMS Page Data. Almost all of the DBMS data and metadata is stored in pages; thus, it can be represented in DB3F and searched with DF-Toolkit. However, connecting metadata and data outside of DBMSes to DB3F files would create more complete timelines. These sources include audit logs, network packets, and

files which are referenced by DBMS records. Section 2 discussed just some of the tools used to store and searched these data and metadata. We hope that bringing this discussion to the DFRWS community will help bridge the gap between these different domains within digital forensics.

7. Disclosure statement

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

References

Access Data, 2019. Forensic Toolkit. https://accessdata.com/products-services/forensic-toolkit-ftk.
Carrier, B., 2019. The Sleuth Kit. https://www.sleuthkit.org/sleuthkit/.
Carrier, B., 2019. Autospy. https://www.sleuthkit.org/autopsy/.
DFRWS, 2018. Dfrws Iot Challenge 2018 - 2019. https://github.com/dfrws/dfrws2018-challenge/.
Drinkwater, R.. Carving SQLite Databases from Unallocated Clusters. http://forensicsfromthesausagefactory.blogspot.com/2011/04/carving-sqlite-databases-from.html.
Dykstra, J., Sherman, A.T., 2013. Design and implementation of frost: digital forensic tools for the openstack cloud computing platform. Digit. Invest. 10, S87–S95.
Garfinkel, S.L., 2009. Automating disk forensic processing with sleuthkit, xml and python. In: Systematic Approaches to Digital Forensic Engineering, 2009. SADFE'09. Fourth International IEEE Workshop on, IEEE, pp. 73–84.
Garfinkel, S.L., 2010. Digital forensics research: the next 10 years. Digit. Invest. 7, S64–S73.
Garfinkel, S., 2012. Digital forensics xml and the dfxml toolset. Digit. Invest. 8 (3), 161–174.
Garfinkel, S., Farrell, P., Roussev, V., Dinolt, G., 2009. Bringing science to digital forensics with standardized forensic corpora. Digit. Invest. 6, S2–S11.
Group, T.P.G.D.. https://www.postgresql.org/docs/8.0/storage-page-layout.html.
Guidance Software, 2018. Sqlite Free-Page Parser. https://www.guidancesoftware.com/app/sqlite-free-page-parser.
Guidance Software, 2018. Encase Forensic. https://www.guidancesoftware.com/products/efindex.asp.
IBM. https://www.ibm.com/developerworks/data/library/techarticle/0212wieser/index.html.
Lattner, C., Adve, V., 2004. Llvm: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization. IEEE Computer Society, p. 75.
H.-Y. Lock, A. Kliarsky, Using Ioc (Indicators of Compromise) in Malware Forensics, SANs Institute.
Martin, R.A., 2008. Making security measurable and manageable. In: Military Communications Conference. MILCOM 2008. IEEE, IEEE, 2008, pp. 1–9.
Microsoft. https://docs.microsoft.com/en-us/sql/relational-databases/pages-and-extents-architecture-guide?view=sql-server-2017.
Neil, P.O., Neil, E.O., Chen, X., Revilak, S., 2009. The star schema benchmark and augmented fact table indexing. In: Performance Evaluation and Benchmarking. Springer, pp. 237–252.
Oracle Corporation. https://docs.oracle.com/database/121/CNCPT/logical.htm#CNCPT004.
Oracle Corporation. https://dev.mysql.com/doc/internals/en/innodb-page-structure.html.
Percona, 2018. Percona Data Recovery Tool for Innodb. https://launchpad.net/percona-data-recovery-tool-for-innodb.
SQLite. https://www.sqlite.org/fileformat.html.
Wagner, J., Rasin, A., Grier, J., 2015. Database forensic analysis through internal structure carving. In: DFRWS.
Wagner, J., Rasin, A., Grier, J., 2016. Database image content explorer: carving data that does not officially exist. In: DFRWS.
Wagner, J., Rasin, A., Malik, T., Hart, K., Jehle, H., Grier, J., 2017. Database forensic analysis with dbcarver. In: CIDR.
Wagner, J., Rasin, A., Glavic, B., Heart, K., Furst, J., Bressan, L., Grier, J., 2017. Carving database storage to detect and trace security breaches. Digit. Invest. 22, S127–S136.
Wagner, J., et al., 2018. In: Detecting database file tampering through page carving. EDBT.