2018 IEEE 34th International Conference on Data Engineering

Auditing DBMSes through Forensic Analysis

James Wagner
Supervised by Dr. Alexander Rasin
Expected graduation date: June 2019
School of Computing, DePaul University
Chicago, IL 60604, U.S.A.
jwagne32@depaul.edu

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

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

I. Introduction

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

DBMSes are targeted by criminals because they serve as repositories of data. Therefore, investigators must have the capacity to examine and forensically interpret contents of a DBMS. Currently, an audit log with SQL query history is a critical (and perhaps only) source of evidence for investigators [1] when a malicious operation is suspected. In field conditions, a DBMS may not provide the necessary logging granularity (unavailable or disabled). Moreover, the storage itself might be corrupt or contain multiple DBMSes.

Digital forensics provides an independent analysis with minimal assumptions about the environment. A particularly important and well-recognized technique is file carving [2], [3], which extracts files (but not DBMS files) from a disk image, including deleted or corrupted files. Traditional file carving techniques interpret files (e.g., JPEG, PDF) individually and rely on file headers. DBMS files, on the other hand, do not maintain a file header and are never independent (e.g., table contents are stored separate from table name and logical structure information). Even if DBMS files could be carved, they cannot be meaningfully imported into a different DBMS and must be parsed to retrieve their content. Therefore, DBMSes need their own set of digital forensics rules and tools.

Even an environment with ideal log settings, DBMSes can not necessarily guarantee log accuracy or their immunity from tampering. For example, log tampering is a concern when a breach originated from a privileged user such as an administrator (DBA or an attacker who obtained DBA privileges). Tamper-proof logging mechanisms were proposed in related work [4], [5], but these only prevent logs from atypical modifications and do not account for attacks that skirt logging (e.g., logging was disabled). Knowing that even privileged users have almost no control of the lowest level storage, an analysis of forensic artifacts provides a unique approach to identify data tampering in an untrusted environment.

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

Sec	Summary
II	Our DB forensics methods approached from the page level: • We describe our forensic method, page carving [6], [7]. **Future work will address column-store and NoSQL DBMSes.** • We present our page carving implementation: `DBCarver` [8]. • **We will build a meta-querying system to answer multi-DBMS forensic questions with `DBCarver` output.** • **Extensions in DB anti-forensics to protect against data theft.**
III	Our DB forensic solutions to attacks in untrusted environments. • We addressed the scenario where logging has been disabled by the DBA. We present our solution, `DBDetective`. [9]. • We addressed DBMS file tampering without SQL by a sys admin. We present our solution, `DBStorageAuditor` [10]. • **Future work to address tampering/backdating logs in a DBMS.** • **Future work will quantify the accuracy of our attack reports. A reproducible analysis will support our evidence.**
IV	Our uses of DB forensics that go beyond digital investigations, and focus on optimization of database storage layout: physical location index (PLI) [11], **external page building**, and **query reordering**.

TABLE I
SUMMARY OF THE REMAINING PAPER.

II. Database Forensics

Unlike traditional files (e.g., PDF), DBMS files do not contain headers that allow for file identification. At the same time, all row-store DBMSes use fixed-size pages to store user data, auxiliary data (e.g., indexes and materialized views), and the system catalog. DBMS data is accessed and cached in a unit of pages. Pages maintain a consistent structure,

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00219

Parameter	Oracle	PostgreSQL	SQLite	Firebird	DB2	SQLServer	MySQL	ApacheDerby
Row Identifier	No	Yes		No			Yes	
Column Count	Yes			No		Yes	No	Yes
Column Sizes	Yes			No			Yes	
Column Directory	No			Yes			No	
Numbers Stored w/ Strings	Yes			No			Yes	

TABLE II
A SUMMARY OF SIGNIFICANT TRADE-OFFS MADE BY DBMSES IN PAGE LAYOUT.

whereas individual record structure varies throughout DBMS storage, which is why we approach database forensics at the page level. In this section, we describe page carving including our implementation (DBCarver), future work to answer forensic questions from DBCarver output, and anti-forensics techniques that can sanitize and hide data in DBMS storage.

A. Page Carving

Database page carving is a method we previously introduced for the reconstruction of relational DBMSes without relying on file system or the DBMS. Page carving is similar to traditional file carving [2], [3] in that data, including deleted data, can be reconstructed from images or RAM snapshots without the use of a live system. Forensic tools, such as Sleuth Kit [12] and EnCASE Forensic [13], are commonly used by investigators to reconstruct file system data but are incapable of parsing DBMS files. None of the third party recovery tools (e.g., [14], [15]) are helpful for independent audit purposes because they only recover "active" data from current tables. A database forensic tool (just like a forensic file system tool) should also reconstruct unallocated pieces of data including deleted rows, auxiliary structures (indexes, MVs), or buffer cache space.

While each DBMS uses its own page layout, a great deal of overlap between page layouts allowed us to generalize storage for many row-store DBMSes. In [6] we presented a comparative page structure study for IBM DB2, Oracle, Microsoft SQL Server, PostgreSQL, MySQL, SQLite, Firebird, and Apache Derby. In this work, we also described a set of parameters that define the layout of a page for the purpose of reconstruction.

Table II demonstrates just a few example characteristics shared between DBMS pages. The row identifier is an internal DBMS pseudo-column which is sometimes explicitly stored in rows. If a DBMS stores the sizes of (string) columns, then numbers and strings are kept together (and column directory is not used). Alternatively, if a DBMS does not store the column sizes, then it maintains pointers to all string columns, and stores numbers separately from strings in each record.

a) Deleted Data: When data is deleted, the DBMS initially marks it as deleted, rather than explicitly overwriting it. This data becomes unallocated (free listed) storage – our work in [7] described the expected lifetime of forensic evidence within database storage following deletion and defragmentation. We described three categories of deleted data: records, pages, and values. A record is the minimum deletion unit and can be attributed to a DELETE, the old version of an UPDATE, or failed (aborted) transactions. A deleted record can be identified by its delete marking during page reconstruction. Dropped or rebuilt objects can create deleted pages, which are identified by carving system catalog tables. Deleted values are found in auxiliary objects – e.g., indexes; they are identified by mapping pointers back to records (only records but *not* index values are deleted). We presented generalized pointer deconstruction and pointer-record mapping in [10].

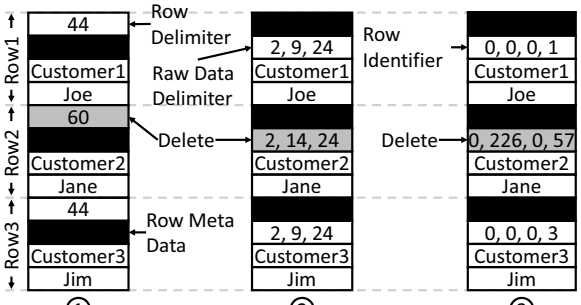

Fig. 1. Deleted row examples: 1-MySQL/Oracle, 2-PostgreSQL and 3-SQLite

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

b) Column-Store and NoSQL DBMSes: Page carving only supports row-store DBMSes. Column-store and NoSQL DBMSes do not typically use pages similar to row-store DBMSes. Future work will expand our database forensic methods to support column-store and NoSQL DBMSes.

B. DBCarver

We previously presented our implementation of page carving called DBCarver [8]. Figure 2 provides an overview of DBCarver architecture, which consists of two main components: the parameter collector (A) and the carver (F).

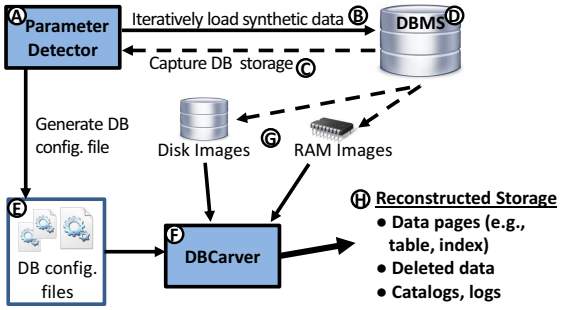

Fig. 2. DBCarver architecture.

The parameter detector loads synthetic data into a DBMS (B), captures storage (C), finds pages in storage, and captures

page layout parameters in a configuration file (E) – a text file that describes page-level layout for that particular DBMS. The parameters we store include those described in [6], and have since been expanded to extract additional metadata. `DBCarver` automatically generates parameters values for new DBMSes, or new DBMS versions. While most DBMSes retain the same page layout across versions, we observed different parameter values for PostgreSQL 7.3 and 8.4.

The carver (F) uses the the parameter values from the configuration files to reconstruct any database content from disk images, RAM snapshots, or any other input file (G). The carver returns storage artifacts (H), such as user records, metadata describing user data, deleted data, and system catalogs.

C. Meta-Querying

After storage artifacts were extracted by `DBCarver`, they must be analyzed to determine their evidential significance. By connecting reconstructed metadata and data, investigators can ask questions such as "Return all deleted records within a file." No such command is supported by any DBMS. More complex questions can be answered by combining both disk and memory data. As future work, we are currently interviewing and collaborating with law enforcement agencies to build a querying system that can answer real-world forensic questions.

In [8] we offered a preliminary view of this system. We presented several scenarios which an investigator may wish to explore. We termed them "meta-queries" because such queries are not executed on the original active DBMS but rather on reconstructed DBMS internals obtained through page carving.

a) Scenario 1: Reconstruction of Deleted Data: An analyst wants to query the carved database storage for deleted values. Deleted row identification is of particular interest when the audit log is missing or altered. For example, the following logged query obfuscates what records were actually deleted:

```
DELETE FROM Customer
WHERE Name LIKE NameFunction()
```

With database carving analysis, deleted records can be trivially identified with the following query:

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

We note that determining whether extracted rows were deleted due to normal or malicious operations requires incorporating audit logs and other evidence sources.

b) Scenario 2: Detecting Updated Data: An investigator wants to find the most recent updates. For example, consider the problem of searching for all recent product price changes in RAM. In order to form this query, we join disk and memory storage, returning the rows for which price is different:

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

D. Anti-Forensics

Anti-forensics (AF) is the field of interfering with forensic techniques [16], [17]. We note that digital forensic tools can be used by either investigators and criminals, to both protect data and to interfere with a criminal investigation. In this section, we discuss future work that uses AF to protect data.

Two of the most representative AF techniques we consider are data wiping and steganography [18]. A corporation can apply data wiping to erase already-deleted customer information to prevent potential data theft. Steganography is the process of hiding data – e.g., a means to discretely whistle-blow. Most prior work in database AF is highly DBMS-specific. Stahlberg erased deleted MySQL data by modifying the purge thread in source code [19]. We propose a more generalized sanitization method for all DBMSes (including closed-source DBMSes). We distinguish four categories of deleted DBMS data to wipe in order to prevent unintended data exposure: records, auxiliary data (e.g., indexes), system catalog, and unallocated pages. To effectively erase this data, the data itself must be overwritten and page metadata (e.g., checksums and pointers) must be updated accordingly. We further propose steganography that additively alters the database state, which can bypass all constraints and logging.

a) Steganography Example: We added the record shown in Figure 3 to a PostgreSQL file containing the `LINEORDER` [20] table. The composite primary key is underlined (solid line), the foreign keys are underlined (dashed line), and values that bypassed a constraint are highlighted.

Orderkey	Linenumber	Custkey	Partkey	Suppkey	Orderdate	Orderpriority	Shippriority	Quantity	Extendedprice	Ordtotalprice	Discount	Revenue	Supplycost	Tax	Commitdate	Shipmode
NULL	NULL	-1	-1	-1	-1	LOW	0	1	1	1	1	1	1	0	1800	Hello_World

Fig. 3. The hidden record that was added to the `LINEORDER` file.

The `LO_Shipmode` column was declared as a string of up to 10 characters (`VARCHAR(10)`). Since 'Hello_World' is 11 characters, it violates the domain constraint. Such hidden messages can be easily retrieved by returning only the values that violate domain constraints (which is normally impossible). For example, to return only our message:

```
SELECT LO_Shipmode FROM Lineorder
WHERE LENGTH(LO_Shipmode) > 10;
```

All SSBM queries [20] perform joins using the foreign key columns in `LINEORDER`. For example, Query 1.1 joins `LINEORDER` and `DATE` using `LO_Orderdate`.Our hidden record stores -1 for `LO_Orderdate` which does not match any `D_Datekey` column value. Our record bypassed referential integrity (normally impossible), and will never be returned by a query that performs a join on `DATE`. Similarly, our record bypassed referential integrity for `LO_Custkey`, `LO_Partkey`, and `LO_Suppkey` with -1. None of the SSBM queries return our hidden record because they perform at least one join, thus hiding the message from accidental discovery.

By default, all DBMSes create an index on the primary key column(s). Using (`NULL, NULL`) in the key value (again, normally impossible), we omit the hidden record from the primary key index and make unintentionally retrieval less likely. As with other indexes, rebuild of the primary key index will exclude `NULL` values, remaining blind to the record.

III. Database Security

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

A. DBDetective

Audit logs are a critical piece of evidence for investigators – and existing research has explored tamper-proof logs. However, DBAs are able to disable logging for legitimate operations (e.g., bulk loads). Therefore, we consider an attack where logging was disabled, malicious activity was performed, and logging was re-enabled. We proposed `DBDetective` in our previous work [9] to detect activity missing from the logs.

Peha [4] and Snodgrass [5] used one-way hash functions to verify and validate audit logs. Our approach detects both log tampering and cases when logging was temporarily disabled.

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

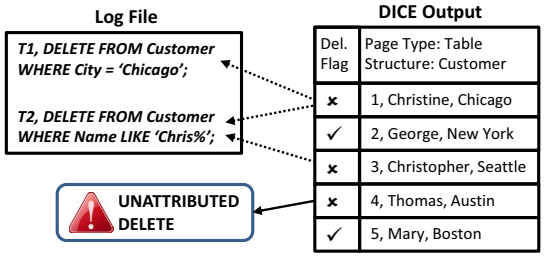

Fig. 4. Detecting unattributed deleted records.

Figure 4 is an example of unaccounted, deleted row detection. `DBCarver` reconstructed 3 deleted rows from *Customer*: *(1,Christine,Chicago)*, *(3,Christopher,Seattle)*, and *(4, Thomas,Austin)*. The log file contains two operations: `DELETE FROM Customer WHERE City = 'Chicago'` (*T1*) & `DELETE FROM Customer WHERE Name LIKE 'Chris%'` (*T2*). After comparing the deleted records to the log file operations, `DBDetective` returned *(4,Thomas,Austin)*, indicating a deleted record that could not be attributed to logged deletes. Here, we cannot conclude whether *T1* or *T2* caused the deletion of *(1,Christine,Chicago)*, which is not necessary to identify record #4 as an unattributed delete.

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

B. DBStorageAuditor

Privileged OS users commonly have access to database files. Consider a SysAdmin who, acting as the root, maliciously edits a DBMS data file in a Hex editor or through Python. The DBMS is unaware of external file write activity taking place outside its own programmatic access and thus cannot log it. Such an attack is a 'black-hat' application of anti-forensics discussed in Section II-D. We proposed `DBStorageAuditor` in our previous work [10] to detect database file tampering.

One-way hash functions were used in [21], [22] to detect file tampering at the file system level. However, we expect DBMS files to be regularly modified by legitimate operations. Thus, the challenge of distinguishing a malicious tampering operation and a legitimate SQL operation remains.

To detect database file tampering, `DBStorageAuditor` [10] uses indexes to verify the integrity of table data. We first verify the integrity of the indexes by checking for tampering-based inconsistencies within the B-Tree structure. Once the index integrity is verified, we deconstruct the index pointers and match them to table records using the table page metadata; we generalized the deconstruction of index pointers for all major DBMSes. We organize the index pointers based on physical location to keep our matching approach scalable. Finally, any extraneous data or erased data found from the index and table comparison is flagged as suspicious.

C. LogEventAnalysis

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

In such an environment, any global or logical clock can not be assumed to be reliable. Therefore, to create a timeline of events, we believe it is necessary to use storage metadata, which even a privileged user cannot modify. The internal RowID pseudo-column is of particular interest to construct a timeline. RowID is used by indexes and reflects the physical location of a record including its PageID. Whenever a page is modified, we can store the PageID to know when data was modified. Thus, the order of the PageIDs must be consistent with the order of the log files commands. We are exploring tamper-proof techniques to store the PageID. One method involves storing the PageID of the previously modified record in a page using steganography (Section II-D). Another method

involves storing the PageID and log commands offsite to later compare to storage snapshots for inconsistencies.

D. Quantitative Analysis and Reproducibility

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

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

IV. Database Storage Optimization

`DBCarver` allows us to analyze storage beyond what a DBMS exposes: physical ordering of data, object fragmentation, and specific data in the buffer cache. In this section, we present database forensics uses to optimize storage.

a) PLI: RDBMSes only support one clustered index per table. DBMS applications, that continually ingest large amounts of data, incur a very high overhead ensuring that the clustered index ordering is maintained. In [11], we showed that clustering slowdown can often be avoided if we use `DBCarver` to expose the physical location of attributes that are approximately clustered. Toward this, we proposed PLI, a physical location index, constructed by determining the physical ordering of an attribute and creating approximately sorted buckets that map physical ordering to attribute values in a live database.

b) External Page Building: `DBCarver` creates parameters for the purpose of deconstructing DBMS storage. At the same time, our future work uses these same parameters to construct DBMS files externally. Once the DBMS files are constructed, we believe they can be appended to a database instance with minor changes to system and file meta data.

c) Query Reordering: While most databases are good at managing memory, randomly ordered queries typically result in non-optimal buffer cache utilization. We explored caching patterns and DBMS memory management for the purpose of security (Section III-A). Our future work leverages this same information to create a method that analyzes RAM and reorders queries to achieve the most efficient I/O.

V. Conclusion

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

Acknowledgment

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

References

[1] R. T. Mercuri, "On auditing audit trails," Communications of the ACM, vol. 46, no. 1, pp. 17–20, 2003.

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

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

[4] J. M. Peha, "Electronic commerce with verifiable audit trails," in Proceedings of ISOC. Citeseer, 1999.

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

[6] J. Wagner, A. Rasin, and J. Grier, "Database forensic analysis through internal structure carving," Digital Investigation, vol. 14, pp. S106–S115, 2015.

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

[8] J. Wagner, A. Rasin, T. Malik, K. Hart, H. Jehle, and J. Grier, "Database forensic analysis with dbcarver." in CIDR, 2017.

[9] J. Wagner et al., "Carving database storage to detect and trace security breaches," Digital Investigation, vol. 22, pp. S127–S136, 2017.

[10] J. Wagner et al., "Detecting database file tampering through page carving," 21st International Conference on Extending Database Technology, p. to appear, 2018.

[11] J. Wagner, A. Rasin, D. H. T. That, and T. Malik, "Pli: Augmenting live databases with custom clustered indexes," in Proceedings of the 29th International Conference on Scientific and Statistical Database Management. ACM, 2017, p. 36.

[12] B. Carrier, "The sleuth kit," TSK). http://www. sleuthkit. org/sleuthkit/.[Online, 2011.

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

[14] OfficeRecovery, "Recovery for mysql," http://www.officerecovery.com/.

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

[16] C. S. J. Peron and M. Legary, "Digital anti-forensics: Emerging trends in data transformation techniques," 09 2017.

[17] R. Harris, "Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem," digital investigation, vol. 3, pp. 44–49, 2006.

[18] S. Garfinkel, "Anti-forensics: Techniques, detection and countermeasures," in 2nd International Conference on i-Warfare and Security, vol. 20087. Citeseer, 2007, pp. 77–84.

[19] 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, ACM. Citeseer, 2007, pp. 91–102.

[20] 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. Springer, 2009, pp. 237–252.

[21] G. H. Kim and E. H. Spafford, "The design and implementation of tripwire: A file system integrity checker," in Proceedings of the 2nd ACM Conference on Computer and Communications Security. ACM, 1994, pp. 18–29.

[22] M. T. Goodrich, M. J. Atallah, and R. Tamassia, "Indexing information for data forensics," in ACNS, vol. 5, Springer. Citeseer, 2005, pp. 206–221.