Forensic Science International: Digital Investigation 45 (2023) 301567

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS 2023 USA - Proceedings of the Twenty Third Annual DFRWS Conference

Database memory forensics: Identifying cache patterns for log verification

James Wagner [a, *], Mahfuzul I. Nissan [a], Alexander Rasin [b]

[a] University of New Orleans, New Orleans, LA, USA
[b] DePaul University, Chicago, IL, USA

ARTICLE INFO

Article history:

Keywords:
Memory forensics
Database forensics
Digital forensics

ABSTRACT

Cyberattacks continue to evolve and adapt to state-of-the-art security mechanisms. Therefore, it is critical for security experts to routinely inspect audit logs to detect complex security breaches. However, if a system was compromised during a cyberattack, the validity of the audit logs themselves cannot necessarily be trusted. Specifically, for a database management system (DBMS), an attacker with elevated privileges may temporarily disable the audit logs, bypassing logging altogether along with any tamper-proof logging mechanisms. Thus, security experts need techniques to validate logs independent of a potentially compromised system to detect security breaches.

This paper demonstrates that SQL query operations produce a repeatable set of patterns within DBMS process memory. Operations such as full table scans, index accesses, or joins each produce their own set of distinct forensic artifacts in memory. Given these known patterns, we propose that collecting forensic artifacts from a trusted memory snapshot allows one to reverse-engineer query activity and validate audit logs independent of the DBMS itself and outside the scope of a database administrator's privileges. We rely on the fact the queries must ultimately be processed in memory regardless of any security mechanisms they may have bypassed. This work is generalized to all relational DBMSes by using two representative DBMSes, Oracle and MySQL.

1. Introduction

In a compromised environment, security experts can employ forensic techniques to verify the integrity of data and files, including the audit logs. Research has considered one-way hash functions (Peha, 1999; Snodgrass et al., 2004; Pavlou and Snodgrass, 2008), hash chains (Sinha et al., 2014; Crosby and Wallach, 2009; Ahmad et al., 2018), and append-only files (e.g. (Ahmad et al., 2022),) to verify an audit log and detect tampering. However, none of these approaches consider activity that completely bypasses the logs. For example, consider a privileged user (or an attacker who gained access to such an account) who has the legitimate ability to suspend logging. The audit logs could be temporarily disabled while a malicious query is executed. Not identifying this missing activity could significantly delay security breach detection and response.

In this paper, we analyze database management systems (DBMS) memory contents and demonstrate that SQL query operations produce repeatable patterns in the buffer cache; we generalize these patterns by evaluating query artifacts in two representative DBMSes, Oracle and MySQL. We then present an example use case that leverages these patterns to verify the integrity of audit logs files and explore the lifetime of memory artifacts. A tool that is ready for deployment and considers a wide variety of workload scenarios is beyond the scope of this paper. We argue that our approach is a promising direction to address this security gap because while queries can be omitted from logs, they must ultimately be processed in memory. For example, some query operations read data into memory and other operations manipulate this data in memory; all of these operations produce memory artifacts.

In our adversary model (see Section 3), we assume a trusted DBMS process snapshot as an input and use memory forensics to carve (extract) data and metadata. We then use forensic artifacts to identify patterns produced by query operations; we demonstrate that query operations (e.g., index accesses, joins, sorting) produce

* Corresponding author.
E-mail addresses: jwagner4@uno.edu (J. Wagner), minissan@uno.edu (M.I. Nissan), arasin@depaul.edu (A. Rasin).

https://doi.org/10.1016/j.fsidi.2023.301567

repeatable patterns in memory. For log verification, if an operation cannot be attributed to a known logged query, we flag it as suspicious. The major paper contributions are:

1. We demonstrate that query operations produce repeatable patterns in memory for two representative DBMSes, Oracle and MySQL (Section 4).
2. We define atomic query operations (e.g., a table scan) and a process to reverse-engineer memory patterns into individual operations (Section 4).
3. Given a trusted memory snapshot, we present an approach to identify missing log activity, specifically retrieval queries, i.e., `SELECT` (Section 5).
4. We explore the lifetime of query artifacts in various workload sizes (Section 6).

2. Related work

2.1. Database forensics

We achieve our unique approach in this paper by using database forensics, specifically data carving. Traditional file carving directly reconstructed files without relying on the file system as a "dead analysis" on disk images (Garfinkel, 2007; Richard and Roussev, 2005). More recently, memory forensics has sought to perform a "live analysis" (Case and Richard, 2017). One example includes inspecting runtime code for malware (e.g. (Case and Richard, 2016),).

Since DBMSes manage their own internal storage separately from a file system, they require their own carving approaches. Database forensics was explored in (Stahlberg et al., 2007; Wagner et al., 2015, 2016, 2017a). However, the work in database forensics has only been concerned with carving data as part of a "dead analysis", where our goal in this paper works toward detecting unusual DBMS access patterns as a "live analysis" similar to the ideas of malware detection in memory. Nissan et al. extracted values from database memory using string searches, and then used support vector machines to determine the query operations that cached the data (Nissan et al., 2023). Our approach extracts forensic artifacts from memory snapshots using database page carving (Wagner et al., 2017b). In this paper, we borrow from the idea of page carving to extract metadata, which allows us to collect additional information beyond simple string searches.

Wagner et al. previously abstracted DBMS memory architecture into four main areas based on DBMS documentation and database textbooks (Wagner and Rasin, 2020). In this paper, we focus on analysis of the I/O buffer and the sort area in memory. DBMS-specific names for the I/O buffer include buffer pool (MySQL & PostgreSQL), buffer cache (Oracle), and page cache (SQL Server); specific sort area names include sort buffer (MySQL), work_mem (PostgreSQL), SQL work areas (Oracle), and work table (SQL Server). The I/O buffer caches table and index pages recently accessed from disk, typically with the least recently used (LRU) algorithm. DBMSes reserve the sort area for memory-intensive operations, e.g., `DISTINCT`, `ORDER BY`, merge joins, and hash joins. For example, hash join constructs a key-based hash table in memory to perform the join operation.

To facilitate an understanding of our results, we visualize some of our data collected from memory using RAM spectroscopy graphs, which were proposed by Wagner et al. in (Wagner and Rasin, 2020). RAM spectroscopy graphs measure the amount of data stored at a given memory offset. Fig. 1 provides an example of a RAM spectroscopy graph for I/O buffer layout. The x-axis represents the byte offset with the DBMS memory snapshot (normalized to a percentage). For example, 50% represents 50 MB offset in a 100 MB snapshot or 800 MB offset in a 1.6 GB snapshot.

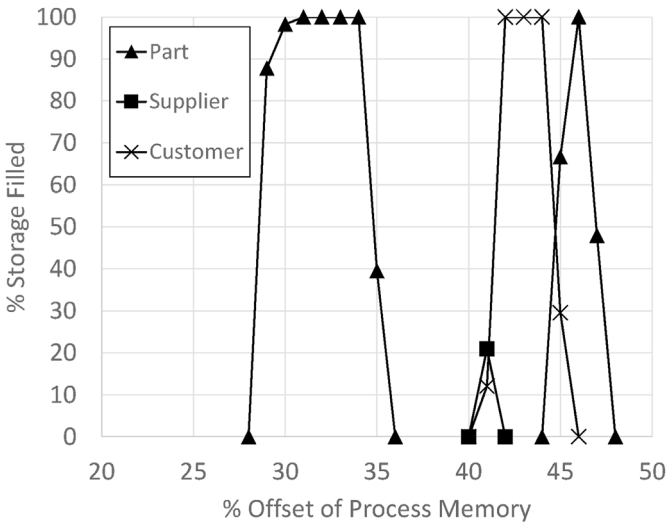

Fig. 1. MySQL full table scans in RAM spectroscopy.

2.2. Log tampering

Adedayo et al. (Adedayo and Olivier, 2012) reconstructed records using inverse relational algebra. Their algorithms assume not only a trusted audit logs but also require other trusted logs to be configured with special settings. While this algorithm is useful to reconstruct user behavior to identify suspicious actions, it requires particular log settings to be enabled and does not consider compromised log files. In contrast, the goal of this paper is verify the accuracy of log files based on the forensic artifacts in memory.

Query activity logs can be generated using triggers. While DBMSes do not support triggers for `SELECT` statements, a `SELECT` trigger was explored for the purpose of logging (Fabbri et al., 2013). While this work can log all query activity, a privileged user can disable triggers or even bypass a `SELECT` trigger by creating a temporary view to access the data.

ManageEngine's EventLog Analyzer (Eventlog analyzer) provides audit log reports and alerts for Oracle and SQL Server for all user activity. However, the Eventlog Analyzer creates these reports based on the DBMS logs, and thus, is vulnerable to a privileged user who can alter or disable logging.

The work in (Wagner et al., 2017b) presented an approach to detect activity missing from DBMS logs if a privileged user disabled logging. However, their work only accounted for query operations that modify data (e.g., `INSERT`, `DELETE`, `UPDATE`) and thus, changes were observed in persistent storage that could be compared to log files. In this paper, we address a similar adversary, but consider detecting missing retrieval queries (i.e., `SELECT`). Such queries would only leave artifacts in memory as opposed to persistent storage. The work in (Wagner et al., 2018) detected DBMS file tampering via the file system without accessing the DBMS API. However, we do not consider an adversary with system administrator privileges to be within the scope of this paper.

Network-based monitoring tools and methods provide a separation of privileges by operating independent of the DBMS, and thus, can be kept outside of the database administrator's control. IBM Security Guardium Express Activity Monitor for Databases (Ibm security guardium express activity, 2017) monitors incoming packets for abnormal query activity. If abnormal activity is

suspected, this tool restricts access for a specific user. Liu et al. (Liu and Huang, 2009) monitored logs network packets containing SQL statements, which prevented their tool from being disabled by database administrators. At the same time, monitoring only network activity does not account for local DBMS connections and does not address obfuscated queries designed to fool the system. By monitoring memory activity in this paper, our approach accounts for both local and network activity. Furthermore, if a query is obfuscated, it must ultimately be processed in memory producing artifacts from the accessed data. However, this paper does not attribute activity to a specific user.

3. Threat model

We assume a secure environment with separate accounts for a system administrator (SA) and a database administrator (DBA) as two different roles within an organization. We further assume the principle of least privilege for the two roles, which protects the system from misuse of privileges, accidents, or a compromised account.

This assumption is consistent with the motivation behind the two-tiered key protocol for transparent data encryption (TDE) to assign separate duties to SA's and DBA's. TDE is supported by DBMSes such as SQL Server and Oracle (e.g. (Huey, 2017),). The DBMS encrypts files so they cannot be accessed by anyone with just server access. TDE stores a master key in a module that is external from the DBMS; the DBA cannot access them, but the SA can access them. The master key encrypts/decrypts the TDE table keys. The table keys are stored in the DBMS where the DBA does have access to these, but the SA does not.

The DBA can issue privileged SQL commands against the DBMS, including disabling logs but cannot suspend any OS processes. The DBA can misuse elevated privileges to bypass most of internal DBMS security mechanisms, including the audit log. Specifically, the DBA can either modify the DBMS audit logs or temporarily disable logging altogether. Tamper-proof audit logs (e.g [1, 2, 3, 4, 5, 6]) could be conceptually implemented to prevent modifications of DBMS audit logs. We also discussed related work (Wagner et al., 2017b), that addressed attack scenario when the audit logs were disabled, detecting modifications (e.g., `INSERT`, `DELETE`, or `UPDATE`) to data. The novel threat (not previously addressed in related work) that we consider in this paper, is a DBA that disables the audit logs and performs read-only operations (i.e., `SELECT`) against the database. Table 1 summarizes how the audit logs can be disabled for several DBMSes. If the attacker gained both SA and DBA privileges, it could be problematic if the attacker also has the ability to suspend other processes and OS logging. If the DBMS were running in a VM, trusted snapshots can be collected from the host even if the attacker gained SA and DBA privileges to the VM.

We also bring attention to DBA-level commands that flush the DBMS I/O buffer. However, these commands do not prevent us from collecting forensic artifacts. These commands only free list the DBMS buffers, and they do not explicitly overwrite or zero out the buffers.

Table 1
Commands to disable the audit log.

DBMS	Command
Oracle	`NOAUDIT session` `NOAUDIT SELECT ON [table]`
Postgres	`set pgaudit.log='none'`
MySQL	`set global` `audit_log_connection_policy=NONE`
SQL Server	`ALTER SERVER AUDIT [file name]` `WITH (STATE = OFF)`
DB2	`db2audit stop`

4. Database memory patterns

This section demonstrates that query operations produce repeatable patterns in memory that we identify by analyzing forensic artifacts. We represent all queries as one or more operations, which are what we match to artifacts. For example, consider a query: `SELECT Name FROM Employee ORDER BY Salary`. This query consists of two operations. The first operation reads the entire table (i.e., a full table scan) into the DBMS I/O buffer. The second operation takes the data (now stored in memory) and sorts it in the sort area, which is separate from the I/O buffer. From a memory perspective, all query operations are either a data access or a data manipulation.

Data access operations read data into the DBMS I/O buffer from disk (or read cached data from the I/O buffer). Furthermore, all data access operations are ultimately either a **full table scan** (i.e., read the entire table) or an **index access** (e.g., a B-Tree index performs direct access on specific pages). A query can consist of a series of data access operations. For example, a join query accesses the two tables using either a full table scan or an index access depending on the type of join (i.e., nested loop join, hash join, or merge join). Other operations process data loaded from disk. For example, processing data based on the join condition requires a data manipulation operation (e.g., building a hash table or sorting a table).

Data manipulation operations process data in the dedicated memory-intensive sort area other than the I/O buffer. Examples include building additional data structures for sorting (e.g., `ORDER BY` or a merge join) or a hash table for hash joins. While identifying the patterns for data manipulation operations is useful for many other applications (e.g., Section ?), we show that this information is not necessary to detect query activity not captured by logs. However, if one wanted to completely reverse-engineer a query from a memory snapshot alone, identifying data manipulation operations in the sort area would provide more precise results. Experiments in this section include a discussion of data found in the sort area for Oracle, but not MySQL. This is because we observed that Oracle reads large, memory intensive I/O operations directly into the sort area.

Table 2 lists the unit operations evaluated in this section. We describe these operations using an Oracle DBMS and a MySQL DBMS. We will demonstrate these operations produce repeatable patterns and discuss how they generalize to other DBMSes.

4.1. Experimental setup

Dataset. Experiments used Scale 10 of the Star Schema Benchmark (SSBM) (O'Neil et al., 2009); Table 3 summarizes the data tables. SSBM represents a data warehouse environment. It combines a realistic distribution of data with a synthetic data generator that of creates datasets of different scales.

Configuration. We performed experiments on two representative DBMSes. A MySQL 8.0.28 instance was deployed on a Linux Debian 10 server, and an Oracle 19c instance was deployed on a Windows 10 server. We used the default page size for each DBMS: 16 KB for MySQL and 8 KB for Oracle. We configured the I/O buffer to 400 MB (25,600 16-KB pages or 53,200 8-KB pages) for each DBMS instance. We consider these two DBMSes to be representative of a variety of relational DBMSes for several reasons. First, MySQL is one of the most popular open source DBMSes and Oracle is the most widely used commercial DBMSes. Second, Oracle use

Table 2
Summary of query operations described in Section 4.

Operation	Summary
Full Table Scan	Read entire table to retrieve a record(s).
Index Access	Retrieve a record(s) by traversing a B-Tree index to find a pointer(s) that directly references the record(s).
Nested Loop Join	Join two tables using two nested for-loops.
Hash Join	Join two tables w/a hash table built on the (smaller) table. Iterate over the (larger) table. Locate the (smaller) table records using the built hash table.

Table 3
SSBM scale 10 table sizes.

Table	# Records	Size (MB)	Oracle 8 KB Page	MySQL 16 KB Page
DWDate	2556	0.4	32	24
Supplier	20K	2	242	134
Customer	300K	34	4067	2188
Part	800K	84	7342	5375
Lineorder	60M	5600	732K	361K

heap tables by default, and MySQL builds index organized tables (IOT) on the primary key by default. Other major DBMSes choose one of these two options. For example, Microsoft SQL Server and SQLite use IOTs by default. PostgreSQL and IBM DB2 use heap tables by default. As we will show, each of these approaches results in distinct caching pattern behavior. Finally, we used both Windows and Linux servers to demonstrate that our approach is not dependent on the OS, but rather the DBMS.

Algorithm 1. MySQL Query Plan for Nested Loop Join

Algorithm 2. MySQL Query Plan for Hash Join

4.2. Workload to isolate operations

We designed a SQL workload to evaluate the individual operations summarized in Table 2. We verified all queries used the desired operation by inspecting the query plan (a query plan describes the post-optimization query operations structure in a DBMS, e.g., Algorithm 2).

Full Table Scan. We designed four queries for each of the five SSBM tables in Table 3 (i.e., 20 queries in total). Each query used different attributes in the `SELECT` clause and variations to the `WHERE` to demonstrate that these different types of queries ultimately perform a full table scan. The queries below show our queries for the Supplier table. The DWDate, Customer, Part, and Lineorder tables each had four equivalent queries of their own.

```
SELECT * FROM Supplier;

SELECT Name FROM Supplier;

SELECT Phone FROM Supplier WHERE Nation ='CANADA';

SELECT Name FROM Supplier WHERE Region='No Match';
```

Index Access. We designed two queries for each of the five SSBM tables (i.e., 10 queries in total). One query used the B-Tree index (constructed by default) on the primary key, and the other query used a secondary B-Tree index we built on a column that we selected based on the number of distinct values in a column. A DBMS query optimizer will not use a B-Tree index if the expected number of records selected by a query is high. In a column with few distinct values, selected number of records will be high (e.g., with 10 distinct values, an equality predicate may select about 10% of the table). The queries below illustrate our workload choices for the Supplier table. The first query performs an index access on the primary key index. The second query perform an index access on the secondary index we created on the Phone column. The DWDate, Customer, Part, and Lineorder tables each had two equivalent queries of their own in the workload.

```
SELECT * FROM Supplier WHERE Suppkey = 10000;

SELECT * FROM Supplier

WHERE Phone = '14-290-375-5897';
```

Nested Loop Join. We designed a single query to demonstrate the cache pattern for a nested loop join. Only one query was used because a nested loop join ultimately uses a combination of full table scans and index accesses, which were previously presented. Algorithm 1 displays the MySQL query plan for our designed query below. The query plan demonstrates that a nested loop join was used, and that table Lineorder is accessed using a full table scan and table Part is accessed using an index access. The Oracle query plan was similar to Algorithm 1 except that table Part was accessed using a full table scan. In Oracle, we also had to explicitly request a nested loop join by adding the following optimizer hint to the `SELECT` clause: `/*+ ORDERED USE_NL(Part) */`.

```
SELECT SUM(size) FROM Lineorder JOIN Part;
```

Hash Join. We designed a single query to demonstrate the cache pattern for a hash join. Similar to nested loop joins, a hash join ultimately uses a combination of full table scans and index accesses. Algorithm 2 displays the MySQL query plan for our designed query below. The query plan demonstrates that a hash join used a full table scan to access both the Customer and Supplier tables. The Oracle query plan was equivalent to Algorithm 2.

```
SELECT Count(S.City) FROM Supplier S, Customer C

WHERE C.Region = S.Region;
```

4.3. Procedure

Experiments ran each of the four workloads on both DBMS instances. Before each workload, we restarted each instance and cleared the cache files. For each workload, we collected a series of memory snapshots: a before and after the execution of a query to verify the data cached. Procdump v9.0 (Russinovich and Richards, 2017) was used to collect DBMS process snapshots on Windows, and the process snapshot data under */proc/$pid/mem* was read on Linux.

To analyze memory contents, we passed each snapshot to DBCarver (Section 2.1). The most relevant information that DBCarver returned that we reference were the Object ID, an internal object identifier maintained by the DBMS, and the Page ID, an identifier stored in each page that is unique for each object ID.

For example, the Supplier table may have the Object ID 100 and a series of Page IDs 1, 2, …, N. Similarly, the Customer table may have the Object ID 101 and a series of Page IDs 1, 2, …, M. Thus, carving the Object ID and Page ID from a page header allowed us to uniquely identify each page in memory.

4.4. Results & discussion: full table scans

Oracle. Table 4 summarizes the full table scan results in Oracle. For table DWDate, all 32 table pages were cached in the I/O buffer for all four queries. Based on each page's Page ID, we observed that the pages occurred in the same order. The 32 pages were spread across a space of approximately 3 MB, and we observed four groups of eight pages. This is explained by extents, a logical storage unit used by Oracle. An extent for our instance was 64 KB (or eight pages). Therefore, data was read in a unit of extents for the DWDate full table scans. We also note that Oracle uses a another, larger logical storage unit called a segment. For our instance, a segment was 1 MB (or 128 pages). An extent is not completely filled with data pages; some pages are used to store metadata describing the extent.

For table Supplier, each query cached all 242 pages were cached in the I/O buffer. The 242 pages were spread across 3 MB, and we observed four groups of 61, 60, 60, and 61 pages. Each such group corresponds to a chunk of 8 extents (8 extents $\times$ 8 pages $=$ 64 pages), with additional metadata in storage.

For table Customer, the first query cached 63 of the 4067 pages in the sort area. Based on the PageIDs, these pages were the last 63 pages stored on disk for table Customer. This page count also corresponds to 8 extents in Oracle. Since this table is a medium size (34 MB) relative to the I/O buffer size (400 MB), we conclude that the DBMS used the sort area to efficiently read the query rather than letting it occupy a significant portion of the I/O buffer. For the remaining three queries in the same workload, all 4067 Customer table pages were read into the I/O buffer. The second query cached these pages, and the remaining two queries re-used the cached pages. These results indicate two possible outcomes from a full table scan: a) the entire table is read into the I/O buffer or b) the table is read in chunks (8 extents) into the sort area. In the latter case, the pages remaining in the sort area are those that correspond to pages with the highest Page ID (or the pages with the highest file address on disk) because the scan proceeds in chunks from beginning to end.

For table Part, all four queries cached 125 of 7342 pages in the sort area. Based on the Page IDs, these pages were the last 125 pages stored on disk for the Part table. Pages were spread across approximately 4 MB in two groups of 63 and 62 pages. This result is consistent with the observations for the first Customer table query; since this table was large (84 MB) relative to the I/O buffer, the DBMS processed this large I/O request directly in the sort area.

For table Lineorder, all four queries cached 188 of the 732K table pages in the sort area. Based on the Page IDs, these were the last 188 pages stored in the table file on disk. Pages were spread across approximately 7 MB in three groups of 63, 63, and 62 pages. Similar to table Part, the large table ($\gg$ I/O buffer size) was processed as chunks in the sort area with units of 8 extents.

Table 4
Oracle full table scan results in # of Pages.

Table	I/O Buffer	Sort Area
DWDate	32	0
Supplier	242	0
Customer	4067	63
Part	0	125
Lineorder	0	188

MySQL. All queries in the MySQL instance cached the entire table for tables Date (24 pages), Supplier (134 pages), Customer (2183 pages), and Part (5375 pages). This also included the IOT root and intermediate nodes. Since table Lineorder was larger than the 400 MB I/O buffer, each query cached approximately 27K - 28K pages (out of 361k). This number varied slightly within this range across all query runs. All pages were cached in the I/O buffer, and no pages were in the sort area as for Oracle. There was also no evidence of caching in larger logical units as with extents for Oracle. This is best explained by the IOTs behaving more similar to an index access (index access patterns are presented next in this section).

Fig. 1 uses RAM Spectroscopy graphs to visualize an example snapshot containing a full table scan for tables Part (▲), Supplier (■), and Customer (×). Tables Supplier and Customer each had localized data forming one peak. Two separate peaks were observed for table Part in separate areas on the I/O buffer for a single query.

Summary. Oracle, our representative for heap tables, had two flavors of forensic artifacts for full table scans: cache the smaller tables ($<$ ~10% of I/O buffer size) in the I/O buffer or read larger tables ($>$ ~10% of I/O buffer size) in chunks directly into the memory intensive sort area. The chunks in the sort area were a unit of a larger storage (in the case of Oracle, multiple extents). The pages remaining in the sort area corresponded to the pages with the highest address in the file on disk, which we confirmed with the Page IDs (verifying the sequential scan).

MySQL, our representative for IOTs, had one flavor of forensic artifacts for full table scans: cache the entire table (if smaller than the I/O buffer). We reason for this difference from the heap tables in Oracle is due to the behavior of IOTs. The leaf nodes containing the table records are accessed by traversing the IOT B-Tree structure. Therefore, each leaf node access is virtually an index access.

Given the results from these two DBMSes, we expect similar results for DBMSes that either use heap tables or IOTs. Of course, DBMS architecture specifics need to be accounted for. For example, PostgreSQL uses heap tables but not the same logical storage structures, extents and segments, as Oracle. We also mention the locality of the pages in memory. When entire tables were cached in the I/O buffer for Oracle and MySQL, pages were mostly clustered together. This may not hold for more extensive workloads since most DBMSes employ some variation of the LRU page replacement algorithm.

4.5. Results & discussion: index access

Both databases produced similar results in the I/O buffer. In general, each time an index access was performed, the index pages (including root nodes and intermediate nodes) and the corresponding table page(s) were cached. However, we delve into some technical differences between the heap tables in Oracle and IOTs in MySQL.

Oracle. Oracle used a straight-forward B-Tree index structure with value-pointer pairs for both the primary key indexes and the secondary indexes. The Oracle index pointers consist of a File ID, Page ID, and Row Position. While we did not use the File IDs for memory analysis, they are available for applications that require interpreting both a RAM snapshot and disk image. The Page ID is stored in a table page's header. The Row Position refers to the row position within the table page. Therefore, a Page ID and a Row ID were used to match an index value to a table page. We note that the table pages cached for an index access were individual pages and not larger units, such as extents observed in full table scans.

MySQL. The MySQL IOT was organized on the primary key index. Therefore, a primary key index in MySQL simply read the root node, any intermediate nodes, and the leaf node(s) containing the table

records into the I/O buffer. The MySQL secondary index was a typical B-Tree structure with value-pointer pairs. The pointers were primary key values themselves, which allowed records to be accessed using the IOT structure. Therefore, for a secondary index access, not only were the secondary index pages and the IOT leaf pages containing the records cached, but also the IOT root and intermediate nodes.

Summary. While a table page can be associated with a particular index pointer using Page IDs, there is still the challenge of accurately matching data in presence of multiple overlapping index range scans. However, such accurate matching is primarily needed for future tasks such as query reconstruction from RAM snapshots. In order to validate the audit logs based on the memory artifacts, memory artifacts need to be explained by *any* of the logged queries. For example, Query 1 scans the cities in the range between 'Austin' and 'Detroit', and Query 2 scans the cities in the range between 'Chicago' and 'Eugene'. The range 'Chicago' and 'Detroit' is explained by either query, but it does not matter which query they are matched to, as long as a value is explained by a logged operation.

4.6. Results & discussion: nested loop join

Oracle. Table 5 summarizes the nested loop join results for Oracle. 125 Lineorder pages were cached in two groups of 62 and 63 in the sort area. The Lineorder artifacts were consistent with our previous full table scan results except that only two 8-extent chunks were observed (rather than three). The entire Part table (7342 pages) was cached in the I/O buffer. Since table Part was in the inner for-loop of the join, the DBMS chose to cache the entire table rather than repeatedly read it from disk. If table Lineorder is read in 125-page chunks and compared to table Part (the inner for-loop), that would require >5800 scans of table Part. Thus, the DBMS wisely chose to cache it even though it was ~20% (i.e., a significant fraction) of the I/O buffer.

MySQL. There were 4035 table Part pages and 27606 table Lineorder pages in the I/O buffer. The entire Part table was not cached because an index access was used, rather than a full table scan, which was consistent with the primary key index accesses previously discussed. The entire Lineorder table was not cached even though a full table scan was used because the Lineorder table is larger than the I/O buffer size. These results are consistent with the full table scan results and index access we reported earlier in this section.

Summary. The nested loop joins for both DBMSes were consistent with the behavior we previously reported for full table scans and index accesses. We conclude that other DBMSes besides Oracle and MySQL will also have nested loop join behavior that is consistent with their full table scan and index access patterns.

4.7. Results & discussion: hash join

Oracle. Table 5 summarizes the hash join results for Oracle. The entire Supplier table was cached in the I/O buffer, and 63 Customer pages were found in the sort area, which is consistent with the full table scan results previously reported. Since a hash table was built for table Supplier, table Customer only required one scan, which explains why Oracle decided to read it into the sort area rather than caching the entire table in the I/O buffer.

MySQL. There were 134 Supplier and 2183 Customer pages in the I/O buffer. These results are consistent with the full table scan results reported earlier in this section.

Summary. The hash joins for both DBMSes were consistent with the behavior previously reported. Similar to nested loop joins, we conclude that other DBMSes besides Oracle and MySQL will maintain hash join patterns that are consistent with full table scans and index accesses.

5. Example application: log verification

Purpose. This experiment demonstrates how the cache patterns described in Section 4 can be used to detect missing activity from audit log files. This experiment uses a workload that overwrites the hidden activity; this will be used to support our discussion in what a formal log verification system would need to incorporate.

Setup & Workload. The dataset and DBMS configurations described in Section 4 were used. We designed a workload to simulate a scenario that makes obfuscated query activity difficult to detect due to overlapping data accesses and a large memory usage, reducing the lifetime of memory artifacts. The following provides the individual steps in our workload. A DBMS process snapshot was collected after each step.

T1 SSBM query #2.1 simulates typical user behavior:
```
SELECT sum(revenue), year, brand1.
FROM lineorder ⋈ dwdate ⋈ part ⋈ supplier.
WHERE category = 'MFGR#12'AND region = 'AMERICA'.
GROUP BY year, brand1 ORDER BY year, brand1.
```
T2 Simulate obfuscated data access with the following:
a. Disable audit log (Table 1 commands).
b. `SELECT * FROM Part.`
c. Re-enable audit log.

T3 SSBM query #4.2 simulates typical user behavior:
```
SELECT year, snation, category, sum(revenue).
FROM lineorder ⋈ dwdate ⋈ part ⋈ supp ⋈ cust.
WHERE cregion = 'AMERICA'AND sregion = 'AMERICA'.
AND year IN (1997,1998).
AND mfgr IN ('MFGR#1','MFGR#2').
GROUP BY year, snation, category.
ORDER BY year, snation, category.
```

Table 5
Pages cached for joins in Oracle.

Nested Loop Join		
Table	**IO Buffer**	**Sort Area**
Part	7342	–
Lineorder	–	125
Hash Join		
Table	**IO Buffer**	**Sort Area**
Supplier	242	–
Customer	–	63

5.1. Oracle results

Table 6 summarizes the memory artifacts found after running the workload against Oracle. The goal is to map the artifacts to a pattern from Section 4 and determine if they are explained by any query operations in the log.

T1. The full table scan patterns for DWDate (32 pages) and Supplier (242 pages) in the I/O buffer and the full table scan pattern for Lineorder (125 pages) in the sort area are explained by the query in the log at T1 (SSBM #2.1). This query performed full table scans on DWDate, Supplier, Part, and Lineorder. No evidence of the Part full table scan at T1 is not a problem because we are searching for activity that *cannot* be explained by the audit log.

T2. The full table scan patterns for DWDate and Supplier are still explained by the audit log entry at T1. However, there is a full table scan pattern for Part (125 pages), but do not a full table scan on Lineorder. While the Part full table scan pattern could be explained by the audit log entry at T1, this result is inconsistent with query engine operations. For a hash join, the larger table (Lineorder) should be in the outer table in the hash join, and thus, the last table that is scanned. Therefore, these artifacts are flagged as potentially missing log activity.

T3. There are full table scan patterns for DWDate (32 pages) and Supplier (242 pages) in the I/O buffer and the full table scan pattern for Lineorder (125 pages) in the sort area. These patterns are explained by the audit log entry at T3 (SSBM #4.2). We note that these operations are also explained by the entry at T1. We do not consider this to be problematic since the goal is find activity that cannot be explained by *any* recent log activity. Additionally, we notice at T3 that evidence of the malicious query at T2 was overwritten, and the traces of the full table scan on Part are erased. The discussion section for this experiment uses this result to present future challenges.

5.2. MySQL results

Fig. 2 summarizes the memory artifacts found after running the same workload against MySQL. At T1, the index access pattern for Part (4035 pages), discussed in Section 4.2, Algorithm 1, and the full table scan patterns on DWDate (20 pages), Lineorder (~27K pages), and Supplier (134 pages) are explained by the log at T1.

T2. The full table scans for DWDate (20 pages), Supplier (134), and Lineorder (~26k pages) are still attributed to operations at T1 in the audit log. However, the full table scan pattern for Part is inconsistent with the index access pattern used by SSBM query #2.1. Therefore, this is flagged as potentially missing log activity.

T3. The index access pattern for Part (4035 pages) and the full table scan patterns for DWDate (20 pages), Supplier (134 pages), Customer (2188 pages), and Lineorder (~25k pages) are explained by the query operations at the T3 SQL entry (SSBM query #4.2) in the audit log. We note that most of these operations can also be matched to the entry at T1. We do not consider this to be problematic since the goal is find activity that cannot be explained by *any* recent log activity. Additionally, the evidence of the malicious query was overwritten, and all traces were erased. Again, the discussion section for this experiment uses this result to present future challenges.

Table 6
IO buffer/sort area memory artifacts (pages).

Table	T1	T2	T3
DWDate	32/0	32/0	32/0
Supplier	242/0	242/0	242/0
Customer	0/0	0/0	0/0
Part	0/0	0/125	0/0
Lineorder	0/125	0/0	0/125
Total	**274/125**	**274/125**	**274/125**

5.3. Discussion

One obvious limitation of this application is memory volatility; we explore this in Section 6. For both Oracle and MySQL, all memory artifacts were explained at T3 and all traces of the hidden query at T2 were overwritten. When determining the frequency in which to collect snapshots, the size of the DBMS memory areas and the query workload should be considered. Intuitively, system activity is more accurately described if snapshots are taken more frequently than the time it takes to overwrite the entire I/O buffer and (for a DBMS like Oracle) the sort area.

Another challenge is detecting a hidden query that produces a repeated pattern similar to a logged query. For example, a logged query performs a full table scan on Supplier and then an unlogged query also performs a full table scan on Supplier. The second unlogged query would re-use the cached pages and not create new memory artifacts in the I/O buffer. Distinguishing forensic artifacts in such cases would require identifying additional artifacts in memory. However, for log validation use cases that are only concerned about detecting *unusual* activity, we argue this would pose a lesser challenge. For example, if users issue a standard set of queries through a web interface, the log validation system would be focused on detecting unlogged activity that deviates from the expected workload.

6. Log verification with a larger workload

Purpose. This experiment explores the lifetime of query artifacts, building on Section 5 by considering larger workloads. Specifically, we look for activity that deviates from an expected workload (i.e., a hidden operation does not overlap with expected workload query features).

Setup & Workload. The dataset and DBMS configurations described in Section 4 were used. To represent an expected query workload, we used the SSBM query flight #4, where each query accesses all five tables. We generated sets of 10, 100, and 1000 queries.

We also created a query, $Query_M$, that deviates from this workload to simulate malicious activity. Artifacts that are explained by the current workload are not considered evidence of malicious activity. $Query_M$ used an index access on the primary key, whereas the expected workload queries perform full table scans on the Part table. We performed the following steps for each DBMS:

T1 Execute $Query_M$:
`SELECT * FROM Part WHERE PartKey = 10000`.
T2 Run the expected workload of 10, 100, or 1000 queries from SSBM query flight #4.
T3 Capture memory snapshot

6.1. Oracle results

This experiment began by running $Query_M$ to verify the data it cached. We observed that a single data page from the Part table and the corresponding index leaf and root page were cached in the I/O buffer. Our experiments will test how long these artifacts remain in memory.

The runtimes for our workloads of 10, 100, and 1000 queries were 2.3 min (or 14.0 s/query), 22.7 min (or 13.6 s/query), and 3 h

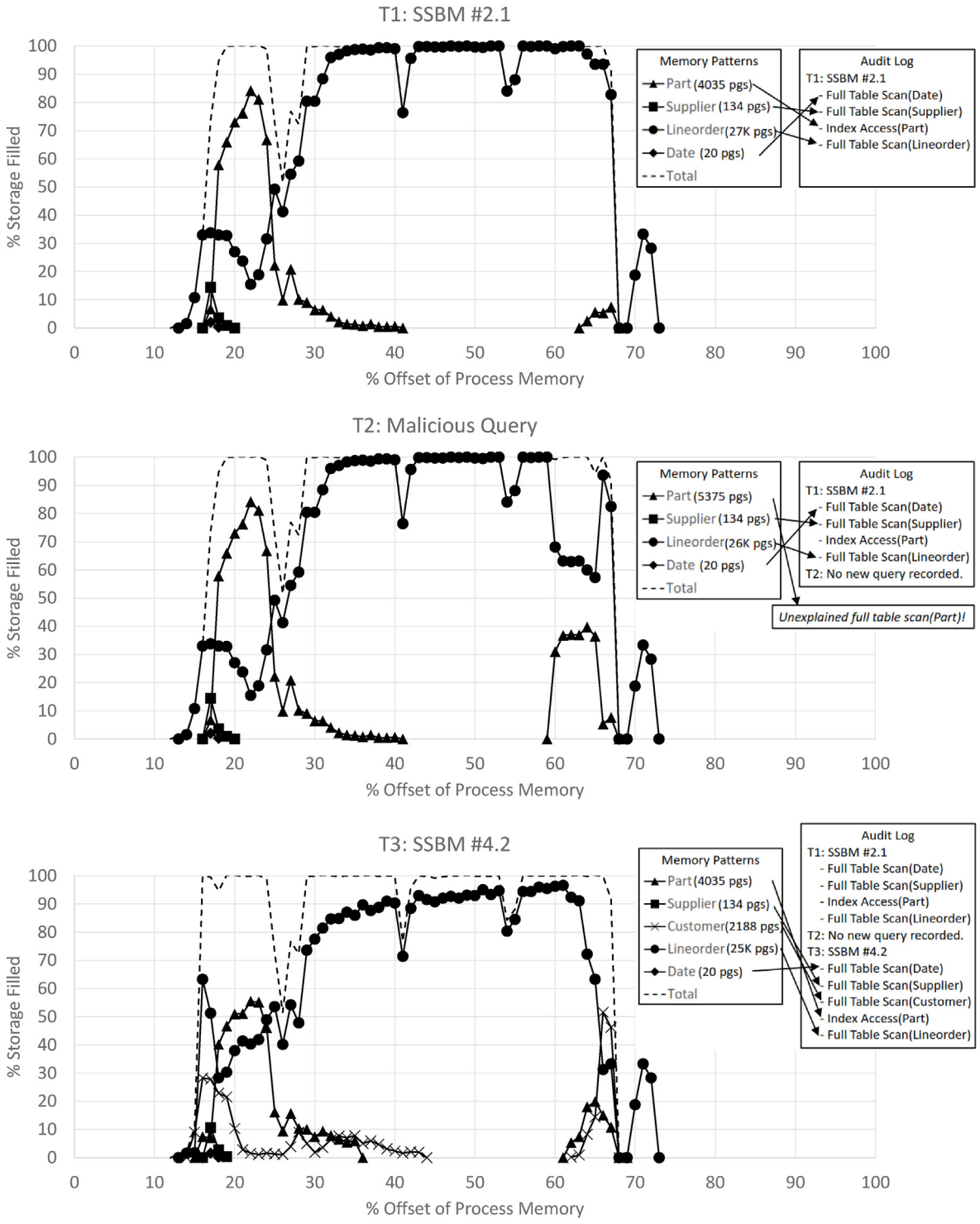

Fig. 2. Ram spectroscopy graph for the MySQL experiments.

and 43.3 min (or 13.4 s/query), respectively. Table 7 summarizes the results of artifacts found in the I/O buffer and the sort area. All three workloads produced similar results with the exception of a portion of the Lineorder table in the I/O buffer for the 1000 query workload. We found the DBMS used the pages cached in I/O buffer for the DWDate, Supplier, and Customer tables for all queries in each workload. The result of the full table scan on the Part table was 30 pages in the sort area for all workloads.

After running each workload, we also observed a single page from table Part remained in the I/O buffer along with a leaf index page and the root index page belonging to the primary key index of table Part. All three pages corresponded to the pages accessed by $Query_M$. Therefore, we conclude the lifetime of $Query_M$ is dependent on when the I/O buffer is overwritten (based on the LRU algorithm). The expected query workload performed similar table access patterns for all queries, which resulted in the I/O buffer not being

Table 7
Oracle results. IO buffer/sort area artifacts (pages).

Table	10 queries	100 queries	1000 queries
DWDate	32/0	32/0	32/0
Supplier	242/0	242/0	242/0
Customer	4067/0	4067/0	4067/0
Part	1/30	1/30	1/30
Lineorder	0/0	0/0	27214/0
Total	**4342/30**	**4342/30**	**31556/30**

overwritten, and thus, a longer lifetime for the $Query_M$. Given these results, we anticipate that artifacts from $Query_M$ could reside in memory much longer than what our experiments explored. Again, this is dependent on the type of data access operations the workload performs and the size of the I/O buffer.

6.2. MySQL results

Similar to the Oracle experiment, $Query_M$ was first ran to verify the data it cached. We observed a single table Part (IOT) data page and the corresponding IOT root and intermediate node pages all at consecutive addresses.

The runtimes for our workloads of 10, 100, and 1000 queries were 13.1 min (or 1.3 min/query), 2.4 h (or 1.5 min/query), and 25 h (or 1.5 min/query), respectively. Table 8 summarizes the artifacts found in the I/O buffer. All three workloads produced similar results. As expected from the experiments in Section 5, the entire tables for DWDate, Supplier, and Customer were found in memory, along with a portion of the Part table (4035 pages) and the Lineorder table (~25K pages).

The data page and two index pages that corresponded to the $Query_M$ artifacts were cached. However, we cannot attribute those artifacts to $Query_M$ for two reasons. First, the index pages and the data page were no longer consecutive as we previously observed when running $Query_M$ alone. Second, we found a total of 4035 Part table pages, which included the $Query_M$ artifacts. All of these pages together indicate a different operation (a full table scan) than the secondary index access performed by $Query_M$. As these pages can be explained by query operations in the workload, we consider $Query_M$ to be overwritten.

6.3. Discussion

The query runtimes were about 5–6 times longer in MySQL than in Oracle. This is explained by the different usage of the I/O buffer for the query operations. MySQL cycled through the I/O buffer to process each query, while Oracle chose to only cache the small - medium sized tables in the I/O buffer and process the large table scans in the sort area. This observation is consistent with how the full table scans are processed for the MySQL IOTs and the Oracle heap tables in Section 4. Therefore, the runtimes correspond with how the I/O buffer was used and thus, provide an indication of the lifetime of the artifacts.

Regardless of the workload size, the results were the same for both DBMSes; the artifacts from $Query_M$ were identified in the Oracle experiments, but they could not be identified in the MySQL experiments. Therefore, the lifetime of the artifacts is dependent on both the query operations in the workload and the DBMS storage management, rather than the number of queries or a global time. In both DBMSes, the queries primarily used full table scans to process the queries, but the difference in storage management (heap table vs. IOTs) resulted in different outcomes. However, we would expect similar results for both DBMSes if the workload primarily used index accesses because in Section 4, index accesses produced the same artifacts: the data page itself along with the B-Tree index pages. Therefore, each query will cache similar artifacts for each DBMS. The lifetime would then be dependent on how these pages are re-used and new pages are brought into the I/O buffer based on the LRU algorithm.

The malicious queries in our experiments were designed to access tables that were also accessed by the expected workload to further test the limitations of our methods. We anticipate that a malicious query accessing a table not included in the expected workload will produce artifacts with a longer lifetime. For example, if the malicious query in Section 5 had accessed a different table for the Oracle experiments, its artifacts will have a longer lifetime consistent with the results in this section.

Table 8
MySQL results. IO buffer memory artifacts (pages).

Table	10 queries	100 queries	1000 queries
DWDate	20	20	20
Supplier	134	134	134
Customer	2188	2188	2188
Part	4035	4035	4035
Lineorder	25K	25K	25K
Total	**31230**	**31231**	**31230**

7. Conclusion & future work

This paper demonstrated that query operations (in two representative DBMSes, MySQL and Oracle) produce repeatable patterns in memory. When a query accesses data, either a full table scan or an index access is performed. In both MySQL and Oracle, an index access caches the relevant index pages (along with any intermediate and root nodes) and the corresponding data pages. Since MySQL use IOTs, a full table scan forces an entire table into memory at once (if it is smaller than the buffer cache size). Alternatively, in Oracle, a memory intensive sort area was used to perform a full table scan on the large heap tables.

We then demonstrated how these patterns can verify log integrity and explored artifact lifetime. Artifact lifetime is dependent on the query operations, not necessarily the number of queries or a global time. While someone with DBA privileges can bypass security or logging mechanisms, their malicious query operations must still be processed in memory. Therefore, we propose the work in this paper can be used to build more formal tools and methods to verfiy the integrity of DBMS audit logs.

The log verification approach demonstrated in Sections 5 & 6, is a first step towards developing a formal log verification tool. Besides log verification, we envision several other applications that are supported by the contributions in this paper. One notable application that is of interest to the digital forensics and cybersecurity communities is the ability to describe data exfiltration.

Acknowledgments

This work was partially funded by the Louisiana Board of Regents Grant LEQSF (2022-25)-RD-A-30 and by US National Science Foundation Grant IIP-2016548.

References

Adedayo, O.M., Olivier, M.S., 2012. On the completeness of reconstructed data for database forensics. In: International Conference on Digital Forensics and Cyber Crime. Springer, pp. 220–238.
Ahmad, A., Saad, M., Bassiouni, M., Mohaisen, A., 2018. Towards blockchain-driven, secure and transparent audit logs. In: Proceedings of the 15th EAI International

Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, pp. 443–448.
Ahmad, A., Lee, S., Peinado, M., 2022. Hardlog: practical tamper-proof system auditing using a novel audit device. In: 2022 IEEE Symposium on Security and Privacy (SP), IEEE Computer Society, 1554–1554.
Case, A., Richard III, G.G., 2016. Detecting objective-c malware through memory forensics. Digit. Invest. 18. S3–S10.
Case, A., Richard III, G.G., 2017. Memory forensics: the path forward. Digit. Invest. 20, 23–33.
Crosby, S.A., Wallach, D.S., 2009. Efficient data structures for tamper-evident logging. In: USENIX Security Symposium, pp. 317–334.
Eventlog analyzer. https://www.manageengine.com/products/eventlog/.
Fabbri, D., Ramamurthy, R., Kaushik, R., 2013. Select triggers for data auditing. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE), IEEE, pp. 1141–1152.
Garfinkel, S.L., 2007. Carving contiguous and fragmented files with fast object validation. Digit. Invest. 4, 2–12.
Huey, P., 2017. Introduction to transparent data encryption. https://docs.oracle.com/database/121/ASOAG/introduction-to-transparent-data-encryption.htm#ASOAG10117.
Ibm security guardium express activity monitor for databases. http://www-03.ibm.com/software/products/en/ibm-security-guardium-express-activity-monitor-for-databases, 2017.
Liu, L., Huang, Q., 2009. A framework for database auditing. In: Computer Sciences and Convergence Information Technology, 2009. ICCIT'09. Fourth International Conference on, IEEE, pp. 982–986.
Nissan, M.I., Wagner, J., Aktar, S., 2023. Database memory forensics: a machine learning approach to reverse-engineer query activity. Forensic Sci. Int.: Digit. Invest. 44, 301503.
O'Neil, P., O'Neil, E., Chen, X., Revilak, S., 2009. The star schema benchmark and augmented fact table indexing. In: Technology Conference on Performance Evaluation and Benchmarking. Springer, pp. 237–252.
Pavlou, K.E., Snodgrass, R.T., 2008. Forensic analysis of database tampering. ACM Trans. Database Syst. 33 (4), 30.
Peha, J.M., 1999. Electronic commerce with verifiable audit trails. In: Proceedings of ISOC.
Richard III, G.G., Roussev, V., 2005. Scalpel: a frugal, high performance file carver. In: DFRWS, Citeseer.
Russinovich, M., Richards, A., 2017. Procdump v9.0. In: https://docs.microsoft.com/en-us/sysinternals/downloads/procdump.
Sinha, A., Jia, L., England, P., Lorch, J.R., 2014. Continuous tamper-proof logging using tpm 2.0. In: International Conference on Trust and Trustworthy Computing. Springer, pp. 19–36.
Snodgrass, R.T., Yao, S.S., Collberg, C., 2004. Tamper detection in audit logs. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases-Volume 30. VLDB Endowment, pp. 504–515.
Stahlberg, P., Miklau, G., Levine, B.N., 2007. Threats to privacy in the forensic analysis of database systems. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 91–102.
Wagner, J., Rasin, A., 2020. A framework to reverse engineer database memory by abstracting memory areas. In: International Conference on Database and Expert Systems Applications. Springer, pp. 304–319.
Wagner, J., Rasin, A., Grier, J., 2015. Database forensic analysis through internal structure carving. Digit. Invest. 14, S106–S115.
Wagner, J., Rasin, A., Grier, J., 2016. Database image content explorer: carving data that does not officially exist. Digit. Invest. 18, S97–S107.
Wagner, J., et al., 2017a. Database forensic analysis with dbcarver. In: Conference on Innovative Data Systems Research.
Wagner, J., et al., 2017b. Carving database storage to detect and trace security breaches. Digit. Invest. 22, S127–S136.
Wagner, J., Rasin, A., Heart, K., Malik, T., Furst, J., Grier, J., 2018. Detecting database file tampering through page carving. In: 21st International Conference on Extending Database Technology.