A Framework to Reverse Engineer Database Memory by Abstracting Memory Areas

James Wagner[1] and Alexander Rasin[2]

[1] University of New Orleans, New Orleans LA 70148, USA
jay.wagner88@gmail.com
[2] DePaul University, Chicago IL 60604, USA
arasin@cdm.depaul.edu

Abstract. The contents of RAM in an operating system (OS) are a critical source of evidence for malware detection or system performance profiling. Digital forensics focused on reconstructing OS RAM structures to detect malware patterns at runtime. In an ongoing arms race, these RAM reconstruction approaches must be designed for the attack they are trying to detect. Even though database management systems (DBMS) are collectively responsible for storing and processing most data in organizations, the equivalent problem of memory reconstruction has not been considered for DBMS-managed RAM.

In this paper, we propose and evaluate a systematic approach to reverse engineer data structures and access patterns in DBMS RAM. Rather than develop a solution for specific scenarios, we describe an approach to detect and track any RAM area in a DBMS. We evaluate our approach with the four most common RAM areas in well-known DBMSes; this paper describes the design of each area-specific query workload and the process to capture and quantify that area at runtime. We further evaluate our approach by observing the RAM data flow in presence of built-in DBMS encryption. We present an overview of available DBMS encryption mechanisms, their relative advantages and disadvantages, and then illustrate the practical implications for the four memory areas.

1 Introduction

Database managements systems (DBMS) serve as the main data repositories for applications ranging from personal use (e.g., text messaging, web browsers) to enterprise data warehouses (e.g., airlines, merchants). In order to perform "live" (i.e., runtime) forensic, security, or performance analysis in a DBMS, an understanding of its RAM layout is necessary. There are currently no approaches or tools that can reverse engineer RAM contents of a DBMS. Current work in OS RAM analysis (see Section 2) seeks to detect specific malware patterns, offering no generalized solution. Although OS RAM may be too general, DBMS memory can be abstracted by identifying and quantifying each type of its memory area. In this paper, we describe our approach and validate its generality on four major RAM areas across several representative DBMSes.

DBMSes allocate multiple RAM areas within their process memory to serve a particular purpose. For example, I/O buffer area caches pages accessed from disk (and some other operations); in Section 5 we describe four ubiquitous memory areas and other special-purpose RAM areas. A memory area can be detected and quantified by executing a customized synthetic workload and capturing the resulting RAM snapshots. In Section 7, we illustrate how to capture any memory area, describing our process and include the link to our query workloads.

A significant contribution of our approach to reverse engineering memory is assigning context to data. We demonstrate this with DBMS encryption as a use case (Section 9 outlines other use cases). While data can be encrypted outside of a DBMS, all major DBMSes (e.g., IBM DB2, Microsoft SQL Server, Oracle, MySQL, SQLite) manage their own encryption. "Foreign" encryption imposes trade-offs between protection guarantees and limiting DBMS functionality. Section 2 summarizes two encryption types: disk encryption and client-side encryption. Disk encryption protects data at rest (i.e., in persistent storage) with software between the I/O subsystem and DBMS. While this approach is transparent to a DBMS and, thus, does not interfere with DBMS functionality, it offers little control over encryption granularity; a malicious system administrator (or an attacker who gained similar privileges) can access DBMS files at byte-level. Alternatively, data encrypted and decrypted by a client application protects data both in-motion and at-rest. A major trade-off for this approach is a loss of DBMS functionality. The built-in encryption DBMS mechanisms offer a balanced solution between disk and client-side encryption. Section 8 demonstrates how to assess encryption vulnerabilities based on the purpose of each memory area. For example, a decrypted credit card number could appear in memory as part of an `INSERT` or `SELECT` query, as an internal copy in buffer cache, or an intermediate computation. The major contributions of the paper are:

- A survey of encryption mechanisms supported by popular DBMSes (Section 4). We review encryption options in IBM DB2, Microsoft SQL Server, Oracle, MySQL, PostgreSQL, SQLite, Firebird, and Apache Derby.
- A taxonomy that abstracts four ubiquitous categories of DBMS memory architecture: the I/O buffer, sort area, transaction buffer, and query buffer.
- A framework for isolating and identifying DBMS memory areas (Section 6).
- An evaluation of our framework (Section 7) demonstrating successful RAM analysis for three representative DBMSes: MySQL, Oracle, and PostgreSQL.
- A use-case study demonstrating how to assign context to encrypted data in RAM (Section 8) using a MySQL DBMS instance.

2 Related Work

Assigning Context to Forensic Data. Foundational digital forensic analysis applies file carving techniques, which reconstruct data without using file system metadata. The work in [7, 20] presented some of the earliest research around file carving performed as a "dead analysis" on disk images. As the field of digital

forensics matured, memory forensics "live analysis" has emerged [6]. An important application for memory forensic investigation is inspecting runtime code to detect malware (e.g., [5]). Such work requires not only carving but a complicated analysis of application and kernel data structures.

Since DBMSes manage their own internal storage separately from the OS and DBMS files are not standalone (unlike PDFs or JPEGs), file carving cannot be applied to DBMS data. Carving DBMS storage was explored in [25, 27]. However, database carving has only been part of a "dead analysis." Combining the work in this paper with database carving would enable a "live analysis", such as detecting unusual DBMS access patterns similar to malware detection.

Query Processing for Encrypted Data. Client-side encryption (i.e., encrypting data before loading it into a database) prevents the DBMS from processing data unless data properties are preserved. Deterministic encryption always produces the same ciphertext for a given plaintext and thus supports equality predicates (e.g., `WHERE Name = 'Alice'`), equality based joins, and `DISTINCT` operations. `GROUP BY` operations can be used, but beyond the columns in the `GROUP BY` clause, deterministic encryption is essentially limited to the `COUNT` function (e.g., `SELECT City, COUNT(*) FROM Customer GROUP BY City`).

Order preserving encryption (OPE) produces ciphertext that preserves the plaintext value ordering [1,4]. OPE supports sorting (i.e., `ORDER BY`), range scans (e.g., `WHERE Salary BETWEEN 50K AND 80K`), and covering indexes. Homomorphic encryption (e.g., [12]) supports computations on ciphertext, returning an encrypted result. Fully homomorphic encryption supports unbounded computations, but research identified major trade-offs [13, 17]. Partially homomorphic encryption offers a more balanced solution by supporting only bounded computations [10]. Support for the standard SQL string wildcard operators (i.e., % and _) on encrypted data was explored in [23]. However, it is only suitable for strings with known patterns. There are no solutions that support query processing on ciphertext with arbitrary wildcard expressions or regular expressions.

Systems such as CryptDB [19], Cipherbase [3], and Microsoft SQL Server's Always Encrypted [28] extend SQL and relational DBMSes to support query processing on encrypted data. However, these systems still sacrifice important functionality, such as nontrivial computations (e.g., multiplication and addition in the same expression) and regular expressions. More importantly, the encryption schemes should be designed with knowledge of the query workload. For example, homomorphic encryption does not support a workload that requires sorting. These systems also remain vulnerable to inference attacks since the ciphertext still preserves data properties [2, 14]. Alternatively, the use case in this paper considers encryption that is natively supported by DBMSes. These mechanisms do not sacrifice DBMS functionality and provide access granularity.

3 Background

Global vs. Local DBMS Memory. DBMSes divide memory into either a global or local context. Global memory stores data and objects shared by all

users, sessions, or all DBMS processes. Local memory stores data for an individual DBMS process, session, SQL statement, or an operation (e.g., sorting) within a SQL statement. All components in global memory remain active once the DBMS instance is started. Components in local memory may be allocated when the process, session, SQL statement, or operation starts and and de-allocated when it ends. Data that is loaded into local DBMS memory is therefore likely to leak into the OS RAM after it has been de-allocated.

Temporary Table. Temporary tables are used to simplify DBMS procedures and improve performance of processing intermediate query results. Temporary tables are only visible to their user session. They are automatically dropped when the session ends; most DBMSes support the option to drop a temporary table on `COMMIT`. Temporary tables are typically stored in local memory.

DBMS	Instance Level	Column Level
Apache Derby	✓	✗
DB2	TDE	pre-built functions, masking
Firebird	✓	✗
MySQL	TDE	pre-built functions, masking
Oracle	TDE	TDE, masking, pre-built functions
PostgreSQL	✗	pre-built functions
SQLite	✓	✗
SQL Server	TDE	client-side TDE, masking, pre-built functions

Table 1. Encryption features supported by major DBMSes.

4 Native DBMS Encryption

Table 1 summarizes the 8 popular DBMSes investigated in this paper their encryption mechanisms. At a high-level we partition all native database encryption into two categories: instance-level and column-level.

Instance-Level Encryption. Instance-level encryption supports encrypting DBMS storage at the granularity of individual files, or other storage structures (e.g., tablespaces). We further categorize the instance-level encryption mechanisms into the standard encryption and transparent data encryption (TDE).

Standard instance-level encryption encrypts all reads and writes to and from DBMS storage; the encryption key is only provided when the DBMS is started or during a new session login. This mechanism works by encrypting entire pages (e.g., table data, binary large objects, and indexes) that make up the DBMS files. Encrypted data typically includes not only user data in tables and indexes, but also WAL files and temporary files created by the DBMS. Standard instance-level encryption is supported by Apache Derby [26], Firebird (user-customized crypt plug-ins [18]), and SQLite (SQLite Encryption Extension, SEE [24]).

TDE is a more advanced version of the standard instance-level encryption, offered primarily by enterprise DBMSes. The major difference between TDE and standard instance-level encryption is a two-tier encryption key architecture. To implement TDE, a DBMS explicitly manages data encryption key(s) to encrypt/decrypt data. The data encryption key(s) themselves are stored in DBMS

storage and are further encrypted with a master key(s) created by the user. The master key is stored in a key store that exists externally and independently from the DBMS files. Two-tier key management creates a further diffusion of privilege required to decrypt the data; the master key can remain hidden from the database administrator. TDE is supported by DB2 (Native Encryption [9]), MySQL Enterprise [16], Oracle [15], and SQL Server [11].

Column-Level Encryption. Column-level encryption refers to the DBMS ability to encrypt individual columns or values in a column. The most common form of column-level encryption is pre-built functions in the DBMS engine (implemented in DB2, MySQL, PostgreSQL, Oracle, and SQL Server). To encrypt new data with pre-built encryption functions, the user must include both the plaintext to encrypt and the encryption key with `INSERT` or `UPDATE` statements. Similarly, to query encrypted data the user must provide the encryption algorithm, the encrypted value, and the encryption key with `SELECT`, `DELETE`, or `UPDATE` statements. The following query illustrates how these functions are used:

```
SELECT Decrypt(Name, key1) FROM Employee
WHERE SSN = Encrypt('123-45-6789', key2);
```

Another form of pre-built encryption functions offered by enterprise DBMSes is masking (or redaction). Masking allows users to specify a function describing which parts of a value must be hidden. Common examples of masking include revealing only the last four digits of a social security number or the last digits of a credit card number. Masking is supported by DB2, MySQL Enterprise, Oracle, and SQL Server.

In addition to the instance-level TDE, Oracle also supports a TDE mechanism for columns-level encryption. Column-level TDE still uses the two-tier encryption key architecture. SQL Server also supports a form of column-level TDE with Always Encrypted. The main difference with Always Encrypted is that the master key(s) is designed to be stored on the client-side application.

5 Abstracting DBMS Memory Structures

This section describes the abstraction of DBMS memory areas, based on the type of runtime operations each area supports. Area categories can be identified with the help of DBMS documentation and database textbooks; each type of DBMS operation can be consistently mapped to an area. For example, regular table access (e.g., table scan or index-based access) uses the I/O buffer in RAM to cache pages; hash-join execution uses a memory-intensive operation area.

We chose four areas that best represent the power of area-based memory abstraction: I/O buffer, the area for memory-intensive operations (or sort area), transaction (TXN) buffer, and query cache. Each DBMS uses some variant of these four areas – Table 2 lists their DBMS-specific names. An area may exhibit DBMS-specific configuration properties (e.g., sort area is allocated at a different granularity across DBMSes). DBMS can include other specialized memory areas, which can similarly be abstracted through the process described in this paper.

DBMS	I/O Buffer	Sort Area	TXN Buffer	Query Buffer
Apache Derby	Page Cache	JVM Sort Heap	Write Cache	Statement Cache
DB2	Buffer Pool	Sort Heap	Log Buffer	Query Heap
Firebird	Page Cache	TempCache	Undo Log Buffer	Metadata Cache
MySQL	Buffer Pool	Sort Buffer	Redo Log Buffer	Query Cache
Oracle	Buffer Cache	SQL Work Areas	Redo Log Buffer	Result Cache
PostgreSQL	Buffer Pool	work_mem	WAL Buffer	Query Plan Cache
SQLite	Page Cache	Transient Index*	Journal Buffer	Tokenizer
SQL Server	Page Cache	Work Table*	Log Cache	Procedure Cache

*Stored in the I/O buffer

Table 2. DBMS-specific names for major memory areas.

I/O Buffer. The I/O buffer caches table, index, and materialized view pages recently accessed from files on disk. While each DBMS uses a custom algorithm to decide when to store or evict data from the I/O buffer, some variation of the least recently used (LRU) policy is typically used. When at least one page record is accessed by a query, the entire page is cached in RAM and possibly decrypted. In most DBMSes, the I/O buffer contains a significant number of index pages, including the intermediate nodes and leaf pages of B-Tree indexes.

Sort Area. DBMSes reserve a separate area(s) for memory-intensive operations, which we refer to as the sort area. Sorting-like operations include the straightforward `ORDER BY` and `DISTINCT` clauses along with certain types of `JOIN`s, such as merge-join or hash-join. Nested loop join does not require as much memory (for sorting or hashing) and is typically performed in the I/O buffer. Our experiments illustrate the variations in sort area implementation. Oracle creates a sort area per session (i.e., per user connection); MySQL allocates a sort area for each query, even for the same session; PostgreSQL allocates a sort area for each operation (potentially allocating multiple sort areas for a single query). Once the operation associated with the sort area concludes, the sort area is de-allocated.

DBMSes almost always use temporary tables for sorting. This allows the DBMS to process data-intensive operations in parts, while storing the rest of the data in temporary files in persistent storage. The temporary tables are created in a dedicated sort area. Two DBMSes are an exception to that rule. SQL Server also uses temporary tables for sorting (called Work Tables) but actually stores them in the I/O buffer rather than in a dedicated local memory area. SQLite sorts data using temporary indexes rather than temporary tables. This is a consistent approach for SQLite since their tables are in the form of index organized tables.

Transaction Buffer. DBMSes use a TXN buffer to store write-ahead log (WAL) entries, sometimes referred to as redo or journal log entries. These log entries describe the transactional change history to data, including information needed to rollback or recover from the changes made to the database through DML operations (e.g., `DELETE` or `UPDATE`). DBMSes typically write to the TXN buffer in a circular pattern while a background process writes the entries to the WAL (or redo) log files on disk. The TXN buffer must be a part of global DBMS memory to avoid conflicting modifications among different users.

Query Cache. The query cache corresponds to operations that store raw SQL code in RAM as well as query execution plans. DBMS can subsequently reuse cached query execution plans in optimizing similar queries. Query cache area may also contain the DBMS-specific general programming code, e.g., PL/SQL (Oracle), PL/pgSQL (PostgreSQL), or T-SQL (SQL Server). Prepared statements and bind variable values are also stored in this area. Depending on the DBMS, the query cache can be part of global memory or local memory.

Other Areas. DBMSes reserve memory areas for background or user-issued maintenance operations. For example, the MySQL Change Buffer maintains indexes in the background, and the PostgreSQL maintenance_work_mem is reserved for the user-issued `VACUUM` and `REINDEX` operations. DBMSes often maintain custom resource scheduling information. Examples include the PostgreSQL commit log which stores the current state of each transaction (i.e., in-progress, committed, or aborted), the Firebird LockMem and Oracle Library Cache acquire locks for database objects, and the DB2 locklist that maintains a list of currently locked objects.

6 Experiment Overview

This section describes using our framework to isolate and identify the four memory areas from Section 5. Section 7 demonstrates the effectiveness of this framework on MySQL, Oracle, and PostgreSQL (chosen as representative of different internal storage implementations in a DBMS). Section 8 further shows how to apply this framework to assign context to decrypted data in memory for MySQL.

Our experimental analysis does not consider an exhaustive list of DBMSes and possible configurations; rather, our framework is designed to be independent of such variables. For example, Section 7 considers three representative DBMSes of the eight DBMSes listed in Section 5, but the same process can be applied to any relational DBMS. Similarly, Section 8 only considers MySQL, although the same analysis could be performed for the other DBMSes. This framework focuses on how DBMSes manage their internal process memory. Although we consider default implementations, a researcher could further explore a specific environment (e.g., compare DBMS memory behavior for ptmalloc2 vs. tcmalloc).

	DWDate	**Supplier**	**Customer**	**Part**	**Lineorder**
Size	200KB	700KB	10MB	50MB	2.3GB
Records	2556	8000	120K	600K	24M

Table 3. SSBM Scale 4 table sizes used for experiments.

6.1 Setup

Dataset. In our experiments we used the Star Schema Benchmark (SSBM) Scale 4 (∼2.4GB or ∼25M records). SSBM is widely used in database research community to represent a data warehouse evaluation. It combines a realistic distribution of data (maintaining data types and cross-column correlations) with a synthetic data generator that can create datasets at different scale. Table 3 summarizes the sizes of the SSBM tables used throughout the experiments.

DBMS	I/O Buffer	Sort Area	TXN Buffer	Query Buffer	Proc Mem
MySQL	100MB	256KB	1MB	10MB	383MB
Oracle	1.6GB	262MB	7MB	200MB	3.6GB
PostgreSQL	128MB	4MB	4MB	12MB	248MB

Table 4. DBMS memory area configurations used for experiments

DBMS Configuration. Table 4 lists the DBMSes we chose for an evaluation as well as their memory area parameter settings. We chose the settings for memory size in consultation with each DBMS' documentation and the established best practices. For example, MySQL and PostgreSQL are relatively lightweight engines, while Oracle requires significantly more memory. Furthermore, although the memory area serves the same function across DBMSes, the setting depends on DBMS engine implementation. For example, 4MB for PostgreSQL vs 262MB for Oracle is not as different as it appears: PostgreSQL initializes a sort area per operation (thereby creating multiple 4MB buffers per query in many cases), while Oracle uses a shared sort area.

Oracle 12c and MySQL 5.7 were deployed on a Windows 10 server. PostgreSQL 9.6 was deployed on a CentOS 6.5 server. Based on our experimental analysis, DBMS behavior remains similar between Windows and Linux servers.

6.2 Workload

We designed a SQL workload to populate each memory area with data. This includes three specialized sets of queries: 1) for filling the I/O buffer, 2) for filling the sort area with data, and 3) for filling the TXN buffer. For evaluation of the query cache area, we used the queries from the other three custom workloads. We next discuss the workload design in the context of each memory area. The workloads and workload generators can be downloaded from our research group website: `http://dbgroup.cdm.depaul.edu/downloads/DB_Mem_Workloads/Workloads.zip` These queries are designed sepcifically to highlight the different memory areas. While randomized queries would populate the same memory areas, they do not contribute to the goal of identifying the different memory areas, thus we do not include any.

I/O Buffer. We generated a total of 300,000 `SELECT` queries: 290,000 for Lineorder, 8,000 for Part, 1,500 for Customer, 400 for Supplier, and 100 for DWDate. All queries included a predicate that accessed equality on a value from an indexed column (to produce query execution with index-based access). An index-based access caches and retains all accessed data pages. Alternatively, full table scan may only cache a small portion of the table in memory and the DBMS is likely to immediately free-list that data. Since the primary key column contains all unique value and an index is automatically created on a primary key column, random values were accessed based on the primary key column. The following query template was used to generate this workload; '?' is a placeholder that was replaced by a (uniformly distributed) random value.

```
SELECT * FROM [Lineorder/Part/Supplier/Customer/DWDate]
WHERE [LO_Orderkey/P_Partkey/S_Suppkey/C_Custkey/D_Datekey] = ?;
```

Sort Area. We designed a memory-intensive query to perform a `JOIN` on all five tables in SSBM. To force result sorting, the query used a four column composite `ORDER BY` clause. The `SELECT` clause used 8 columns (as this is what is sorted in memory); these columns were arranged to uniquely identify them as sorted result among any records found in the SSBM tables and thus in the RAM snapshot. We experimentally chose the number of columns to be sufficiently large to fill each DBMS respective sort area.

```
SELECT S_Name,C_Name,P_Name,D_Day,S_City,S_Nation,S_Phone,C_Nation
FROM Lineorder JOIN Part JOIN Customer JOIN Supplier JOIN DWDate
ORDER BY S_Name, C_Name, P_Name, D_Dayofweek;
```

Transaction Buffer. We issued 10 `UPDATE` queries against the Part table. Each query updated 150,000 different records. To definitively detect entries in transaction buffer area, every query updated the container column to a (string + a unique ID) value not already used in the table. We used the following template for our update queries. The first question mark was replaced by a unique ID, the second question mark was replaced by a value from the P_Container column.

```
UPDATE Part SET P_Container = 'DEXA'+ ? WHERE P_Container = ?;
```

6.3 Experimental Procedure

We performed the experiments in the following sequence of steps for each DBMS: 1) Set up a new DBMS instance, 2) Load the SSBM tables into the DBMS, 3) Run the I/O cache query workload, 4) Run the transaction buffer query workload, 5) Run the sort area query workload. The RAM snapshot was generated *during* step #5 while the sort area workload was still running. Since sort area is part of local memory, it would become de-allocated after the sort area workload was completed. Therefore, the memory had to be captured while this local area was still allocated to the DBMS process. We verified that, if taken after step #5, the sort area was no longer a part of the captured DBMS process memory for all three evaluated DBMS. We used procdump v9.0 [21] to collected DBMS process snapshot on the Windows server, and read the process snapshot data under `/proc/$pid/mem` on the Linux server.

To evaluate the contents of the memory snapshots, used regular expressions with Python 2.7 to locate matching data values and their offsets. We designed the regular expression to search for known string values introducing enough slack for metadata content (varies by DBMS). For example, we used the following regular expression to detect customer records. Each string represents possible values (e.g., 'Customer#000000042', 'EUROPE', '85-234-621-3704') plus the additional wildcards for numeric columns and metadata characters.

```
''Customer#[0-9]{9}.{5,60}((EUROPE)|(AFRICA)|(AMERICA)|(MIDDLE EAST)|(ASI
  ↪ A)).{1,10}[0-9]{2}-[0-9]{3}-[0-9]{3}-[0-9]{4}''
```

7 Memory Experiments

For each experiment, we performed at least five evaluations and chose a representative snapshot (snapshots were always consistent with minor variations).

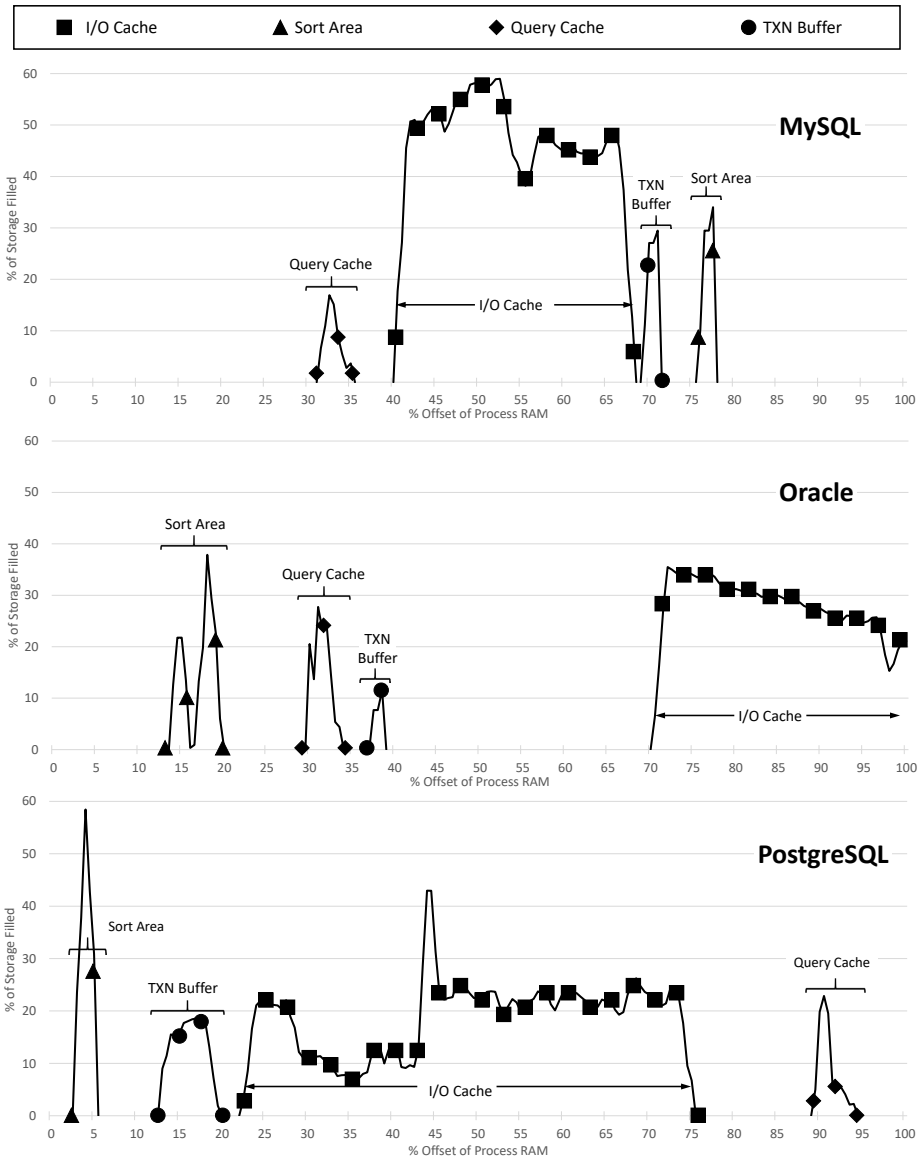

Fig. 1. Process memory representation for MySQL, Oracle, and PostgreSQL

7.1 RAM Spectroscopy Graphs

Figure 1 summarizes the memory contents; each DBMS is represented by a separate graph to describe and quantify contents of its process memory. The four memory areas from Section 5 are annotated with the following legend: I/O cache line is highlighted by square points, query cache line is denoted by diamonds, sort area is identified by triangles, and the TXN buffer is marked by circles. We term these graphs as *RAM spectroscopy*, which was inspired by infrared (IR) spectroscopy commonly used in analytical chemistry [22]. IR spectroscopy graphs measure the amount of infrared light absorbed by a chemical sample at different wavelengths. In an analogous manner, the purpose of our RAM spectroscopy graphs is to visualize the amount of data found at different memory offsets. We observed that each DBMS maintained a consistent shape throughout multiple session connections and system restarts. RAM spectroscopy cannot be applied to full OS RAM snapshots due to heavy fragmentation of the DBMS data.

For each RAM spectroscopy graph in Figure 1, the x-axis represents the byte offset within the DBMS process snapshot, normalized as a percentage. For example, 50% represents 50MB in a 100MB process snapshot or 800MB in a 1.6GB process snapshot. We summarized the data to 200 points (i.e., a point at every 0.5%) to normalize the snapshots for DBMSes across different RAM sizes. The y-axis represents an estimated amount of memory storage filled at a given offset. To estimate the percent of the storage filled by our data values, we assumed an additional 20% overhead to the data found. That is, for 'Customer#000000042' we accounted for (a total of 18×1.20) 21.6 bytes. This overhead is based on a generally accepted estimate of metadata associated with a DBMS page. While metadata varies between DBMSes, we chose a constant estimate to simplify our measurements. We also note that not all memory areas use pages (e.g., I/O buffer uses pages but sort area buffer does not). However, we only consider the relative heights of the peaks and we do not compare across areas (e.g., we do not compare I/O buffer peaks vs sort area buffer peaks).

7.2 Memory Observations

Memory Area Data. Figures 1 and 2 reflect only the SSBM table data distribution. Each area contains other data that we do not consider; we therefore never expect to observe values close to 100%. It is likely that memory areas are not densely packed or contain data from DBMS system tables (we did not load other data tables, but all DBMSes use internal "system" tables). Moreover, the memory areas typically contain auxiliary data or metadata in addition to raw table data. For example, the I/O cache includes index pages, which we did not measure in our report (I/O buffer regular expressions search for table rows and not index entries). The indexes used integer columns, and integers have their own DBMS-specific encodings that vary both in format and in size. Although index access and caching behavior would share similarities across DBMSes, we measured cached table rows (or SQL query result rows for query cache area) as the most consistent and representative way to detect the relevant memory areas.

Identifying Memory Area Regions. For all DBMSes the size of each memory area was consistent with sizes in the configuration files (see Table 4). When repeating and verifying these results, we observed the memory areas maintained the same order with slight shifts within the process memory snapshot. Therefore, we concluded that when a process memory snapshot is taken, data found in those offset regions belongs to the respective memory region. Each snapshot is a chosen representative of at least five independent snapshots we recorded. However, all of the snapshot were similar enough that any one of them could have been chosen for the spectroscopy figure report. Figure 1 also indicates how much of the overall DBMS memory process is occupied by the four memory areas. PostgreSQL snapshot uses relatively little space outside of these areas, while both Oracle and MySQL allocate a significant quantity of other RAM.

Local Memory. When the sort area query finished executing and the user session was disconnected, the sort area was no longer present in the process memory snapshot. This is consistent with the behavior of a local memory buffer, which stores the sort area. After de-allocation, the sort area data values (the output of the sort area SQL workload) could still be found in the full OS RAM snapshot, outside of the DBMS process. However, the sort area contents were now fragmented across OS RAM. Therefore, we concluded that when local DBMS memory is deallocated, its contents are effectively leaked into global OS RAM.

Memory Area Shapes. In Figure 1, MySQL I/O cache buffer fills by approximately 50%, in contrast with Oracle (approximately 25%) and PostgreSQL (approximately 20%). This is consistent with our expectations because MySQL uses index-organized tables. As a result, query access does not fetch index pages independently of the data pages (as there is no separate index structure). Specifically, B-Tree leaf pages with value-pointer pairs do not exist because data is in the leaf page of the B-Tree. Alternatively, both Oracle and PostgreSQL fetch a significant number of index pages, filling the I/O buffer cache with non-table pages. As a result, while the number of pages in the I/O buffer is similar, there are fewer table pages in Oracle and PostgreSQL compared to MySQL.

Oracle sort area in Figure 1 exhibits two distinct peaks for the single query we executed. This is also consistent with our expectations because Oracle uses hash-join which is a memory-intensive operation that targets the sort area buffer. We therefore observed data originating from two different operations in Oracle's sort area: the results sorting and the hash-joins. PostgreSQL sort area in Figure 1 exhibits only one peak. While PostgreSQL also uses a hash-join, it allocates a separate sort area for each operation. Therefore, the PostgreSQL hash-join operations use a different sort area that was de-allocated at the time the process snapshot was taken. MySQL uses nested loop join which will execute in the I/O buffer. Therefore, the MySQL sort area is dedicated to the result sorting.

8 Encryption Experiments

The purpose of this experiment is to demonstrate the importance of assigning context to data. We extend the Section 7 experiments using a new MySQL 5.7

instance with encryption enabled. The same setup and procedure described in Section 6 were used except TDE was enabled for all five SSBM tables. Since finding decrypted data in memory is an expected result, we emphasize that assigning context to this data can anticipate vulnerabilities.

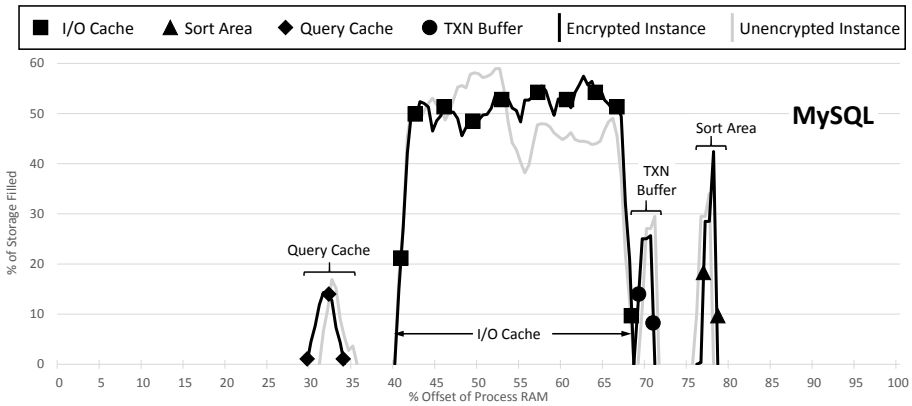

Fig. 2. Process memory representation for encrypted MySQL instance superimposed over the unencrypted MySQL instance from Figure 1

Figure 2 displays the resulting RAM spectroscopy graph for the encrypted MySQL instance combined with the MySQL instance data from Figure 1. The old unencrypted instance is represented with the gray line and the encrypted instance is represented with a black line.

All memory area peaks were observed equivalent, confirming that all data read into memory with TDE is decrypted and accessible in RAM. As a result, TDE has no significant impact on protecting the data from RAM perspective. However, it does not exhibit new vulnerabilities as does column-level encryption. Figure 2 also illustrates the consistency of the peak detection by superimposing results from two different snapshots. We note that the sort area buffer exhibited the same de-allocation behavior; as a result the decrypted data was released into global OS RAM. This data is particularly vulnerable because it could be observed in RAM and potentially captured with `malloc` from another process.

The experiment in Figure 2 measured the data cached by MySQL using instance-level TDE. The column-level encryption that relies on pre-built functions can manifest additional data vulnerabilities, depending on the memory area. The I/O buffer will expose less data with column-level encryption compared TDE. While the column-level encryption pages are visible in the I/O buffer, individual values in pages will remain encrypted in RAM. In contrast, both the query cache area and transaction buffer area will expose the encryption key in column-level encryption schemes. Pre-built encryption queries explicitly specify the encryption key in SQL commands which are cached in query cache and transaction buffer. The sort area will expose a similar amount of data for both column-level and TDE encryption because both TDE page requests and column-level encryption `SELECT` clause decrypts the queried values.

9 Future Work

The work in this paper supports future directions for third-party tools to assign context to data in addition to carving raw content from DBMS memory and providing detailed data flow tracking. Current DBMS APIs do not support data flow tracking and offer few limited system analysis features. For example, Oracle allows users to query the number of pages associated with table in the I/O buffer, but not the information about specific pages or records. Most DBMSes do not even offer the features provided by Oracle. We believe that data flow tracking has two primary application: security monitoring and performance analysis.

Current work in memory forensics detects activity patterns indicative of malware. The equivalent for DBMSes is detecting unusual data access patterns in RAM. Tools such as IBM Guardium [8] detect unusual patterns by observing SQL queries. While useful, this approach is limited – an obfuscated SQL query or a query that bypassed the monitoring proxy will escape detection. However, the approaches discussed here would allow monitoring memory operations in the event that an attacker circumvents current detection mechanisms.

DBMSes use a complex set of configuration settings. Our experiments demonstrated that these settings are not consistent across DBMSes; even for a corresponding setting (e.g., sort area buffer) the actual implementation can lead to a radically different behavior. For example, it is a known issue that increasing PostgreSQL area buffer setting (seemingly a good idea!) leads to significant performance deterioration as too many buffers are allocated in some workloads. Database memory forensic tools would allow administrators and researchers to more precisely identify performance bottlenecks and monitor memory utilization.

10 Conclusion

This paper presented a systematic approach to reverse engineering DBMS-controlled memory. We evaluated our approach by creating a taxonomy defining several common memory areas. Experiments demonstrated how to identify and isolate DBMS memory areas through design and evaluation of custom query workloads. We validated our approach on four memory areas using three representative DBMSes (PostgreSQL, Oracle, and MySQL). Finally, experiments showed the significance of assigning context to data in memory, an inherent feature of our reverse engineering approach.

References

1. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. In: SIGMOD conference. pp. 563–574 (2004)
2. Akin, I.H., Sunar, B.: On the difficulty of securing web applications using cryptdb. In: Conference on Big Data and Cloud Computing. pp. 745–752. IEEE (2014)
3. Arasu, A., et al.: Orthogonal security with cipherbase. In: CIDR. Citeseer (2013)
4. Boldyreva, A., Chenette, N., Lee, Y., O'neill, A.: Order-preserving symmetric encryption. In: EuroCrypt conference. pp. 224–241. Springer (2009)

5. Case, A., Richard III, G.G.: Detecting objective-c malware through memory forensics. Digital Investigation **18**, S3–S10 (2016)
6. Case, A., Richard III, G.G.: Memory forensics: The path forward. Digital Investigation **20**, 23–33 (2017)
7. Garfinkel, S.L.: Carving contiguous and fragmented files with fast object validation. digital investigation **4**, 2–12 (2007)
8. IBM: Security guardium. `http://www-03.ibm.com/software/products/en/ibm-security-guardium-express-activity-monitor-for-databases` (2017)
9. IBM: Db2 native encryption. `https://www.ibm.com/support/knowledgecenter/SSEPGG_11.1.0/com.ibm.db2.luw.admin.sec.doc/doc/c0061758.html` (2019)
10. Liu, J., Mesnager, S., Chen, L.: Partially homomorphic encryption schemes over finite fields. In: SPACE conference. pp. 109–123. Springer (2016)
11. Microsoft: Transparent data encryption. `https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/transparent-data-encryption?view=sql-server-ver15` (2019)
12. Microsoft: Microsoft seal. `https://www.microsoft.com/en-us/research/project/microsoft-seal/` (2020)
13. Naehrig, M., Lauter, K., Vaikuntanathan, V.: Can homomorphic encryption be practical? In: Workshop on Cloud computing security. pp. 113–124 (2011)
14. Naveed, M., Kamara, S., Wright, C.V.: Inference attacks on property-preserving encrypted databases. In: SIGSAC Conference. pp. 644–655 (2015)
15. Oracle: Database advance security guide. `https://docs.oracle.com/database/121/ASOAG/toc.htm` (2017)
16. Oracle Corporation: Innodb data-at-rest encryption. `https://dev.mysql.com/doc/refman/5.7/en/innodb-data-encryption.html` (2020)
17. Peng, Z.: Danger of using fully homomorphic encryption: A look at microsoft seal. arXiv preprint arXiv:1906.07127 (2019)
18. Peshkov, A., Firebird Foundation: Encrypting firebird databases. `https://firebirdsql.org/file/documentation/release_notes/html/en/3_0/rnfb30-security-encryption.html` (2016)
19. Popa, R.A., Redfield, C.M., Zeldovich, N., Balakrishnan, H.: Cryptdb: protecting confidentiality with encrypted query processing. In: SOSP. pp. 85–100 (2011)
20. Richard III, G.G., Roussev, V.: Scalpel: A frugal, high performance file carver. In: DFRWS (2005)
21. Russinovich, M., Richards, A.: Procdump v9.0. `https://docs.microsoft.com/en-us/sysinternals/downloads/procdump` (2017)
22. Skoog, D., West, D., Holler, J., Crouch, S.: Fundamentals of Analytical Chemistry, chap. Molecular Absorption Spectroscopy. Brooks-Cole (2014)
23. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: IEEE S&P conference. pp. 44–55. IEEE (2000)
24. SQLite: Sqlite encryption extension. `https://www.sqlite.org/see` (2019)
25. Stahlberg, P., Miklau, G., Levine, B.N.: Threats to privacy in the forensic analysis of database systems. In: SIGMOD conference. pp. 91–102 (2007)
26. The Apache Software Foundation: Configuring database encryption. `http://db.apache.org/derby/docs/10.13/security/cseccsecure24366.html` (2016)
27. Wagner, J., Rasin, A., Malik, T., Heart, K., Jehle, H., Grier, J.: Database forensic analysis with dbcarver. In: CIDR conference (2017)
28. Ward, B.: SQL Server 2019 Revealed, chap. New Security Capabilities. Springer (2019)