Forensic Science International: Digital Investigation 44 (2023) 301503

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS 2023 EU - Selected papers of the Tenth Annual DFRWS Europe Conference

Database memory forensics: A machine learning approach to reverse-engineer query activity

Mahfuzul I. Nissan*, James Wagner, Sharmin Aktar

University of New Orleans, New Orleans, LA, USA

ARTICLE INFO

Article history:

Keywords:
Memory Forensics
Database Forensics
Digital Forensics
Machine Learning
Support Vector Machines
Supervised Machine Learning

ABSTRACT

Memory analysis allows forensic investigators to establish a more complete timeline of system activity using a snapshot of main memory (i.e., RAM). Investigators may rely on such analysis to detect malicious activity and understand the scope of what data was exfiltrated. This is of particular interest in the presence of incomplete or untrusted logs, where a privileged user (or an attacker with such capabilities) can altogether bypass or disable logging. In such instances, a forensic investigator can still rely on the fact that data must ultimately be processed in memory, regardless of the information that is recorded in audit logs.

In this work, we propose methods to reverse-engineer query activity from a database management system (DBMS) process snapshot. Since DBMSes are used to manage and store an organization's most sensitive data, they are of particular concern for data exfiltration. A DBMS processes queries using a series of operations, such as index sort, file sort, or joins, which produce their own set of distinct forensic artifacts in memory. Our methods use these artifacts to make conclusions about recent query activity even in the presence of untrusted or incomplete logs. Our methods use a supervised learning based model using support vector machines (SVM) to approximate recently executed queries given these memory artifacts. We extract feature vectors from the byte frequencies in a special area of the DBMS process called the sort area fragment, and use SVM to predict the type of the query operation under supervised learning. We demonstrate the capabilities and the accuracy of our methods for two representative DBMSes, PostgreSQL and MySQL. Experimental results show that, our model achieved an accuracy of 92% and 90% on MySQL and PostgreSQL datasets, respectively.

1. Introduction

Memory forensics has received significant attention from researchers and practitioners in the digital forensics community. This is because memory forensics has the potential to provide more information about the current system state (i.e., most recent process activity and data versions) as opposed to analyzing images of persistent storage (e.g., hard disk drives or solid state drives). The most common applications of memory forensics include malware detection and inspection in main memory (e.g.,Manna et al., 2022; Case et al., 2020; Manna et al., 2021).

Database management systems (DBMSes) have also received attention in the digital forensics community since they are commonly used to store personal information (e.g., mobile devices and web browsers) and are used to store and manage an organizations most sensitive data. However, the majority of database forensics research has focused on analyzing images (or files) from persistent storage, and little attention has focused on database memory forensics. DBMSes require a specialized subdomain of memory forensics because most DBMSes replicate most of the functionality supported by an operating system. Examples of this replicated functionality include, audit logs, recovery logs (e.g., write ahead logs), access control policies, encryption mechanisms, and storage & memory management. For memory management specifically, DBMSes manage their own data and query processing separate from the OS. By this we mean, DBMSes usually define their own pages (i.e., minimum I/O unit), and they manage their own I/O buffer cache (with separate caching policies from the OS) and other

Abbreviations: SVM, Support Vector Machine; DBMS, Database Management System; RBF, Radial Basis Function; ASCII, American Standard Code for Information Interchange; I/O, Input/Output; API, Application Programming Interface; RAM, Random Access Memory.

* Corresponding author.
E-mail addresses: minissan@uno.edu (M.I. Nissan), jwagner4@uno.edu (J. Wagner), saktar@uno.edu (S. Aktar).

https://doi.org/10.1016/j.fsidi.2023.301503

memory structures for query processing.

Since the main advantage of memory forensics is to explore the current state of the system, we foresee a few distinct applications of database memory forensics and questions this field has the potential to answer. This paper works towards solutions to these problems and answering these questions. First, a forensic investigator may not only want to know what data is currently in a memory snapshot, but the current activity in a snapshot. For example, can the recent query activity be reversed-engineered from the current artifacts in the snapshot? Second, a memory snapshot may provide a trusted view of the DBMS in an untrusted environment. For example, one current motivation of memory forensics is to inspect user code for malware. To equate this example for a DBMS, can malicious code introduced into the source code for an open-source DBMS (e.g., PostgreSQL, MySQL, or SQLite) be detected from a memory snapshot? As another example, even though tamper-proof logging has been well researched, DBMSes allow the legitimate ability to disable logs, and other ways to obfuscate logged activity (e.g., temporary tables or views). Therefore, while logs are not guaranteed to contain a complete and accurate timeline, can memory snapshots verify or complete these timelines?

The goal of this paper is to reverse-engineer query activity from a DBMS process snapshot. The high-level idea of our approach is that SQL queries break down into a finite series of operations that are processed in memory. We propose and demonstrate that these operations produce repeatable patterns in memory. We then use machine learning methods to identify these patterns from a memory snapshot to make conclusions about the current system activity. Our major contributions are the following:

1. We present a novel approach to predict query activity from a memory snapshot using machine learning techniques.
2. We evaluate our approach for two representative DBMSes: MySQL and PostgreSQL.
3. We test this approach against a series of unknown memory snapshots.

The remainder of the paper is as follows. Section 2 provides an overview of related work in database forensics and machine learning in digital forensics. Section 3 discusses background concepts in machine learning and database forensics. Section 4 presents an overview of our proposed framework. Sections 5 - 7 provide detailed descriptions of our framework. Section 8 presents experiments that provide unknown snapshots to our trained framework for query predictions. Section 9 discusses future work and concludes the paper.

2. Related work

2.1. Database memory forensics

Case et al. presented a survey of memory forensics research and proposed future directions (Case and Richard, 2017). The authors concluded that database memory forensics can provide a significant amount of information that cannot be retrieved from the DBMS API or other database server mechanisms. Database memory forensics has the potential to retrieve recently executed queries, query results, commands executed on the server.

The current state of database forensics research has primarily focused on analyzing images of persistent storage, but some work has considered memory analysis. Stahlberg et al. performed some of his earliest research in database forensics (Stahlberg et al., 2007). However, the focus of their work analyzed tables, logs, and other files stored on persistent storage for deleted data that could be forensically reconstructed. In (Wagner et al., 2015, 2016), the authors outlined an approach to generalize database forensics, which was later formalized in (Wagner et al., 2017a, 2020). While the work in (Wagner et al., 2015) primarily concentrated on persistent storage, they performed preliminary experiments on carving memory artifacts from the DBMS process buffer cache. This was achieved by carving database data at the page level (or the minimum I/O unit).

Wagner et al. later used these database carving techniques to detect inconsistencies between audit logs and forensic artifacts (Wagner et al., 2017b). Again, this work primarily focused on persistent storage, but they presented experiments that outlined how query activity could be reversed-engineered from a snapshot; a similar to what the authors in (Case and Richard, 2017) proposed. Our goal in this paper is to expand upon this idea of capturing the current state of database activity from a memory snapshot. However, we take a different approach to the page carving used in (Wagner et al., 2017b). Two limitations we see with the page carving approach for database memory forensics is that it primarily only applies to the DBMS process buffer cache (and typically no other areas in the DBMS process), and it is limited to row-store relational DBMSes that implement their own page architecture. Alternatively, we rely mostly on byte frequencies.

The work in (Wagner and Rasin, 2020) focused specifically on reverse-engineering database memory. The authors described how to identify and distinguish different areas on the DBMS process and what data can be collected from these different areas. Table 1 summarizes DBMS-specific names for I/O buffer and sort area. In this paper, we focus on creating histogram of ASCII characters from sort area of each memory snapshot. Then use the histogram as a feature vector for machine learning algorithm. The job of I/O buffer is to cache table, index, and materialized view pages recently accessed from files on disk. To manage the cache, DBMSes usually use some variation of the least recently used (LRU) algorithm. Again, the memory analysis has relied on page carving outlined in (Wagner et al., 2017b) only applies to some of these memory areas because the not all of the DBMS process stores data in pages. In this paper, we specifically consider a memory-intensive area of the DBMS process used to satisfy queries, rather than the I/O buffer cache, which is used to cache table pages in memory.

2.2. Machine learning techniques in digital forensics

Our methods in this paper use file fragmentation classification techniques previously used in forensics, but we classify database memory artifacts as certain query operations. The first file fragment classification technique was proposed by McDaniel et al. (McDaniel and Heydari, 2003). Their approach built histograms of the frequency of American Standard Code for Information Interchange (ASCII) code (0 … 255) for each file. The histogram of the frequency was used as a 256-element vector (feature vector) to apply a clustering algorithm. The classification achieved 27.50% and 45.83% accuracy for the BFA and BFC algorithms, respectively. The next important file fragment classification work was done by Li et al. (2005). They modified the BFD algorithm that is actually 1-gram analysis to formulate a centroid or multiple centroids derived from the byte frequency distribution as the signature of a file type. Interestingly, this method achieved nearly perfect accuracy for 20-byte fragments. Karresand and Shahmehri (2006) proposed a similar method to (Li et al., 2005), called the Oscar Method. It introduced a new metric called rate-of-change (RoC) that defined the difference of the ASCII values of consecutive bytes. It's accuracy is near-perfect for JPEG file recognition.

The typical approach to solving file fragment classification problems is accomplished using simple statistics, such as the statistical distance of byte histograms and the entropy. However, this

method is inefficient for high entropy data, i.e., compressed files and multimedia. To resolve the issue, Li et al. (2011) utilized a supervised learning method (SVM) to predict the fragment type where they used byte frequencies as feature vectors. They achieved high accuracy using only byte frequencies histogram alone, without combining other methods.

We use nearly similar techniques as file fragment classification. In file fragment classification, researchers calculated different positions and frequencies of bytes in a file and classify the file type (e.g., PDF, jpeg, text) using SVM. In our case, instead of identifying file type from a file, we are identifying query type.

3. Background

3.1. Support vector machine (SVM)

Support vector machines (SVMs) are a group of supervised machine learning algorithms. The goal of the SVM algorithm is to find a hyperplane in an N-dimensional space (N is the number of features) that distinctly classifies the data points. Here, data points are support vectors which are closer to the hyperplane. Support vectors influence the position and orientation of the hyperplane. These support vectors are used to maximize the margin of the classifier.

Fig. 1 shows the visual representation of a SVM model with two support vectors. SVM algorithms use a set of mathematical functions to manipulate data known as kernel. Kernel take the data as input and convert it into the required form. Some of the most common kernel types of SVMs are: (1) Linear Kernel, (2) Polynomial Kernel, (3) Radial Basis Function (RBF), and (4) Sigmoid Kernel. Each of these kernel requires different set of hyperparameters for optimization.

We selected the SVM algorithm to solve our classification problem because it is one of the powerful supervised learning models used for classification problem and regression analysis (Huang et al., 2018). It is also fairly easy to implement using scikit-learn libraries. As we are solving 5-class classification problem with 2 feature and our dataset is small, we select SVM algorithm for our model. SVM classification performs well on small sized class and feature. It is not suitable for large datasets as the training complexity of SVM is largely dependent on the dataset size (Cervantes et al., 2008).

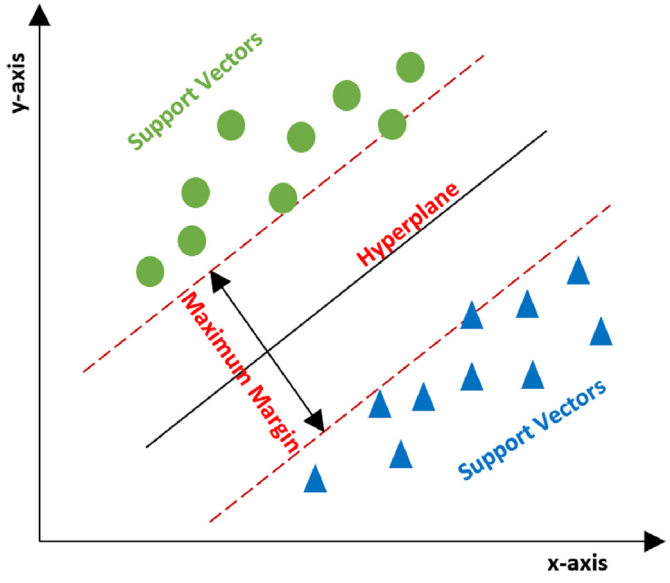

Fig. 1. SVM model representation.

3.2. Database memory patterns

All SQL queries consist of one or more operations. For example, consider these two queries: (i) SELECT * FROM customer, and (ii) SELECT name FROM customer ORDER BY city. Query (i) performs a full table scan, which reads the entire table into the DBMS I/O buffer. Query (ii) consists of two operations. The first operation reads the entire table into the DBMS I/O buffer using a full table scan. The second operation takes the data from I/O buffer (stored in the memory) and sorts it in the sort area. The sort area is a separate area from I/O buffer. From a database memory point of view, all query operations are either a **data access** or a **data manipulation**.

Data access operations either read cached data from the DBMS I/O buffer or bring data into the DBMS I/O buffer from disk. Moreover, there are two types of data access operations: full table scan or index access. For a full table scan operation, the DBMS reads the entire table. For an index access operation, the DBMS uses B-Tree to directly access specific pages.

Data manipulation operations read data from I/O buffer and process it according to the query plan. For example, constructing a hash table for hash joins or additional data structure for sorting (e.g., ORDER BY or a merge join) needs data manipulation operation. DBMS has a dedicated memory-intensive area for data manipulation operation and it is separate from the I/O buffer area. While different DBMSes have different names for this area (Table 1), we universally refer to it as the sort area.

A query may also need to perform series of operations to execute a query plan. For example, a join query either uses index access or full table scan based on the type of join (i.e., merge join, nested loop join, hash join) to access the two tables. Then, it uses data manipulation operations (e.g., sorting one of the tables or building a hash table) required by the join condition to process this data.

In this paper, our work is based on identifying the patterns of data manipulation operations. As we reverse-engineer a query from a memory snapshot, identifying data manipulation operations in the sort area would provide us more precise results.

4. Framework overview

Fig. 2 displays an overview of our proposed framework. At a high-level, our framework is divided into three components: 1) Dataset Collection & Preparation (green), 2) SVM Model Implementation (blue), and 3) SVM Model Evaluation (red). Sections 5 - 7 provides more detailed descriptions of each component. Section 8 demonstrates how this framework can be used by forensic investigators with a snapshot containing unknown memory artifacts. While, we performed the experiments on a MySQL and a PostgreSQL DBMSes, our presented work is designed to work for any row-store relational DBMS.

The first component, Dataset Collection & Preparation, produces a dataset that is provided to the SVM model in the next step. To build this dataset, we first execute our custom-designed query workload and collect a series of DBMS process snapshots. Next, we

Table 1
DBMS-specific names for I/O buffer and sort area.

DBMS	I/O Buffer	Sort Area
MySQL	Buffer Pool	Sort Buffer
PostgreSQL	Buffer Pool	work_mem
Oracle	Buffer Cache	SQL Work Areas
SQL Server	Page Cache	Work Table

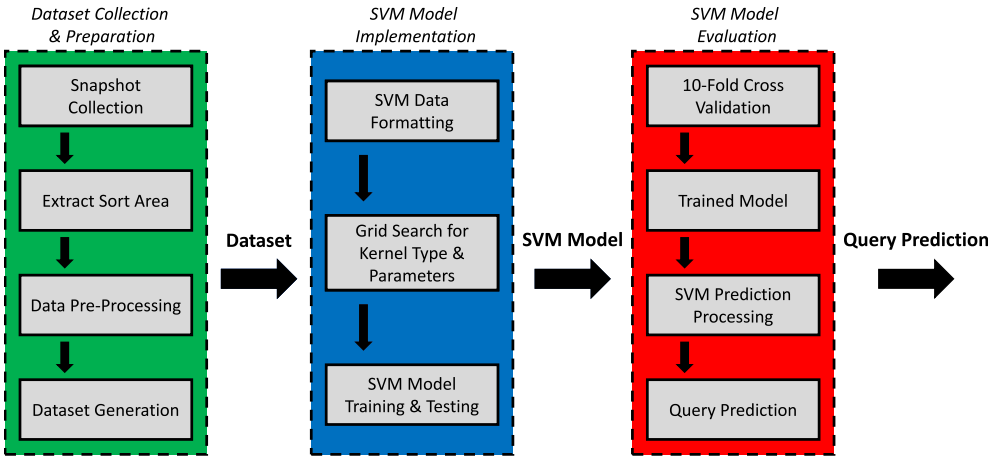

Fig. 2. Overview of proposed project.

isolate a segment from the snapshot called the sort area. We then perform pre-processing on the sort area using frequency analysis to extract artifact features, building our data set. Section 5 further discusses the details for this component.

The second component, SVM Model Implementation, builds our SVM model. To build our SVM model, we first format our dataset (from the first component) into data frame. Next, we use best kernel and hyperparameters from grid search to train and test our SVM model. Section 6 provides more detailed steps to build the SVM model.

The third component, SVM Model Evaluation, performs accuracy measurements for our proposed SVM model. Specifically, we used both 10-fold cross validation and manual input of unknown data for the evaluation purpose. Section 7 describes this evaluation.

4.1. Setup

We evaluated our framework on two representative DBMSes; a MySQL 8.0.29 instance and a PostgreSQL 12.11 instance deployed on a Ubuntu (v.20.04 LTS) server. We used the default configurations for each DBMS. MySQL used a 16 KB page size, 128 MB I/O buffer, and 2 MB sort area. PostgreSQL used a 8 KB page size, 128 MB I/O buffer, and 4 MB sort area. We consider these two DBMSes to be representative for several reasons. First, MySQL and PostgreSQL are the two popular open-source DBMSes. Second, MySQL builds index organized tables (IOT) on the primary key by default, whereas PostgreSQL use heap tables by default. Other major DBMSes use one of these two options. For example, Microsoft SQL Server and SQLite use IOTs by default. Oracle and IBM DB2 use heap tables by default.

4.2. Workload

We used the Star Schema Benchmark (SSBM) (O'Neil et al., 2009), scale 10 to populate each of our databases for evaluation; Table 2 summarizes the five tables in this benchmark. SSBM simulates a data warehouse environment. The synthetic data generator produces datasets at different scale by combining a realistic distribution of data (maintaining cross-column and data types).

We designed a workload with five query templates (Table 3). Each template was designed to perform a specific operation that we later seek to identify from an unknown snapshot. For each template, many queries can be generated by using different tables, columns, and values.

Table 2
SSBM scale 10 table sizes.

Table	Records	Size
DWDate	2556	0.4 MB
Supplier	20K	2 MB
Customer	300K	34 MB
Part	800K	84 MB
Lineorder	60M	5.6 GB

5. Dataset Collection & Preparation

The first component in our framework has four main steps: 1) Snapshot Collection, 2) Extract the Sort Area, 3) Data Pre-Processing, and 4) Dataset Generation.

5.1. Snapshot Collection

Each DBMS process snapshot collected contained the artifacts after running an individual query from our custom-built workload in Table 3. Before running each query, we restarted the DBMS, and snapshots were collected after the query finished execution. Snapshots were collected using ProcDump v1.2 (Russinovich and Richards, 2021).

For each query template in Table 3, we generated 100 queries (i.e., a total of 500 queries). Therefore, we collected a total of 500 memory snapshots for each DBMS. Each MySQL process snapshot size was 2.1 GB, while each PostgreSQL snapshot size was 159 MB.

5.2. Extract the Sort Area

After collecting our snapshots, we extracted the portions of each snapshot that was of interest to us, the sort area, to simplify our data processing. To achieve this, we defined the boundaries of the sort area within the snapshot. As discussed in Section 3, the DBMS memory areas for each process maintain consistent positions as long as the DBMS keeps its same memory configurations. Therefore, we analyzed several snapshots (for each DBMS) to find these boundaries. There are two main process areas that contain DBMS data: the I/O buffer, and the sort area. The I/O buffer contains the DBMS pages, which store the full table records. The sort area contains our manipulated query results, i.e., not the full table records. Therefore, we identified memory area boundaries by searching for entire records and our manipulated results. While we do not use the data stored in the I/O buffer, we identified this region to confirm

Table 3
Custom-designed query workload to demonstrate different query processing operations.

Operation	Summary	SQL Template
Index Sort	A record(s) is retrieved by traversing a (typically) B-Tree index to find a pointer(s) that directly references the record(s).	SELECT * FROM [table_name] ORDER BY [indexed_column]
File Sort	File sort is used when a sorting operation can't be performed using index access	SELECT * FROM [table_name] ORDER BY [non_indexed]
Join	Two nested for-loops, merge join or hash join	SELECT [table_x & table_y] FROM [table_x] JOIN [table_y] ON [table_x.ID] = [table_y.ID]
Filter	Filter rows based on a WHERE clause condition.	SELECT * FROM [table_name] WHERE [where_condition]
Aggregate	It often used with GROUP BY clause to group values into subsets.	SELECT [column], [aggregate_cond.] FROM [table_name] GROUP BY [column]

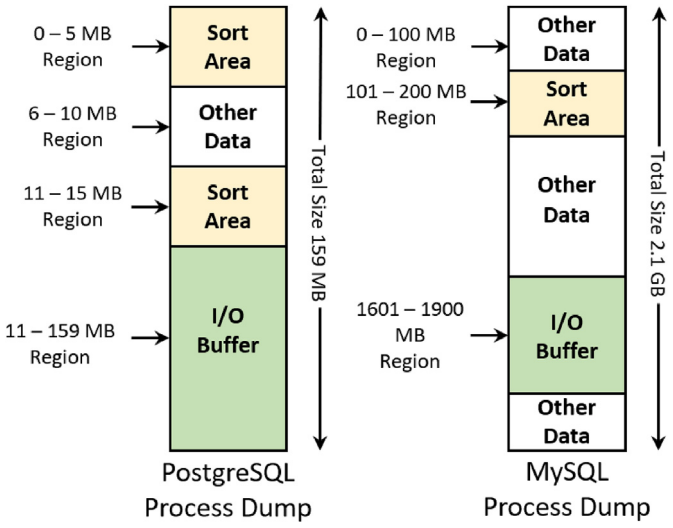

Fig. 3. Process dump of PostgreSQL & MySQL.

our results. Since the DBMS configurations did not change, boundaries remained constant for all snapshots. For each snapshot, we extracted the sort area using these boundaries.

Fig. 3 summarizes the sort area and I/O buffer boundaries we identified for each DBMS. For PostgreSQL, we identified the sort area in 0 MB – 5 MB memory region from queries containing ORDER BY and/or WHERE clauses and 11 MB – 15 MB memory region from queries containing GROUP BY clauses and/or JOIN conditions. For MySQL, we identified the sort area in the 101 MB – 200 MB for all queries. We used the Linux 'Split' utility to extract the sort area accordingly.

5.3. Data Pre-Processing

From each of the sort areas we extracted, we next determined the ASCII code frequency and the features from these frequencies. To determine the ASCII code frequencies, we considered each byte (with decimal values from 0 to 255) and counted the frequencies. Next, we built a histogram using the byte frequencies.

We observed that the NULL ASCII character (decimal value 0) was used only as padding between data values or to delimit the end of values in both MySQL and PostgreSQL process snapshots. Therefore, we used this feature, which we termed *fractions*, to capture the empty spaces between data stored in the sort area. Next, we collected the sequences of bytes (decimal values 1–255) between each fraction; this represented another feature, which we termed *bytes*. We considered each of the operations listed in Table 3 as a class. Hence, we have a total of five classes for our model: Index Sort, File Sort, Join, Filter, and Aggregate. We generated a feature vector for the byte frequencies and a feature vector for the fraction frequencies.

5.4. Dataset Generation

Using our feature extraction and classification, we generated a dataset for each DBMS. Each dataset stored three columns: the fractions (feature vector), bytes (feature vector), and the class label. A row was generated for each snapshot that was collected; we generate a dataset containing 500 rows for each DBMS.

We used the Python data visualization library Seaborn to display the data distribution in Fig. 4 & Fig. 5. These figures visualize the fractions and bytes features of our datasets through a cloud of points.

Fig. 4 shows the data distribution of MySQL dataset. In this figure, file sort, index sort, filter, join, and aggregate data points are represented by red, lime, green, blue, and purple color, respectively. Here, the x-axis represents the fractions values and the y-axis represents the bytes values. The figure shows that file sort data points have no overlap with other data points. File sort does an extra sorting phase to generate result because it does not use index of the database table. Thus, the operation is different from the other four types of operation shown here. Hence, its data pattern in sort area region is different, and the data points are easily distinguishable. As index sort and in some cases (i.e., according to query plan)

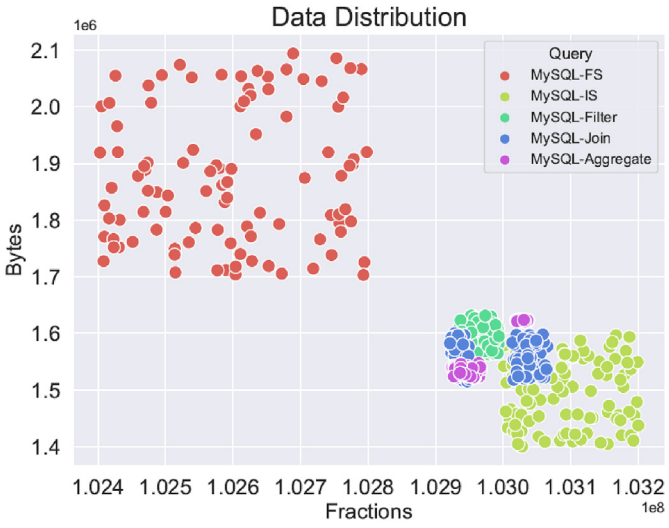

Fig. 4. Data distribution of MySQL dataset.

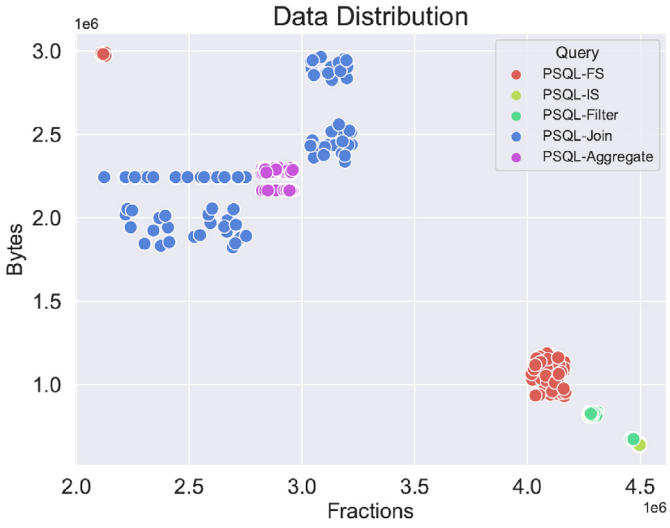

Fig. 5. Data distribution of PostgreSQL dataset.

filter, join, and aggregate operation use index of the database table to generate result, there are some overlapping data patterns in their sort area region. Therefore, the figure shows that data points related to these operations have some overlaps.

Fig. 5 shows the data distribution of PostgreSQL dataset. In this figure, file sort, index sort, filter, join & aggregate data points are also represented by red, lime, green, blue, and purple color, respectively. Similar to Figs. 4 and 5 shows that file sort data points have no overlap with other data points. The reason for this similarity to the MySQL is that PostgreSQL performs an extra sorting phase for the file sort operation to generate the result. Thus, its data pattern in the sort area region is different from other four types of operations, and the data points are easily distinguishable. Since index sort, join, and aggregate operations (and in some cases filter) use an index to generate the result, there are some overlapping data patterns in their sort area region. The figure shows that index sort and filter data points have overlaps with each other. Furthermore, join and aggregate data points are situated near to each other.

Overall, we can observe in both Figs. 4 and 5 that some data points overlap with each other making it difficult to distinguish them manually. Hence, machine learning model is useful to separate these types of categories from any dataset.

6. SVM Model Implementation

The second component of our framework has three main steps: 1) SVM Data Formatting, 2) Grid Search for Best Kernel & Parameters, and 3) SVM Training & Testing.

6.1. SVM Data Formatting

The SVM Model requires data to be loaded into memory, which we achieved using DataFrame objects. We converted our datasets from Section 5 into DataFrame objects using the Python based Pandas library.

6.2. Grid Search for Best Kernel & Parameters

To achieve the best result for the SVM model, we searched for the appropriate SVM kernel and hyperparameters (discussed in Section 3.1) for our dataset. There are several techniques and algorithms available to find the best kernel and hyperparameters. Two common search techniques are grid search and random search. Given the modest size of our dataset, we used grid search. Grid search produces more accurate results for smaller datasets, whereas random search is more suitable for larger datasets at the expense of accuracy. Grid search tests all values within a given range, whereas random search randomly selects a subset of the values to be tested.

For the MySQL dataset, our model performed best with Radial Basis Function (RBF) kernel: cost C = 300 and gamma = 2. For the PostgreSQL dataset, our model performed best with Radial Basis Function (RBF) kernel: cost C = 600 and gamma = 0.1.

6.3. SVM Training & Testing

To build our SVM model, we divided each dataset into two parts: training data and testing data. The SVM model uses training data to learn classification. The test data is unknown to the model and only used to test the model. From MySQL and PostgreSQL dataset, we use 80% data for training and 20% data for testing the model. We also random shuffled the rows to get the different training and testing sets. We scaled our training and testing set using the StandardScaler function from scikit-learn (Scikit-Learn). Scikit-learn is an open-source library written in Python. Scaled data helps the model to learn and understand the problem easily. Next, we fit our train set into the SVM model using RBF kernel and used cost C = 300, gamma = 2 for MySQL and cost C = 600, gamma = 0.1 for PostgreSQL.

7. SVM Model Evaluation

To evaluate the SVM model's performance on our dataset, we use four standard performance metrics: Precision, Recall, F1-Measure, and Accuracy.

Precision

Precision is the fraction between the true positives (TP) and sum of the true positives and false positives (FP).

$$Precision = \frac{TP}{TP + FP}$$

Recall

Recall is the fraction between the true positives and sum of the true positives and false negatives (FN).

$$Recall = \frac{TP}{TP + FN}$$

F1-measure

F1-Measure combines precision and recall to provide an overall measure of a model's accuracy. It is defined as the harmonic mean between precision and recall.

$$F1{-}Measure = 2{*}\frac{Precision{*}Recall}{Precision + Recall}$$

Accuracy

Accuracy is defined as the number of correctly predicted classes out of all the classes.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

Macro average

Macro average is the arithmetic mean of individual classes' precision, recall, and F1-measure scores.

Weighted average

Weighed average is the total number true positive of all classes divided by total number of objects in all classes.

Tables 4 and 5 summarize the query prediction results from MySQL and PostgreSQL memory snapshots using our SVM classifier. On average, our model achieved a recall of 93% for MySQL and 90% for Post-greSQL. Besides, our macro-averaged F1-measure was 93% for MySQL and 90% for PostgreSQL. For MySQL, we achieved the highest recall score with aggregate, file sort and filter and highest F1-Measure with file sort and filter. Moreover, for PostgreSQL, we achieved the highest recall score with aggregate, and file sort and highest F1-Measure with file sort. Finally, our model attained an accuracy of 92% and 90% on MySQL and PostgreSQL datasets, respectively.

We used 10-fold cross-validation to evaluate the performance of our model. In the cross-validation method, a dataset is divided into k non-overlapping folds (here, $k = 10$). Each of the k folds is used as test set and remaining folds are the training set. Hence, a total of k number of times model's performance are evaluated. This method is mainly used in machine learning to evaluate the skill of a machine learning model on unseen data. We used cross-validation libraries from scikit-learn to evaluate our model's performance with 10-fold cross-validation.

Table 6 presents the 10-fold cross-validation results for MySQL and PostgreSQL. Overall, we achieved better result with MySQL dataset where highest accuracy was 100% in fold number 8. For PostgreSQL dataset, we achieved highest accuracy, i.e., 90% in fold number 2, 3, & 7. We achieved better result with MySQL because it has less overlapping data points (Fig. 4) in the dataset than the PostgreSQL dataset.

Confusion matrix

We also use confusion matrix to visualize our model's performance. A confusion matrix is a specific table that visualizes the performance of machine learning algorithms. Each row of the matrix represents predicted class and each column of the matrix represents actual class, or vice versa. The matrix table compares actual values as true and false and predicted values as positive and negative.

Table 4
Classifier results for MySQL.

Query Type	Precision	Recall	F1-Measure
Aggregate	0.78	1.00	0.88
File Sort	1.00	1.00	1.00
Filter	1.00	1.00	1.00
Index Sort	1.00	0.84	0.91
Join	0.87	0.80	0.83
Macro Avg.	0.93	0.93	0.93
Weighted Avg.	0.93	0.92	0.92

Table 5
Classifier results for PostgreSQL.

Query Type	Precision	Recall	F1-Measure
Aggregate	0.95	1.00	0.97
File Sort	1.00	1.00	1.00
Filter	0.88	0.67	0.76
Index Sort	0.71	0.89	0.79
Join	1.00	0.96	0.98
Macro Avg.	0.91	0.90	0.90
Weighted Avg.	0.91	0.90	0.90

Table 6
10-Fold cross-validation for MySQL and PostgreSQL.

Fold	MySQL Accuracy	Postgres Accuracy
1	92%	87%
2	87%	90%
3	95%	90%
4	90%	87%
5	88%	84%
6	95%	87%
7	92%	90%
8	100%	89%
9	95%	87%
10	87%	87%

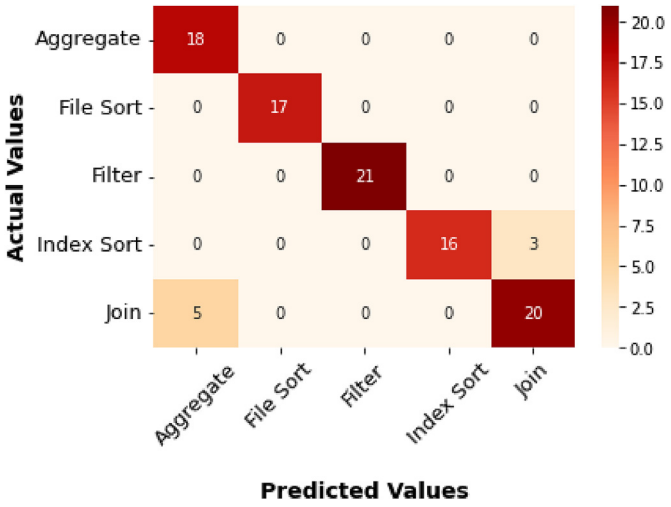

Fig. 6. Confusion matrix for MySQL dataset.

Figs. 6 and 7 display the confusion matrices for MySQL and PostgreSQL, respectively. Here, the rows represent the actual value and the columns represent the predicted value from our model. For MySQL, we observed inaccuracies for the join query prediction. Our model correctly predicted 20 out of 25 instances (Fig. 6). It misclassified the other five operations as aggregate operations. In the case of index sort query prediction, our model correctly predicted the operation in 16 out of 19 instances and misclassified the other three instances as join operations. The model predicted the other operations (i.e., Aggregate, File Sort, Filter) correctly in all instances.

For PostgreSQL, we observed inaccuracies for the filter operation prediction. Our model correctly predicted 14 out of 21 instances (Fig. 7). It misclassified the other seven operations as index sort operations. In the case of index sort query prediction, our model correctly predicted the operation in 17 out of 19 instances and misclassified the other two instances as filter operations.

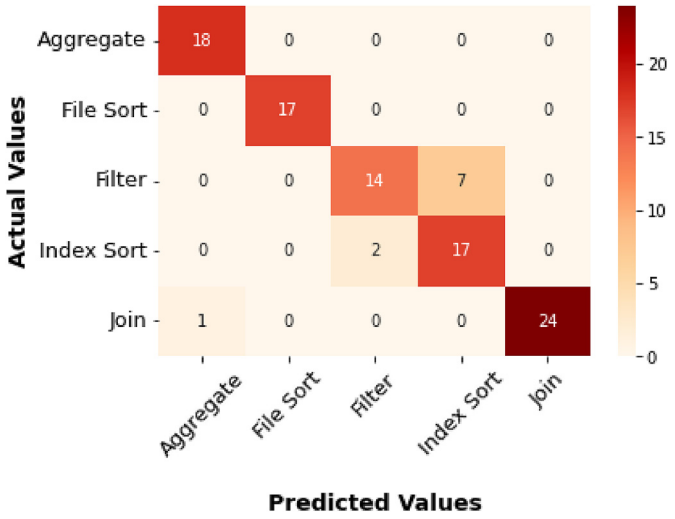

Fig. 7. Confusion matrix for PostgreSQL dataset.

Furthermore, for the join query prediction, our model correctly classified the operation in 24 out of 25 instances and misclassified it as aggregate in one instance. The model predicted rest 2 operations (i.e., Aggregate, File Sort) correctly in all instances.

Overall, our model achieved the better result for the MySQL dataset in all our evaluation metrics. Our model performed better with MySQL dataset because we observed fewer overlapping samples in Fig. 4 than we did for the PostgreSQL dataset in Fig. 5.

To fully reverse engineer the query, the table names must also be identified. In the I/O buffer, each page of the table has an object ID which corresponds to the plain text table name. By identifying table's object ID in process snapshot, it is possible to know the table names that were used in the query plan. Developing and testing the method to identify the tables that were used in the query plan is beyond the scope of this paper.

8. Experiments

The purpose of this experiment is to demonstrate the accuracy of the trained model in our framework given unknown snapshots from a MySQL and PostgreSQL instance.

8.1. Procedure

For this experiment, we used the same setup and DBMS configurations as those described in Section 4. For our workload, we generated five queries for each of the five query templates in Tables 7 and 8. This resulted in a total of 25 queries for each DBMS.

After running each of the queries, we collected a DBMS process snapshot, which served as our new "unknown" data to run against our SVM model. For each snapshot, we followed each of the steps described in Section 5. This included the data collection and preparation, extraction of the sort area, and analysis of the ASCII code frequency to collect the feature vectors (factions & bytes). Finally, we passed these features to our trained SVM model.

8.2. Results

Tables 7 and 8 show our experimental results. Our trained model predicted query activity from the unknown snapshots based on the knowledge it gained in Sections 5 - 7.

8.2.1. MySQL

For MySQL, we achieved the correct result for #1) index sort, #2) file sort, and #3) join operations for all five queries. For #4) filter operation, we achieved the correct result 4 out of 5 times. Our model incorrectly predicted a join operation for one of the filter operations. Finally, the model failed to correctly predict all of the queries in #5) aggregates. For these queries, the model incorrectly showed the output as a filter operation one time and as join operation four times.

8.2.2. PostgreSQL

For PostgreSQL, we achieved the correct result for #1) index sort, #2) file sort, and #3) join operations for all five queries. For #4) filter operation, we achieved the correct result 4 out of 5 times. Our model incorrectly predicted an index operation for one of the filter operations. Finally, for #5) aggregates, the model correctly predicted the operation as aggregate 3 times. It incorrectly predicted a join operation and a filter operation in other two aggregate experiments.

8.2.3. Discussion

Overall, we found incorrect results in some experiments from experiments #4) filter operation and #5) aggregate operation for both MySQL and PostgreSQL. This is due to the fact that we observed overlapping data points (discussed in Section 5) between these misclassified operations and the actual operations.

9. Conclusion

In this paper, we presented a novel framework to predict query activity from a memory snapshot using a trained SVM model. While, we evaluated our approach for two representative DBMSes,

Table 7
Experimental result of our trained model on MySQL data.

Exp.	Experimental Query	Operation	Result	Remarks
01	SELECT * FROM [table_name] ORDER BY [indexed_column]	Index Sort	Index Sort ⇒ 5/5	Correctly predicted in all 5 experiments
02	SELECT * FROM [table_name] ORDER BY [non_indexed]	File Sort	File Sort ⇒ 5/5	Correctly predicted in all 5 experiments
03	SELECT [table_x & table_y] FROM [table_x] JOIN [table_y] ON [table_x.ID = table_y.ID]	Join	Join ⇒ 5/5	Correctly predicted in all 5 experiments
04	SELECT * FROM [table_name] WHERE [where_condition]	Filter	Filter ⇒ 4/5 Join ⇒ 1/5	Correctly predicted in 4 experiments
05	SELECT [column], [aggregate_cond.] FROM [table_name] GROUP BY [column]	Aggregate	Filter ⇒ 1/5 Join ⇒ 4/5	Incorrectly predicted in all 5 experiments

Table 8
Experimental result of our trained model on PostgreSQL data.

Exp.	Experimental Query	Operation	Result	Remarks
01	SELECT * FROM [table_name] ORDER BY [indexed_column]	Index Sort	Index Sort ⇒ 5/5	Correctly predicted in all 5 experiments
02	SELECT * FROM [table_name] ORDER BY [non_indexed]	File Sort	File Sort ⇒ 5/5	Correctly predicted in all 5 experiments
03	SELECT [table_x & table_y] FROM [table_x] JOIN [table_y] ON [table_x.ID = table_y.ID]	Join	Join ⇒ 5/5	Correctly predicted in all 5 experiments
04	SELECT * FROM [table_name] WHERE [where_condition]	Filter	Filter ⇒ 4/5 Index ⇒ 1/5	Correctly predicted in 4 experiments
05	SELECT [column], [aggregate_cond.] FROM [table_name] GROUP BY [column]	Aggregate	Aggregate ⇒ 3/5 Join ⇒ 1/5 Filter ⇒ 1/5	Correctly predicted in 3 experiments

MySQL and PostgreSQL, our approached is designed to work for all DBMSes that maintain a similar memory architecture. Finally, we performed experiments that tested our trained model against a series of unknown snapshots for both MySQL and PostgreSQL. Results from these experiments showed that our model achieved an accuracy of 92% and 90% on MySQL and PostgreSQL datasets, respectively.

Future work for this project will explore the lifetime of memory artifacts. Our experiments collected a process snapshot after each query. This approach would be costly in a monitoring environment and forensic investigator may not have that granularity at their disposal. From a preliminary analysis, we observed that larger sets of artifacts (e.g., those produced by file sort or join operations) survived in a snapshot after running a series of queries. Whereas, a query that produced a smaller quantity of artifacts (e.g., index sort operations) did not survive in a snapshot containing a series of queries. Our goal is to formalize how long these artifacts survive and determine the accuracy of partial query artifacts in our approach.

Another larger vision for this project is to develop formal methods for audit log verification. For example, a database administrator (or an attacker that has gained such elevated privileges) can not only tamper with the audit logs, but bypass them completely by disabling the logs, which is a legitimate operation for something such as bulk loading. If such a malicious user were to disable the logs, perform a malicious query, and then re-enable the logs, their activity would be completely hidden from the logs. We propose to use the work in this paper to identify such missing activity.

Acknowledgments

This work was partially funded by the Louisiana Board of Regents Grant LEQSF (2022–25)-RD-A-30.

References

Case, A., Richard III, G.G., 2017. Memory forensics: the path forward. Digit. Invest. 20, 23–33.
Case, A., Maggio, R.D., Manna, M., Richard III, G.G., 2020. Memory analysis of macOS page queues. Forensic Sci. Int.: Digit. Invest. 33, 301004.
Cervantes, J., Li, X., Yu, W., Li, K., 2008. Support vector machine classification for large data sets via minimum enclosing ball clustering. Neurocomputing 71 (4–6), 611–619.
Huang, S., Cai, N., Pacheco, P.P., Narrandes, S., Wang, Y., Xu, W., 2018. Applications of support vector machine (svm) learning in cancer genomics. CANCER GENOMICS PROTEOMICS 15 (1), 41–51.
Karresand, M., Shahmehri, N., 2006. Oscar—file type identification of binary data in disk clusters and ram pages. In: IFIP International Information Security Conference. Springer, pp. 413–424.
Li, W.-J., Wang, K., Stolfo, S.J., Herzog, B., Fileprints, 2005. Identifying file types by n-gram analysis. In: Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop. IEEE, pp. 64–71.
Li, Q., Ong, A., Suganthan, P., Thing, V., 2011. A novel support vector machine approach to high entropy data fragment classification. In: Proceedings of the South African Information Security Multi-Conf. SAISMC), pp. 236–247.
Manna, M., Case, A., Ali-Gombe, A., Richard III, G.G., 2021. Modern macOS userland runtime analysis. Forensic Sci. Int.: Digit. Invest. 38, 301221.
Manna, M., Case, A., Ali-Gombe, A., Richard III, G.G., 2022. Memory analysis of .NET and .NET core applications. Forensic Sci. Int.: Digit. Invest. 42, 301404.
McDaniel, M., Heydari, M.H., 2003. Content based file type detection algorithms Proceedings of the. In: 36th Annual Hawaii International Conference on System Sciences. IEEE, p. 10, 2003.
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.
Russinovich, M., Richards, A., 2021. Procdump v1.2. In: https://docs.microsoft.com/en-us/sysinternals/downloads/procdump.
Scikit-Learn. Support vector machines library from scikit-learn. URL. https://scikit-learn.org/stable/modules/svm.html.
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., et al., 2015. Database forensic analysis through internal structure carving. Digit. Invest. 14.
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., Rasin, A., Malik, T., Heart, K., Jehle, H., Grier, J., 2017a. Database forensic analysis with dbcarver. In: CIDR 2017. 8th Biennial Conference on Innovative Data Systems Research.
Wagner, J., et al., 2017b. Carving Database Storage to Detect and Trace Security Breaches. DFRWS.
Wagner, J., Rasin, A., Heart, K., Malik, T., Grier, J., 2020. Df-toolkit: interacting with low-level database storage. Proceed. VLDB Endowment 13 (12), 2845–2848.