Ransomware Detection in Databases through Dynamic Analysis of Query Sequences

Christoph Sendner, Lukas Iffländer, Sebastian Schindler, Michael Jobst,
Alexandra Dmitrienko, Samuel Kounev
University of Würzburg, Germany
firstname.lastname@uni-wuerzburg.de

***Abstract*—Ransomware is an emerging threat that imposed a $ 5 billion loss in 2017, rose to $ 20 billion in 2021, and is predicted to hit $ 256 billion in 2031. While initially targeting PC (client) platforms, ransomware recently leaped over to server-side databases—starting in January 2017 with the MongoDB Apocalypse attack and continuing in 2020 with 85,000 MySQL instances ransomed. Previous research developed countermeasures against client-side ransomware. However, the problem of server-side database ransomware has received little attention so far. In our work, we aim to bridge this gap and present DIMAQS (Dynamic Identification of Malicious Query Sequences), a novel anti-ransomware solution for databases. DIMAQS performs runtime monitoring of incoming queries and pattern matching using two classification approaches (Colored Petri Nets (CPNs) and Deep Neural Networks (DNNs)) for attack detection. Our system design exhibits several novel techniques like dynamic color generation to efficiently detect malicious query sequences globally (i.e., without limiting detection to distinct user connections). Our proof-of-concept and ready-to-use implementation targets MySQL servers. The evaluation shows high efficiency without false negatives for both approaches and a false positive rate of nearly 0%. Both classifiers show very moderate performance overheads below 6%. We will publish our data sets and implementation, allowing the community to reproduce our tests and results.**

***Index Terms*—Database, Ransomware, Colored Petri Nets, Machine Learning, Attack Modeling, MySQL**

I. Introduction

In today's era of digital transformation, data has become more critical than ever before. The amount of data we produce daily is astonishing—every day, hundreds of millions of people are taking photos, make videos, and exchange messages. Furthermore, data is not only a valuable asset for users nowadays but has also become wheels for the digital transformation vehicle that requires a lot of data for AI-enabled services and training machine learning models. Given such trends, the importance of database security is hard to overestimate—the rapid growth of the data volume stored in the databases in cloud environments and enterprise data centers makes them attractive attack targets.

Traditionally, attacks on data aim at undermining confidentiality and authenticity. More recently, however, attacks against the availability of data, services, and users became more common. Modern attackers deploy ransomware, malicious software that claims to have encrypted all data—while in numerous instances deleting the data—and requires the victim to pay a ransom for the decryption key. The financial loss from ransomware is significant—it reached 5 billion USD in 2017, rose to 20 billion by 2021, and is predicted to hit 256 billion by 2031 [27], [2].

While the first ransomware attacks targeted client platforms (e.g., information stored in users' files), such attacks also leaped over to server-side databases. In January 2017, tens of thousands of MongoDB servers suffered from an attack called MongoDB Apocalypse [8], followed by a second attack wave targeting MySQL servers [35]. A more recent large-scale campaign hit 85,000 MySQL instances in 2020 [23].

One reason for the rise of the server-side ransomware is that enterprises can afford to pay higher ransoms than private users. As a comparison, the typical ransom amount for regular users lies in the range of a few hundred dollars. At the same time, businesses can pay more—for instance, Colonial Pipeline paid 4.4 million USD of ransom [1]. Second, recently, researchers and antivirus companies focused on countermeasures against client-side ransomware. However, to date, no solutions exist against ransomware targeting database servers, thus making databases easy attack targets.

Existing anti-ransomware solutions limit themselves to client-side ransomware detection and follow two dominant strategies: Signature-based detection of malicious binaries and runtime monitoring and behavioral analysis for anomaly detection. The first one builds upon the detection of malicious binaries and is typically used by antivirus vendors. In contrast, the second strategy originates from research papers [10], [9] and relies on runtime monitoring of file accesses and the detection of malicious activity based on heuristics. Unfortunately, both strategies are not suitable for detection of server-side ransomware attack scenarios, where attackers connect to the database remotely and, hence, there is no local malicious binary to detect. Furthermore, monitoring at the file system level for abnormal activity is not appropriate either, due to a lack of correlation between attacker activity and file access patterns.

Dealing with ransomware in databases faces the following challenges. Foremost, one couldn't straightforwardly apply existing Intrusion Detection Systems (IDS) such as [15], [32], as they typically rely on detection of single malicious queries, while here an attacker would typically use a sequence of queries where each isolated query is seemingly benign (e.g., listing available tables, or entering a data record with instructions for ransom payments). Second, existing methods for profiling using activities ([5], [33], [6]) won't help either, as an attacker can potentially use multiple accounts to insert

different queries of a malicious query sequence. Third, an attacker could interleave malicious queries from the attack sequence with benign queries, which stretches attack duration, yet significantly complicates detection. Fourth, even though DB admins generally backup data more often than regular users, the updates in between backups remain vulnerable, and restoring backups results in undesirable downtime. Finally, potentially useful machine learning approaches require data for training. However, there exist no data sets that include query sequences of server-side ransomware.

Contributions. In this paper, we aim to improve security of databases and tackle the above discussed challenges imposed by server-side ransomware. In particular, we make the following contributions:

- We design DIMAQS (Dynamic Identification of Malicious Query Sequences), a novel intrusion detection framework for databases that, in contrast to existing IDSes that focus on detection of isolated malicious queries, can detect malicious query sequences. In its heart, DIMAQS has a classifier that analyzes arriving queries along with preceding ones (within a pre-defined time window) and classifies them as being malicious or benign. We instantiated DIMAQS classifier using two methods: (i) Colored Petri Nets (CPNs) and (ii) supervised Machine Learning (ML). The CPN-based classifier models the series of attack events as transitions in between the states of the CPN and detects if any of the arriving queries transits the system into an attack state. Our CPN leverages several novel techniques (dynamic color creation, token merging, and token expiration) we introduced to reduce system representation complexity and improve detection efficiency. The second classifier relies on a Recurrent Neural Network (RNN), the deep neural network-based detection technique that does not require labor-intensive engineering of features. To train our RNN, we collected a first dataset of malicious query sequences, which is available at https://github.com/sss-wue/DIMAQS.
- We implemented an end-to-end attack detection tool for MySQL databases in a form of a MySQL plugin that is easily installable on existing MySQL servers—thus preserving compatibility with legacy software—while only imposing a very moderate performance overhead of 6% for the ML-based classifier and only 0.2% for the CPN-based classifier. The tool can perform system-wide monitoring and detect malicious sequences injected through several user sessions and interleaved with benign queries, eliminating the most obvious evasion strategies.
- We evaluated ability of DIMAQS to detect ransomware attacks and compared effectiveness of both classifiers. Our experiments show that both classifiers perform pretty well: They can detect attacks with 100% success rate, and CPN-based classifier has no false positives, while ML-based has a very low rate of false positives of 0.003%. On the other hand, CPN-based approach requires manual effort of CPN engineering—the disadvantage which is eliminated with the ML-based approach. To summarize, CPN-based approach is more efficient and has superior detection performance at the cost of additional manual effort, while ML-based approach is more powerful when it comes to adaptation to new attack scenarios.

We also note that, while the design of DIMAQS was motivated by server-side ransomware attacks, its ability to detect malicious query sequences is likely applicable to other advanced attack scenarios that involve query sequences rather than isolated malicious queries (e.g., an advanced SQL injection aiming at remote code execution [12]). Adaptation of a CPN-based classifier to new attack scenarios will require manual effort of CPN engineering, while an ML-based approach has the advantage of being easily adaptable without such a manual effort by simply retraining the ML model on a new corpus of attack data. We leave evaluation of applicability of DIMAQS for detection of other attack scenarios for future work.

II. Framework

We first introduce the attack scenario and the adversary model guiding our design decisions. Next, we describe the system architecture, its components and provide details of implementation.

A. Attack Scenario

Our attack scenario is inspired by real-world attack campaigns [35], [23]. The attack begins when an attacker connects remotely to the database using a TCP connection and gains root access (e.g., brute-forcing the 'root' password). Next, they enumerate the database data by retrieving the list of the databases present. Subsequently, the attacker creates a new table with an arbitrary name (e.g., with the name 'WARNING'), either in a new database (e.g., named 'PLEASE_READ') or in an already existing database. This table includes a ransom message containing a contact email address as well as payment instructions to a bitcoin address. Finally, the attacker deletes the databases on the server and disconnects.

While the scenario above describes attack steps recorded in real-world attacks, we also accept permutations of attack steps.

B. Adversary Model

We make the following assumptions about the goal and the capabilities of the attacker. The attacker's goal is to destroy the available data and claim a ransom. We assume the remote attacker without physical access to the database. We trust the software running on the server, i.e., the attacker has no malicious software installed on the system. However, the attacker has full access to the network and can communicate with the Database Management Systems (DBMS) [5], [33], [6] without any restrictions. Furthermore, we assume an attacker with administrator-level privileges to the DBMS. This assumption often proves true in practice since the problem of weak or re-used passwords [18] is well known and has not been satisfactorily solved for decades. While two-factor authentication can resolve such issues, and most databases implement it, widespread usage is still rare. Alternatively, an

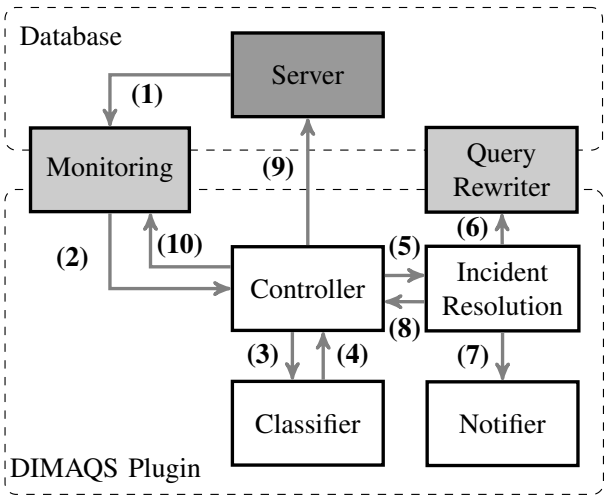


Fig. 1: System architecture of DIMAQS. Dark gray boxes are components provided by the database, light gray boxes are components that interface between DIMAQS and the database, and white boxes belong to DIMAQS itself.

attacker might exploit a security vulnerability like [14] to gain administrative privileges. Additionally, we assume an attacker can leverage multiple user accounts to access the DBMS and, thus, use several connections to evade detection.

We, however, do not assume administrator privileges of the attacker to the operating system. Furthermore, we leave DoS attacks out of our attacker model since an attacker with administrator privileges to DBMS can always cause a denial of service, e.g., through the creation of fake DBs or tables and exhausting DB's memory.

C. System Architecture

Figure 1 shows the DIMAQS system architecture. DIMAQS comprises six components: (i) `Monitoring`, (ii) `Classifier`, (iii) `Incident Resolution`, (iv) `Notifier`, (v) `Query Rewriter` and (vi) `Controller`. The `Monitoring` and `Query Rewriter` components use the query parser embedded in the database server. Hence, the figure shows them as belonging to both the DIMAQS plugin and the database server. In the following, we describe the role of every component in more detail.

Monitoring The `Monitoring` component monitors all incoming queries for potentially malicious query sequences. This module monitors all queries arriving through different connections, not specific to user sessions (c.f., Section II-B).

Classifier The `Classifier` component decides whether DIMAQS deems a query sequence as malicious. Since we consider query sequences, any viable type of `Classifier` keeps an internal state updated after each query. Further, `Classifier` have an inherent limit to the number of observable sequences—otherwise, misclassification would rise. Thus, we define a time window t, a system parameter of DIMAQS, in which an attacker can be detected. We instantiate `Classifier` using two methods—Colored Petry Nets (CPNs) and Machine Learning (cf. Sections III and IV, respectively).

Incident Resolution When an event in the `Classifier` component issues an action, it must be carried out by the `Incident Resolution` module. Possible actions are "create backup," "rewriting" and "create notification." `Incident Resolution` performs the rewriting of malicious queries as well as creates backups.

Create backup action. Whenever the system detects a potential attack, the `Incident Resolution` component will move the database or the table dropped by an attacker to a safe place instead of deleting it. The backup copy is invisible to users with and without admin rights (and, hence, from the attacker) so that an attacker cannot drop it again or even identify that such a backup exists. `Incident Resolution` uses a "rewriting" action to hide backed-up tables and databases from users. While performing such a move, `Incident Resolution` renames the protected tables to avoid name collisions.

Rewriting Action. Rewriting actions rewrite queries to exclude tables and databases created by DIMAQS. The `Query Rewriter` component performs these actions.

Notification action. The `Incident Resolution` component uses notification action to notify an administrator about a detected attack. The `Notifier` component performs this notification as described below.

Notifier The `Notifier` component informs about security incidents by emailing the DIMAQS administrator. The gathered information relevant to the incident is attached to the notification so that the administrator can evaluate the incident and respond accordingly (e.g., restore the deleted table).

Query Rewriter The `Query Rewriter` component rewrites queries to exclude tables and databases created by DIMAQS from query results. For a 'rewriting' action, the `Query Rewriter` receives the name of the table and, if applicable, the name of the database from the `Incident Resolution` component. If the queries are nested, the `Query Rewriter` extracts them into sub-queries, rewriting each sub-query separately. For instance, a query dropping a table will be rewritten to move the table to a safe storage space. This operation happens without any indication to the attacker. Additionally, some statements that list tables and databases will be rewritten to exclude the hidden information from query results.

Controller The `Controller` component connects all other DIMAQS components. It is the central element orchestrating the incoming query processing by other components, e.g., through the invocation of the `Classifier` component for state classification or the `Incident Resolution` component to initiate incident resolution upon attack detection.

D. Component Interaction

Figure 1 depicts the interaction between the components during query processing. DIMAQS classifies query sequences by sequentially processing queries. The `Classifier` internally tracks the DBMS state (i.e., the query sequences). The database server first receives the query and then notifies `Monitoring` (1). If `Monitoring` receives a potentially malicious query

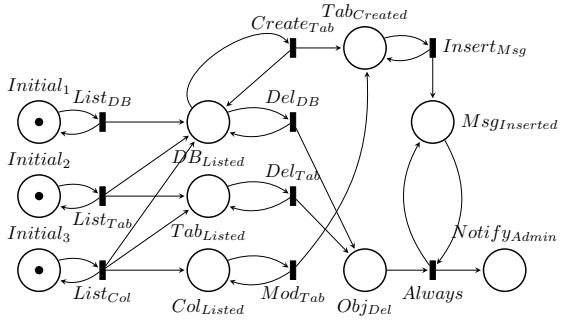
type, the Controller is notified (2). The Controller then forwards the relevant query to the Classifier (3) for evaluation. The Classifier returns the classification result to the Controller (4). There are two possible outcomes: the query's classification is either benign or part of a potential attack. In the former case, the Controller returns, and the server executes the query as-is (9). In the latter case, the query is considered malicious, and the Controller calls Incident Resolution (5), which in turn backs up dropped tables and rewrites the malicious query using Query Rewriter (6). It then invokes the Notifier to inform the administrator (7). The Controller then receives the rewritten "disarmed" query from Incident Resolution (8). The database server then executes the query (9). The Controller informs Monitoring when additional objects need to be observed (10), e.g., when a query creates new tables.

E. Component Implementation

DIMAQS design is generic and can be applied to different database technologies. For illustration, we have chosen to prototype it for MySQL servers—our implementation is realized as MySQL plugin compatible with MySQL server versions 5.7.x. To function, DIMAQS requires the mysqlservices library provided by the MySQL server. We chose the C++11 language for DIMAQS since it is the default language for MySQL plugins. DIMAQS consists of 4908 lines of code (LoC). In the following, we detail the implementation of DIMAQS modules.

MySQL Integration: The plugin is loaded during MySQL server start-up and registers itself as an auditing plugin. Per default, the MySQL server does not provide any event that returns the atomic values of database elements affected by INSERT, UPDATE, and DELETE queries. These queries are typically used by attacks like mimicry, e.g., for the insertion of ransom messages. To allow us to access the atomic values, we generate "before INSERT/UPDATE" triggers for every table. In these triggers, we execute a user-defined function, forwarding the values affected by the queries to the controller for evaluation. As detailed in the MySQL trigger syntax [11], a trigger becomes associated with a table named tbl_name. For other queries, the MySQL server plugin interface provides multiple notifications [11].

Monitoring: Additional triggers are required to access information that is not transparent to the DIMAQS plugin when using MySQL's audit features. Trigger creation occurs when loading the plugin, and existing triggers are recreated after server start-up since the database structure might have changed. Trigger creation within so-called "stored procedures" or "stored functions," the conventional concepts supported by the MySQL server, is not possible. Due to this limitation, the creation must be within the plugin code. The function dimaqs_plugin_init() performs the creation of the additional triggers and is called directly after initialization of the server and before entering the listening state. dimaqs_plugin_init() creates a trigger for every non-virtual database. Virtual databases contain read-only views and have no database files associated with them. Hence, the protection of virtual databases is not necessary.



States: $Initial_x$: initial states; $List_x$: objects listed, $Tab_{Created}$: table created; Obj_{Del}: object (database or table) deleted; $MSG_{Inserted}$: ransom message inserted; $Notify_{Admin}$: notification sent
Transitions: $List_{DB}$: list databases; $List_{Tab}$: list tables; $List_{Col}$ list columns; $Create_{Table}$: create table; $Drop_{Table}$: drop table; $Modify_{Table}$: modify table; $Insert_{Msg}$: insert ransom message

Fig. 2: The CPN used to classify database transactions. All arcs are weighted with a value of 1 token.

The INSERT and UPDATE triggers call eval_value(). Several values are passed to that function, namely (1) schema name, (2) table name, and (3) new column values. Using this structure, we can identify inserted/updated values.

Classifier The Classifier implementation depends on the used approach. We describe the implementations for a Petri-Net-based classifier in Section III and a ML-based classifier in Section IV-D.

Incident Resolution The Incident Resolution backs up dropped databases and deleted values. The renaming of databases is not trivial due to MySQL limitations. MySQL added a command to carry out a database renaming called 'RENAME DATABASE <database_name>.' However, this command was only active through a few minor releases before its discontinuation. The simplest way to rename a database is to move its tables to another database and recreate the affected triggers. The function renameTable() performs this renaming. If a database drop occurs, renameDatabase() calls the renameTable() for every table. For backup actions, a 'DROP DATABASE <db_name>' does not require rewriting. However, before executing, *renameTable* or *renameDatabase* back up the database tables.

Notifier The Notifier messages the administrator with all available information about the suspected attack. Administrator credentials are configurable.

Query Rewriter The Query Rewriter rewrites a query by adding a WHERE/AND condition to hide sensitive information or rewrites it entirely, e.g., for backup operations.

Controller The Controller is implemented using the *visitor* design pattern. This visitor extracts the nested statements from inside to outside. It then forwards each extracted query to Classifier.

III. Petri Net Classifier

Petri Nets are a commonly used mathematical modeling language for the description of distributed systems [28]. A Petri Net is a directed bipartite graph, in which nodes represent *places* and *transitions*, while edges, called *arcs*, connect either a place to a transition or a transition to a place, but never connect two places or two transitions directly. Transitions are events in the system, and places are conditions that need to be satisfied for the transition to fire.

Places may contain a discrete number of marks called *tokens*. Transitions *fire* if they are *enabled*, which is achievable by placing enough input tokens on the input places—i.e., places directly connected to the transition. Colored Petri Nets (CPNs) [19] enable support for tokens of different types, also known as *token colors*.

Extensions to CPNs. New for CPNs, we use token colors to attach runtime information to the tokens, such as timestamps, table names, and modified cell values. The token colors also provide additional information in the case of an incident. Since such token colors are dynamic and unbounded, conventional CPNs would be unable to represent all the possible states. Hence, we introduce extensions to CPNs to deal with dynamic information.

In particular, we extend CPNs with three new features. The first is the dynamic creation of colors for storing information inside the tokens. The second is the ability to merge tokens that are identical except for their timestamps. This extension improves performance and does not impede classification accuracy. The third extension allows for token expiration. Since each place in the CPN can have timeout information, this feature can be used to limit the time window of analyzed query sequences. The expiration timeout is a security parameter since, on the one hand, reducing it can minimize false positives but increases the chances of attacks that stretch over time to escape detection. On the other hand, increasing this value can decrease this risk while increasing the chance of false-positive detection. The observed attacks usually transpired during a few seconds, making low values an option for false-positive prone workloads.

The `Security Policy` component configures the CPN classifier describing its places, place actions, transitions, transition actions, transition conditions, and arcs. We depict the CPN constructed based on the observed attacks in Figure 2. All places and transitions have names, and the arcs have a weight of one token. Each place can trigger several place actions upon CPN transitions to the corresponding place. Transitions allow checking for the execution of a step in a malicious query sequence. They become active when the source place contains at least one token. Each transition triggers one transition action, representing conditions for incoming queries. For instance, they may specify the query type (e.g., query that lists tables) and the actual content of the query. A transition may also have an arbitrary number of transition conditions to evaluate the token data from the source place against the query values. Our policy includes only one transition condition to detect the insertion of a suspicious message (here, containing a cryptocurrency address).

Transitions fire when an action occurs that is specified as malicious by the `Security Policy` component. Note that no single action alone is enough to transit the CPN to the "attack detected" state. Typically, the sequence of actions is required, and their execution requires a specific order (defined by the CPN configuration) to reach the state that corresponds to attack detection. The policy is easily adaptable to include new attack signatures by modifying the Petri Net. While reconfiguration is a manual process, it is not cumbersome and can be accomplished in a reasonable amount of time.

Implementation We implemented the `Classifier` using our Petri Net implementation library libPetri comprising 1008 LoC. libPetri is a C++ library implementing the functionality of colored Petri Nets. It includes dynamic coloring, token timeout, and token merging features mentioned above. Since libPetri has been developed explicitly for DIMAQS, it carries no additional feature overhead.

libPetri keeps track of all active transitions. Since all our arcs in `Classifier` are weighted with the value one, active transitions have tokens on all input places. If the to-be-classified query matches the action attributed to an active transition, that transition fires. When transferring a token to a place with an associated action, that action executes with the corresponding parameters. Until completion of these actions, the `Classifier` does not accept additional queries.

IV. Machine Learning Classifier

To instantiate ML-based `Classifier`, we formulate the problem of ransomware detection as a binary classification problem of textual data and use a supervised learning technique for detection of malicious query sequences. We chose Deep Neural Nets (DNN) targeting text recognition as our ML technique because query sequences are fundamentally text sequences. In the following, we describe each of the steps in more detail.

A. Data Augmentation and Labeling

Deep Learning techniques demand a substantial amount of data for training. However, the challenge with the real-world data is that most DBMSs contain sensitive data (e.g., credit card information and passwords) and are not openly available. We employ a sliding window technique to extract query sequences of variable length and random starting points in query sequences to address this challenge. Since no malicious data sets are readily available either, we record a limited amount of malicous query sequences ourselves and then increase the volume of data by changing the order of malicious queries in sequences, by interleaving malicious queries in sequences with various amounts of benign queries, and by altering ransom payment instructions. The procedure of sliding window for benign data and the alteration of malicious data is done on-the-fly during training and evaluation. With that, we create two streams of accordingly labeled benign and malicious query sequences.

B. Data Pre-processing

The next step is to pre-process the sequences. A query of a sequence can have very complex forms with arbitrary data. This complexity is a problem for deep learning, as we usually map each data point to a single number that can then be processed for deep learning.

A common solution is to train a relation between data input and a corresponding number. However, this allocates a tremendous amount of memory capacity for text parts—like table names or textual values of updates—which have no useful information for our attack scenario.

As such, we opted to use one-hot encoding for our embedding layer. Here, we extract features from the query and encode them into a binary vector (i.e., feature vector). Focusing on features—not useless text parts—benefits both detection performance and resource impact.

In total, our feature vector has eleven features. Each feature can either be available (one in the vector) or not (zero in the vector). Ten features are related to a query type. Query types can range from table creations to data insertions and deletions. It is worth noting that the query types are mutually exclusive. For example, a query to update data cannot also be a query for data deletion. The important point is that we focus on the type of query and not specific values specified in the query, such as table names in the table deleting queries (in contrast to the Petri Net `Classifier`). This generalization allows us to detect generic attack patterns instead of focusing on specific attack instantiations.

The eleventh feature concerns the ransom message. Here, we use regular expressions to identify cryptocurrency addresses, such as Ethereum, Bitcoin, and Monero addresses. Please note that these regular expressions can easily be extended with new cryptocurrencies without retraining the model.

C. Model Training and Parameter Tuning

Query sequences are sequential data streams and largely depend on previous input (e.g., one can only alter a table that exists). As such, RNNs seem to be well-suited for classifying such data. Two classic types of RNNs are Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU), both retaining previous outputs for the subsequent calculations on a new input.

Model training is an iterative process that includes definition of the model architecture, model training and evaluation, and tuning model parameters for the next iteration. Here, one can alter the number and kind of layers that comprise the model. Further, the training and evaluation duration depends on the number of iterations (i.e., epochs). We can also alter the optimizers (i.e., Adam), their learning rates, and dropout factors of different layers.

The model is trained on pre-processed query sequences from the data stream generated as elaborated in Section IV-B. Ratio benign:malicious defines the ratio of benign and malicious queries in a sequence. We tune the query sequence length for training and evaluation separately. Thus, we explore the ratio of malicious sequences in training necessary for high effectiveness on unseen data.

Variable	Setting
Layer Type	Input, Embedding, LSTM, Output
#Hidden Units	LSTM:8, Output:1
Optimizer	Adam
Loss Function	Mean Squared Error
Learning Rate	0.15
Dropout	0.00
Minimum Sequence Length	20
Training Sequence Length	50
Evaluation Sequence Length	6,000
Iterations	200,000
Ratio benign:malicious queries in sequence	7:4

TABLE I: Model Hyper-parameters

We repeated the hyper-parameter tuning process 232 times to find configurations that yield good classification performance. Our best performance model consists of four layers: (i) input layer, (ii) embedding layer, (iii) LSTM layer, and (iv) output layer.

The *input layer* defines an interface between the raw data and an embedding layer. Here, the computer transforms the multiple binary vectors of a sequence into an internal matrix representation. The *embedding layer* transforms the binary vector matrix from the input layer into higher dimensional matrix representation—ready for the LSTM layer. The next *LSTM layer* has 8 hidden units. We utilize the Adam optimizer to update the internal weights of the RNN while aiming to optimize Mean Squared Error based on the loss function. The last *output layer* accumulates the outputs of the LSTM into a single value. This value represents the probability whether the query sequences within a specified Sequence Length are malicious or not.

Table I summarizes all the hyper-parameters of our model. We want to emphasize that while we train on a query sequence length of only 50 queries, we still achieve excellent results in the evaluation (cf. Section V-C) using a much longer sequence length of 6,000.

We recall that it took 232 various hyper-parameter combinations to find the optimal configuration for our model. The key takeaways of our extensive tests are the following: 1) Training on a smaller sequence length will increase the accuracy of the model in evaluation with longer sequence lengths. 2) Stacking multiple RNNs has only a marginal effect on detection performance, but impacts inference runtime negatively. 3) We did not see a performance improvement of GRUs compared to LSTMs.

D. Classifier Implementation

We implemented the ML-based `Classifier` using C++ to integrate with the other DIMAQS components. Further, we utilize the C++ library Torch to build, train, and infer the model. The entire implementation comprises 2,954 LoC.

We implement rotating models, since we defined a time window t for attack detection in Section II. We chose the sequence length as a time delimiter and hyper-parameter (i.e., number of queries, instead of time-frame) and reset the internal

state of our model after this pre-defined number of sequences is processed.

To eliminate a potential attack vector, where an attacker stretches their malicious sequence across the boundaries of the time window, we rotate three instances of the model with overlapping time windows. While one model instance classifies the sequences, another model also receives the data and updates its internal state, preparing a hand-off. The third instance is reset in the background.

V. Evaluation

In this section, we evaluate both classifier instantiations of DIMAQS. First, we explain our data sets. Then, we explore the effectiveness of Petri Net `Classifier` and the ML `Classifier`. Further, we compare the performance impact of both classifiers with standard benchmarks for DBMS. Last, we consider different security aspects of DIMAQS.

Testbed: We execute performance and security tests on a Lenovo P50 workstation laptop featuring a four Core i7-6700HQ CPU with 32 GB RAM and a 512 GB NVMe. For the software, we chose Arch Linux running with kernel 5.11.13 and a MySQL server 5.7.22 via Docker version 20.10.5. All benchmarks run directly on this hardware to eliminate possible network bottlenecks.

A. Data Sets

We employ three data sets during our evaluation. These are selected to test different aspects of DIMAQS. We will publish the data sets along with the paper to allow third parties to reproduce our tests and enable follow-up works to compare our results.

The first set (*malicious set*) includes malicious query sequences, which we generated ourselves using information about real-world attacks collected at [35], which contain real-world attack query sequences. Our resulting query set contains query sequence variations with an expected malicious classification, as well as their possible permutations (since an attacker may execute them in arbitrary order). The full test set contains 13,485 tests with 53,940 queries in total.

The second set (benign *Bibspace set*) is from the publication management system *Bibspace* [30], which was gathered over 40 days and contains a total of 52,085 queries.

The third set (benign *MediaWiki set*) is from a locally run *MediaWiki* [26] with the *Semantic MediaWiki* [31] plugin enabled, collected for 50 days and containing 2,514,764 queries.

B. Petri-Net Effectiveness Evaluation

In the following, we evaluate the precision of the Petri-Net classifier module. Thus, we evaluate whether a wrongful classification of benign queries as malicious (false positives) or malicious query sequences as benign (false negatives) occurs.

Security Policy: The execution policy for the `Classifier` is as described in Section II-C. Our policy is quite generic in the sense that we do not look for specific table or database names but instead detect the removal or renaming of any table or database. However, we are looking for specific patterns of cryptocurrency addresses.

False Negatives: We used the *malicious set* described in Section V-A to test for false negatives. After processing all the queries from the data set by our CPN, we achieved 100% attack detection rate and received no false negative results. This result confirms that our CPN correctly models each attack from our malicious data set including different cryptocurrency addresses.

False Positives: To test for false positives, we utilize the *Bibspace set* and the *MediaWiki* sets. They contain a total of 2,566,849 benign queries. The `Classifier` performs classification of every set. Afterward, the `Classifier` state shows if DIMAQS wrongfully detected attacks and how many false detections occurred. If tokens reach place N in `Classifier`, the amount of tokens represents raised alerts. To put the system to its limits, we disable the time window limit (i.e., token timeout) to increase the potential for false positives. In other words, we evaluate the entire sequence of queries in both datasets. We detected no false positives.

C. ML Effectiveness Evaluation

In this section, we evaluate the effectiveness of ML-based DIMAQS `Classifier`. Similarly to Section V-B, we evaluate false positives, false negatives, and F1-score.

Classification performance: We evaluated our model against our *malicious set*, *Bibspace set*, and *MediaWiki set*—dynamically combined into the streams as explained in Section IV-A. We use the F1-score as our performance metric. The F1-score is the harmonic mean of recall (true positives divided by all positives) and precision (true positives divided by all true positives plus false negatives). Our model achieves an F1-score of over 99.99%. We detected no false negatives of malicious sequences and only 93 false positives out of 2,932,190 samples. Thus, the model has a false negative rate of zero and a false positive rate of 0.003%.

Time window t: We want to identify the query sequence length threshold producing a low amount of false positives while remaining large enough for attack detection. The larger the sequence length, the higher the chance for false positives. Moreover, the shorter the sequence length with higher false positives is, the easier an attacker can avoid detection. For this test, we do not rotate model instances and continuously issue sequences until detecting a rise of false positives. For this test, we use both benign data.

We observed a rise in false positives after 3,357,746 queries in a single sequence. Processing 3,357,746 queries takes 8.82 minutes in our testbed. We see a time window of under eight minutes to be sufficient to detect an attacker without the rise of false positives. In DIMAQS we reset the classifying model after 7.5 minutes with an overlap of 2.5 minutes to the fresh model instance (see Section IV-D).

D. Performance Evaluation

In this section, we evaluate DIMAQS's performance overhead. We apply the model rotation in the performance eval-

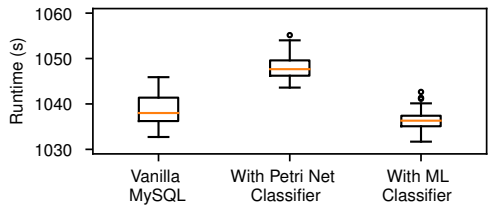


Fig. 3: Box-plot for TPC-H Runtimes with and without the Different Plugin Implementations

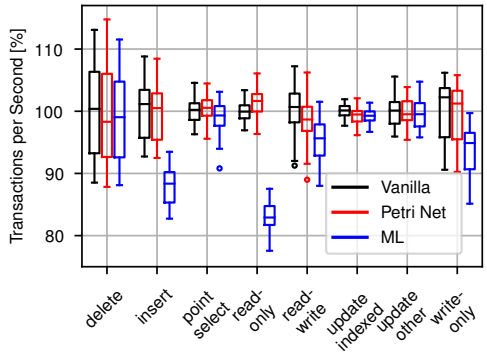


Fig. 4: Box-plot for Sysbench Throughput with and without the Different Plugin Implementations

uation of the ML-based `Classifier` as described in Section IV-D.

Workloads: In this evaluation, we use two synthetic benchmark tools: Sysbench [21] 1.0.20 (different delete, insert, point select, read-only, read-write, update and write only workloads) and TPC-H [3] 2.18.0_rc2 (suite of business-oriented ad-hoc queries and concurrent data modifications). We executed three performance benchmarks: (1) without the plugin as a baseline measure, (2) with the plugin enabled using the Petri Net classifier, and (3) with the plugin enabled using the ML-based classifier. We repeated every benchmark for over 50 iterations. The results are presented in the default box-plot style, where the box consists of the second and third quantile, and the whiskers include the last value within 1.5 interquartile range (IQR).

Results: Figure 3 visualizes these runtimes for the TPC-H benchmark (lower values are better). We see a slight increase of around one percent when using the Petri Net classifier compared to without the plugin. The ML classifier has no noticeable impact. Overall, the impact of either implementation is negligible, and operators with a TPC-H-like workload pattern can expect to use any DIMAQS implementation without a performance loss.

We present a more fine-grained per-workload view for the Sysbench benchmark in Figure 4. We normalized the transactions per second to the average value for vanilla MySQL (larger values are better). Here, we see little difference between vanilla MySQL and both plugin implementations for most workloads. However, the ML classifier diverges largely for the insert, read-only, and write-only and slightly for the read-write workloads. Write-only and insert are tightly connected, since the former comprises many insert statements. A potential cause is that the ML-classifier has to process each query while the Petri Net classifier processes only the configured queries. Thus, the ML classifier might produce significantly reduced performance compared to vanilla MySQL and the Petri Net classifier. On average, over all use-cases with equal weight, the Petri Net implementation reaches 99.8% of throughput, and the ML-based implementation reaches 94.6% of throughput. However, most database workloads resemble a read-write pattern, where the ML-implementation performs without issues.

The performance evaluation shows that DIMAQS does not create a significant performance loss for the Petri Net classifier and for most use-cases with the ML classifier. Depending on the use-case, potential users of DIMAQS can weigh the expected transactions specific to their workload and use this data to decide on which implementation and its security characteristics. Our proof-of-concept prototype is not yet optimized for performance. Neither Petri Net-based nor ML-based implementation of DIMAQS received extensive profiling for potential bottlenecks. Thus, further performance improvements are likely possible.

E. Security Considerations

In the following, we discuss potential attack scenarios against DIMAQS and show how we defend against them.

DIMAQS disabling: An attacker may try to disable DIMAQS to avoid detection. However, such a scenario would not be successful since administrative privileges to the database are insufficient to perform this task. One would need to have administrative privileges to the file system to manipulate corresponding config files. As an additional burden, it is also non-trivial for an attacker to detect that the system runs under DIMAQS observation because the `Query Rewriter` component of DIMAQS rewrites the queries in such a way that it excludes information about DIMAQS from the results.

DIMAQS triggers removal: The next possible attack vector is specific to MySQL implementation, which uses triggers. An attacker may attempt to delete triggers, which are used to deliver additional information to the DIMAQS plugin.

DIMAQS detects the removal of its specific triggers to defend against this attack vector. Their absence becomes obvious whenever the plugin does not receive information about atomic values affected by the queries. Upon detection, DIMAQS generates a notification for the administrator and backups all the databases and tables affected by subsequent queries.

Evasion of detection: An attacker may try to evade detection of DIMAQS. However, we based our detection on elementary features—query type and presence of payment information. If an attacker attempts evasion, the attack itself will morph into a new attack form. For example, an attacker cannot get paid without leaving payment information in a ransom message. Otherwise, if the attacker neither scans nor deletes the DBMS, then there is no attack. Further, we introduced a variable time window for attack detection. An attacker may stretch out the attack pattern beyond the window. However, this would require the attacker to diverge from the scalability goals described in Section II-B and, therefore, constitute a different type of attack.

VI. Related Work

Many previous works explored intrusion detection systems in databases. However, none explicitly focused on detecting ransomware so far. Compared to our approach, related work focuses on analyzing single queries [15], [32], while we aim to detect malicious query sequences. Intrusion detection frameworks [4], [7], [34], [22], [29], [24] examine database audit logs to identify anomalous queries using role profiles. However, those frameworks are limited to analyzing single queries. Hu et al. [16], [17], who proposed a database intrusion detection system utilizing (uncolored) Petri Nets, is the most comparable to ours. However, they detect anomalies by modeling data dependency relationships and regular data update patterns. In contrast, we derive a model from malicious query sequences and compare the sequences at runtime. Luckham et al. [25] perform intrusion detection through complex event processing (CEP). Romano et al. [13] propose a generic framework for intrusion detection using CEP examining different intrusions vectors. CEP systems are only passive information processing and monitoring tools, whereas DIMAQS actively prevents attacks. Other works also proposed ML as a classifier for intrusion detection. For instance, Kim et al. [20] use a combination of CNN and LSTM to classify roles. Related works [5], [33], [6] focus on role-based anomaly detection applying ML techniques. However, none of these works to track a global state of the DBMS to detect ransomware attacks using multiple accounts.

VII. Conclusion

In this work, we present DIMAQS, the first solution against server-side ransomware. In its heart, DIMAQS uses two classifiers to model malicious query sequences and matches them against query sequences captured at runtime. The first is a Colored Petri Net (CPN)-based classifier that introduces several novel extensions to CPNs, reducing the system representation complexity. The second classifier uses machine learning (ML), eliminating the required hand-crafted modeling of observed attacks and learning based on existing datasets.

Our solution is a MySQL plugin targeting and easily installable on existing servers. We evaluated our solution regarding the precision of the attack detection as well as its performance, showing promising results. We report no false negatives for both classifiers, no false positives for the CPN classifier, and 0.003% false positives for the ML classifier. Performance-wise, we see no significant impact on the TPC-H benchmark. For the Sysbench benchmark, we only find significant performance deviations for two out of eight scenarios and only regarding the ML classifier. On average, the performance loss is below 1% for the CPN classifier and below 6% for the ML classifier.

References

[1] Colonial Pipeline boss confirms $4.4m ransom payment, May 2021. [Online; accessed May 2022].

[2] Global Ransomware Damage Costs Predicted To Exceed $265 Billion By 2031, Jun 2021. [Online; accessed May 2022].

[3] TPC-H Homepage, Jun 2022. [Online; accessed May 2022].

[4] E. Bertino, A. Kamra, E. Terzi, and A. Vakali. Intrusion Detection in RBAC-administered Databases. In *ACSAC*, 2005.

[5] S.-J. Bu and S.-B. Cho. A convolutional neural-based learning classifier system for detecting database intrusion via insider attack. *Information Sciences*, 512:123–136, 2020.

[6] S.-G. Choi and S.-B. Cho. Adaptive database intrusion detection using evolutionary reinforcement learning. In *SOCO'17-CISIS'17-ICEUTE'17*.

[7] C. Y. Chung, M. Gertz, and K. Levitt. DEMIDS: A Misuse Detection System for Database Systems. In *Integrity and Internal Control in Information Systems (IICIS)*, 1999.

[8] C. Cimpanu. MongoDB Apocalypse: Professional Ransomware Group Gets Involved, Infections Reach 28K Servers. *Bleeping Computer*, 2017.

[9] A. Continella, A. Guagnelli, G. Zingaro, G. De Pasquale, A. Barenghi, S. Zanero, and F. Maggi. ShieldFS: The Last Word in Ransomware Resilient Filesystems. In *Black Hat USA*, 2017.

[10] A. Continella, A. Guagnelli, G. Zingaro, G. D. Pasquale, A. Barenghi, S. Zanero, and F. Maggi. ShieldFS: A Self-healing, Ransomware-aware Filesystem. In *ACSAC*, 2016.

[11] O. Corporation. *MySQL 5.7 Manual*, 2018.

[12] M. Dzulfakar. Advanced MySQL Exploitation. In *Black Hat USA*, 2009.

[13] M. Ficco and L. Romano. A Generic Intrusion Detection and Diagnoser System Based on Complex Event Processing. In *International Conference on Data Compression, Communications and Processing (CCP)*, 2011.

[14] D. Golunski. MySQL-Exploit-Remote-Root-Code-Execution-Privesc-CVE-2016-6662, 2017.

[15] W. G. J. Halfond and A. Orso. Preventing SQL Injection Attacks Using AMNESIA. In *ICSE*, 2006.

[16] Y. Hu and B. Panda. Identification of Malicious Transactions in Database Systems. In *IDEAS*, 2003.

[17] Y. Hu and B. Panda. A Data Mining Approach for Database Intrusion Detection. In *ACM Symposium on Applied computing (SAC)*, 2004.

[18] B. Ives, K. R. Walsh, and H. Schneider. The Domino Effect of Password Reuse. *Communications of the ACM*, 47(4), 2004.

[19] K. Jensen. *Coloured Petri Nets*. 1997.

[20] T.-Y. Kim and S.-B. Cho. Optimizing cnn-lstm neural networks with pso for anomalous query access control. *Neurocomputing*, 2021.

[21] A. Kopytov. akopytov/sysbench, 2022.

[22] V. C. S. Lee, J. A. Stankovic, and S. H. Son. Intrusion Detection in Real-time Database Systems Via Time Signatures. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2000.

[23] Lindsey O'Donnell. PLEASE_READ_ME Ransomware Attacks 85K MySQL Servers, December 2020. [Online; accessed May 2022].

[24] W. L. Low, J. Lee, and P. Teoh. DIDAFIT: Detecting Intrusions in Databases Through Fingerprinting Transactions. In *International Conference on Enterprise Information Systems (ICEIS)*, 2002.

[25] D. C. Luckham and B. Frasca. Complex Event Processing in Distributed Systems. Technical report, Stanford University, 1998.

[26] MediaWiki. MediaWiki/de — MediaWiki, The Free Wiki Engine, 2018.

[27] S. Morgan. Cybersecurity Business Report. Ransomware Damage Costs predicted to hit USD 11.5B by 2019, 2017.

[28] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, 1981.

[29] A. Roichman and E. Gudes. DIWeDa - Detecting Intrusions in Web Databases. In *DBSEC*, 2008.

[30] P. Rygielski. vikin91/BibSpace, 2022.

[31] semantic mediawiki.org. Semantic MediaWiki, 2018.

[32] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2006.

[33] S. Subudhi and S. Panigrahi. Application of optics and ensemble learning for database intrusion detection. *Journal of King Saud University - Computer and Information Sciences*, 2019.

[34] F. Valeur, D. Mutz, and G. Vigna. A Learning-Based Approach to the Detection of SQL Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2005.

[35] O. Ziv. 0.2 BTC strikes back, now attacking MySQL databases, 2017.