

Hands Off my Database: Ransomware Detection in Databases through Dynamic Analysis of Query Sequences

Lukas Iffländer
University of Würzburg

Alexandra Dmitrienko
University of Würzburg

Christoph Hagen
University of Würzburg

Michael Jobst
University of Würzburg

Samuel Kounev
University of Würzburg

Abstract

Ransomware is an emerging threat which imposed a \$ 5 billion loss in 2017 and is predicted to hit 11.5 billion in 2019. While initially targeting PC (client) platforms, ransomware recently made the leap to server-side databases – starting in January 2017 with the MongoDB Apocalypse attack, followed by other attack waves targeting a wide range of DB types such as MongoDB, MySQL, Elasticsearch, Cassandra, Hadoop, and CouchDB. While previous research has developed countermeasures against client-side ransomware (e.g., CryptoDrop and ShieldFS), the problem of server-side ransomware has received zero 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 Colored Petri Nets (CPNs) for attack detection. Our system design exhibits several novel techniques to enable efficient detection of malicious query sequences globally (i.e., without limiting detection to distinct user connections). Our proof-of-concept implementation targets MySQL servers. The evaluation shows high efficiency with no false positives and no false negatives and very moderate performance overhead of under 5%. We will publish our data sets and implementation allowing the community to reproduce our tests and compare to our results.

1 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 an asset for users nowadays, but has also become the key component of digitization and transformation of today’s businesses globally – enterprises collect data on consumer preferences, purchases, and trends and use it to optimize their business models and strategies. Given such trends, the importance of database security

is hard to overestimate – the rapid growth of the data volume stored in the databases of service providers, in cloud environments and enterprise data centers, as well as their increasing importance, make them attractive attack targets.

Traditionally, attacks on data have aimed to undermine confidentiality and authenticity. More recently, however, attacks against the availability of data, services, and users have become common as well – modern attackers deploy ransomware, malicious software that encrypts data and holds the decryption key until the victim pays a ransom. They still claim the ransom pretending to have encrypted the data. The financial loss from ransomware is significant – it reached 5 billion USD in 2017 and is predicted to hit 11.5 billion by 2019 [50].

The rise of server-side ransomware While the first ransomware attacks targeted client platforms (information stored in users’ files), recently such attacks made a leap to server-side databases that store, accumulate and process (big) data. In January 2017 tens of thousands of MongoDB servers were hit in an attack called MongoDB Apocalypse [9, 10], followed by a second attack wave targeting MySQL servers [60]. Since then, server-side ransomware attacks spread to a wide range of server technologies, including Elasticsearch [11], Cassandra [7], Hadoop and CouchDB [8].

Attack scenario The typical attack scenario of server-side ransomware observed so far is as follows: First, an attacker gains remote privileged access to the database database through the exploitation of configuration vulnerabilities such as the usage of default passwords¹. Once connected, they execute commands for data enumeration (e.g., to learn names of databases and tables hosted), then drop (delete) data and insert the ransom message with instructions how to pay the ransom. Remarkably, in contrast to client-side ransomware, the new attack form wipes the data without making any plaintext or encrypted copy, e.g., acting as a *wiper*. This strategy has, on the one hand, more dramatic implications for the vic-

¹Note that default passwords and other misconfiguration errors are prevalent real-world problems. For instance, Mirai botnet [1] used similar vulnerabilities to take over more than 600,000 IoT devices around the globe.

tim, since the data is unrecoverable even if the ransom is paid. On the other hand, the attack is stealthier, since no intensive and easily detectable operations required, such as bulk encryption or massive data copying, and no back channel to the attacker needed (e.g., for delivering the decryption key or recovered data) that could be used to trace them back.

Motivation for server-side ransomware to spread While server-side ransomware is more recent and to this day less widespread than client-side ransomware, there are reasons why the situation might change quite soon. First, 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. However, businesses can pay much more – for instance, in a recent attack, a Los Angeles Hospital paid USD 17000 of ransom to attackers [49]. Second, in recent years, researchers and antivirus companies developed countermeasures against client-side ransomware. However, to date, no solutions exist against ransomware targeting database servers. This lack of protection makes databases easy attack targets.

Do victims pay the ransom to a wiper? Note that there is evidence that even though server-side ransomware is a wiper, some desperate victims paid the ransom, nonetheless. We identified that two known ransomware addresses involved in MySQL attacks [60] received 0.6 BTC (equivalent to 3 payments). For the attacks against MongoDB, we identified a total of 160 ransom payments to the addresses collected in [9], totaling in 26.35 BTC. Moreover, the survey [9] reveals that even production systems lack sufficient protection by strong passwords and sensible backup strategy: Among 123 surveyed ransom victims, only 11% had recent backups, and 8% paid the ransom.

State of the art Existing anti-ransomware solutions are aiming at detection of client-side ransomware only. They follow two dominant strategies: Signature-based detection of malicious binaries and runtime monitoring and behavioral analysis for anomaly detection. The first one builds upon detection of malicious binaries and is typically used by anti-virus vendors, while the second strategy originates from research papers [12, 13, 37, 54] and relies on runtime monitoring of file accesses and the detection of malicious activity based on heuristics, such as access to multiple files, their modification, and renaming. Unfortunately, both strategies are not applicable for detection of database wipers. Since in server-side ransomware attack scenario an attacker connects to the database remotely, there is no malicious binary on the platform that could be detected. Furthermore, monitoring at the file system level for abnormal activity is not adequate either since there is no direct correlation between an attacker’s activity and file access patterns.

Our contributions In this paper, we aim to improve the security of database systems and propose DIMAQS (Dynamic

Identification of Malicious Query Sequences), signature-based intrusion detection tool that can detect sequences of malicious queries. Generally, the tool is not limited to ransomware detection and can potentially be applied to detection of other attack classes as long as they rely on malicious sequences of queries (e.g., advanced SQL injections aiming at removing code execution [15]). However, motivated by the rise of server-side ransomware we apply it to the problem of ransomware detection. We make the following contributions:

- We provide design and implementation of DIMAQS, a framework that can detect sequences of malicious queries. To keep track of queries and to perform detection, our solution leverages Colored Petri Nets (CPNs) to model the series of events used in attacks and to match them to known malicious patterns. Our system design exhibits several novel techniques (dynamic creation of colors, merging of tokens and token expiration) to reduce the complexity of the system representation and achieve better performance. Our framework performs system-wide monitoring and as such can detect malicious sequences injected through several user sessions and interleaved with benign queries – a quite interesting feature that eliminates most obvious evasion strategies. Our implementation targets MySQL, one of the most popular database management systems, and imposes only a very moderate performance overhead under 5%. We realize our solution in the form of a MySQL plugin that is easily installable on existing MySQL servers, thus preserving compatibility with legacy software. We will publish the source code on GitHub along with the paper.
- We apply DIMAQS to the challenging problem of server-side ransomware. To make detection of such attacks possible, we analyze previously observed attacks and extract their distinctive properties that provide a basis for attack detection. We then evaluate the effectiveness and practicality of our solution using three data sets: Malicious data set recorded by us, and benign query sets from a publication management system and a MediaWiki server. The results demonstrate the high efficiency of our approach with no false negatives or false positives. We will publish our data sets along with the paper to the benefit of the research community. To the best of our knowledge, our malicious data set will be the first one publicly available.

Outline. The remainder of this paper is structured as follows. In Section 2 we present the necessary background followed by system design of DIMAQS in Section 3. In Section 4, we reveal the details of our prototype implementation. Prototype evaluation results are presented in Section 5. After a review of the related work in Section 6, we conclude the paper and outline future work in Section 7.

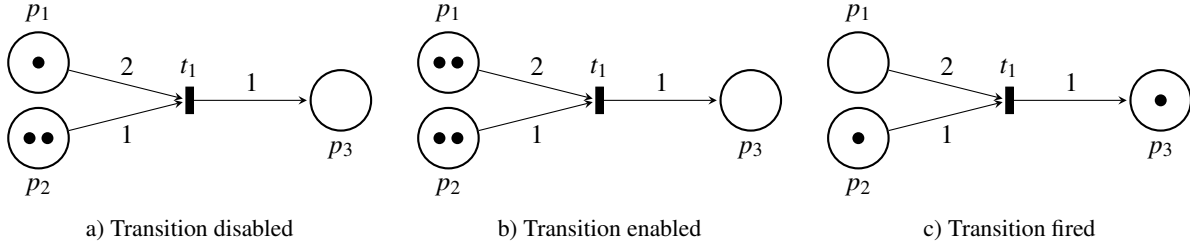


Figure 1: Demonstration of Petri net execution using a simple example

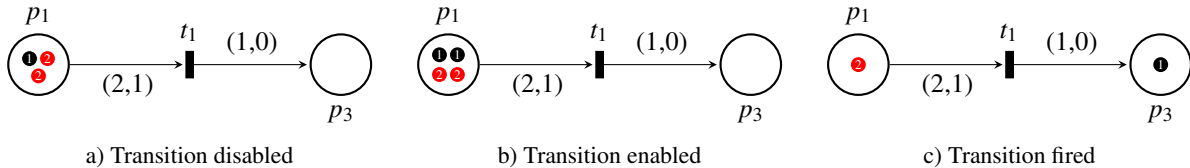


Figure 2: Colored Petri Net example. In comparison to the regular Petri net depicted in Figure 1, the number of required *places* is reduced from two to one without reducing functionality.

2 Background

In this section, we provide the necessary background on Petri nets and their enhanced version, colored Petri nets.

Petri Nets are a commonly used mathematical modeling language for the description of distributed systems [51] named after their inventor Carl Adam Petri. They are a class of discrete event dynamic systems. 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. The value of the arc defines the number of tokens required per place. Once a transition fires, it consumes the required number of input tokens from the input places. The transition results in creating the specified number of output tokens on the places with arcs from the transition to them (output places).

Figure 1 shows a simple example of a Petri net. The depicted Petri net consists of three places (depicted as circles), one transition (depicted as a bar), and three arcs. Enabling the transition requires three tokens: Two tokens at place p_1 and one token at place p_2 . In Figure 1a only one token is available at p_1 . Regardless of the total count being three tokens, with only one token on p_1 , the transition is not yet enabled. Adding another token to p_1 in Figure 1b satisfies the requirement and thus enables the transition. When the transition fires, two tokens are subtracted from the token set

at p_1 as well as one token from p_2 . At the same time, the transition adds one token to p_3 . Figure 1c shows the state after the transition firing.

Petri nets are a powerful tool for modeling [5] and allow for extensions to suit various tasks like queuing Petri nets for performance modeling. In this work, we use colored Petri nets, an extension to ordinary Petri nets.

Colored Petri Nets (CPNs) enable support for tokens of different types, also known as *token colors*. Places can now contain tokens of multiple colors. Arcs can define any combination of the colors for the number of input and output tokens. This addition allows for making Petri nets more compact.

Figure 2 illustrates the reduction in representation complexity by presenting a CPN derived from the previous example. The places p_1 and p_2 depicted in Figure 1 are now merged into a single place denoted as p_1 , while tokens are now assigned different colors: Tokens formerly placed in p_1 are now black (1) and those placed in p_2 are red (2). The transition now requires two black and one red token instead of requiring two tokens from p_1 and one from p_2 . The overall Figure 2 depicts the same process as before. In Figure 2a, one black token is missing for the transition to be enabled. In Figure 2b this token is added, thus enabling the transition. Finally, in Figure 2c the transition has fired, subtracting two black and one red token from p_1 and adding a black token to p_3 .

3 Design

DIMAQS is the first system that aims at the detection of ransomware attacks in databases. In a nutshell, it represents an intrusion detection system that leverages knowledge about

the attack pattern (or signature) and performs real-time system monitoring and pattern matching to detect intrusion attempts. For pattern matching, we leverage a CPN to encode the system states and their transitions inside the color information to detect when the system transitions to the state associated with the attack description.

The usage of (colored) Petri nets is a known technique for pattern matching, and their application to intrusion detection problems was investigated in previous works [26, 40]. However, typical application scenarios of CPN-based intrusion detection systems target other environments, e.g., networks [59] and operating systems [2].

The application of Petri nets for intrusion detection in databases was only considered by Hu et al. [26], who aimed at detection of anomalies of any sort, not specific to ransomware. However, they use *uncolored* Petri nets and leverage them to model benign states of a database system rather than attack states. Hence, their solution requires a training phase to gain knowledge about the underlying data structure as well as about benign data update patterns. In contrast, our system does not require similar training. Moreover, their work is theoretical. Hence, they did not provide any implementation or evaluation results with which to compare.

In our work, we aim to fill the gap and address the problem of ransomware attacks targeting databases. As such, we investigate the applicability of CPNs for ransomware attack detection in databases. We observe that databases are complex systems and modeling their state regarding dependency relationships and update patterns, as, e.g., done in [26], may lead to overly complicated system representations (for large and complex databases) and non-trivial overhead. Hence, we tackle the problem differently and choose to model malicious query sequences – an approach which results in a much simpler system representation, and independence from the structure of the underlying data and update patterns.

Our approach is system-centric and allows for detection of attacks that are carried out over multiple sessions or multiple user accounts. We also develop several novel techniques that even further to simplify the system representation, namely (1) dynamic color creation (creating an infinite color space), (2) token merging and duplication, and (3) token expiration making the use of CPNs practical.

The remaining part of this section is structured as follows: We first describe a typical ransomware attack scenario (Section 3.1). Next, we present our adversary model (Section 3.2) followed by the system architecture description (Section 3.3). Finally, we show the interaction of the system components when handling incoming queries (Section 3.4).

3.1 Attack Scenario

Our attack scenario originates from an analysis of a large-scale ransomware attack targeting MySQL servers that took place in February 2017 [60]. The attacker performs the at-

tack remotely by connecting to the database using a TCP connection. Once connected, an attacker gains root access through, e.g., brute-forcing the ‘root’ password of the database. Next, they enumerate the data in the database through retrieval of the list of the databases present. After that, the attacker creates a new table with an arbitrary name (e.g., the table 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 (drops) the databases on the server and disconnects.

The scenario above describes the attack steps recorded in real-world attacks. Additionally, we accept that attack steps can deviate from this scenario: For instance, an attacker could first perform the database deletion and only after that insert the ransom message. Also, attackers may use arbitrary names for databases and tables and arbitrary patterns for the ransom message. We, however, assume that the attacker demands payments in cryptocurrency (such as Bitcoin or Ethereum) since they provide at least some level of anonymity in contrast to more traditional payment methods that involve banks². We also assume that an attacker continues to wipe data and does not aim to keep any data copies, since this would slow down the attack significantly, and would require storage on attacker’s side and a communication channel between the victim and the attacker, which demands additional resources and increases chances of exposure. We also assume an attacker does not perform on-site database encryption since we did not identify any standard SQL commands that could be used to do so.

3.2 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 the ransom. We assume the remote attacker who is accessing the server over the Internet has no physical access to it. The software running on the server is trusted, 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 DBMS without any restrictions. Furthermore, we assume an attacker with administrator-level privileges to the DBMS. This assumption is often fulfilled in practice since the problem of weak or re-used passwords [33] is well known and not satisfactory solved for over decades. For instance, findings show that most of the MySQL servers had no root password set due to using an insecure default configuration [14]. Alternatively, an attacker might exploit a security vulnerability like [19] to gain administrator privileges for the database.

We, however, do not assume administrator privileges of the attacker to the operating system. Also, we leave DoS

²Since banks are obliged to follow "know your customer" policy.

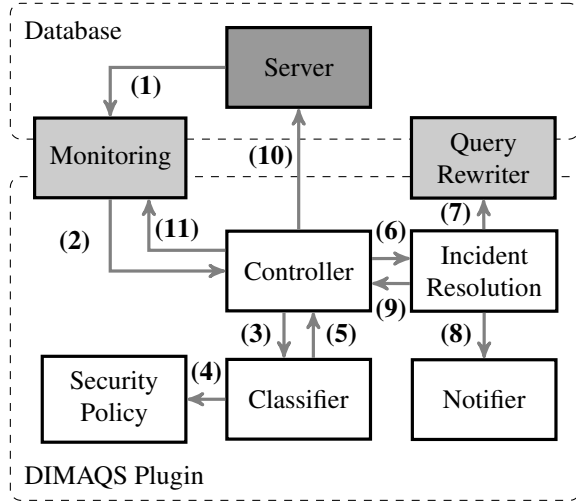


Figure 3: System architecture of DIMAQS. Dark grey boxes are components provided by the database, light grey boxes are components that interface between DIMAQS and the database, and white boxes belong to DIMAQS itself.

attacks are 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. The attacker wants to perform a hit-and-run attack without considering other services and ways of communication.

3.3 System Architecture

Figure 3 shows the DIMAQS system architecture. DIMAQS is comprised of six components: (i) Monitoring, (ii) Classifier, (iii) Security Policy, (iv) Incident Resolution, (v) Notifier, (vi) Query Rewriter and (vii) 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, 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. Note that this module monitors all queries arriving through different connections, not specific to user sessions. Notifications on the occurrence of incoming queries result from the database server’s audit functionality.

Classifier The `Classifier` component processes the incoming queries and produces a verdict whether a query is benign or malicious. For the classification, DIMAQS uses a CPN with our extensions. The token colors are used to attach run-time information to the tokens, such as time-stamps, table names and modified cell values. Since such token colors are

dynamic and unbounded, conventional Petri nets would be unable to represent all the possible states. This information also provides additional information to the DIMAQS administrator in the case of an incident³.

Extensions to CPNs. For our purposes, 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. It is highly unlikely that a malicious query sequence spawns over a long period (e.g., days), since this increases the risk of detection and complicates the attack (the database can change considerably over time). Large or absent timeouts can additionally result in a higher false positive rate since eventually all transitions might be triggered by unrelated queries. The timeout threshold is, therefore, a security parameter, which enables a trade-off between effectiveness and false alerts. In real-world attacks observed so far, attackers did not stretch malicious query sequences over long periods. Hence, even short timeouts (1-2 minutes) would work well against them. Attackers might increase the attack time window to avoid detection. However, the longer they stay connected, the higher the burden for them (since the attacks are not generally automated), and the higher the risk of being uncovered, especially given the fact that they do not know the currently used threshold parameter and, hence, have no understanding for how long they should stay connected to remain undetected.

Security Policy The `Security Policy` component holds information about patterns of malicious query sequences (or attack signatures). The CPN configuration represents it in our system – it describes CPN’s places, place actions, transitions, transition actions, transition conditions, and arcs.

All places and transitions are named, and the arcs are each weighted with a value of 1 token. Each place can be assigned several place actions executed upon CPN transitions to the corresponding place. Transitions are used to check for the execution of a (next) step in a malicious query sequence. They become active when the source place contains at least one token. Each transition is assigned 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 (such as a table name or a typical ransom message).

A transition may also have an arbitrary number of transition conditions which are used to evaluate the token data from the source place against the query values. Our policy includes only one transition condition, ensuring ransom mes-

³Note that DIMAQS administrator and database administrator are different entities

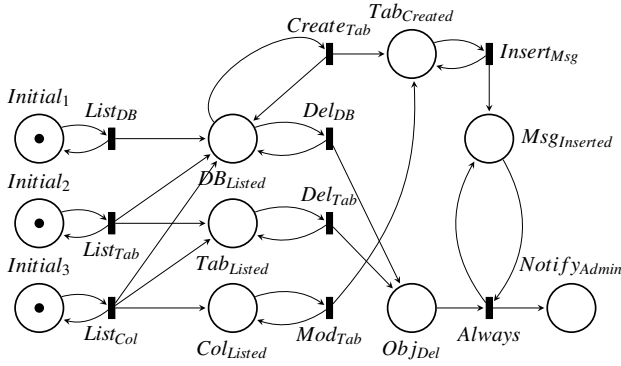


Figure 4: The CPN used to classify database transactions. All arcs are weighted with a value of 1 token.

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

Place	Description
<i>DB_{Listed}</i>	Rewriting
<i>Tab_{Listed}</i>	Rewriting
<i>Col_{Listed}</i>	Rewriting
<i>Tab_{Created}</i>	Trigger creation
<i>Obj_{Del}</i>	Create backup
<i>Notify_{Admin}</i>	Create notification

Table 1: Configured actions for the places inside the CPN in Figure 4. When a token reaches a place, the specified action can be executed.

sage insertion into a previously created or modified table.

We depict the CPN that was tailored to the observed attacks configured according to our security policy in Figure 4. Table 1 shows the place actions executed after putting a token on the place.

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 would be 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 time⁴.

⁴Our estimate is 30 min.

Incident Resolution When an event in the *Classifier* component issues an action, an action 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 (and, hence, from the attacker) so that an attacker cannot drop it again or even identify that such a backup exists. To hide backed up tables and databases from users, *Incident Resolution* uses a "rewriting" action. 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. Notification actions are used by the *Incident Resolution* component whenever there is a need 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 sending an email to 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 system segments. It is the central element that orchestrates the processing of incoming queries by other components, e.g., through invocation of the *Classifier* component to classify the query as malicious or benign, or the *Incident Resolution* component to initiate incident resolution upon attack detection.

3.4 Component Interaction

Figure 3 depicts the interaction between the components during query processing. The database server first receives the query and then notifies `Monitoring` (1). If `Monitoring` raises an alert for a potentially malicious query type, the `Controller` is notified (2). The `Controller` then forwards the suspicious query to the `Classifier` (3) for evaluation. The `Classifier` is configured using the security policy from the `Security Policy` (4) and returns the classification result to the `Controller` (5). There are two possible outcomes: the query’s classification is either benign or malicious. In a former case, the `Controller` terminates its actions, and the server executes the query as-is (10). In the latter case, the query is considered malicious, and the `Controller` calls `Incident Resolution` (6), which in turn backs up dropped tables and rewrites the malicious query using `Query Rewriter` (7). It then invokes the `Notifier` to inform the administrator about an incident (8). The `Controller` then receives the rewritten "disarmed" query from `Incident Resolution` (9). The database server then executes the query (10). The `Controller` informs `Monitoring` when additional objects need to be observed (11), e.g., when a query creates new tables.

4 Implementation

DIMAQS design is generic and can be applied to different database technologies. For the sake of 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 our own Petri net implementation library `libPetri` as well as the `mysqservices` 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), while `libPetri` results in 1008 LoC.

4.1 Plugin Integration

The plugin is loaded during MySQL server start-up and registers itself as an auditing plugin.

The MySQL server plugin interface provides notifications [14] for the following useful events:

- `MYSQL_AUDIT_CONNECTION_CLASS`,
- `MYSQL_AUDIT_CONNECTION_CONNECT`,
- `MYSQL_AUDIT_CONNECTION_DISCONNECT`,
- `MYSQL_AUDIT_PARSE_CLASS`,
- `MYSQL_AUDIT_PARSE_POSTPARSE`.

Notifications of the `MYSQL_AUDIT_PARSE_CLASS` class provide an event of a single to-be-executed query. Queries, however, could also be nested.

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 typical for the use in attacks like mimicry, e.g., for the insertion of ransom messages. To allow us to access the atomic values, we create triggers. We generate “before `INSERT/UPDATE`” triggers for every table. In these triggers, we execute a user-defined function. This function forwards the values affected by the queries to the controller for evaluation.

As detailed in the MySQL trigger syntax [14], a trigger becomes associated with a table named `tbl_name`. This name must refer to a permanent table, which means that a trigger does not apply to a temporary table or a view. This limitation does not affect our solution since it is unlikely that an attacker would attack data stored in temporary tables.

4.2 Component Implementation

In the following, we detail the implementation of DIMAQS modules.

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 startup 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 `dimags_plugin_init()` performs the creation of the additional triggers and is called directly after initialization of the server and before entering the listening state. `dimags_plugin_init()` creates a trigger for every non-virtual database. Virtual databases are databases that contain read-only views rather than base tables and have no database files associated with them. Hence, protection of virtual databases is not necessary.

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` is implemented using our library `libPetri`. `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. Thus, `libPetri` contains all necessary functionality within around 1008 of LoC.

`libPetri` keeps track of all active transitions. Since all our arcs in `Classifier` are weighted with the value one as seen in Figure 4, 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.

Security Policy The `Security Policy` is a database that contains tables holding the information about the actions that can fire transitions (e.g., the regular expression for detecting the ransom message) and the places with their associated actions. Classifier processes this information on startup and during classification.

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. Each moved table requires recreation of the affected triggers. Table renaming follows the following schema "`<storagespace>.<object prefix>_<dbname>_<tablename>_<timestamp>`" with `storagespace` being a preconfigured variable of DIMAQS. 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` is executed to back up the database tables.

Notifier The `Notifier` sends an email with all transmitted information about the suspected attack to the administrator. The administrator's address can be configured inside the database or in a configuration file.

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.

5 Evaluation

In this section, we describe our test setup and evaluate our implementation with regards to effectiveness and performance. We conclude by discussing security considerations.

5.1 Test Setup

Testbed To execute performance and security tests, we use the following setup. For the database server, we use an HPE

ProLiant DL360 Gen9 server [16]. The server is equipped with a single 8-core Haswell generation Xeon E5-2640 CPU with a base clock of 2.60 GHz and a turbo clock of 3.40 GHz and packaged with a total of 20 MB of cache [32]. Simultaneous multithreading is enabled allowing the execution of 16 threads in parallel. The server features 32 GB of DDR4 RAM at 2133 MHz with dual channel capability. A 500 GB 3.5-inch hard drive provides storage I/O turning at 7.200 rpm.

For the operating system, we chose Ubuntu 16.04.4 LTS running Linux kernel 4.4.0-121. To provide a DBMS to evaluate against we install and run MySQL server 5.7.22 on this server.

All tests are executed directly on this server. Thus, the network is not a limiting factor for the benchmarks. Due to the performance of the server, the resources consumed by the client running in parallel to the server are expected to be negligible, and their performance influence is therefore not evaluated in this work.

Data Sets We employ three data sets during our evaluation. The first set (malicious set) includes malicious query sequences, which we generated ourselves using information about real-world attacks collected at [60]. Our resulting query set contains query sequence permutations with an expected malicious classification, as well as their possible permutations (since an attacker may execute them in an arbitrary order). The full test set contains 13 485 tests. Each test contains nine queries. The first five queries of each test are to set up two databases and a table at the beginning of the experiment and remove them at the end. Relevant to the detection are four queries: (i) listing all databases, (ii) creating a table, (iii) inserting a ransom message into this table, and (iv) dropping a table or database. Therefore, the set performs 53 940 queries in total.

The second set (*Bibspace set*) is from the publication management system *Bibspace* [53], which was gathered over 40 days from 13th of April 2018 to 22nd of May 2018 and contains a total of 52 085 queries. Among them, 24 430 are `CREATE_TABLE_IF_NOT_EXISTS` queries, 8 357 `INSERT` queries, and 38 `DROP_TABLE_IF_EXISTS` queries.

The third query set (*MediaWiki set*) is from a locally run *MediaWiki* [47] with the *Semantic MediaWiki* [55] plugin enabled, collected for 50 days from 3rd of April 2018 to 22nd of May 2018. Containing 2 514 764 queries, it includes 69 261 `INSERT` statements, 29 830 `CREATE_TEMPORARY_TABLE` statements, and 29 797 `DROP_TEMPORARY_TABLE` statements.

We will publish the data sets along with the paper, to allow third parties to reproduce our tests and to enable follow up works to compare with our results.

5.2 Effectiveness

In the following, we evaluate the precision of the classifier module. Thus, we evaluate whether a wrongful classification

Query set	$Initial_1$	$Initial_2$	$Initial_3$	DB_{Listed}	Tab_{Listed}	Col_{Listed}	$Tab_{Created}$	$Object_{Deleted}$	$Notify_{Admin}$
Bibspace	1	1	1	2	2	0	24	0	0
MediaWiki	1	1	1	7	5	1	0	0	0

Table 2: Petri net state after execution of query sets

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 3.3. 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 a specific pattern of the ransom message. We search for the occurrence of a BTC or Bitcoin string inside the inserted message since attackers until now requested ransom in Bitcoins⁵. We used the regular expression `'(d*[.]){0,1}d+s*(BTC|Bitcoin)'` (case insensitive). The matching expressions are, e.g., `5 BTC|Bitcoin`, `.5 BTC|Bitcoin`, `20.1 btc|Bitcoin`.

False Negatives: To test for false negatives, we used the *attack set* described in Section 5.1. After processing all the queries from the data set by our CPN, we achieved 100% attack detection rate and received no false negative result. This result confirms that our CPN correctly models each attack from our malicious data set.

False Positives: To test for false positives, we choose to use the *Bibspace set* and the *MediaWiki set*. The sets 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, their number represents raised alerts. For this evaluation, we disable the token timeout, to increase the potential for false positives.

Table 2 shows the population of the CPN after running all the queries from the *Bibspace set* through Classifier. No token has reached the state N , that would have triggered an alert to the administrator. Next, the Classifier processed the queries of the *MediaWiki set*. Table 2 shows the state of CPN from Figure 4 after classification. Again, no token has reached the state N , and no ransom attack was detected, which is a favorable result.

5.3 Performance Evaluation

To evaluate the performance of the DIMAQS plugin, we used two data sets: The *MediaWiki set* described in Section 5.1 and the synthetic benchmark sysbench [38]. We use sysbench 0.4.12 with 16 active threads. We performed three performance benchmarks: (1) without the plugin as a baseline measure, (2) operating on a newly initialized Petri net, and (3) with a fully occupied Petri net with tokens in each

⁵Our policy can be trivially extended to detect ransom messages requesting payments in other cryptocurrencies.

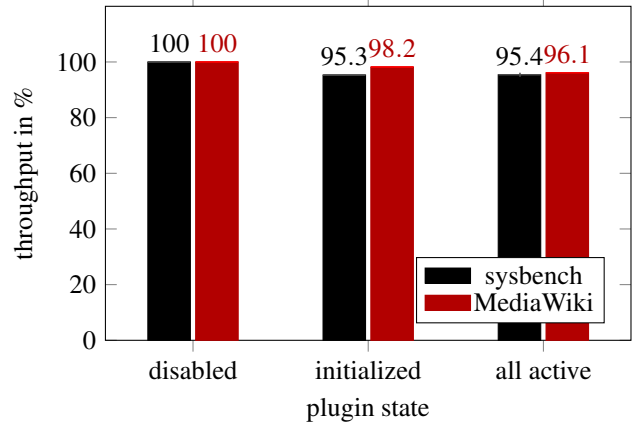


Figure 5: Performance influence of DIMAQS for sysbench and MediaWiki. Values are normalized to the respective value for the disabled plugin.

Test	Transactions per second			relative to baseline [%]		
	mean	stdev	conf int	mean	stdev	conf int
sysbench						
disabled	9 245	28	±9	100.0	0.3	±0.1
initialized	8 806	30	±11	95.3	0.3	±0.1
full	8 823	19	±7	95.4	0.2	±0.0
MediaWiki						
disabled	2 008	5	±2	100	0.2	±0.1
initialize	1 971	7	±2	98.2	0.3	±0.1
full	1 930	6	±13	96.1	2.9	±0.3

Table 3: Performance without the plugin, with the plugin enabled, and with tokens in each Petri net state.

state. Sysbench benchmarks were run for 60 seconds per iteration, while the *MediaWiki set* was classified entirely every time. We performed every benchmark for over 50 iterations. Table 3 shows the resulting measurements (database transactions per second). We report average values with standard deviation and confidence intervals (5% quantile according to the Student’s t-distribution). Figure 5 visualizes these results.

The results show that the usage of the DIMAQS plugin results in performance degradation of about 5 % for sysbench. There is no substantial difference whether the Petri net is only initialized or entirely populated (overlapping confidence intervals). This marginal difference suggests that the overhead is not a result of querying the Petri net, but from analyzing and parsing the queries themselves. For the *MediaWiki set*

performance degradation is about 2% for the initialized Petri net and 4% for an entirely populated net. This time, the influence of the set population has a more significant impact.

Our proof-of-concept prototype is not yet optimized for performance. Neither DIMAQS nor libPetri has received extensive profiling for potential bottlenecks. Also, no compiler optimizations were enabled. Thus, performance improvements are likely possible.

5.4 Security Considerations

In the following, we discuss potential attack scenarios against DIMAQS itself and show, how our system defends itself 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: A 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.

To defend against this attack vector, DIMAQS detects the removal of DIMAQS-specific triggers. 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 DIMAQS administrator and backups all the databases and tables affected by subsequent queries.

6 Related Work

In this section, we provide an overview of the related work in three domains: (i) intrusion detection for databases, (ii) ransomware detection, and (iii) application of Petri Nets for intrusion detection in various application domains.

Intrusion Detection for Databases There is a plethora of previous works on intrusion detection systems in databases, but none of them explicitly focused on detection of ransomware so far. The first line of works in this category concentrate on detection of SQL injections. Fonseca et al. [18] and Kemalis et al. [34] detect anomalies in SQL commands given a training set of known valid query structures or their specifications. Buehrer et al. [21] and Bockermann et al. [4] use tree structure when parsing SQL statements and then dynamically compare them with the intended queries. AMNE-

SIA [22,23] checks the application code for SQL queries generating automata for each query to match against dynamic requests during operation. SQLCheck [57] validates queries by adding a key at the beginning and the end of each user's input and validate syntactic correctness of the "augmented" queries at runtime. In contrast to our work, all these approaches concentrate on the analysis of single queries, while we aim at the detection of malicious query sequences.

Intrusion detection frameworks [3, 6, 58] analyze database audit logs to detect anomalous queries by matching against role profiles. In contrast to our work, their analysis concentrates on irregular access patterns of single SQL queries. Moreover, their analysis is bound to user profiles, while DIMAQS performs global monitoring across user sessions.

DAIS [42] and the solution by Liu et al. [43] combine intrusion detection with the dynamic isolation of malicious and suspicious activities through rewriting of SQL statements. As a result, potentially malicious modifications are performed on a shadowed incremental copy of the database. In our work, we use a similar approach to preserve copies of the values affected by potentially malicious queries.

The most similar work to ours is by Hu et al. [26,27], who proposed an intrusion detection system for databases using (uncolored) Petri Nets. However, Hu et al. choose to model data dependency relationships and regular data update patterns and then detect anomalies, while we model malicious query sequences and compare the sequences captured at runtime with the derived model. As such, their system requires knowledge about the legitimate state of the system, while our approach represents a signature-based misuse detection system and needs knowledge about attack patterns. As a result, our solution applies to databases of arbitrary complexity and without the need to learn about underlying data structure (which can be complex), while the solution by Hu et al. requires a training phase to gain knowledge about the database under protection. On a positive side, their approach is likely to detect previously unseen malware. The feasibility of the approach by Hu et al. however was not practically verified, since authors concentrated on theoretical aspects and did not provide any implementation and evaluation. Their concept also relies on several assumptions that simplify the model but might be too restrictive in practical scenarios. For instance, they assume low database load and that users only update the database through a limited number of fixed transactions modifying the same data items. Our solution, in contrast, operates on databases of arbitrary complexity and with good performance.

Lee et al. [41] target real-time databases with regular access patterns, which occur, e.g., in data collection from sensors. They use time signatures to capture expectations about update rates and flag unexpected and possibly malicious operations. DIWeBa [52] is an anomaly-based intrusion classifier for web databases that works at the session level by fingerprinting user sessions. DIDAFIT [44] models benign

query sequences and maps them to a directed graph, where graph vertices represent query signatures. Enforcement of sequence orders on the graph prevents anomalous queries. In contrast to our work, solutions above require a training phase to learn the benign behavior of users, manual setup and knowledge of the database content, or to construct graphs of benign queries.

Mathew et al. [46] argue that query classification based on syntax is more error-prone than observing the accessed data points since syntactically similar queries can produce significantly different results. Their system is another example of observing anomalous database access patterns, which need a training phase or some predetermined knowledge of acceptable behavior.

It is also possible to perform intrusion detection through complex event processing (CEP) [45]. Romano et al. [17] propose a generic framework for intrusion detection through CEP, where they examine different intrusions, including policy violations, buffer overflows and SQL injections. It should be noted that CEP is not a single algorithmic concept, but rather the more general idea to infer not directly observable events from multiple, related events. In a way, our implementation with CPNs acts similarly, observing individual queries that together form a ransomware attack. On the other hand, CEP systems are mostly merely a monitoring and information processing tool, while our solution includes active components, such as the automatic table backup functionality.

Commercial solutions, such as IBM Guardium [29] and IMPERVA SecureSphere [20], offer intrusion detection for databases for detection of misbehaving users. While detailed evaluation of these products is impossible due to their proprietary nature, we speculate that an attacker could easily evade their detection, since their analysis is bound to user sessions.

Ransomware Detection Several solutions have been proposed to detect and prevent ransomware at the file level. CryptoDrop [54], ShieldFS [12, 13] and Redemption [36] all monitor the file system to detect intrinsic ransomware behavior, such as file type changes, file entropy, and file similarity. They differ by their choice of observed properties, and by the mechanisms provided to prevent data loss, such as providing shadowed copies of files to possibly malicious processes. UNVEIL [35] tries to detect evasive ransomware by generating artificial user environments for dynamic analysis. However, their approach does not apply to server-side database ransomware. PayBreak [37] observes the use of symmetric keys commonly used by ransomware to encrypt files and holds them in escrow. This observation enables the recovery of the decryption keys upon ransomware detection. For the observed attacks on databases this approach hardly applicable since the files were deleted instead of encrypted.

FlashGuard [28] and RWGuard [48] propose ransomware-tolerant Solid-State Drives (SSDs) which are based on the property of SSDs to perform out-of-place writes in order to mitigate long erase latency. Both operate on the firmware

level and are effective in recovering encrypted files without impacting performance or lifetime.

The related work presented in this section targets client-side crypto-ransomware and is not applicable for detection of wipers at databases, as those do not use crypto primitives and do not access the file system directly.

Petri Nets and State Analysis Previous work has explored the concept of state analysis and more specifically the use of Petri nets for intrusion detection. Kumar et al. [39, 40] present a generic model and a misuse detection system for OS kernel audit logs using CPNs. This work is conceptually comparable to our work regarding the use of a Petri net to match attack patterns but focuses on intrusions in UNIX systems. Ilgun et al. [31] also focus on UNIX systems and use states and transitions to identify the necessary steps for penetrations, resulting in a flexible rule-based system to detect intrusions. Similarly, Shieh et al. [56] propose a pattern-oriented model with system states and transitions to identify context-dependent patterns of intrusion. USTAT [30] is a similar state transition analysis tool for UNIX systems, which describes penetrations as sequences of state changes and uses rule-based analysis of audit trails to identify intrusions. Ho et al. [25] describe the use of Petri nets for intrusion detection through the example of privilege escalation, again in UNIX systems. Helmer et al. [24] describe a general approach using Software Fault Trees to create CPNs for intrusion detection. The work focuses on modeling of intrusions and concentrates on the detection of FTP bounce attacks.

Overall, all the works discussed above are intended for intrusion detection in other environments, mostly in UNIX systems, and are not explicitly aimed at anomaly detection in databases or for ransomware detection.

7 Conclusion and Future Work

Ransomware attacks are an emerging threat, and their server-side variance that appeared recently imposes a significant threat to databases and stored data. In this work, we present DIMAQS (Dynamic Identification of Malicious Query Sequences), the first solution against server-side ransomware. In its heart, DIMAQS has colored Petri nets (CPN)-based classifier, which models malicious query sequences and matches them against query sequences captured at runtime. We introduce several novel extensions for the CPN, which allow us to reduce the complexity of the system representation and achieve better performance.

Our solution is implemented for MySQL servers and realized as a MySQL plugin, which is easily installable on existing servers. We evaluated our solution with regards to the precision of the attack detection as well as its performance and report no false positives, no false negatives and performance overhead under 5% for our non-optimized implementation.

In our future work, we plan to extend DIMAQS for detection of other attack types, since generally the framework

can be used for detection of arbitrary malicious query sequences and thus not necessarily limited to ransomware detection. Moreover, we will investigate possibilities for automated policy generation, which is potentially achievable given more elaborate malicious data sets and by applying machine learning techniques. Furthermore, we plan to perform performance optimization to decrease the imposed overhead further. Finally, we plan to develop new prototypes that target other database technologies⁶.

References

- [1] Manos Antonakakis, Tim April, Michael Bailey, Matthew Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai Botnet. In *USENIX Security Symposium*, 2017.
- [2] Stefan Axelsson. Intrusion Detection Systems: A Survey and Taxonomy. Technical report, Department of Computer Engineering, Chalmers University of Technology, Goteborg, Sweden, 2000.
- [3] E. Bertino, A. Kamra, E. Terzi, and A. Vakali. Intrusion Detection in RBAC-administered Databases. In *Annual Computer Security Applications Conference (ACSAC)*, 2005.
- [4] Christian Bockermann, Martin Apel, and Michael Meier. Learning SQL for Database Intrusion Detection Using Context-Sensitive Modelling (Extended Abstract). In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2009.
- [5] H. Chen, L. Amodeo, F. Chu, and K. Labadi. Modeling and Performance Evaluation of Supply Chains Using Batch Deterministic and Stochastic Petri Nets. *IEEE Transactions on Automation Science and Engineering (T-ASE)*, 2005.
- [6] Christina Yip Chung, Michael Gertz, and Karl Levitt. DEMIDS: A Misuse Detection System for Database Systems. In *Integrity and Internal Control in Information Systems (IICIS)*, 1999.
- [7] Catalin Cimpanu. A Benevolent Hacker Is Warning Owners of Unsecured Cassandra Databases. *Bleeping Computer*, 2017. URL: <https://bit.ly/2SiAnLz>.
- [8] Catalin Cimpanu. Database Ransom Attacks Hit CouchDB and Hadoop Servers. *Bleeping Computer*, 2017. URL: <https://bit.ly/2iVbas0>.
- [9] Catalin Cimpanu. Massive Wave of MongoDB Ransom Attacks Makes 26,000 New Victims. *Bleeping Computer*, 2017. URL: <https://bit.ly/2wAfq3X>.
- [10] Catalin Cimpanu. MongoDB Apocalypse: Professional Ransomware Group Gets Involved, Infections Reach 28K Servers. *Bleeping Computer*, 2017. URL: <https://bit.ly/2idWSRn>.
- [11] Catalin Cimpanu. MongoDB Hijackers Move on to Elasticsearch Servers. *Bleeping Computer*, 2017. URL: <https://bit.ly/2NX0SYk>.
- [12] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barengi, Stefano Zanero, and Federico Maggi. ShieldFS: The Last Word in Ransomware Resilient Filesystems. In *Black Hat USA*, 2017.
- [13] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barengi, Stefano Zanero, and Federico Maggi. ShieldFS: A Self-healing, Ransomware-aware Filesystem. In *Annual Conference on Computer Security Applications (ACSAC)*, 2016.
- [14] Oracle Corporation. *MySQL 5.7 Manual*, 2018. URL: <https://bit.ly/2xQAe8F>.
- [15] Muhaimin Dzulfakar. Advanced MySQL Exploitation. In *Black Hat USA*, 2009.
- [16] Hewlet Packard Enterprise. HPE ProLiant DL360 Generation9 (Gen9), 2014. URL: <https://bit.ly/2XL6iKt>.
- [17] Massimo Ficco and Luigi Romano. A Generic Intrusion Detection and Diagnoser System Based on Complex Event Processing. In *International Conference on Data Compression, Communications and Processing (CCP)*, 2011.
- [18] José Fonseca, Marco Vieira, and Henrique Madeira. Detecting Malicious SQL. In *Trust, Privacy and Security in Digital Business (TrustBus)*, 2007.
- [19] Dawid Golunski. MySQL-Exploit-Remote-Root-Code-Execution-Privesc-CVE-2016-6662, 2017. URL: <https://bit.ly/2SjtMAC>.
- [20] Rob Gravelle. *IMPERVA SecureSphere Database Audit and Protection*, 2018. URL: <https://bit.ly/2NZk2gm>.
- [21] Gregory T. Buehrer and Bruce W. Weide and Paolo A. G. Sivilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. In *International Workshop on Software Engineering and Middleware (SEM)*, 2005.

⁶E.g., for Prolog databases the ransom message insertion and table deletion could be mapped to the `assert` and the `retractall` commands.

- [22] William G. J. Halfond and Alessandro Orso. AMNESIA. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2005.
- [23] William G. J. Halfond and Alessandro Orso. Preventing SQL Injection Attacks Using AMNESIA. In *International Conference on Software Engineering (ICSE)*, 2006.
- [24] Guy Helmer, Johnny Wong, Mark Slagell, Vasant Honavar, Les Miller, Yanxin Wang, Xia Wang, and Natalia Stakhanova. Software Fault Tree and Coloured Petri Net-based Specification, Design and Implementation of Agent-based Intrusion Detection Systems. *International Journal of Information and Computer Security*, 1(1/2), 2007.
- [25] Yuan Ho, Deborah Frincke, and Donald Tobin. Planning, Petri Nets, and Intrusion Detection. In *National Information Systems Security Conference (NISSC)*, 1998.
- [26] Yi Hu and B. Panda. Identification of Malicious Transactions in Database Systems. In *International Database Engineering and Applications Symposium (IDEAS)*, 2003.
- [27] Yi Hu and Brajendra Panda. A Data Mining Approach for Database Intrusion Detection. In *ACM Symposium on Applied computing (SAC)*, 2004.
- [28] Jian Huang, Jun Xu, Xinyu Xing, Peng Liu, and Moinuddin K. Qureshi. FlashGuard: Leveraging Intrinsic Flash Properties to Defend Against Encryption Ransomware. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.
- [29] IBM. *IBM Security Guardium*, 2018. URL: <https://ibm.co/2ShttWW>.
- [30] K. Ilgun. USTAT: A Real-time Intrusion Detection System for UNIX. In *IEEE Computer Society Symposium on Research in Security and Privacy*, 1993.
- [31] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State Transition Analysis a Rule-based Intrusion Detection Approach. *IEEE Transactions on Software Engineering*, 21(3), 1995.
- [32] Intel®. Xeon® Processor E5-2640 v3 Specifications, 2014. URL: <https://intel.ly/2qFbGJX>.
- [33] Blake Ives, Kenneth R. Walsh, and Helmut Schneider. The Domino Effect of Password Reuse. *Communications of the ACM*, 47(4), 2004.
- [34] Konstantinos Kemalis and Theodoros Tzouramanis. SQL-IDS: A Specification-based Approach for SQL-Injection Detection. In *ACM Symposium on Applied Computing (SAC)*, 2008.
- [35] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. UNVEIL: A Large-scale, Automated Approach to Detecting Ransomware. In *USENIX Security Symposium*, 2016.
- [36] Amin Kharraz and Engin Kirda. Redemption: Real-Time Protection Against Ransomware at End-Hosts. In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2017.
- [37] Eugene Kolodenker, William Koch, Gianluca Stringhini, and Manuel Egele. PayBreak: Defense Against Cryptographic Ransomware. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2017.
- [38] Alexey Kopytov. akopytov/sysbench, 2018. URL: <https://bit.ly/2jjjeuf4>.
- [39] Sandeep Kumar and Eugene Spafford. A Software Architecture to Support Misuse Intrusion Detection. Technical report, Department of Computer Science, Purdue University, 1999. URL: <https://bit.ly/2Sij6C6>.
- [40] Sandeep Kumar and Eugene H. Spafford. A Pattern Matching Model for Misuse Intrusion Detection. Technical report, Purdue University, 1994. URL: <https://bit.ly/2YVb3xA>.
- [41] 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.
- [42] P. Liu. DAIS: A Real-Time Data Attack Isolation System for Commercial Database Applications. In *Annual Computer Security Applications Conference (ACSAC)*, 2001.
- [43] Peng Liu. Architectures for Intrusion Tolerant Database Systems. In *Annual Computer Security Applications Conference (ACSAC)*, 2002.
- [44] Wai Lup Low, Joseph Lee, and Peter Teoh. DIDAFIT: Detecting Intrusions in Databases Through Fingerprinting Transactions. In *International Conference on Enterprise Information Systems (ICEIS)*, 2002.
- [45] David C. Luckham and Brian Frasca. Complex Event Processing in Distributed Systems. Technical report, Stanford University, 1998. URL: <https://bit.ly/2YUIa4J>.
- [46] Sunu Mathew, Michalis Petropoulos, Hung Q. Ngo, and Shambhu Upadhyaya. A Data-Centric Approach to Insider Attack Detection in Database Systems. In *Lecture Notes in Computer Science*, RAID, 2010.

- [47] MediaWiki. MediaWiki/de — MediaWiki, The Free Wiki Engine, 2018. URL: <https://bit.ly/2XR0loW>.
- [48] Shagufta Mehnaz, Anand Mudgerikar, and Elisa Bertino. RWGuard: A Real-Time Detection System Against Cryptographic Ransomware. In *Research in Attacks, Intrusions, and Defenses (RAID)*, 2018.
- [49] Steve Morgan. Los Angeles Hospital Pays Hackers \$17,000 After Attack, 2016. URL: <https://nyti.ms/2Gr1It1>.
- [50] Steve Morgan. Cybersecurity Business Report. Ransomware Damage Costs predicted to hit USD 11.5B by 2019, 2017. URL: <https://bit.ly/2VNjsB1>.
- [51] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, 1981.
- [52] Alex Roichman and Ehud Gudes. DIWeDa - Detecting Intrusions in Web Databases. In *Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSEC)*, 2008.
- [53] Piotr Rygielski. vikin91/BibSpace, 2018. URL: <https://bit.ly/2JBr07c>.
- [54] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin R. B. Butler. CryptoLock (and Drop It): Stopping Ransomware Attacks on User Data. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2016.
- [55] semantic mediawiki.org. Semantic MediaWiki, 2018. URL: <https://bit.ly/30tny3U>.
- [56] Shih-Pyng Shieh and V. D. Gligor. On a Pattern-oriented Model for Intrusion Detection. *IEEE Transactions on Knowledge and Data Engineering*, 9(4), 1997.
- [57] Zhendong Su and Gary Wassermann. The Essence of Command Injection Attacks in Web Applications. In *SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2006.
- [58] Fredrik Valeur, Darren Mutz, and Giovanni Vigna. A Learning-Based Approach to the Detection of SQL Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2005.
- [59] Theuns Verwoerd and Ray Hunt. Intrusion Detection Techniques and Approaches. *Computer Communications*, 25(15), 2002.
- [60] Ofri Ziv. 0.2 BTC strikes back, now attacking MySQL databases, 2017. URL: <https://bit.ly/2JImQsR>.