DIMAQs - Dynamic Identification of Malicious Query Sequences

Master Thesis of

Michael Jobst

At the Department of Computer Science
Chair for Computer Science II
Software Engineering
Secure Software Systems

Advisor: Prof. Dr.-Ing. Alexandra Dmitrienko
I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Cologne, 09.06.2018

(Michael Jobst)
# Contents

1. Introduction .............................................. 1

2. Background ............................................. 5
   2.1. Petri Nets ........................................... 5
   2.2. Colored Petri Nets ................................. 7
   2.3. ShieldFS ............................................ 7

3. Related Work ........................................... 11
   3.1. Intrusion Detection for Databases ................. 11
   3.2. Ransomware Detection .............................. 13
   3.3. Petri Nets and State Analysis .................... 13
   3.4. Query analysis for detection of SQL injections .. 14

4. Attack Analysis ......................................... 15
   4.1. Problem Areas ...................................... 15
   4.2. Threat Analysis .................................... 15
   4.3. Threat Model ....................................... 17
       4.3.1. Areas of Reliance ............................ 17
       4.3.2. Attack Process ............................... 19

5. Requirement Analysis ................................. 23
   5.1. Database Backup Strategy ......................... 23
   5.2. Colored Petri Net .................................. 24
   5.3. MySQL – Information Exposure .................... 25
   5.4. MySQL – Variables and Storage .................. 27

6. Design .................................................. 29
   6.1. System Architecture ............................... 30
   6.2. Components ......................................... 30
       6.2.1. Monitoring .................................... 30
       6.2.2. Classifier ..................................... 34
       6.2.3. Policy .......................................... 37
       6.2.4. Resolution .................................... 40
       6.2.5. Notifier ........................................ 44
       6.2.6. Parser .......................................... 44
       6.2.7. Controller ..................................... 44
   6.3. Component Interaction ............................. 45

7. Implementation ......................................... 51
   7.1. LibPetri Implementation .......................... 51
## Contents

7.2. Plugin Integration ........................................ 54
7.3. Component Implementation ................................ 58
   7.3.1. Monitoring ........................................ 58
   7.3.2. Classifier .......................................... 61
   7.3.3. Policy .............................................. 63
   7.3.4. Resolution .......................................... 66
   7.3.5. Notifier ............................................ 68
   7.3.6. Controller .......................................... 68
7.4. Development Details ....................................... 69
7.5. Implementation Issues .................................... 70
7.6. Debugging .................................................. 71

8. Evaluation .................................................. 73
   8.1. Testbed .................................................. 73
   8.2. Data Sets ............................................... 74
   8.3. Effectiveness ........................................... 74
      8.3.1. Security Policy .................................. 75
      8.3.2. False Negatives .................................. 75
      8.3.3. False Positives .................................. 75
   8.4. Performance Evaluation ................................ 76
   8.5. Limitations ............................................. 77

9. Conclusion and Future Work ................................ 79
   9.1. Conclusion ............................................. 79
   9.2. Future Work ............................................ 80

Acronyms ...................................................... 81

Appendix ...................................................... 83
   A. SQL ...................................................... 83
   B. CPN Policy Examples .................................... 84
   C. Auditable Notifications .................................. 87

Bibliography ................................................. 89
Abstract

Ransomware is an emerging threat which imposed 5 billion USD loss in 2017 and is predicted to hit 11.5 billion in 2019. While initially targeting PC (client) platforms, recently it made a leap to server-side databases – in January 2017 we faced MongoDB Apocalypse attack, followed by other attack waves targeting a wide range of DB technologies such as MongoDB, MySQL, ElasticSearch, Cassandra, Hadoop, and CouchDB. While previous research has developed countermeasures against client-side ransomware (e.g., CryptoDrop and ShieldFS), no attention was given to the problem of server-side ransomware yet.

This thesis aims to bridge this gap and presents design and implementation of the tool DIMAQS (Dynamic Identification of Malicious Query Sequences) for MySQL servers that can efficiently and effectively detect server-side ransomware. DIMAQS performs run-time monitoring of incoming queries and pattern matching using a Colored Petri Net (CPN) for attack detection. The system design of DIMAQS exhibits several novel techniques to enable efficient detection of malicious query sequences globally (i.e., without limiting detection to distinct user connections). Evaluation results show high efficiency with no false positives and no false negatives, and a very moderate performance overhead (for a run-time monitoring tool) of under 5% in worst case scenarios.

Zusammenfassung


1. Introduction

In today’s era of digital transformation, data has become more important than ever before. The amount of data we produce on a daily basis is astonishing – every day hundreds of millions of people make videos, take photos, and exchange messages. Nowadays, data is not only a valuable asset for users that produce them, 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 them to optimize their business models and strategies. Given such trends, the importance of database security is hard to overestimate – the rapid growth of data volumes stored in 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 aim to undermine such security goals as confidentiality and authenticity. More recently, however, attacks against availability became common as well – modern attackers deploy ransomware, malicious software that encrypts data and holds the decryption key until the victim pays a ransom or locks out the users and applications so that the data is not accessible anymore. The most familiar and recent attack is the ‘WannaCry’ ransomware attack that infected Microsoft Windows platforms, encrypted user’s data, and demanded payment in the form of Bitcoin crypto currency from users in order to have their data restored. The attack affected 200 000 computers across 150 countries including large organizations such as the National Health Service in the UK, the American delivery company FedEx, and the Spanish telecommunications company Telefonica [12, 54]. The attack was described as ‘unprecedented in scale’ [8, 57] due to the scale and manner in which it brought affected services and production to a halt and the ensuing chaos thereof.

Financial loss from ransomware is very significant – it reached 5 billion USD in 2017 and is predicted to hit 11.5 billion by 2019 [52]. While the first ransomware attacks targeted client platforms and in particular information stored in user’s files, recently such attacks also started to aim at server-side databases storing, accumulating and processing big data. In particular, since December 2016 tens of thousands of MongoDB servers were hit in an attack called MongoDB Apocalypse [19, 20], and the first attack wave was followed by the second one targeting MySQL servers [72, 15].
Since then, ransom attacks have also spread to other server technologies, such as ElasticSearch \cite{17}, Cassandra \cite{16}, Hadoop and CouchDB \cite{18}.

The database ransomware attackers, using brute-force, obtained passwords and gained access to Database Management Systems (DBMS). Upon retrieving the access to the DBMS, the attackers ceased the databases and inserted a ransom message. The message contained information for database users with instructions on how to pay ransom to an attacker, if they were wishing to re-gain access to their data. The attackers demanded a value of 0.05 to 1 Bitcoins to be sent to a certain address in order to retrieve the database dump. There was however, no guarantee that the databases would be restored upon payment as there was no evidence of the created database dumps, meaning the data may have been lost forever if no personal backups existed.

This new kind of ransomware is called database ransomware and is different from known ransomware attacks. Already known types of ransomware are Locker ransomware and Crypto ransomware. Locker ransomware does not encrypt the files of the victim but instead, it denies the access to the device. The device’s user interface is locked and a ransom is demanded from the victim of the attack. Crypto ransomware infiltrates the victim’s device and silently identifies and encrypts valuable files. After successful encryption, the ransomware asks the user for a fee to access their files. Without the decryption key held by the attackers, the user loses access to the encrypted files. Crypto ransomware often includes a time limit. The prominent example of crypto ransomware attack is the WannaCry campaign which took place in 2017 \cite{6,28,38,40}.

Both, Locker and Crypto ransomware types are programs which are usually executed on the victim’s machine. Crypto ransomware can also access files via network protocols like Samba or NFS, but in such cases the files have to be read and deleted, or respectively overwritten. In contrast to that database ransomware consists of SQL statements only which are executed remotely. The attacker only needs knowledge of the log-in credentials for a user who has common rights for several databases and can thereby remotely execute the statements. Usually, the compromised servers are accessed from the Internet, but also internal attacks are not impossible due to Trojan horses, or other malware versions which can find their way into the internal network.

While ransomware attacks targeting databases are more recent and to this day less widespread than client-side ransomware, this situation might change soon. First of all, enterprises can afford to pay higher ransoms than private users. For 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 17 000 in ransom to its attackers \cite{51}. Secondly, according to a recent study, only 48% of victims of more traditional ransomware attacks (targeting file system) pay the ransom \cite{58}, contrastingly enterprises might have higher incentives to do so, given the higher value of data for business. Thirdly, in recent years, researchers and anti-virus companies developed countermeasures against client-side ransomware, however, to date no solutions exist against ransomware targeting databases. This makes databases easy attack targets.

Existing solutions against client-side ransomware are not applicable to the problem of server-side attacks. In particular, they follow two dominant strategies. The first one
builds upon detection of malicious binaries. Such an approach is typically used by
anti-virus vendors. The second strategy originates from research papers [21, 22, 61]
and relies on run-time monitoring of file accesses and detection of malicious activity
based on heuristics, such as access to multiple files, their modification, and renaming.
These defense strategies are not suitable for server-side ransomware, since these
attacks are mounted differently. Rather than dropping a malicious binary on the
platform, in the server-side scenario, an attacker connects to the database remotely
and then executes a sequence of malicious queries to select, modify and drop database
tables, and to insert ransom messages. Hence, no malicious binary exists that can
be detected. Furthermore, detection at file system level is ineffective either, since
there is no direct correlation between attacker’s activity and file access patterns.

Another potential solution to the problem would be the adaptation of existing
database security solutions, such as IBM Security Guardium [1] and Imperva [2].
Unfortunately, such solutions are limited to the analysis of single queries and their
expressions, but do not model attacks using query sequences. However, detection of
server-side ransomware would require analysis of query sequences – for instance, it
would be necessary to detect the combination of the individual queries to select and
drop a table, as well as the insertion of a ransom message (i.e., giving instructions
on how to pay the ransom). It is unlikely that analysis based on single queries would
be sufficient in telling malicious and benign activity apart, and would most likely
lead to a high rate of false positives and false negatives.

This thesis is aims to solve the problem of server-side ransomware and presents the
design and implementation of DIMAQS (Dynamic Identification of Malicious Query
Sequence) – the tool that monitors queries and identifies malicious query sequences
of ransom attackers. When the ongoing attack is detected, the tool makes a backup
copy of the targeted database and all the relevant tables before they are deleted. The
heart of DIMAQS’s attack detection engine is a colored Petri net (CPN), which is
used to model the sequences of events used for attacks. To the best of the our knowl-
edge, the tool is the first to model attacks considering query sequences rather than
single queries for malware detection, as well as the first tool to use colored Petri nets
with dynamic color creation for such purposes. The implementation targets MySQL
servers and imposes small performance overhead – the reference implementation is
only 5% slower than a MySQL server deployment without protection. The tool is
realized in the form of a MySQL plugin that can be easily installed on existing
MySQL servers, thus preserving compatibility with legacy software.

Outline. The remaining part of the thesis is structured as follows: Chapter 2
provides background information on Petri nets, Colored Petri nets (CPN), and ShieldFS.
Next, we analyze work that is related to the topic in Chapter 3. The typical attack
scenario and corresponding threats are analyzed in Chapter 4, and the resulting
threat model is derived. Requirement analysis for the MySQL plugin and corres-
dponding security requirements are covered in Chapter 5. The system design of
DIMAQS is presented in Chapter 6. Chapter 7 provides implementation details of
the developed system and describes how it can be used. Security and performance
evaluation of the proposed solution are provided in Chapter 8. The work is concluded
and possible directions for future work are discussed in Chapter 9.
2. Background

In this Chapter we provide the necessary background on Petri nets and on their enhanced version – Colored Petri nets. We also characteristics the technical properties of ShieldFS. Those concepts are partly used for the design and implementation of the proposed solution.

2.1. Petri Nets

Petri nets are a commonly used mathematical modeling language for the description of distributed systems [56] named after their inventor Carl Adam Petri. They are a class of discrete event dynamic systems. A Petri net give a graphical notation for step-wise processing that include iteration, concurrent execution, and choice. It describes a directed bipartite graph, in which nodes represent places and transitions, while directed 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 in order for the transition to be fired. The places from which an arc points to a transition are called the input places of the transition; the places to which arcs point from a transition are called the output places of the transition.

Places may contain a discrete number of marks called tokens. Transitions fire if they are enabled, which can be achieved by placing enough input tokens on the input places – i.e., places directly connected to the transition. The number of tokens required per place is defined by the value of the arc. Once a transition fires, it consumes the required number of input tokens from the input places. As a result of the transition, the specified number of output tokens is created on the places where the arcs are directed away from the transition (output places).

Figure 2.1 shows a simple example of a Petri net. Circles represent places, bars are transitions, arrows are arcs, and dots are tokens. The depicted Petri net consists of three places, one transition, and three arcs. For the transition to be enabled, three tokens are required: Two tokens at place $p_1$ and one token at place $p_2$. In Sub-figure 2.1a only one token is available at $p_2$. Regardless of the total count of three tokens, with only one token on $p_1$, the transition is not yet enabled. Adding
2. Background

Figure 2.1.: Demonstration of Petri net execution using a simple example

Figure 2.2.: Colored Petri Net example. In comparison to the regular Petri net depicted in Figure 2.1, the number of required places is reduced from two to one

another token to $p_1$ in Sub-figure 2.1b satisfies this requirement and thus enables the transition $t_1$. 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 $p_3$. Sub-figure 2.1c shows the state after the transition firing.

**Formal definition of CP-nets**

The definition of a non-hierarchical CPN [39, p. 70] is a tuple

$$CPN = (\Sigma, P, T, A, N, C, G, E, I)$$

with the following requirements:

- $\Sigma$ is a finite set of non-empty types, called *colour sets*.
- $P$ is a finite set of *places*.
- $T$ is a finite set of *transitions*.
- $A$ is a finite set of *arcs* such that $P \cap T = P \cap A = T \cap A = \emptyset$.
- $N$ is a *node* function. It is defined from $A$ into $P \times T \cup T \times P$.
- $C$ is a *color* function. It is defined from $P$ into $\Sigma$.
- $G$ is a *guard* function. It is defined from $T$ into expressions such that $\forall t \in T: [Type(G(t)) = Bool \land Type(Var(G(t)))) \subseteq \Sigma]$ where $p(a)$ is the place of $N(a)$.
- $E$ is an *arc expression* function. It is defined from $A$ into expressions such that: $\forall a \in A: [Type(E(a)) = C(p(a))_{MS} \land Type(Var(E(a)))) \subseteq \Sigma]$ where $p(a)$ is the place of $N(a)$. 


2.2. Colored Petri Nets

Petri nets are a powerful tool for modeling and allow for extensions to suit various tasks like queuing Petri nets for performance modeling. In this work, we use the extension named colored Petri nets.

2.2. Colored Petri Nets

A colored Petri net is an extension of a Petri net. Colored Petri nets (CPNs) enable support for tokens of different types, which are then referred to as token colors. Places can now contain tokens of multiple colors. Arcs can define any combination of the colors required for input and output tokens. This addition allows to model complex relations with a reduced number of places and transitions compared to regular Petri nets.

Figure 2.2 demonstrates an example of a CPN derived from the previous example and illustrates the reduction in representation complexity. In particular, the places $p_1$ and $p_2$ depicted in Figure 2.1 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 colored in 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 2.2 depicts the same process as before. In Sub-figure 2.2a, one black token is missing for the transition to be enabled. In Sub-figure 2.2b, this token is added, thus enabling the transition. Finally, in Sub-figure 2.2c, the transition has fired, subtracting two black and one red token from $p_1$ and adding a black token to $p_3$.

2.3. ShieldFS

ShieldFS [21] is a self-healing, ransomware-aware filesystem. The proposed solution is an add-on driver that makes the Windows native filesystem immune to ransomware attacks on file level. ShieldFS dynamically toggles a protection layer for each running process and acts as a copy-on-write mechanism according to the outcome of its detection component. The low-level filesystem activity is monitored internally to update a set of adaptive models that profile the system activity over time. As soon as one or more processes violate these models, their operations are deemed malicious and the side effects on the filesystem are transparently rolled back. ShieldFS requires a learning phase to classify benign and malicious behaviors.

ShieldFS is a detection system based on the combined analysis of entropy of write operations, frequency of read, write, and folder-listing operations, dispersion of per-file writes, fraction of files renamed, and the file-type usage statistics across multiple processes and OS facilities. The detection approach is applied in a real-time, self-healing virtual filesystem that shadows the write operations. Thus, if a file is surreptitiously altered by one or more malicious processes, the filesystem presents the original, mirrored copy to the user space applications. This shadowing mechanism is dynamically activated and deactivated depending on the outcome of the detection logic. The copied files are stored at a shadow drive, a so-called Volume
Figure 2.3.: ShieldFS: On the right ShieldFS shadowing a file offended by ransomware malicious write (MW), in comparison to standard filesystems (on the left); Graphic: ShieldFS Shadow Copy Service, which is only accessible by ShieldFS. Figure 2.3 depicts the logical activity of ShieldFS in comparison with a traditional filesystem.

The advantages of ShieldFS lie in the capability of copying the affected data before they are overwritten or deleted by a suspicious process. Other anti-ransomware software such as CryptoDrop only observe processes until they are classified as malicious and then stop them from further execution. By then, the ransomware would have already encrypted some files, which are very likely to be lost unless there is a possibility to obtain the decryption key after the ransom is paid. With ShieldFS, paying a ransom can be avoided if the system recognizes the ransomware as malicious in time. Furthermore, ShieldFS is advantageous by running in a privileged kernel mode, and it is implemented to be “non unloadable” at runtime, even by administrator users. Additionally, ShieldFS is able to deny any operation that attempts to delete or modify the driver binaries. The system was designed to also detect multi-process malware by implementing an incremental, multi-tier classification approach. Process-centric and system-centric models provide both the detection of single-process malware and multi-process malware, even if they are executed by multiple system users. When one or multiple processes were detected by ShieldFS, the system recovers the shadowed file copies automatically, and prevents the identified malware from further execution.

The disadvantages of ShieldFS are its false positives and false negatives, performance issues, and security. False positives can for instance be triggered when processes such as compression tools are not white-listed. False negatives can occur when the adversary performs the encryption process very slowly. If this process is very slow, the low percentage of files accessed during a specific period of time could avoid detection. In an ideal situation, a slow encryption process could be recognized by the user, who could then stop the process. A malware could attempt to compromise ShieldFS by filling up the shadow drive. This could lead to a state where the system is not able to continue anymore. An administrator-privileged process could try to prevent ShieldFS service from starting at boot time by modifying the Windows registry and force a reboot. A possible solution is to mitigate this limitation by embedding the approach directly into the kernel without the need for a service. The proposed system can then only be bypassed by compromising the OS kernel. The
required learning phase of ShieldFS is a disadvantage for database systems, because single queries such as 'Drop database' are often benign actions. Only additional queries like inserting a ransom message into a new or modified database table exposes them as malicious.
3. Related Work

In this Chapter, we provide an overview of the related work in four domains: (3.1) intrusion detection for databases, (3.2) ransomware detection, (3.3) application of Petri Nets for intrusion detection in various application domains, (3.4) SQL query analysis to detect SQL injections.

3.1. Intrusion Detection for Databases

Regarding Intrusion Detection Systems (IDS), we distinguish between anomaly detection [55] and misuse detection. Anomaly detection defines normal system behaviour and classifies all other behaviour as abnormal. Misuse detection defines abnormal behaviour and treats other behaviour as normal.

Previous work addressed the problem of intrusion detection in databases, but none of them focused on detection of ransomware, since this attack vector emerged very recently. In particular, Bertino et al. [9] used database logs to detect anomalous queries by matching against role profiles generated in a training phase. In contrast to our work, this approach analyses single SQL queries and checks only for irregular access patterns. DEMIDS [14] is a conceptually similar misuse detection approach to Bertino et al., which requires knowledge about the data structures in the database to identify average user working scopes (frequent item sets, attributes referenced together). Valeur et al. [67] provided an IDS that applies various models (e.g., character distribution and query length) to detect anomalous queries to a database. This approach targets attacks such as SQL injections, but cannot provide adequate protection against ransomware attacks, since it detects anomalies in single queries. Vaccaro and Liepins [66] defines a rule based approach in collecting benign activity queries from historic data to indicate the differences between current and normal patterns. Vigna et al. [69] goes a step further and provides a hybrid solution which combines analysis of web requests and SQL queries for more accurate results. Both solutions require a learning phase. DAIS [47] is an attack isolation system for Oracle Server that can re-write SQL statements to isolate database users that are deemed suspicious. All changes to the database by such users are performed on a shadowed incremental copy of the database. Similarly, Liu et al. [48] proposed a solution that
combines intrusion detection with the dynamic isolation of malicious and suspicious users. Their system creates virtual isolated databases that store changes to the database until a malicious activity is validated and either confirmed or declined. In our work, we use similar approach to preserve copies of database tables that might have been dropped by a ransom attack and inform an administrator about detected misbehaviour, providing an opportunity to revert malicious changes.

The most relevant work to ours is by Hu et al. [33], who proposed an intrusion detection system for databases that uses a Petri net to model data dependency relationships and regular data update patterns. The model is then used to detect irregular updates to the database. Their model is similar to our solution, in that they also rely on a Petri net. However, it is applied in a different way: We model malicious query sequences and look for their occurrences, while Hu et al. model the data structure and legitimate changes. As a result, the solution by Hu et al. requires a training phase to gain knowledge about the database under protection (dependency relationships and legitimate update patterns), while our solution can be applied to any database without learning. Furthermore, modeling data dependency relationships using a Petri net would likely result in a more complex system representation than ours, since we model a (relatively) simple sequence of queries. Furthermore, we use a number of optimizations (color information to reduce number of places, as well as token merging and token expiration) that help us to further reduce complexity and improve performance. In contrast, Hu et al. do not use color information and no similar optimizations. Moreover, Hu et al. didn’t provide any implementation and, hence, did not prove their performance overhead as being practical. Finally, this solution was intended for detection of other attack classes and it is not clear if it could be straightforwardly applied for ransomware detection.

In other work targeting real-time databases, Lee et al. [46] propose time signatures to capture expectations about update rates (e.g., sensor transactions) and flag unexpected and possibly malicious operations. This approach is likely to result in a high false positive rate for database systems with irregular usage patterns. DIWeBa [59] is an anomaly-based intrusion classifier for web databases that works at the session level by fingerprinting user sessions. The sessions are clustered through their sets of SQL statements, and anomalies are detected by matching against profiles generated in a training phase. In contrast to our work, this solution requires a training phase to learn benign behaviour of users, manual setup and knowledge of the database content. Furthermore, its detection does not analyze the query order and is limited to a single user session, while an attacker could use different sessions in order to compose a malicious query sequence. Our solution is not limited to single sessions and is capable of analyzing queries across all database connections. DIDAFIT [49] is another database intrusion detection system that creates and checks signatures of SQL statements and imposes sequence orders (which statements are allowed to follow previous statements) to identify anomalous queries. It is proposed that these orders are enforced by directed graphs containing the signatures as vertices, although no further details are presented.

Commercial solutions, such as IBM Guardium [1] and IMPERVA SecureSphere [2], offer intrusion detection for databases. These products claim to analyze and profile user activity and to identify malicious or compromised users. While detailed evaluation is impossible due to the proprietary nature of these products, we speculate
3.2. Ransomware Detection

Since ransomware was initially targeting computer systems, not databases, all anti-ransomware solutions were designed for PC platforms and focus on file-level detection. CryptoDrop [61] performs malware detection by monitoring ransomware indicators such as file type changes, file entropy, and file similarity through a Windows file system filter. It is effective against different classes of ransomware due to their intrinsic properties of accessing files and rewriting them with high entropy. UNVEIL [42] tries to detect evasive ransomware by generating artificial user environments for dynamic analysis through a similar file monitoring mechanism. Performing system testing on a large data set with 13,637 ransomware samples yielded zero false positives. Another example of a file system monitoring solution is ShieldFS [21, 22], a Windows file system add-on with a copy-on-write mechanism and anomaly-based ransomware detection. After a malicious process is discovered, it can be automatically stopped, resulting in a minimal number of corrupted files. ShieldFS, CryptoDrop and UNVEIL all work on the file system level, and are therefore not suitable for ransomware detection in databases.

3.3. Petri Nets and State Analysis

The concept of state analysis and more specifically the use of Petri nets for intrusion detection has been explored in previous work. Kumar et al. [45, 44] present a generic model and a misuse detection system for OS kernel audit logs using CPNs. Intrusions attempts are classified in UNIX systems via Existence, Sequence, Partial order, Duration, and Interval characteristics. This work is conceptually similar to our work regarding the use of a Petri net to match attack patterns, but focuses on intrusions in UNIX systems such as privilege escalation, whereas we focus on ransomware in databases. This approach’s level of complexity was analyzed by Myers et al. [53].

Ilgun et al. [35] 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. [63] propose a pattern oriented model with system states and transitions to identify context-dependent patterns of intrusion. Denning et al. [24] follows an analogical approach, but as anomaly detection instead of misuse detection. Profiles are serving as a signature or description of normal activity. USTAT [34] 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. [32] describe the use of Petri nets for intrusion detection through the example of privilege escalation, again in UNIX systems. By representing system events (such as the UNIX commands chmod or touch) as transitions between the Petri net states, they can detect a malicious event sequence though the processing
Related Work

Helmer et al. [31] describe a general approach using Software Fault Tree to create CPNs for intrusion detection. The work focuses on modeling of intrusions, and in particular concentrates on 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 or detecting ransomware in databases.

3.4. Query analysis for detection of SQL injections

Query analysis is a conventional technique to detect intrusions in databases and has been covered extensively by previous work, and is often used to detect and prevent SQL injection attacks. Fonseca et al. [26] propose an IDS based on anomaly detection, which checks SQL commands against a training set of valid query structures to observe SQL injections, and analyses transactions to detect more elaborate data-centric attacks. Kemalis et al. [41] declared milestones towards next-generation intrusion detection and performs detection of SQL injection attacks based on differences in query structure, where specifications are used to analyze the query parts for mismatches with the intended queries. These specifications require knowledge about every statement issued to the database by an application. Instead of specifications, Buehrer et al. [13] convert SQL requests into a parse tree structure to dynamically compare them with the intended queries. Similarly, Bockermann et al. [11] use parse trees and add tree kernels to compare SQL statements. AMNESIA [29, 30] checks the application code for SQL queries and generates automata for each query to match against the dynamic requests during operation. SQLCheck [65] validates queries by adding a key at the beginning and the end of each user’s input. At runtime, the “augmented” queries that are not in valid syntactic form are considered attacks. A potential threat for database systems is the use of mimicry and evasion attacks [27]. In such attacks, a query is split into partial sub queries which are malicious as a whole. A general analysis of mimicry attacks on IDS was done by Wagner et al. in [70].

The works discussed in this Section are aimed at SQL injection detection, and therefore different to our approach, which is aimed at detecting ransomware. Furthermore, none of them uses Petri nets for system representation.
4. Attack Analysis

In this Chapter, we discuss the problem areas, analyze the threat, and describe the threat model by demonstrating a recent database ransomware attack step by step.

4.1. Problem Areas

In the past, we observed ordinary ransomware attacks against big companies such as Deutsche Bahn, Nissan, and many other companies and institutions [28, 38]. City Councils [40], Hospitals [6] and other public institutions are not immune against such attacks. In 2015, the Wall Street Journal already declared ransomware as a growing threat to small businesses [64], as they are not as well prepared as large companies in terms of security.

Regarding database ransomware attacks, more than 45,000 servers were already compromised in total [15] since 2016. Companies are usually careful not to tell such information to the public so as not to scare consumers. However, a spreadsheet referenced by Binaryedge [10] shows that there were worldwide 124 companies and institutions victims of such attacks from January 3rd 2017 to January 15th 2017. The analyzed data shows that none of the ten victims who paid the ransom got the data back. Only 11.3% had a recent backup meaning that the data was lost to the other 88.7% of the victims.

Based on the date of the mentioned spreadsheet we can group the attacks by country and business field. The following graphics in Figure 4.2 show the evaluation.

The chart 4.2a shows the deviation of successfully attacked different companies grouped by country. 53% of the attacked companies are located in the United States of America and United Kingdom. Chart 4.2b expresses that 72.5% of the attacked companies working in the production business. The data, however, does only include recorded attacks. There is a high probability that the unreported number of attacks is much higher.

4.2. Threat Analysis

In the following, we provide an analysis of a large-scale ransomware attack specifically targeting MySQL servers that took place in Feb 2017 [72]. While we concentrate
4. Attack Analysis

![Threat Analysis Charts](chart1)

**Figure 4.1.** Threat analysis charts

**a)** 11.3% of different victims had recent database backups, 88.7% had not

**b)** 10 victims paid ransom, 114 did not

![Analysis by Country and Business Sector](chart2)

**Figure 4.2.** Analysis of attacks by country and business sector

- **a)** Analysis: companies grouped by country
- **b)** Analysis: companies grouped by business sector

On attacks targeting MySQL servers, attack scenarios against MongoDB servers were very similar.

In an initial step of the attack, an attacker determines IP addresses hosting MySQL servers by online providers like shodan.io and other lists (1). In the next step, the attacker gains access to a MySQL server (2) by brute-forcing the 'root' password of the database. This step can be accomplished on databases with insecure or default passwords. At the time of attack occurrence, most of the MySQL servers had no root password set, due to insecure default configuration [23]. Even with a configured password, this step could succeed in many cases due to the well-known problem of weak or re-used passwords [37].

In a third step of the attack (3), the gained root access was used to retrieve a list of the existing databases. This step could be done by the `SHOW DATABASES` command, or by selecting from information_schema tables. Next, the attacker creates a new table with the name `WARNING`, either in a new database named `PLEASE_READ`, or in an existing database. This table includes a ransom message containing contact email address, as well as payment instructions to a Bitcoin address and a Bitcoin value.
After insertion of the ransom message, the attacker dumps the databases, though some attack instances skip this step entirely. Finally, the intruder deletes (drops) the databases on the server and disconnects from the MySQL server.

The scenario above describes an exemplary attack scenario with real attacks showing slight variations regarding the sequence of steps and used table names. For instance, the database containing the ransom message had a variety of names, such as WARNING, PLEASE_READ, README_MISSING_DATABASES, READ1, PLEASE_READ_ME and many more. A full list of observed database names is listed in Table 4.1. Also, the attacker could first insert the ransom message and then perform the database drop.

It is to be expected that even if the victims are prepared to pay the ransom, the contact and crypto currency payment information were already overwritten by other attackers, making the payment impossible. The following Figure 4.4 shows an observation of the top 20 MongoDB database name usages on the 16th and 17th of January 2018. It shows that within one day, many of the accessible databases had been changed. For example; on the 16th, there were 18613 databases with the name WARNING observed. Only one day later, there were only 312 databases with that name, but 19051 with the name PLEASE_READ. This makes payment of the ransom almost impossible.

As analyzed by Binaryedge [10], 278 Bitcoin transactions have been made with a total value of 25,061,868,18 BTC. The last change in the study was made on January 10th 2018, meaning that the actual number might be higher.

### 4.3. Threat Model

In this Section, we analyze the attack vectors in detail. First, we define the areas of reliance and thereby determine which affected elements are trustworthy. Next, we describe the attack process which is always carried out in a similar scheme as mentioned in the previous Section.

#### 4.3.1. Areas of Reliance

The areas of reliance characterize a set of actors, software and hardware components that influence the security of the system and need thereby to be evaluated for weaknesses.
4. Attack Analysis

We identify the three potential actors. They are system administrators, database users, and attackers. As it is not possible to distinguish between database users and attackers due to password brute-forcing and leaking, we consider both untrustworthy. System administrators usually have access to the physical machines and its operating system, and must therefore be trusted. An attacker should not have access to the physical machine and its operating system.

Regarding software components, we identify the DBMS and other software which is running on the system. On UNIX systems, the DBMS usually runs under a dedicated MySQL user account, and can thereby only be modified by that user, or a privileged user. On Windows systems, there are two situations. First, a DBMS runs usually as a service. Second, it can also be started by a user directly. In the first situation, an attacker would have to gain administrative rights in order to access the DBMS directly. This can be prevented by making use of strong password policies and the activation of Windows’ User Account Control (UAC). In the second situation, the DBMS is directly accessible by Software running under that user account. However, this situation is more likely practiced by developers and will therefore not be further taken into account.

Hence, local executed software should not be able to access the DBMS directly, but by the networking interface. As such communication between software applications and database systems is the usual practice, we classify the communication via TCP/IP and local sockets as non-problematic.

The Operating System (OS) and the hardware need to be trustworthy, as these are central components. If these are compromised, the attacker has unlimited access to the MySQL configuration file and the database storage files. Thereby, the attacker could change the configuration, or respectively delete files in order to deactivate the proposed MySQL plugin and destroy valuable data.
4.3.2. Attack Process

In the following, we analyze the attack process in detail and give an overview of the queries executed by an attacker.

List Databases

Once the attacker is connected to a MySQL server, he tries to list the existing databases and/or tables. There exist multiple ways to do that. A list of MySQL query types exposing that information is contained in the appendix Section Table 5.1. After receiving the names of existing databases or tables, the attacker emits their deletions and the insertion of a ransom message. The order of execution of these steps vary in observed attacks.

Drop Database or Table

As soon as the attacker knows the names of the existing databases, the databases can be deleted. Additionally, the user could also only delete the tables in each database.

The following commands exist for dropping databases and tables:

- Database:
  \[
  \text{DROP \{DATABASE | SCHEMA\} db\_name}
  \]

- Table:
  \[
  \text{DROP TABLE [IF EXISTS] tbl\_name [, tbl\_name] ...}
  \]

The database names db\_name and table names tbl\_name must be written in hard code in the query. For MySQL server, it is not possible to build the name of the object in a dynamic query.

Create Database

The following command exist for creating a database:

\[
\text{CREATE \{DATABASE | SCHEMA\} [IF NOT EXISTS] db\_name}
\]

Table 4.1 contains database names that were used during observed MongoDB and MySQL server attacks.

Creating a database however, is not a necessary step in executing the attack. The attacker could also create a table within an existing database.
4. Attack Analysis

WARNING
README_MISSING_DATABASES
PLEASE_READ
PWNED
PWNED_SECURE_YOUR_STUFF_SILLY
ReadmePlease
CONTACTME
WARNING_ALERT
to get_DB_back_send1BTC_to_1DGztzLNz1eufswtqMDWPMWSgwthdpXRtC
DELETED_BECauses_YOU_DIDNT_PASSWORD_PROTECT_YOUR_MONGODB
PLEASE_READ_56b41cc944bd390932e79827
README
LEIA_ME
AVISO_LEIA_ME
IHAVEYOURDATA
READ_ME
READMEPLS
ENCRYPTED
READ1
README_YOU_DB_IS_INSECURE
AVISO
DB_H4CK3D
PLEASEREAD
DB_DROPPED
REQUEST_YOUR_DATA
BACKUP_DB
Attention
PLEASE_READ_ME
PLEASEREADTHIS

Table 4.1.: Used database names in MongoDB
4.3. Threat Model

Create Table

The following commands exist for creating a table:

- \texttt{CREATE TABLE [IF NOT EXISTS] tbl_name}
  ...
- \texttt{CREATE TABLE [IF NOT EXISTS] tbl_name [AS] query_expression}
- \texttt{CREATE TABLE [IF NOT EXISTS] tbl_name \{ LIKE old_tbl_name \| (LIKE old_tbl_name)}

The only observed table name used during the attacks was \texttt{WARNING}, but also other table names, as listed in Table 4.1, can be used. The attacker could use any combination of malicious table names, therefore, the name of the table is not a reliable attack indicator of a possible next step in a malicious query sequence.

Insert Ransom Message

The following commands exist for inserting values into a table:

- Insert value list:
  \texttt{INSERT [INTO] tbl_name \{VALUES | VALUE\} (value_list) [, (value_list)]}
- Insert select:
  \texttt{INSERT [INTO] tbl_name SELECT ...}

A ransom message consists of a text containing specific information about what happened, and how to pay to get the data back. Regarding GuardiCore [72], the following two versions were observed:

**Version 1**

\begin{verbatim}
INSERT INTO PLEASE_READ.'WARNING'(id, warning, Bitcoin_Address, Email)
VALUES('1', 'Send 0.2 BTC to this address and contact this email with your
ip or db_name of your server to recover your database!
Your DB is Backed up to our servers!',
'1ET9NHZEXXQ34qSP46vKg8mrWgT89cfZoY',
'backupservice@mail2tor.com')
\end{verbatim}

**Version 2**

\begin{verbatim}
INSERT INTO 'WARNING'(id, warning)
VALUES(1, 'SEND 0.2 BTC TO THIS ADDRESS 1Kg9nGFdAoZwmrn1qPMZtam3CXLgcxPA9
AND GO TO THIS SITE http://sognd75g4isasu2v.onion/ TO RECOVER YOUR DATABASE!
SQL DUMP WILL BE AVAILABLE AFTER PAYMENT! To access this site you have use
the tor browser https://www.torproject.org/projects/torbrowser.html.en')
\end{verbatim}

The two versions differ in the structure and in the information they expose. In Version 1, the ransom information is split into separate columns, whereas Version 2 has only one text column containing all the information. Both versions contain a value of Bitcoins (‘0.2 BTC’) and a 34 characters long Bitcoin address consisting of letters and numbers. This information is crucial for the detection of an inserted ransom message.
A very detailed description of attacks and their ransom information is exhibited in a Google Docs spreadsheet, which is linked by the article on Binaryedge [10]. The analysis shows that each attack left a Bitcoin value. Almost every attack left a Bitcoin address, but some only left an email address that the victim has to contact to get the payment information.

**Database Dumps**

It is expected that the attacker would create a database backup (dump) as mentioned in the ransom messages. However, the attacker does not always carry out this step. Therefore, the creation of database dumps is not a requirement for a malicious query sequence. Without these dumps, the data is lost after dropping if there was no personal backup done by the DBMS administrator.
5. Requirement Analysis

The priority of this master thesis is to prevent data from being lost, and notify the administrator after a database ransomware attack detection. The proposed system has to fulfill the following characteristics:

- system that tracks the executed queries
- allow authentication for privileged mode
- backup data tables
- hide backed up database tables from unprivileged users
- restore backed up tables by privileged user
- send email notification to an administrator

The following Sections provide information on how we achieve these characteristics.

5.1. Database Backup Strategy

A mechanism is required that copies dropped database tables which should be restorable by a privileged user. The database tables are backed up immediately when deleted, and then we wait for further malicious queries. The deletion of a data table is not enough, as it could also be a benign command. An administrator is notified about an incident if an attack has been detected. The backed up data should not be accessible by a normal database user, and also its existence should be hidden.

The proposed mechanism is comparable to the ShieldFS [21] approach as described in Section 2.3. The big difference is that we cannot observe operations on file level but on database query level. Queries such as DROP DATABASE and DROP TABLE can be monitored by an auditing plugin in MySQL server. The plugin is able to intercept the command, rewrite the query, and execute further commands.
5.2. Colored Petri Net

Section 4.3 shows how database ransomware attacks are executed. An appropriate proposal for detecting malicious query sequences is to trace significant queries and affected database objects using a colored Petri net (CPN). For a detailed explanation of CPNs see Section 2.2.

The benefit of using a CPN is it only acts on relevant events. This means that the state of the Petri net does not change on queries which are not part of the defined malicious query sequence. This behavior provides a high performance and low memory consuming query sequence analysis. The results of performance and memory consumption are provided in Chapter 8.

When a place contains at least one token, we declare such a place as active. A transition must add at least one token to a place marking that place as active during transition. A Petri net can consist of multiple active places. Only active transitions are checked for malicious queries. Using this approach, it is possible to also track variations of query sequences with only one Petri net. Therefore, a path from one place to another specifies the dependence of single queries. For instance, the attacker has to know which databases exist before they can be dropped.

An illustration of the used Petri net and its description are provided in Section 6.2.

The usage of the CPN cannot be used in its original form, but needs to modified and some aspects such as the Color Function, Forward-Copy Arcs, and Transition Actions must be added in order for the identification of database ransomware to occur.

Modification: Dynamic Color Creation

The coloring function adds a data object at the transition point to a moved token. The data object contains information such as a time stamp, the name of the triggered transition, and information which explains why the transition was triggered. An example is the name and corresponding database name of a deleted table.

Modification: Token Duplication

Transitions having an inward and outward arc to the same place, leave the tokens at this place, and forward a copy of the affected tokens extended by the transition data to the other outward places. Each transition may only be connected to at least one place with inward and outward arcs.

Addons – Transition Action

Each transition is triggered by one specific action. An action has a isMatching function which returns a positive or negative result based on query matching. If the result is positive, the transition is triggered.

Addons – Transition Always Action

Each transition $t \in T$ is triggered if the corresponding Transition Action matches. In the case of two or more executed steps (e. g. a table or database was deleted and a ransom–like message was inserted), we have at least two places with tokens. The Transition Always Action which is connected to these two places can immediately emit further steps when activated.
Addons – Transition Condition

A transition condition is an extension of Transition Actions which checks the token colors for specific values. This feature is used for actions that act on the same database object, e.g., the same table.

Addons – Place Action

The places $p \in P$ are extended by Place Actions. Each place $p$ can have multiple action assigned to it. A Place Action is emitted at the end of the classification process when a token is transferred to the place where the Place Action has been assigned. A Place Action is e.g., a notification to the Administrator about an incident or backing up a database before deletion.

Addons – Token Merging

During transition, tokens are moved or copied from one place to another. This could lead to a high amount of tokens at the same place which contain equal data, excluding the time stamp of token creation. To reduce the amount of tokens for performance and memory reasons, the tokens are merged by only using the newer time stamp at the target place.

Addons – Token Expiration

Each place is extended by a timeout value. The time out expresses how long a token is occupied by a place before the token is removed. This feature reduces the amount of false positives due to ‘forgetting’ executed queries after a period of time.

5.3. MySQL – Information Exposure

As a requirement, we want to hide sensitive information from potential attackers. Therefore, we analyzed which commands and tables expose such information.

Table 5.1 in the appendix Section shows commands and tables that expose information about existing database schemes, tables, and columns. Those commands and tables marked with a $\times$ are relevant because they show information about backed up data tables and their existence. This information should be hidden from potential attackers. The entries marked with a $/$ either do not show a complete list entities, or are dependent on the used storage engine. They are therefore not relevant for attackers.
### Table 5.1: The table shows commands and tables which expose relevant data for database ransomware detection.

<table>
<thead>
<tr>
<th>Command</th>
<th>Reference</th>
<th>Exposed Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHOW {DATABASES</td>
<td>SCHEMAS}</td>
<td></td>
</tr>
<tr>
<td>SHOW TABLES</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SHOW {COLUMNS</td>
<td>FIELDS}</td>
<td></td>
</tr>
<tr>
<td>SHOW TRIGGERS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHOW OPEN TABLES</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SHOW TABLE STATUS</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schema</th>
<th>Table</th>
<th>Reference</th>
<th>Exposed Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLUMNS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>COLUMN_PRIVILEGES</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>EVENT</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>FILES</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>KEY_COLUMN_USAGE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PARTITIONS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>REFERENTIAL_CONSTRAINTS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>ROUTINES</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SCHEMA</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SCHEMA_PRIVILEGES</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>TABLES</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>TABLE_CONSTRAINTS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>TABLE_PRIVILEGES</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>TRIGGERS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>VIEWS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>INNODB_BUFFER_PAGE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>INNODB_BUFFER_PAGE_LRU</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>INNODB_SYS_COLUMNS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>INNODB_SYS_DATAFILES</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>INNODB_SYS_FIELDS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>INNODB_SYS_FOREIGN</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>INNODB_SYS_FOREIGN_COLS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>INNODB_SYS_TABLES</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>INNODB_SYS_TABLESPACES</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>INNODB_SYS_TABLESTATS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

| mysql | | | X |
| db | | | X |
| innodb_index_stats | | | X |
| innodb_table_stats | | | X |
| proc | | | X |
| procs_priv | | | X |
| tables_priv | | | X |

| performance_schema | | | X |
| file_instances | | | X |
| file_summary_by_instance | | | X |
| objects_summary_global_by_type | | | X |
| table_handles | | | X |
| table_io_waits_summary_by_index_usage | | | X |
| table_io_waits_summary_by_table | | | X |
| table_lock_waits_summary_by_table | | | X |

| information_schema | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |
| X | | | X |

Table 5.1.: The table shows commands and tables which expose relevant data for database ransomware detection.
5.4. MySQL – Variables and Storage

To configure the proposed solution and store the database backups, we need to be able to store specific types of values. Therefore, we make use of variables and the database storage engine provided by MySQL server.

Variables

We require the following MySQL variables for our solution:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object_prefix</td>
<td>String</td>
<td>Name of backed up database objects and hidden values from result sets</td>
</tr>
<tr>
<td>secret</td>
<td>String</td>
<td>Secret value to enter privileged mode</td>
</tr>
<tr>
<td>email</td>
<td>String</td>
<td>Email address of the recipient being informed</td>
</tr>
<tr>
<td>verbosity</td>
<td>Integer</td>
<td>0: OFF - No output is generated (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Only plugin init and deinit messages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: Output of (1) and Place Action</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3: Output of (2) and Query Rewriting</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4: Output of (3) and Classification</td>
</tr>
<tr>
<td>excluded_dbs</td>
<td>String</td>
<td>Comma separated database names to be excluded from of Classification</td>
</tr>
<tr>
<td>max_capacity</td>
<td>Integer</td>
<td>Maximum capacity (MB) of backed up databases</td>
</tr>
<tr>
<td></td>
<td></td>
<td>default: 0 (unlimited)</td>
</tr>
<tr>
<td>storagespace</td>
<td>String</td>
<td>Database name in which backed up tables are stored.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Automatically added to excluded_dbs</td>
</tr>
</tbody>
</table>

Table 5.2.: Required MySQL variables

Storage Engine

This storage type is used to store backed up tables and should not be accessible to anyone but the administrator. When the tables are moved to this storage after a user or attacker deleted them, an administrator can restore the backed up database objects by simply renaming them. The storage is a simple database. The name of this database is specified by the MySQL variable dimaq_storage as mentioned above. To avoid naming collisions, the tables are named according to the following scheme <object_prefix>_<dbname>_<tablename>_<timestamp> with object prefix being the pre-configured variable. The name parts are concatenated with underscores.
6. Design

DIMAQS is the first system that aims at 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 in order 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 [45, 33]. However, typical existing application scenarios of CPN-based intrusion detection systems target other environments, e.g., networks [68] and operating systems [7]. The application of (uncolored) Petri nets to databases was only considered in a theoretical work by Hu et al. [33]. However, as we discuss in detail in Chapter 2 there are substantial differences in what is modeled and how the detection is performed. Our approach has many practical advantages – for instance, in contrast to the solution by Hu et al., we do not require any prior knowledge about the data structure and regular data update patterns. What is essential to know is the sequence of queries and its characteristics that leads to a malicious and undesired situation.

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 databases in general and for ransomware attack detection in particular. We identify that databases are complex systems. The modeling of their states in terms of dependency relationships and update patterns, as e.g., done in [33], may lead to over-complicated system representations for large and complex databases, and produce 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 also allows us to detect attacks that are carried out over multiple sessions and multiple user accounts.
For our CPN, we develop several novel techniques to even further simplify the system representation. In Section 5.2, we analyzed additional requirements for the Petri net. Of those requirements, the main ones are

1. dynamic color creation,
2. token merging,
3. token duplication, and
4. token expiration.

By using these techniques we reduce the overall complexity and make the use of CPNs practical in the given context.

In the next Section 6.1, we declare the general architecture of the our solution. We then describe the required system components in detail in Section 6.2. Finally, we show how those components interact to handle incoming queries and their classification in Section 6.3.

6.1. System Architecture

This Section gives a general overview of the proposed solution’s architecture.

Our solution is provided as a MySQL server auditing plugin. MySQL server provides a pluggable audit interface that enables information about server operations to be reported to our plugin. We take a closer look at this interface in Section 6.2.1.

Figure 6.1 shows the architecture of the developed DIMAQS MySQL plugin and demonstrates the communication paths between the individual components. The interaction between the components is described in Section 6.3.

The system architecture graphic shows the components which are on the one hand integrated in the MySQL server package, and on the other hand contained in the DIMAQS plugin. The plugin is comprised of the six components: (i) Monitoring, (ii) Classifier, (iii) Policy, (iv) Resolution, (v) Notifier, and (vi) Controller. The Monitoring component consists of fractions of the MySQL server and the DIMAQS plugin, simultaneously. Additionally, DIMAQS uses the query parser embedded in the MySQL server. In the following, we describe the role of every component in more detail. The interaction of the components is explained in Section 6.3.

6.2. Components

In this Section, we describe the developed modules of the DIMAQS MySQL plugin and give a detailed explanation of their functions.

6.2.1. Monitoring

All incoming queries are monitored by the Monitoring component 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 are usually generated by setting triggers directly inside the MySQL server, or by using the server’s audit notification functionality.
6.2. Components

This module uses the trigger feature to alert the Controller of events for which not all relevant information is directly accessible by the database server due to run-time execution. The Monitoring component is thereby also responsible for setting up the triggers during MySQL Server start-up phase and after a new table has been created.

An audit notification can be emitted by the MySQL server for these operations:

- Write a message to the general query log (if the log is enabled)
- Write a message to the error log
- Send a query result to a client

The DIMAQS plugin registers to the MySQL server with specific notifications to be received, that inform about server operations. When an audible event occurs within the server, the server determines whether notification is needed. This allows for high performing surveillance of server operations.

Currently, MySQL supports appropriate notifications during the 5 remarkable checkpoint events: Connect (1), Preparse (2), Postparse (3), Result sent (4) and Disconnect (5). A full list of audible events is provided in Table C.1 in the Appendix Section C.

Figure 6.2 expounds event points at which the plugin can currently receive notifications. We use the events (1) and (5) to monitor established connections. The connection information is forwarded to the Controller where it is used to allow single connections to be used in privileged mode. The parsing events (2) and (3) notify the parsing state and contain additional query information (e.g. query type, table names). Event (4) communicates to DIMAQS that the query result was sent to the client. These events are mainly used for the classification process.

Figure 6.1.: Interaction between the components of DIMAQS.
The auditing functionality is sufficient to monitor almost all events and query classes which are mandatory to detect malicious sequences. The only event class which cannot be properly observed is during inserts and updates (at Execution). The following paragraph gives reasons for this and shows how we bypass this limitation.

**Calculated Values**

MySQL allows the assembling of Data Manipulation Language statements (DML) such as insert and update statements. The following statement (i) declares the actual inserted values (‘TEXT 1’ and ‘TEXT 2’) in a list into table A. Statement (ii) is an example of an assembled construct in which the selected values from table B are inserted into table A. In contrast to (i), where the inserted values can be determined by the query directly as plain text, the actual inserted values in query (ii) are not visible, because they are calculated during execution.

(i) INSERT INTO TABLE A values ('TEXT 1', 'TEXT 2');
(ii) INSERT INTO TABLE A SELECT * FROM B;

In Statement (ii), the real inserted values are calculated during *Execution* between event *Postparse* (3) and *Result sent* (4). Since there is currently no such event provided by MySQL, we use triggers to provide additional information about the inserted and updated values sent to the plugin. In the next paragraph, we describe the sub-module **Trigger Creator** which is responsible for the setup of the triggers for each observed table.
Trigger Creator

The Monitoring component uses a Trigger Creator internally. The Trigger Creator is executed
- during plugin loading,
- after a new table was created, and
- after the modification of a table structure.

Some of the triggers might already exist for observed tables. Therefore, they are always recreated, because the table structure might have changed in the interim. Triggers cannot be created within stored procedures. Hence, there must be a creation from within the plugin code. The plugin initial point is called directly after initialization of the server and before servicing. The (re-)creation of the triggers is done at that point for all observed tables. The Trigger Creator also (re-)creates triggers directly after creating and modifying tables.

Each observed table has its own insert/update trigger. During an INSERT or UPDATE of rows of tables, the corresponding trigger executes a user-defined function which is provided by the plugin. This function call transfers the values to the plugin for evaluation. Currently, we see this method as the only existing possibility to evaluate the actually inserted and updated values as there are currently no hooks available for MySQL server that can be used instead.

The communication paths are illustrated by sequence diagrams in Section 6.3.

Regarding the MySQL Documentation, each trigger must refer to a permanent table. A trigger cannot be associated with TEMPORARY tables and views. Thereby, our solution does not address TEMPORARY tables as they do usually not contain valuable information.

User-Defined Value Function

The DIMAQS plugin provides a function which is activated by triggers directly for evaluating inserted and updated values. This function passes the following values to the plugin:
- scheme name
- table name
- a list of new column values

With this approach it is possible to determine the real values which are inserted into or updated from a database table. If such a trigger exists for the corresponding table, the function is activated for each inserted and updated row. The new column values contain the values which are stored into the columns. Thereby, we are also able to determine mimicry attacks, because all plain values of a inserted and updated rows are transferred for classification. An example of such a mimicry attack type is inserting a line with only a part of the ransom message and then adding additional information to the ransom message by updating the columns. Therefore our solution is able to recognize mimicry attacks and avoid them by sending all row values to the DIMAQS plugin. We provide further information about query classification in the next Section.
6.2.2. Classifier

The Classifier component processes the incoming queries and renders a verdict whether a query is considered benign or malicious. For the classification, DIMAQs uses a CPN with our extensions. In particular, we use the coloring function of the CPN to attach additional information to a token, e.g., a Transition type and a Table name. Since timing information can cover an indefinite space, this also creates an indefinite space of colors. Hence, the attached information can have arbitrary complexity, e.g., objects with member variables. Information stored this way is often used to determine how and why the token reached a specific place. The information inside the color element can also be used to enable a transition only if the color is in a specific range.

Every transition in the CPN colors its output tokens with the current time-stamp as well as information about the occurred action, thus creating colors dynamically at run-time. For example, at transition $D_T$ (Drop Table) the name of the deleted table is added to the token. This information is later useful, e.g., for the administrator notification.

**Token Color Function**

The color of a token results from an ordered list of token types which is usually the name of the transition that adds the information and additional information such as e.g., a table name. The tokens at initial states contain an empty value list. A new value is added to a token during transition firing as described in the next paragraph.

Additionally, each token data contains the timestamp of when the transition was triggered.

The color of two tokens is the same, if and only if:

- the count of token values is equal, and
- the type of the token values exist in the same order, and
- the corresponding object values (except the timestamp) are equal.

The following illustrations in Figure 6.3 show examples of when token colors are equal. Only the first situation (6.3a) produces an equal color for two given tokens. The others (6.3b, 6.3c, and 6.3d) result in different colors.

When two tokens have identical colors, these two two tokens are equal, and a merging of tokens is possible during a transition.

**Transition Firing**

Each query to be analyzed by the defined CPN is checked by all active transitions whether it matches

1. the transition type (see previous paragraph), and
2. the transition condition.

A transition condition denotes a proposition between token values, e.g., a table name, and the analyzed query. A token of the CPN contains a list of information items. A new item is appended to the list during a transition.
### 6.2. Components

#### Figure 6.3.: Token Colour Equality Explanation

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>DD</td>
<td>MyDB</td>
<td>CT</td>
<td>Warning</td>
</tr>
<tr>
<td>a)</td>
<td>equal, token types and object values match</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>DD</td>
<td>MyDB</td>
<td>CT</td>
<td>Warning</td>
</tr>
<tr>
<td>LD</td>
<td>DD</td>
<td>MyDB</td>
<td>CT</td>
<td>Warning</td>
</tr>
<tr>
<td>LD</td>
<td>DD</td>
<td>MyDB</td>
<td>CT</td>
<td>Warning</td>
</tr>
<tr>
<td>LD</td>
<td>DD</td>
<td>MyDB</td>
<td>CT</td>
<td>Warning</td>
</tr>
<tr>
<td>LD</td>
<td>DD</td>
<td>MyDB</td>
<td>CT</td>
<td>Warning</td>
</tr>
<tr>
<td>LD</td>
<td>DD</td>
<td>MyDB</td>
<td>CT</td>
<td>Warning</td>
</tr>
<tr>
<td>LD</td>
<td>DD</td>
<td>MyDB</td>
<td>CT</td>
<td>\texttt{RE/g16DME}</td>
</tr>
<tr>
<td>LD</td>
<td>DD</td>
<td>MyDB</td>
<td>CT</td>
<td>\texttt{RE/g16DME}</td>
</tr>
<tr>
<td>LD</td>
<td>DD</td>
<td>MyDB</td>
<td>CT</td>
<td>\texttt{RE/g16DME}</td>
</tr>
<tr>
<td>LD</td>
<td>DD</td>
<td>MyDB</td>
<td>CT</td>
<td>\texttt{RE/g16DME}</td>
</tr>
<tr>
<td>LD</td>
<td>DD</td>
<td>MyDB</td>
<td>CT</td>
<td>\texttt{RE/g16DME}</td>
</tr>
<tr>
<td>LD</td>
<td>DD</td>
<td>MyDB</td>
<td>CT</td>
<td>\texttt{RE/g16DME}</td>
</tr>
<tr>
<td>LD</td>
<td>DD</td>
<td>MyDB</td>
<td>CT</td>
<td>\texttt{RE/g16DME}</td>
</tr>
<tr>
<td>LD</td>
<td>DD</td>
<td>MyDB</td>
<td>CT</td>
<td>\texttt{RE/g16DME}</td>
</tr>
<tr>
<td>LD</td>
<td>DD</td>
<td>MyDB</td>
<td>CT</td>
<td>\texttt{RE/g16DME}</td>
</tr>
</tbody>
</table>

b) not equal, count of token values do not match

c) not equal, token value order does not match
d) not equal, token object values do not match
<table>
<thead>
<tr>
<th>Literal</th>
<th>Description</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_{1-3}$</td>
<td>Initial places</td>
<td></td>
</tr>
<tr>
<td>$OL_1$</td>
<td>Object “Database” listed</td>
<td>Rewrite</td>
</tr>
<tr>
<td>$OL_2$</td>
<td>Object “Table” listed</td>
<td>Rewrite</td>
</tr>
<tr>
<td>$OL_3$</td>
<td>Object “Column” listed</td>
<td>Rewrite</td>
</tr>
<tr>
<td>$TC$</td>
<td>Table created</td>
<td>Trigger</td>
</tr>
<tr>
<td>$OD$</td>
<td>Object “Database” or “Table” deleted</td>
<td>Backup</td>
</tr>
<tr>
<td>$RI$</td>
<td>Ransom message inserted</td>
<td></td>
</tr>
<tr>
<td>$N$</td>
<td>Admin Notification to be sent</td>
<td>Notification</td>
</tr>
</tbody>
</table>

Table 6.1: Description of the classifier states realized as places inside the CPN in Figure 6.5. When a token reaches a place, the specified action can be executed.

A token item consists of

- a token type and,
- if applicable, a list of object values,

and is added to a transferred token during the transition event. The illustrations in Figure 6.4 demonstrate in detail, when new information is added to the transferred token and when merging takes place.

For our purposes, we also extend CPNs with four new features, namely Token Merging, Always Transition, Token Expiration, and Place Action as followed.

**Extension: Token Merging**

The first is the ability to merge similar tokens, where the color information is identical except for the time stamp.

For performance and memory purposes, the transferred token is merged with at most one token from the target place if the token color of the two compared tokens are equal. If the target place contains a token with the same color, only the time stamp of the token’s last value entry is updated with the same value of the transferred token. If there is no match, the transferred token is simply added to the target place.

Figure 6.4 illustrates the situations in which the tokens can be merged and shows the result, accordingly.

**Extension: Always Transition**

The second extension is called Always Transition, a special type of transition that fires immediately when it becomes active. With this feature we are able to trigger multiple transitions in one query classification step. Our CPN in Figure 6.5 contains one Always Transition. We require that kind of transition in this specific situation, because without it, the CPN could only take the next step of the malicious query sequence with the next query. If there is no additional query, the notification will not be sent the Administrator, even though the attack has already been detected. It is essential that the tokens remain at place $RI$ when $A$ is fired, because a ransom
message is usually inserted only once. However, the transitions $D_T$ and $D_D$ could be triggered multiple times during a single attack. A *Always Transition* starts firing immediately when activated and continues until it is deactivated.

**Extension: Token Expiration**

The third extension allows token expiration. Since each place in the CPN can have a expiration information, this feature can be used to limit the time window of analyzed query sequences. This feature is important, since it is unlikely that a malicious query sequence spawns over a long period of time. Additionally, we take into consideration that unlimited sequences would possibly also increase the complexity leading to false positives.

The timeout threshold is a security parameter which enables different trade-offs between effectiveness and false alerts. In real-world attacks observed so far, attackers did not stretch malicious query sequences over long periods of time. Hence, even short timeouts (1-2 minutes) would work well against them. Attackers might increase the attack time window in order to avoid detection. However, the longer they stay connected, the higher the burden is for them (since the attacks are not largely automated), and the higher the risk of being uncovered. Given the fact that they do not know the currently used threshold parameter, and have no understanding of how long they should stay connected to avoid detection, this too increases the risk of being detected.

**Extension: Place Action**

The fourth extension to our CPN is called *Place Action*. A Place action is bound to a place and is activated when this place receives a token. The action takes arguments from the new place's token and is executed at the end of the classification process by the *Resolution* component.

Our solution provides the four place action types *Rewrite*, *Trigger*, *Backup*, and *Notification* as stated in Table 6.1. Each place of the CPN could bind multiple actions to act in specific situations during ransomware attack detection. The *Trigger* action emits the creation of triggers as described in Section 6.2.1. The other actions are part of the *Resolution* component and are explained in Section 6.2.4.

**6.2.3. Policy**

The *Policy* component is responsible for attack detection and holds information about patterns of malicious query sequences (or attack signatures). In our system, it is represented by the configuration of our CPN – it describes CPN’s places, place actions, transitions, transition actions, transition conditions, and arcs.

The places and transitions are referred by a name, and the arcs are each weighted with a value of 1 token. However, all tokens are transferred during a transition. Each place can be assigned several place actions, which need to be executed once the CPN transits to the corresponding place. Transitions are used to check if a (next) step in a malicious query sequence was executed. They become active when each source place contains at least one token. Each transition is assigned one transition action that is used to represent conditions to be checked against incoming queries.
Table 6.2.: Description of Transition Actions. The type and the corresponding values influence the resulting token color within our CPN (Figure 6.5). A token color is calculated at runtime.

<table>
<thead>
<tr>
<th>Type</th>
<th>Object Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>List Database</td>
<td>[Database Name]</td>
</tr>
<tr>
<td>List Table</td>
<td>[Database Name, Table Name]</td>
</tr>
<tr>
<td>List Column</td>
<td>[Database Name, Table Name, Column Name]</td>
</tr>
<tr>
<td>Create Table</td>
<td>Database Name, Table Name, [Column Name]</td>
</tr>
<tr>
<td>Drop Database</td>
<td>Database Name</td>
</tr>
<tr>
<td>Drop Table</td>
<td>Database Name, Table Name</td>
</tr>
<tr>
<td>Modify Table</td>
<td>Database Name, Table Name, [Column Name]</td>
</tr>
<tr>
<td>Insert Value</td>
<td>Database Name, Table Name, Value</td>
</tr>
<tr>
<td>Always</td>
<td></td>
</tr>
</tbody>
</table>

For instance, they may specify the query type (e.g., SELECT, DROP, CREATE TABLE) and actual content of the query such as a table name, or a typical ransom message. We present all the transitions and their respective actions as used by our policy for ransomware attack detection in Table 6.3.

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 attached to transition 1. It ensures that the message was inserted into a previously created or modified table.

Table 6.2 shows possible token value types and its stored object values. Object values in square brackets denote a non-ordered list. As an example, the transition List Database adds a list of database names to the token data during a transition firing. The Figures in Section B in the Appendix Chapter show examples of step-by-step transitioning.

We depict the CPN configured according to our security policy in Figure 6.5. Table 6.1 shows the state information represented by the places. It also shows which actions can be executed after putting a token on the place. These are handled by the Resolution component or Monitoring component, which are explained in the corresponding paragraphs.

The Petri net in Figure 6.5 was originally designed and simulated using PIPE [5].

Transitions are fired when an action occurs that is specified as malicious by the Policy component. Table 6.3 gives an overview of actions that are deemed as suspicious. Note, that no single action alone is sufficient to transit the CPN to the "attack detected" state (N). Normally, the sequence of actions would be required, and they have to be executed in a specific order (defined by the CPN configuration) to reach the state that corresponds to an attack detection.

Each place has an expiration. This supplement allows the CPN to remove tokens from active states, and thereby reduce the amount of false positives. As an example, we imagine that a user lists all tables of the DBMS which would trigger the LT transition and leave the states I2 and OL2 active. For a given amount of time, none of the transitions are triggered. After approximately 2 days, a table is deleted. The last action is very likely not a part of the attack, as an attacker usually wants to
execute the whole process as quickly as possible. Otherwise, the attack could be detected during the process and stopped by the administrator. In this case, the token in OL2 should be removed from the place after the given amount of time.

### CPN Policy Description

In this paragraph, we describe the defined CPN of Figure 6.5 resulting in the attack analysis of Section 4.3.2.

Our security policy consists of ten places and nine transitions. The places $I_{1-3}$ denote initial places where each contains exactly one neutral token (token without any additional information). The transitions $L_D$, $L_T$, and $L_C$ are thereby always active, due to the backwards directed arc from the connected transition. $I_1$ is connected to the transition $L_D$ that is triggered when a query is executed which sends information about existing databases to the client. Analogous to $L_D$, the transitions $L_T$ and $L_C$ fire when tables or columns are listed respectively. The places $OL_{1-3}$ denote object listed places. When a user gains knowledge about the existing databases, the attacker could either create a table within one of these ($C_T$), or drop them ($D_D$). To drop an existing table ($D_T$), the attacker first needs knowledge about its existence ($OL_2$). Whenever either $D_D$ or $D_T$ is triggered, tokens are transferred to the 'Object Deleted' place ($O_D$). Instead of creating a table, an attacker could also modify an existing table $M_T$. To execute such an action, he must have knowledge about the table structure and thereby know the columns of a table ($OL_3$). A modification of an existing table structure ($M_T$) is treated in the same way as creating a new table ($C_T$), which adds new tokens to the place $TC$. The attacker always inserts a ransom message ($I$) containing payment information in a previously created table ($TC$). After executing this step, we know that a ransom message was inserted ($RI$). Whenever both the places $RI$ and $OD$ contain at least one token, the transition $A$ is fired immediately to inform the administrator about a detected incident. The tokens linger at places $OL_{1-3}$, $TC$, and $RI$ until removal by Token Expiration.

The additional actions are executed at certain places as defined in Table 6.1. At the places $OL_{1-3}$, queries are rewritten so that the backed up tables are not visible to a potential attacker as described in Resolution Section 6.2.4. The rewriting is only active if the privileged mode is disabled (described in Section 6.2.7). The place $TC$ emits the Trigger Creator to create or recreate a trigger belonging to the affected table. Whenever either a database or a table is deleted ($OD$), a backup of the to-be-

<table>
<thead>
<tr>
<th>Transition</th>
<th>Transition Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_D$</td>
<td>List Databases (e.g. SHOW DATABASES)</td>
</tr>
<tr>
<td>$L_T$</td>
<td>List Tables</td>
</tr>
<tr>
<td>$L_C$</td>
<td>List Columns</td>
</tr>
<tr>
<td>$C_T$</td>
<td>Create Table</td>
</tr>
<tr>
<td>$D_D$</td>
<td>Drop Database</td>
</tr>
<tr>
<td>$D_T$</td>
<td>Drop Table</td>
</tr>
<tr>
<td>$M_T$</td>
<td>Modify table</td>
</tr>
<tr>
<td>$I$</td>
<td>Insert ransom message</td>
</tr>
<tr>
<td>$A$</td>
<td>Always</td>
</tr>
</tbody>
</table>

Table 6.3.: Transitions used in the security policy for the CPN in Figure 6.5

execute the whole process as quickly as possible. Otherwise, the attack could be detected during the process and stopped by the administrator. In this case, the token in $OL_2$ should be removed from the place after the given amount of time.

### CPN Policy Description

In this paragraph, we describe the defined CPN of Figure 6.5 resulting in the attack analysis of Section 4.3.2.

Our security policy consists of ten places and nine transitions. The places $I_{1-3}$ denote initial places where each contains exactly one neutral token (token without any additional information). The transitions $L_D$, $L_T$, and $L_C$ are thereby always active, due to the backwards directed arc from the connected transition. $I_1$ is connected to the transition $L_D$ that is triggered when a query is executed which sends information about existing databases to the client. Analogous to $L_D$, the transitions $L_T$ and $L_C$ fire when tables or columns are listed respectively. The places $OL_{1-3}$ denote object listed places. When a user gains knowledge about the existing databases, the attacker could either create a table within one of these ($C_T$), or drop them ($D_D$). To drop an existing table ($D_T$), the attacker first needs knowledge about its existence ($OL_2$). Whenever either $D_D$ or $D_T$ is triggered, tokens are transferred to the 'Object Deleted' place ($O_D$). Instead of creating a table, an attacker could also modify an existing table $M_T$. To execute such an action, he must have knowledge about the table structure and thereby know the columns of a table ($OL_3$). A modification of an existing table structure ($M_T$) is treated in the same way as creating a new table ($C_T$), which adds new tokens to the place $TC$. The attacker always inserts a ransom message ($I$) containing payment information in a previously created table ($TC$). After executing this step, we know that a ransom message was inserted ($RI$). Whenever both the places $RI$ and $OD$ contain at least one token, the transition $A$ is fired immediately to inform the administrator about a detected incident. The tokens linger at places $OL_{1-3}$, $TC$, and $RI$ until removal by Token Expiration.

The additional actions are executed at certain places as defined in Table 6.1. At the places $OL_{1-3}$, queries are rewritten so that the backed up tables are not visible to a potential attacker as described in Resolution Section 6.2.4. The rewriting is only active if the privileged mode is disabled (described in Section 6.2.7). The place $TC$ emits the Trigger Creator to create or recreate a trigger belonging to the affected table. Whenever either a database or a table is deleted ($OD$), a backup of the to-be-
deleted object is created as described in Section 6.2.4. A notification is sent to the database administrator as soon as a Notification Action is emitted at place N.

### 6.2.4. Resolution

Whenever an event in the Classifier component issues an action (except Trigger), this action is carried out by the Resolution module. This module defines and performs the three place action types Backup, Notification, and Rewrite for dealing with database ransomware attacks. In the following, we describe these actions in more detail.

#### Backup Action

Whenever a database or a table is dropped, the Resolution component moves the database or the table dropped by an attacker to a safe place instead of deleting it. This safe place is a database which is hidden from unprivileged users. Thereby, an attacker cannot drop the back up database again, or even identify that such a backup exists. While performing such a move, Resolution renames the protected tables to avoid name collisions. For hiding tables and databases within the safe place from users, the Resolution module makes use of the Rewrite action.

#### Notification Action

Notification actions are used by the Resolution component whenever there is a need to notify an administrator about an attack detection. This is achieved through the invocation of the Notifier component which we describe in Section 6.2.5.

#### Rewriting Action

Rewriting Actions change SQL queries in order to exclude information from the result sets of SHOW and SELECT statements. This action is executed by a Query Rewriter, a sub-component of the Resolution module. The rewriting is only enabled for queries which are executed in a session which has privileged access rights (see Controller ins Section 6.2.7). The backed up database objects (databases and tables) should not be shown to unprivileged users. To exclude this information, the query

```sql
SELECT * FROM information_schema.tables
```

is rewritten to

```sql
SELECT * FROM information_schema.tables
WHERE 'TABLE_SCHEMA' NOT LIKE 'dimaqs%'
```

and then is executed by the MySQL server.

The system rewrites the queries due to reasons analyzed in Section 5.3 by appending WHERE and AND clauses as described in Table 6.4.
6.2. Components

a) Before: Add the new token value and transfer token to $p_3$

b) After: New information was added to the token

c) Before: Add the new token values to each token and transfer tokens to $p_3$

d) After: Tokens from the input places are transferred to the output place with new information attached

e) Before: Add the new token values to each token, transfer tokens to $p_3$, and merge them with tokens at output (if possible)

f) After: Information is added to both source tokens. Each token is merged with the existing token at the output place

Figure 6.4.: Adding a value item to a token during transition. Each line shows a different situation of transition firing. The left side shows the situation before firing, the right side shows the situation after firing, accordingly
Figure 6.5.: The CPN used to classify transactions on the MySQL server. All arcs are weighted with a value of 1 token.

States: \( I_x \): initial states; \( OL_x \): objects listed, \( T_C \): table created; \( O_D \): object deleted; \( R_I \): ransom message inserted; \( N \): notification sent

Transitions: \( L_D \): list databases; \( L_T \): list tables; \( L_C \) list columns; \( C_T \): create database; \( D_T \): drop table; \( M_T \): modify table; \( I \): insert ransom message; \( A \): always
<table>
<thead>
<tr>
<th>Query Command</th>
<th>WHERE</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHOW PLUGINS</td>
<td>no rewriting possible</td>
<td></td>
</tr>
<tr>
<td>SHOW DATABASES</td>
<td>‘Database’ NOT LIKE ’dimaqs%’</td>
<td></td>
</tr>
<tr>
<td>SHOW TABLES</td>
<td>rewriting not needed</td>
<td></td>
</tr>
<tr>
<td>SHOW TRIGGERS</td>
<td>‘Trigger’ NOT LIKE ’dimaqs%’</td>
<td></td>
</tr>
<tr>
<td>SHOW COLUMNS</td>
<td>rewriting not needed</td>
<td></td>
</tr>
<tr>
<td>SHOW VARIABLES</td>
<td>‘Variable_name’ NOT LIKE ’dimaqs%’</td>
<td></td>
</tr>
<tr>
<td>SELECT FROM information_schema.columns</td>
<td>SCHEMA_NAME NOT LIKE ’dimaqs%’</td>
<td></td>
</tr>
<tr>
<td>SELECT FROM information_schema.files</td>
<td>FILE_NAME NOT LIKE ’./dimaqs%’</td>
<td></td>
</tr>
<tr>
<td>SELECT FROM information_schema.key_column_usage</td>
<td>TABLE_SCHEMA NOT LIKE ’dimaqs%’</td>
<td></td>
</tr>
<tr>
<td>SELECT FROM information_schema.partitions</td>
<td>TABLE_SCHEMA NOT LIKE ’dimaqs%’</td>
<td></td>
</tr>
<tr>
<td>SELECT FROM information_schema.schemata</td>
<td>SCHEMA_NAME NOT LIKE ’dimaqs%’</td>
<td></td>
</tr>
<tr>
<td>SELECT FROM information_schema.tables</td>
<td>TABLE_SCHEMA NOT LIKE ’dimaqs%’</td>
<td></td>
</tr>
<tr>
<td>SELECT FROM mysql.db</td>
<td>Db NOT LIKE ’dimaqs%’</td>
<td></td>
</tr>
<tr>
<td>SELECT FROM performance_schema.file_instances</td>
<td>FILE_NAME NOT LIKE ’%/dimaqs%/’</td>
<td></td>
</tr>
<tr>
<td>SELECT FROM performance_schema.objects_summary_global_by_type</td>
<td>OBJECT_SCHEMA NOT LIKE ’dimaqs%’</td>
<td></td>
</tr>
<tr>
<td>SELECT FROM performance_schema.table_handles</td>
<td>OBJECT_SCHEMA NOT LIKE ’dimaqs%’</td>
<td></td>
</tr>
<tr>
<td>SELECT FROM performance_schema.table_io_waits_summary_by_index_usage</td>
<td>OBJECT_SCHEMA NOT LIKE ’dimaqs%’</td>
<td></td>
</tr>
<tr>
<td>SELECT FROM performance_schema.table_io_waits_summary_by_table</td>
<td>OBJECT_SCHEMA NOT LIKE ’dimaqs%’</td>
<td></td>
</tr>
<tr>
<td>SELECT FROM performance_schema.table_lock_waits_summary_by_table</td>
<td>OBJECT_SCHEMA NOT LIKE ’dimaqs%’</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4.: Rewriting with WHERE/AND if not in privileged mode
6.2.5. Notifier
The Notifier component informs an administrator about security incidents by sending an email. 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., to restore a deleted table). The email address is specified by a MySQL variable as described in Section 5.4.

6.2.6. Parser
The query Parser component is responsible for the conversion of queries to objects suitable for evaluation. These objects are used to identify the query type and affected database items of the executed query. Therefore, if the query contains nested queries, the queries will be extracted into sub-queries, and each sub-query will be sent to the Classifier component separately. Since MySQL servers provide query parsing functionality, this component is already a part of the MySQL server. The query Parser is used by the Monitoring component when creating the triggers, and by the Resolution component to facilitate the rewriting actions.

6.2.7. Controller
The Controller component connects all other DIMAQS system components. It is the central component that orchestrates the processing of incoming queries by other components. For example, it invokes the Classifier component to classify the query as malicious or benign, or the Resolution component to initiate incident resolution upon attack detection.

Hence, the Controller could be seen as the main component of the DIMAQS MySQL plugin, it is responsible for

- setting up the plugin and thereby setting up the the other modules,
- observing connections,
- en-/disabling privileged mode for database sessions,
- processing the the extracted queries obtained by the Monitoring component through the Classifier,
- executing actions through the Resolution component, and
- hiding sensitive information from unprivileged MySQL users.

Plugin Setup
The plugin is set up by the Controller during the plugin loading stage. During the setup, the MySQL server is paused to handle queries. First, the system sets up the MySQL variables internally with the values from the MySQL configuration file. Then, it instantiates the Monitoring, Resolution, Notifier, and Classifier components. The Monitoring creates the triggers for database tables that are not excluded by the configuration as described in Section 6.2.1. The specified email address of the administrator is passed to the Notifier. The Policy that describes CPNs (we only have one CPN in our solution) is used to setup the Classifier component.

The other tasks contain component interactions and are thereby described in the next Section 6.3.
6.3. Component Interaction

Figure 6.1 depicts the interaction between the components. The query is first processed by the MySQL server. If a trigger is set, or the auditing function triggers an action, the server notifies Monitoring (1). If Monitoring raises an alert for a potentially malicious query type, the Monitoring notifies Controller (2). Controller then forwards the suspicious query to Classifier (3) for evaluation. Classifier is configured using security policy from Policy (4). Classifier then returns the classification to Controller (5). Based on the classification results, there are two possibilities: In the first case, the query is considered benign. Controller terminates its actions and the server executes the query as-is (10). In the second case, the query is considered malicious. Controller calls Resolution (6), which in turn backups dropped tables and rewrites the malicious query using Parser (7). If a Notification Action is emitted by the Classifier, Resolution then invokes Notifier to inform the administrator about an incident (8). Controller then receives the rewritten "disarmed" query from Resolution (9). This query is then executed on the MySQL server (10). Controller informs Monitoring when new tables need to be observed (11).

Connection Observation

The Controller has a list of the established connections to the server. This information is provided by the Monitoring component. Each entry in the list contains a unique Connection ID, and an Identifier which shows whether the privileged mode is enabled or disabled for the corresponding connection. The connection is removed from the list when the user disconnects from the MySQL Server. The following Figure 6.6 shows the handling of the connections as a sequence diagram.

![Sequence Diagram](image)

Figure 6.6.: Connection Observation Sequences

Privileged Mode

The Controller allows database connections to be marked as privileged for maintaining the DIMAQs MySQL plugin. Only in privileged mode can the connected user be able to list, delete, and recover backed up database tables as well as modify the settings of the plugin within a single database session. The authentication is handled by a certain SQL query which requires a predefined Secret Value, specified in the MySQL configuration. The privileged mode is therefore essential as we do not trust queries executed from any database user account to be good-natured.
In the following, we provide a detailed explanation of the privileged mode authentication as represented in Figure 6.7. For authentication, the user sends a query to the MySQL server. This query, provided by the notification function, is picked up by the Monitoring component. The Monitoring then checks the query structure to identify whether it is a authentication query. If so, the Secret Value is extracted from the query and is matched against the one from the MySQL configuration file. If it matches, a privileged flag is attached to the connection information which is stored in an internal list of the Controller, otherwise the privileged flag is removed.

![Figure 6.7.: Privileged Mode Sequences](image)

**Query Classification**

The *Query Classification* categorizes a query as benign or malicious. A benign query is defined as a query which does not change the status of the colored Petri net during the classification. The CPN changes its state, when a transition is triggered. A malicious query changes the state of the Petri net due to firing transitions and transferring tokens.

A query is parsed by the MySQL Server and then sent to the Controller via the Monitoring component if it is not an authentication query. The Controller forwards the query to the Classifier for evaluation. There are two possibilities:

In the first case, a transition in the colored Petri net was triggered meaning it is a step in the defined malicious query sequence. The places within the colored Petri net may have actions defined which are emitted by a Transition of the CPN. Therefore, if actions are emitted, a list of these actions is sent to the Controller. The Controller then executes these actions in the order of their creation. The query might be rewritten during a Backup Action and Rewrite Action, and has then to be re-parsed accordingly. The result is sent back to the Monitoring component after the classification.

In the other case where the state of the Petri net did not change, actions and other modifications are not executed. The unchanged query is sent back to the Monitoring component. This process is represented by the sequence diagram in Figure 6.8.

---

46
In terms of classification, we distinguish between Data Definition Language (DDL) and Data Manipulation Language (DML) queries, as they are handled differently by reasons of MySQL auditing limitations.

DDL queries are characterized by changing databases and its tables and comprise the following statement types:

- CREATE (e.g. database, index, stored procedure)
- CREATE TABLE
- DROP (e.g. database, table, index, view, procedure)
- ALTER (e.g. table, index, view)
- TRUNCATE (table)

We audit DDL and SELECT queries by using auditing functionality as described in the Monitoring component in Section 6.2.1. This type of queries is then forwarded to the Controller where these queries are classified as described above. Figure 6.8 gives a complete overview of the execution paths.

The following statement types are grouped into the set of DML queries and are used to select, create, modify, and delete information from existing database tables:

- SELECT ...FROM ...
- SELECT ...INTO
- INSERT
- UPDATE
- DELETE
DML operations such as *INSERT*, *UPDATE* and *DELETE* are forwarded to the *Monitoring* module via SQL triggers. The triggers are currently only emitted during *INSERT* and *UPDATE* statements according to the database ransomware problem.

Figure 6.10 explains the the execution paths for DML queries. Instead of DDL statements, a trigger forwards the inserted, updated or deleted information to the *Monitoring* component. The *Monitoring* then sends the values to the *Controller*, where the query is classified as described above.

**Action Execution**

We provide the four action types *Backup*, *Notification*, *Rewrite* (*Resolution*), and *Trigger* (*Monitoring*). As described in Figure 6.8, the actions are executed at the end of the classification. During the execution of a *Rewrite Action*, a *Query Rewriter* sub-component is used internally. Depending on the transferred information, the *Query Rewriter* decides how to rewrite the query according to *Resolution* in Section 6.2.4. *Trigger Actions* are executed after creating a table for accordingly setting up a trigger. Therefore, the table needs to be created by MySQL first, only then can the trigger be created during the *Result sent* event (see Figure 6.2 in Section 6.2.1).
Hiding Sensitive Information

An unprivileged database user should not be able to access backed up database tables, sensitive plugin information, and to gain knowledge about their existence. In Section 6.3 we explained, how a user is authenticated to the plugin as a privileged user (administrator).

When a database user is authenticated, the queries are not evaluated by the Classifier and no query rewriting takes place, because hiding backed up tables and sensitive information is no longer needed. If a database user is not authenticated, the queries are evaluated by the classifiers, and emitted actions are executed.
7. Implementation

DIMAQS is implemented as a MySQL server plugin. It is compatible with all MySQL server 5.7.x versions. To function, DIMAQS requires our own Petri net implementation library libPetri, as well as the mysqlservices library provided by 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.

7.1. LibPetri Implementation

For this project, a self developed Petri net implementation is provided, as existing implementations do not fulfill the requirements. This implementation is bundled in its own library, so that it can be reused in other projects.

Figure 7.1 shows the implemented classes of the libPetri library. The following paragraphs describe the classes and what they are used for.

NamedElement

A NamedElement provides a name attribute for Transition and Place objects. These names are used to identify transitions and places within the PetriNet, and should therefore be unique.

TimestampedElement

This class provides a single timestamp value for classes that refer to time related objects. An example of such objects is TokenData, which stores the time of its creation during a Transition.

Transition Action

The Transition Action is an abstract based class for specialized Transition Action implementations, and provides logic for Transition firing decisions. The functions isMatching and getTransitionTokenData must be implemented by derived classes. Examples for derived classes are mentioned in Table [6.1]
Transition

A Transition object represents a single Transition in the Petri net. It links to exactly one TransitionAction, contains basic functionality for action matching, and supports TransitionConditions. The isMatching function forwards the request to the TransitionHelper of the corresponding PetriNet. Since a TransitionCondition is relatively performance intensive, due to it comparing query and token values, the checkConditions function is called after the isMatching function. The list of TransitionConditions is checked in the same order as they were added to the Transition.

Place Action

PlaceAction is an abstract class of actions that are executed when a Token is transferred to a Place by a Transition. A Place Action provides the attribute executeLater to specify whether the action has to be executed directly after the classification, or after execution of the classified query by the MySQL server. For example, the creation of triggers after a table has been created. Therefore, the MySQL server executes the CREATE TABLE statement first, then executes the TriggerCreator module of the DIMAQs MySQL plugin.

Place

A Place is both, a ConnectableElement and a NamedElement. It represents a place object in the colored Petri net and contains a list of Tokens and PlaceActions. A Place can have a timeout value specified which removes tokens which have exceeded their timeout limit. The addToken function checks whether the tokens contained by the place can be merged with the to-be-added token. If a merge-able token exists, only the time stamp of that token is updated, and the to-be-added token is deleted, otherwise the token is added to the tokens list of the Place. The addToken function returns a list of PlaceActions that must be executed.
Arc

An Arc represents directed connections between two ConnectableElements such as Transitions and Places. The types of the connected elements must be different, so that it is impossible to connect transitions with transitions and places with places.

Connectable Element

Connectable Element is a basis for the Transition and Place classes with lists of incoming and outgoing Arcs. This class provides the basic structure and functionality for representing a directed graph.

TokenData

A TokenData is an abstract class representing stored information that was observed during a transition period. It is TimestampedElement which relates to the Transition that created the data. It provides a basis for cloning itself and for checking if the object itself is merge-able with another TokenData.

Token

Tokens describe a single token in the colored Petri net. It contains an ordered list of TokenData objects. This list stores the transition information in ascending order regarding the time of their creation.

AbstractTransitionHelper

The AbstractTransitionHelper is an abstract class, which provides a structure for decentralized isMatching and trigger functionality. By using the Strategy pattern, it is possible to define specific transition behaviors for various Petri nets.

PetriNet

This class is the basis for a Petri net description. It holds the Places, Transitions, and Arcs. A PetriNet must be associated with exactly one AbstractTransitionHelper.
7.2. 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 \[3\] about the following useful events:

- **MYSQL_AUDIT_CONNECTION_CLASS**
  - MYSQL_AUDIT_CONNECTION_CONNECT
  - MYSQL_AUDIT_CONNECTION_DISCONNECT
- **MYSQL_AUDIT_PARSE_CLASS**
  - MYSQL_AUDIT_PARSE_POSTPARSE

The notifications of the **MYSQL_AUDIT_CONNECTION_CLASS** class are used to track established connections and provide a possibility to disconnect them if a security alert occurs. It is also used to track connections for the privileged mode (see Section \[6.2.7\]). The notification of the **MYSQL_AUDIT_PARSE_CLASS** class provides an event of single to-be-executed queries. A query however, could also contain nested queries. The queries of the notification were already parsed when they are in the **MYSQL_AUDIT_PARSE_POSTPARSE** state. However, an event that allows to read the atomic values of inserted or updated values does not exist. The values of an **INSERT ... SELECT** statement and values which are specified dynamically are calculated during execution. An attacker could execute mimicry attacks to change the values in the table. For example, when inserting the ransom message, it is hard to determine the real values in the table. Therefore, we make use of a before insert/update trigger for every table.

To allow us to access the atomic values, we create triggers as described in Chapter \[6\].

As detailed in the MySQL trigger syntax \[23\], a trigger becomes associated with a table named *tbl_name*. This name must refer to a permanent table, which means that a trigger cannot be associated with a temporary table or a view. This limitation does not affect our solution, since it is unlikely that an attacker would delete data stored in temporary tables.
Plugin Descriptor

A MySQL plugin is declared by a descriptor containing general plugin information. Each shared library may embed at most one descriptor. The descriptor is the real entry point of the plugin, its structure defines the plugin specific information such as the plugin type, type specific descriptor, name of the plugin, the name of the author, and a plugin description. Additionally, it contains license information, the plugin version, a structure of status and system variables, and function addresses for the `init` and `deinit` functions. The following code gives a detailed overview of the implemented descriptor:

```c
#define PLUGIN_NAME "dimaqs"
#define PLUGIN_AUTHOR "Michael Jobst"

/* Plugin descriptor */
mysql_declare_plugin(audit_log) {
  MYSQL_AUDIT_PLUGIN,         /* plugin type */
  &dimaqs_query_descriptor,   /* type specific descriptor */
  PLUGIN_NAME,                /* plugin name */
  PLUGIN_AUTHOR,              /* author */
  "A security plugin to identify"
  "malicious query sequences.", /* description */
  PLUGIN_LICENSE_GPL,         /* license */
  dimaq_plugin_init,          /* plugin initializer */
  dimaq_plugin_deinit,        /* plugin deinitializer */
  0x0001,                     /* version */
  dimaq_plugin_status_vars,   /* status variables */
  dimaq_plugin_sys_vars,      /* system variables */
  NULL,                       /* reserved */
  0,                          /* flags */
} mysql_declare_plugin_end;
```

MySQL supports several plugin types such as `Auditing`, `Authentication`, and `Replication`, etc. The DIMAQS MySQL plugin is an `Auditing` plugin that enables audit functionality. Based on the plugin type, an audit plugin can be notified on specific types of events. These events and the notify function are declared in the `dimaqs_query_descriptor` structure. The notify function has the name `audit_notify`, which forwards the query and connection information to the Monitoring. The `release_thd` function is called as soon as a thread handle is freed. This notification is essential to free thread specific objects. The plugin uses the event `MYSQL_AUDIT_CONNECTION_ALL` to monitor established connections and the event `MYSQL_AUDIT_PARSE_POSTPARSE` to analyze queries after parsing by the MySQL server, before execution. At this stage, queries can also be rewritten.
static struct st_mysql_audit dimaqs_query_descriptor=
{
    MYSQL_AUDIT_INTERFACE_VERSION, /* interface version */
    release_thd, /* release_thd() */
    audit_notify, /* event_notify() */
    { 0,
        (unsigned long) MYSQL_AUDIT_CONNECTION_ALL,
        (unsigned long) MYSQL_AUDIT_PARSE_POSTPARSE,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
    }
};

Variable Declaration

The plugin uses MySQL variables as defined in Section 5.4. Making use of MySQL variables is a fundamental possibility for defining values which influence the behavior of the plugin without much effort.

All variables have a name, a data type, a visibility scope, a description, and options that set whether a variable is optional or required. Each variable is available globally, so that it is displayed by the `SHOW VARIABLES` command. Table 7.1 lists the implemented MySQL variables.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Options</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>dimaqs_secret</td>
<td>CHAR_PTR</td>
<td>RQCMDARG</td>
<td>MEMALLOC</td>
</tr>
<tr>
<td>dimaqs_object_prefix</td>
<td>CHAR_PTR</td>
<td>MEMALLOC</td>
<td></td>
</tr>
<tr>
<td>dimaqs_email</td>
<td>CHAR_PTR</td>
<td>MEMALLOC</td>
<td></td>
</tr>
<tr>
<td>dimaqs_excluded_dbs</td>
<td>CHAR_PTR</td>
<td>OPCMDARG</td>
<td>MEMALLOC</td>
</tr>
<tr>
<td>dimaqs_storagespace</td>
<td>CHAR_PTR</td>
<td>RQCMDARG</td>
<td>MEMALLOC</td>
</tr>
<tr>
<td>dimaqs_mode</td>
<td>BOOL</td>
<td>OPCMDARG</td>
<td></td>
</tr>
<tr>
<td>dimaqs_verbosity</td>
<td>INT</td>
<td>OPCMDARG</td>
<td></td>
</tr>
<tr>
<td>dimaqs_maxcapacity</td>
<td>INT</td>
<td>OPCMDARG</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1.: List of implemented MySQL variables

The options set whether the configuration must specify the value (RQCMDARG), the specification is optional (OPCMDARG), or if the MySQL server must allocate space for the variable if the variable is a string (MEMALLOC).

The variables `mode`, `verbosity`, and `maxcapacity` are initialized with the default value 0 if they were not defined in the MySQL configuration file. Similarly, the variable `object_prefix` is initialized with the default value 'dimaqs' if it was not defined in the MySQL configuration file.
7.2. Plugin Integration

Init Function

The \textit{init} function is set in the Plugin Descriptor and is executed during the MySQL start-up phase if the plugin was already installed, or after the plugin has been installed. The \textit{init} function initializes the plugin. Within that function, the plugin creates an instance of the Controller component and carries out an initialization of the triggers. The function passes in a MYSQL\_PLUGIN reference, that is the reference of itself in the MySQL server scope. That reference is for example used for logging purposes. At the end of the \textit{init} function, the Trigger Creator component is executed to create the triggers for all existing tables.

```c
static int dimaqs_plugin_init(MYSQL_PLUGIN plugin_ref)
{
    plugin_info = plugin_ref;
    status_var_debug_enabled = 0;

    Controller::s_instance = new Controller();
    Controller::s_instance->init();

    std::string msg = "dimaqs_plugin_init";
    my_plugin_log_message(&plugin_info, MY_ERROR_LEVEL, "%s", msg.c_str());

    // init triggers here
    Controller::s_instance->initTriggers();

    return 0;
}
```

Deinit Function

The \textit{deinit} function cleans up the plugin by destroying the Controller component and its sub-components. The \textit{deinit} function is called when MySQL shuts down if the plugin is installed, and after uninstalling.

```c
static int dimaqs_plugin_deinit(void*)
{
    std::string msg = "dimaqs_plugin_deinit";
    my_plugin_log_message(&plugin_info, MY_ERROR_LEVEL, "%s", msg.c_str());

    plugin_info = NULL;
    delete Controller::s_instance;
    Controller::s_instance = 0;
    return 0;
}
```
7.3. Component Implementation

In this Section, we provide the implementation of DIMAQS modules in detail. We thereby follow the same order as in component description 6.2 of the Design Chapter. We begin with the Monitoring component and its Trigger Creator implementation, followed by the Classifier and the used Policy. We then describe the Resolution of detected incidents by the Notifier, and how the Controller was implemented.

In Figure 7.2, we give a visual overview of the main classes of the DIMAQS MySQL plugin. The diagram does not show the Place Actions and Transition Actions, and derivations, due to legibility reasons. The DatabasePetriNetQuery is probably the most used object in the whole classification process, as it contains the current SQL query information which is passed through all main modules. The main components are described in the following Sections. The sub-components Place Actions and Transition Actions are described in the Policy Section 7.3.3.

7.3.1. Monitoring

As explained in Section 6.1 additional triggers are required to access information that is not transparent to the DIMAQS plugin when using MySQL’s audit features. These triggers are created when loading the plugin, and existing triggers are recreated after server start-up, since the database structure might have been changed. Triggers cannot be created within so called “stored procedures” and “stored functions”, the routine concepts supported by the MySQL server. Due to this limitation, the creation must be performed from within the plugin code. The function dimaq_plugin_init performs the creation of the additional triggers and is called directly after initialization of the MySQL server, and before entering the listening state. That function creates a trigger for every non-virtual database table. Virtual
database tables are tables that contain read-only views rather than base tables, and have no database files associated with them. Hence, protection of virtual tables is not necessary.

The query is already parsed by MySQL's Query Parser when it is notified to the plugin during the MYSQL_AUDIT_PARSE_POSTPARSE event. The following Table 7.2 exposes Query Types and its accessible Query Values that are used in the DIMAQs Plugin.

<table>
<thead>
<tr>
<th>Query Type</th>
<th>Query Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCOM_SELECT</td>
<td>multiple Tables, Table names, Column names</td>
</tr>
<tr>
<td>SQLCOM_CREATE_TABLE</td>
<td>DB name, Table name, Column names</td>
</tr>
<tr>
<td>SQLCOM_ALTER_TABLE</td>
<td>DB name, Table name, Column name changes</td>
</tr>
<tr>
<td>SQLCOM_DROP_TABLE</td>
<td>no select statement, DB name, Table name</td>
</tr>
<tr>
<td>SQLCOM_SHOW_DATABASES</td>
<td>DB names</td>
</tr>
<tr>
<td>SQLCOM_SHOW_TABLES</td>
<td>DB names</td>
</tr>
<tr>
<td>SQLCOM_SHOW_FIELDS</td>
<td>DB names, Table name,</td>
</tr>
<tr>
<td>SQLCOM_CREATE_DB</td>
<td>DB name</td>
</tr>
<tr>
<td>SQLCOM_DROP_DB</td>
<td>DB name</td>
</tr>
<tr>
<td>SQLCOM_RENAME_TABLE</td>
<td>DB name, Table name</td>
</tr>
</tbody>
</table>

Table 7.2.: MySQL query types and provided values

Trigger Creator

The Trigger Creator is a sub-component of the Monitoring module which is managed by the Controller component. The function createSingleTrigger creates a trigger which is added to a specific table as demonstrated in Algorithm 1. An example for such a trigger is contained in the Appendix Section A.

Algorithm 1: createSingleTrigger(database, table)

1: lock Table table in database
2: open Table table
3: for all Column c in Table table do
4:   add Column name c to Trigger definition
5: end for
6: close Table table
7: if Trigger exists for table then
8:   drop Trigger
9: end if
10: create Trigger
11: unlock Table table
The **INSERT** and **UPDATE** triggers call the user-defined function `eval_value_in` of the DIMAQS plugin. Several values are passed to that function, namely

1. scheme name,
2. table name, and
3. a list of new column values.

By choosing this structure of transmitted values, we can identify which values are inserted or updated in a table of a database.

During the start up phase we (re-)create triggers for existing tables. Only non-virtual database tables need to be observed. The Algorithm 2 shows how a trigger is created for each table of a non-virtual database. Examples of virtual databases are `information_scheme`, `mysql`, and `performance_scheme`.

**Algorithm 2 createTriggers()**

```
1: for all database in existing databases do
2:  if not isVirtual(database) then
3:    for all table in database.tables do
4:      createSingleTrigger(database, table)
5:    end for
6:  end if
7: end for
```

**Audit Notify Function**

The audit notify function is the callback function for specified event notifications. The arguments passed are the MySQL thread handle (`thd`), which contains information about the server and the current session, the event class (`event_class`), and information about the event itself.

Based on the type of the notification, the **Controller** component is either notified about the parsed query for classification (`handleEvent`), or about the connection which was established or closed (`handleConnection`).

```c
int audit_notify(MYSQL_THD thd, mysql_event_class_t event_class,
                 const void *event) {
  if( event_class == MYSQL_AUDIT_PARSE_CLASS ){
    const struct mysql_event_parse *event_parse=
      static_cast<const struct mysql_event_parse *>(event);
    if (event_parse->event_subclass == MYSQL_AUDIT_PARSE_POSTPARSE){
      handleEvent(thd, event_class);
    }
  } else if( event_class == MYSQL_AUDIT_CONNECTION_CLASS ){
    const struct mysql_event_connection *event_connection =
      static_cast<const struct mysql_event_connection *>(event);
    handleConnection(thd, event_connection);
  }
  return 0;
}
```
User-defined Value Function

The user-defined value function is called from within the installed insert/update triggers. The function has to be referenced separately (see Setup–Paragraph in Section 7.4) before it can be used.

A user-defined function consists of the three sub-functions `init`, the provided function `eval_value_iu` itself, and the function `deinit`. The `init` function is called before `eval_value_iu` is executed and thereby, we can ensure that the DIMAQS plugin was installed and configured properly. The `eval_value_iu` code forwards the received values to the Controller to classify them. The plugin does not require the implementation of the `deinit` function, yet.

```c
my_bool eval_value_iu_init(UDF_INIT *initid, UDF_ARGS *args, char *message) {
    if (get_dmaqs_plugin_info() != NULL)
        return 0;
    strncpy(message, "DIMAQS plugin needs to be installed.", MYSQL_ERRMSG_SIZE);
    return 1;
}

char *eval_value_iu(UDF_INIT *initid, UDF_ARGS *args, char *result, unsigned long *length, char *is_null, char *error) {
    Controller::s_instance->handleInsertUpdate(args);
    const char *message= "OK";
    *length= static_cast<unsigned long>(strlen(message));
    return const_cast<char*>(message);
}
```

7.3.2. Classifier

The Classifier is implemented using our library libPetri. libPetri is a small C++ library that implements the functionality of colored Petri nets. It includes the dynamic coloring, token timeout and token merging functions as mentioned above. Since libPetri has been specifically developed for DIMAQS, it carries no features that are unnecessary for our plugin. Thus, the implementation contains all necessary functionality within around one thousand LoC.

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 is fired. When transferring a token to a place with an associated action, that action is created with corresponding parameters. Until completion of these actions takes place, the Classifier remains locked.

The Classifier is directly managed by the Controller module. It is possible to define multiple classifiers of different types by introducing an additional abstraction. The PetriNetClassifier contains a list of Petri net instances where each of instances describes one malicious query sequence.
Classification

The first step of the classification algorithm is to determine the active transitions of a Petri net as described in Algorithm 3, which starts with the active places. Active places are places containing at least one token. Places that have outgoing arcs are connected to transitions. The algorithm calls the sub-algorithm isActive (Algorithm 4), which checks whether all places of the incoming arcs of the transition have at least one token. If this requirement is fulfilled, the algorithm returns true, otherwise false.

Algorithm 3 \textit{DatabasePetriNet.getActiveTransitions()}

1: \textbf{for all} ap in activePlaces \textbf{do}
2: \hspace{1em} \textbf{for all} arc in ap.outgoingArcs \textbf{do}
3: \hspace{2em} \textbf{if} arc.target.isActive \textbf{then}
4: \hspace{3em} activeTransitions $\leftarrow$ activeTransitions $\cup$ arc.target
5: \hspace{2em} \textbf{end if}
6: \hspace{1em} \textbf{end for}
7: \textbf{end for}
8: \textbf{return} activeTransitions

Algorithm 4 \textit{Transition.isActive()}

1: \textbf{for all} ia in incomingArcs \textbf{do}
2: \hspace{1em} \textbf{if} ia.source has no tokens \textbf{then}
3: \hspace{2em} \textbf{return} false
4: \hspace{1em} \textbf{end if}
5: \hspace{1em} \textbf{end for}
6: \textbf{return} true

After determining the active transitions, the Classifier checks whether each of the transition matches the type of the to-be classified query. This step is part of the Policy and thereby described in Section 7.3.3. The next step will trigger all matching transitions. The triggering of transitions can be defined separately for each Petri net and was outsourced into the TransitionHelper class as described in the next paragraph. In the next step, the system again determines the active transitions and searches that list for Always Transitions that are fired immediately until their deactivation. The triggering of transitions results in a list of Place Actions which is returned to the Controller and executed there. Existing Place Actions are described in Section 6.2.2.

Transition Helper

The Transition Helper was introduced to provide a specific matching and triggering functionality for each Petri net which is defined for classification. Because we only have one Petri net to detect database ransomware, only one Transition Helper exists in our solution.

Due to the fact that each Petri net can have its own Transition Helper defined, we allocate this module to be part of the Classifier and Policy simultaneously. The logic is described in Section 7.3.3.
Token Expiration

Each token, which is at a Place that requires a specified timeout value, is removed if the time is exceeded. The functionality of Algorithm 5 is called periodically. The function does not proceed for a Place if it has no timeout value defined. The function execution time is used for the comparison between the tokens’ time stamps and the places’ timeouts.

Algorithm 5 Place.tokenTimeout()

1: if timeout <= 0 then
2:   return
3: end if
4: time ← current_time()
5: tokensToRemove ← ∅
6: for all token in place.tokens do
7:   if token.timestamp + timeout <= time then
8:     tokensToRemove.add(token)
9: end if
10: end for
11: for all token in tokensToRemove do
12:   place.tokens.remove(token)
13: end for

7.3.3. Policy

The Policy contains tables that hold the information about the Transition Actions e.g. for storing regular expressions for detecting ransom messages. It also stores Places with their associated Place Actions. This information is read by Classifier on start up, and to setup the Petri net, and during classification.

Transition Actions

Figure 7.3.: Class Diagram: DIMAQS Transition Actions

Figure 7.3 shows the implementation of the Transition Actions. They all have a isMatching function that checks whether a query is matching the transition, and
a `getTokenData` function which creates the token value item. The two `Transition Actions`, `CreateTableAction` and `InsertAction`, have an additional value which allows enhanced checking for malicious table names and Bitcoin values.

**Place Actions**

![Class Diagram: DIMAQS Place Actions](image)

Figure 7.4 shows the implementation of the `Place Actions`. They all have an `execute` function for either executing the action through the `Notifier`, or the `Resolution` components. In addition it has a `create` function to clone the action instance bound to the respective `Place`, and set the action values (e.g. `databaseName`), which are required for the later operation.

**Transition Matching**

Transition Matching is a crucial technique for detecting database ransomware. Each of our nine defined `Transition Actions` match with a specific `Query Type` (e.g. `SELECT`, `DROP`), and is also checked against specific values. For instance, we say the `Transition List Database (LD)` is matching when the user executes

- a `SHOW DATABASES` command, or
- a `SHOW OPEN TABLES` command, or
- a `SELECT` command which selects information which exposes information about existing database schemes.

Tables or Views that expose such information are contained in Table 5.1.

The following Table 7.3 describes, when a Transition matches a specific `SQL` query. The entries in column `Additional Check` allow for enhanced checking against query values such as database names, table names, and Bitcoin values.
### Transition Firing

In this paragraph, we describe the algorithm of *Transition Firing* which follows the adding of token value items to tokens in terms of the behavior depicted in Section 6.2.2. Therefore, we explain the process with the the algorithms 6 and 7.

Starting with algorithm 6, we define local variables in lines 1 and 2. In lines 3-4, we ensure that the transition has at least one outgoing arc to a place. Lines 6-11 determine the source places and target places of the current transition. Next, in lines 12-31, all target places $tp$ are run through. For each $tp$, we consider each source place $sp$, and checker whether each token can be copied. If the transition has a backwards arc defined to the source place $sp$, we check whether each token matches the TransitionCondition (line 17). If this condition is true, the token is copied, added to the token in the target place $tp$, and the emitted actions are stored (lines 18-20). For source places $sp$ without a backwards arc, the tokens are not copied, instead they are moved to the target place $tp$ (lines 24-28). The function returns all emitted actions so that they can be executed by the Controller (line 32).

In lines 19 and 26 we use algorithm 7 to add a token to a place. In the following, we describe Algorithm 7. While adding a token, the Place checks whether the new token can be merged with an existing token at the Place. The merging of two tokens is possible if the two tokens have an equal color as defined in Token Color Function and Transition Firing in Section 6.2.2. In the case where merging is possible, only the time stamp of the existing token is updated with the value from the transferred token.

### Data Structure

The Policy is stored in a separate, private database table structure. It is not accessible from the DBMS so that it can not be seen by unprivileged users.
Algorithm 6 TransitionHelper.trigger(transition, query)

1: Actions ← ∅
2: TargetPlaces, SourcePlaces ← ∅
3: if transition has no outgoing arcs then
4:   return Actions
5: end if
6: for all arc in transition.outgoing do
7:   TargetPlaces ← TargetPlaces ∪ arc.place
8: end for
9: for all arc in transition.incoming do
10:   SourcePlaces ← SourcePlaces ∪ arc.place
11: end for
12: for all tp in TargetPlaces do
13:   for all sp in SourcePlaces do
14:     tdata ← transition.createData(query)
15:     if transition has Backwards Arc to sp then
16:       for all token in sp.tokens do
17:         if token.TransitionCondition matches query then
18:           newToken ← copy of token
19:           EmittedActions ← tp.addToken(newToken, tdata)
20:           Actions ← Actions ∪ EmittedActions
21:         end if
22:       end for
23:     else
24:       for all token in sp.tokens do
25:         newToken ← take token from sp.tokens
26:         EmittedActions ← tp.addToken(newToken, tdata)
27:         Actions ← Actions ∪ EmittedActions
28:       end for
29:     end if
30:   end for
31: end for
32: return Actions

Query Rewriter is responsible for hiding these tables from selections due to security reasons.

The database model in Figure 7.3 shows the data structure for each object and their relations required for the Policy to describe a Petri net. The column Additional Information in Place Action and Transition Action are used to store descriptive information for their execution. The column data type needs to be adjusted accordingly to fulfill the action requirements. For the database ransomware, we could store the information within a JSON object.

7.3.4. Resolution

For the Rewriting Actions, Resolution triggers rewriting of queries by adding a WHERE/AND condition to hide sensitive information, or rewrites it entirely. In Table 5.1 we list the commands that require rewriting. Backup Actions also perform
Algorithm 7 Place.addToken(newToken, tokenData)

1: merged = false
2: newtoken.addData(tokenData)
3: for all token in place.tokens do
4:     if token.isMergeable(newtoken) then
5:         token.mergeWith(newtoken)
6:         merged = true
7:     end if
8: end for
9: if not merged then
10:    addToken(newtoken)
11: end if

Figure 7.5.: Database Model

database and table backup operations by rewriting queries. In addition to these actions, the Resolution component also handles Administrator notifications by triggering the Notifier component.

Query Rewriter

The Query Rewriter is a sub-component within the plugin which is managed by the Resolution component.

Renaming databases cannot be trivially done due to MySQL limitations. MySQL added a command to carry out a database renaming called RENAME DATABASE <db>. However, this command was only active through a few minor releases before it was discontinued. The simplest way to rename a database is to move its tables to another database. Each moved table requires recreation of the affected triggers. Tables are renamed according to the following scheme <storagespace>.<object prefix>_<dbname>_<tablename>_<timestamp> with storagespace being a pre-configured MySQL variable of DIMAQS. The renaming of one database is done by function renameDatabase (Algorithm 8), and the renaming of one table is done.
by function renameTable (Algorithm 9). Algorithm 8 executes Algorithm 9 for every table in the database.

For Backup Actions, the DROP DATABASE <db_name> query is not rewritten. However, before executing renameTable, the function renameDatabase is executed to backup the database tables.

Algorithm 8 renameDatabase(oldSchema, newSchema)

1: for all table in oldSchema.tables do
2: renameTable(oldSchema, table, newSchema, table)
3: end for

Algorithm 9 renameTable(oldSchema, oldTableName, newSchema, newTableName)

1: if not exists(newSchema) then
2: createSchema(newSchema)
3: end if
4: for all trigger in oldSchema.oldTableName.triggers do
5: list.append(trigger)
6: end for
7: RENAME TABLE oldSchema.oldTableName TO newSchema.newTableName
8: for all trigger in list do
9: createTrigger(oldSchema, oldTableName, trigger)
10: end for

7.3.5. Notifier

The Notifier simply sends an email with all transmitted information about the suspected attack to the administrator. The administrator’s email address is configured in the configuration file of MySQL server (see Section 5.4).

7.3.6. Controller

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

The Controller is instantiated in the plugin initialization function Init. The audit_notify function forwards the queries from the Monitoring directly to the Controller. The Controller decides based on the connection and query information which component is executed.

A detailed overview of the Controller component is available in the Design Section 6.2.7.

Privileged Mode

A privileged user is authenticated to the MySQL plugin via a SQL query which contains a Secret Value string, that is only known by the DBMS administrator. This value is stored in the MySQL variable dimaq$secret, which is set on start-up.
to the corresponding value of the MySQL configuration file. This value cannot be changed while the MySQL server is running.

The following query is used to enter privileged mode for the current connection:

\[ \text{SET dimaqs\_secret = \langle secret\rangle} \]

The \langle secret\rangle value must be replaced by the correct dimaqs\_secret value for enabling privileged mode. This mode is enabled only for that session, or if the same query \[ \text{SET dimaqs\_secret = \langle secret\rangle} \]
is executed with a different Secret Value. The command

\[ \text{SET \langle variable\_name\rangle \ldots} \]

usually updates the variable value in the MySQL server, but we suppress the new value internally, and use it only for comparisons.

### 7.4. Development Details

#### Compiling the project

The plugin is platform independent, meaning it can be run on several OS and hardware, since only non-platform specific APIs were used. At this point, it must be mentioned that the plugin was tested on a UNIX-based operating system. The classes of the libraries must be exported before the plugin can be used on Microsoft Windows systems, which is not the case with Unix-based systems.

The project contains .pro project files. It is recommended to use the following tools to compile the libraries:

- QMake: To generate Makefiles
- GNU c++ compiler supporting the C++11 standard

#### Setup

After compiling, the bin folder must contain the two shared libraries libdimaqs and libPetriNet. The libdimaqs library must be copied to the MySQL plugin directory (linux default: /usr/lib/mysql/plugin/), and the libPetriNet library must be copied to a directory that is included in the PATH variable.

The plugin is installed and loaded using the following MySQL query commands:

\[
\text{INSTALL PLUGIN dimaqs SONAME 'dimaqs.so'};
\]
\[
\text{CREATE FUNCTION eval\_value\_iu RETURNS STRING SONAME 'dimaqs.so'};
\]

The INSTALL PLUGIN command installs the plugin. The CREATE FUNCTION command creates a link to that function so that it can be used by the triggers.

We recommend to set the FORCE_PLUS_PERMANENT for the dimaqs plugin to tell the server to load the plugin at start-up, and prevent it from being removed while the server is running.

The following variables should be added to the MySQL configuration file:
dimaqs=FORCE_PLUS_PERMANENT
dimaqs_email=<email address>
dimaqs_object_prefix=dimaqs
dimaqs_secret=<Secret Value for administrator authentication>
dimaqs_verbosity=0
dimaqs_excluded_dbs=<comma separated database names>

The plugin is uninstalled and deactivated using the following MySQL query command:

DROP FUNCTION eval_value_iu;
UNINSTALL PLUGIN dimaqs;

7.5. Implementation Issues

In this Section, we describe the three largest problems appeared during implemen-
tation, and how we solved them. These problems occurred during Renaming Tables,
Classifier Concurrency developments, and Token Expiration.

Renaming Tables

The tables that must be backed up are stored in a separate database. The name of
this database is equal to the specified dimaqs_storagespace variable (see Section 5.4).
Tables which contain triggers, cannot simply be moved from one database scheme
to another due to naming issues. When a Backup Action is executed, the action in-
formation is forwarded to the Query Rewriter, a sub-component of Resolution, that
is responsible for carrying out the table renaming. Therefore, the triggers of a table
have first to be dropped. The definition of the trigger is stored in the memory. After
dropping the triggers, the table is renamed using the naming scheme &lt;storages-
pace\&gt;.&lt;object_prefix&gt;&lt;dbname&gt;&lt;tablename&gt;&lt;timestamp&gt;. The time stamp
ensures unique table names as two tables with the same name cannot exist within
one database. After renaming, the triggers are re-created.

Classifier Concurrency

SQL queries are run through each specified Classifier in the same order. Tokens are
the critical elements in the Petri net Classifier, because they are the only objects that
have to be serialized during transitions. Therefore, each Classifier can analyze only
one query at a time to ensure data consistency. This is realized by a Mutex. Each
Petri net classifier locks a Mutex at the beginning of the handleEvent function and
unlocks it at the end.

DatabasePetriNet::handleEvent()
{
    mutex_\rightarrow lock();
    do classification ...
    mutex_\rightarrow unlock();
}
7.6. Debugging

We added a debugging functionality to print information about the current plugin state into the MySQL log file. In Section 5.4 we introduced the variable `dimaqs_verbosity` which controls the debugging verbosity level.

On level 4, the plugin prints a scheme like the following example at the time Transition Action matches a query:

```
Plugin dimaqs reported: '  
-----------------------------  
Query: SHOW FULL TABLES FROM 'test'  
Query Type: ShowTables  
Active Transitions: LD,LT,LC,AV,SV,CT,DD  
Triggered Transition (for matching conditions): LT  
Tokens transferred: 2  
ActivePlaces(Tokens): I1(1),I2(1),I3(1),I4(1),I5(1),OL1(2),OL2(1)'  
```

The code snippet above shows the query which was executed and the query type which was associated with the Transition Action. In this case, a SHOW TABLES command was executed which triggered the transition $LT$, and transferred 2 tokens. The snippet also displays information about which transitions were active when the query was classified, and how many tokens each places has after the classification. With this information, it should be simple to debug the Classifier and the corresponding policy and keep track of the current system state.
8. Evaluation

In this Chapter, we evaluate the effectiveness and performance overhead of DIMAQs. We first present our Testbed and Data Sets. Afterwards, we assess the Effectiveness in Section 8.3, Performance in Section 8.4, and Limitations in Section 8.5.

8.1. Testbed

To execute performance and security testing, we propose a simple setup. For the database server, we use an HPE ProLiant DL360 Gen9 server [25]. The server is equipped with a single 8-core Haswell generation Xeon E5-2640 CPU with a base clock of 2.60 GHz, a turbo clock of 3.40 GHz, and packaged with a total of 20 MB of cache [30]. Simultaneous multi-threading is enabled allowing the execution of 16 threads in parallel. 32 GB of DDR4 RAM at 2133 MHz is installed with dual channel capability. A 500 GB 3.5 inch hard drive provides storage I/O turning at 7,200 rpm.

Ubuntu 16.04.4 LTS is installed on the server running Linux kernel 4.4.0-121. For evaluation purposes, we installed the DBMS 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.
8.2. Data Sets

We employ three data sets during our evaluation. The first set includes malicious query sequences, which we generated ourselves using information about real-world attacks collected at [72]. Our resulting query set contains query sequence permutations that should be classified as malicious, 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:

- listing all databases,
- creating a table,
- inserting a ransom message into this table,
- and dropping a table or database.

Therefore, 53,940 queries are performed in total.

The second set (Bibspace set) is from the publication management system Bibspace [60], 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 [50] with the Semantic MediaWiki [62] 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.

The second and third query log files were modified so that the contained queries can be executed by the MySQL server. Missing databases and table structures were created, and additional information like time stamps, command ids, and query types were replaced by semicolons to separate the queries, and make the log files executable. Server start-up and shutdown information was removed as well. We did not remove any queries from the log files.

We will publish the attack set and the Bibspace set along with the paper to allow third parties to reproduce our tests, and to enable follow up works to compare with our results. The MediaWiki set contains sensitive information and cannot be published as is. Publishing an obfuscated data set is likely to be less useful, since results might differ when obtained using an obfuscated data set.

8.3. Effectiveness

In the following, we evaluate the precision of the Classifier module. Thus, we evaluate whether benign queries are classified as malicious (false positives), or malicious query sequences are classified as benign (false negatives).
8.3 Effectiveness

Table 8.1.: Petri net state after execution of query sets

<table>
<thead>
<tr>
<th>Query set</th>
<th>$I_1$</th>
<th>$I_2$</th>
<th>$I_3$</th>
<th>$OL_1$</th>
<th>$OL_2$</th>
<th>$OL_3$</th>
<th>$T_C$</th>
<th>$O_D$</th>
<th>$N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bibspace</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MediaWiki</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

8.3.1 Security Policy

The execution policy for the **Classifier** is described in Section 6.2.3 and listed in Table 6.1. Our policy is quite generic in the sense that we do not look for specific table or database names, but instead detect removal or renaming of any table or database. However, we are looking for a specific pattern of the ransom message. In particular, we search for occurrence of BTC or Bitcoin string inside the inserted message, since attackers typically request ransom in bitcoins\(^1\). In particular, we used the regular expression `(\d*\.|\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 etc.

8.3.2 False Negatives

To test for false negatives, we used the **attack set** described in Section 8.2. After processing all the queries from the data set by our CPN, we achieved 100% attack detection rate, and have not received a single false negative result. This behavior is somewhat expected, since our policy is designed to capture attacks from the malicious data set\(^2\). However, it confirms that our CPN correctly models each attack.

8.3.3 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 queries, and are known not to contain any ransom attacks. Every set is sent through **Classifier**. Afterwards, **Classifier** state shows if DIMAQS wrongfully detected attacks, and how many of these wrongful detections occurred. If tokens reach place $N$ in the **Classifier**, their number represents the number of alerts raised. For this evaluation, the token timeout is disabled to increase potential for false positives.

After running all the queries from the **Bibspace set** through **Classifier**, the places are populated as shown in Table 8.1. This is also visualized in Figure 8.1. No token has reached the state $N$, and no alert was registered.

Next, the queries of the **MediaWiki set** were run through the **Classifier**. Table 8.1 shows the state of CPN after classification of all queries inside the Bibspace query set. Figure 8.1 gives a visualization of this state.

Again, no token has reached the state $N$, and no ransom attack was detected, which is a very promising result.

Figure 8.2 shows the performance of the same task for the Mediawiki query set.

\(^1\)Our policy can be naturally extended to detect ransom messages requesting payments in other forms.

\(^2\)For instance, it didn’t include query sequences requesting ransom payment in ETH (Ether).
8.4. Performance Evaluation

To evaluate the performance of the DIMAQS plugin, we used two data sets: The *MediaWiki set* described in Section 8.2 and for the synthetic benchmark we utilize *sysbench* [43]. 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 state.

Sysbench benchmarks were run for 60 seconds per iteration, while the *Bibspace set* and *MediaWiki set* were completely classified every time. Each benchmark performed over 50 iterations. The resulting measurements (transactions per second) are given in Table 8.2. We report average values with standard deviation and confidence intervals of 5% quantile according to the Student’s t-distribution. Figure 8.3
Figure 8.3.: Performance influence of DIMAQS for sysbench and MediaWiki. Values are normalized to the respective value for the disabled plugin.

visualizes these results.

The results show that the usage of the DIMAQS plugin results in a 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 reduction of about 2% for the initialized Petri net, and 4% for a fully populated net. In this case the influence of the set population has a more significant impact.

Our implementation is a first proof-of-concept prototype and was not optimized for performance. Neither DIMAQS nor the used Petri net library has been profiled for potential bottlenecks. Also, no compiler optimization has been enabled. This implies that further performance improvements are possible.

8.5. Limitations

While our evaluation results show no false positives and no false negatives, DIMAQS potentially may have both. For instance, an attacker may adjust an attack to avoid using BTC or bitcoin in its ransom message, or the attacker may perform hidden renames of tables (e.g., by moving all content from one table to another) instead

<table>
<thead>
<tr>
<th>Test</th>
<th>Transactions per second</th>
<th>relative to baseline [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
<td>stdev</td>
</tr>
<tr>
<td><strong>sysbench</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>disabled</td>
<td>9 245</td>
<td>28</td>
</tr>
<tr>
<td>initialized</td>
<td>8 806</td>
<td>30</td>
</tr>
<tr>
<td>full</td>
<td>8 823</td>
<td>19</td>
</tr>
<tr>
<td><strong>MediaWiki</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>disabled</td>
<td>2 008</td>
<td>5</td>
</tr>
<tr>
<td>initialize</td>
<td>1 971</td>
<td>7</td>
</tr>
<tr>
<td>full</td>
<td>1 930</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 8.2.: Performance without the plugin, with the plugin enabled, and with tokens in each Petri net state.
issuing explicit \texttt{RENAME\_TABLE} statements. Alternatively, a specific data set may include Bitcoin information. For instance, some database backends for web forums may include a significant amount of Bitcoin addresses, thus producing false positives. This limitation however, is not specific to DIMAQs, but generally for any intrusion detection tool that uses attack signatures for detection.

To capture new attack forms, it will be necessary to tune the security policy accordingly, so that the new variation of the attack is captured by the attack description. False positives however, would not have a significant effect, since the backup tables can be automatically removed if they are not restored by the administrator (e.g., when the storage gets full or after a certain period of time).
9. Conclusion and Future Work

This Chapter summarizes the previously elaborated results in Section 9.1 and pro-
vides ideas for future works regarding DIMAQs in Section 9.2.

9.1. Conclusion

Ransomware attacks are an emerging threat, and their server-side variance that
appeared recently also imposes a significant threat to databases and stored data.
These attacks may have more severe consequences, as attackers do not always create
backups of dropped databases. So, even if the victim pays the demanded ransom,
the data cannot be restored. In this work, we addressed the problem and pre-
SENT DIMAQs (Dynamic Identification of Malicious Query Sequences), the first
solution against server-side ransomware. At its core, DIMAQs has a colored Petri
net (CPN)-based classifier, which classifies query sequences as benign or malicious.
We introduced a number of novel extensions for the CPN, which allows us to reduce
complexity of system representation, and achieve better performance. Our solution
is implemented for MySQL servers, and realized as a MySQL plugin which can be
easily installed on legacy MySQL servers. We evaluated our solution with regards
to precision of the attack detection, as well as its performance. We reported no false
negatives, no false positives, and performance overhead remained under 5% for our
non-optimized implementation. An analysis of the related work showed that there
has been much work done in respect to intrusion detection for databases, malware,
ransomware detection, as well as on Petri nets and state analysis in general. To
the best of our knowledge, no other solution has incorporated all of these methods,
including the support for the analysis of query sequences, and no other solution can
be easily adapted to the problem of server-side ransomware detection.
9.2. Future Work

During the design and testing process, we identified weaknesses in the function of the DIMAQS classifier. We describe these weaknesses in the following and provide information on how we are going to cope with them.

The Token Merging functionality of the CPN classifier should be enhanced in order to hinder it from sending multiple notifications, and thereby improve the performance. According to the examples of Section B in the Appendix Chapter, the notification place N receives two tokens during the transition firing of transition A. This firing currently generates two notifications which should be reduced to only one by merging the tokens.

The transitions $M_T$ (Modify/Alter Table) and $C_T$ (Create Table) currently do not detect Table modifications and creations when a Table was renamed multiple times during the classification process. We plan to track the table renaming so that the identification of the Tables in the system remains possible.

In our future work, we also plan to extend DIMAQS for the detection of other attack types, especially those that require analysis of query sequences rather than of single queries.

The prominent example of such an attack is SQL injection using MySQL’s `INTO_OUTFILE` query\(^1\) which uses a sequence of queries to

1. create a binary file using the `INTO_OUTFILE` query,
2. create functions in MySQL that execute user defined functions in that binary file, and then
3. execute that function.

Furthermore, we also plan to perform performance optimizations to further decrease performance overhead of our prototype. Moreover, we would like to investigate possibilities for automated policy generation, which could potentially be achieved given more elaborate malicious data set, and by applying machine learning techniques.

\(^1\)http://blog.rootcon.org/2012/03/sql-injection-using-mysql-loadfile-and.html
Acronyms

**CPN** Colored Petri net. 3, 30, 38

**DBMS** Database Management System. 18, 38, 65, 68, 73

**DDL** Data Definition Language. 46

**DML** Data Manipulation Language. 32, 46, 47

**FTP** File Transfer Protocol. 13

**IDS** Intrusion Detection System. 11

**LoC** Lines of Code. 51, 61

**LTS** Long Time Support. 73

**MEMALLOC** Memory Allocation. 56

**OPCMDARG** Optional Command Argument. 56

**OS** Operating System. 13, 18, 69

**RQCMDARG** Required Command Argument. 56

**SQL** Structured Query Language. 2, 40, 45, 47, 58, 64, 68, 70

**UAC** User Account Control. 18
Appendix

A. SQL

Insert/Update Trigger

DELIMITER $$
CREATE TRIGGER dimaqs_iu_test
BEFORE INSERT UPDATE ON test
FOR EACH ROW
BEGIN
    declare stat varchar(20);

    /* call the eval_value_iu function
       order: schema, table, [values] */

    set stat = eval_value_iu(
        'test',
        'test',
        new.value1,
        new.value2,
        ...);

END $$
B. CPN Policy Examples

Figure B.1.: Initial state

The initial places $I_{1-3}$ contain one empty token. The places do not contain tokens. The transitions $L_D, L_T,$ and $L_C$ are active and will be triggered on matching queries. All other transitions are disabled.

Figure B.2.: State, after LT triggered

This figure shows the states after transition $L_T$ was triggered. The initial states $I_{1-3}$ still contain tokens. The token from $I_2$ is transferred to the places $OL_1$ and $OL_2$. These tokens contain transition information which tables and databases were listed. The transitions $C_T, D_D,$ and $D_T$ become active.
Figure B.3.: State, after triggering $L_T$ and $L_C$

This figure shows the states after an additional firing of transition $L_C$. The initial states $I_{1-3}$ still contain tokens. The token from $I_3$ is transferred to the places $OL_{1-3}$ and are not merged, because they contain different information. The transition $M_T$ becomes active.

Figure B.4.: State, after triggering $L_T$, $L_C$, and $C_T$

The figure shows the states after an additional firing of transition $C_T$. The tokens from $OL_1$ are copied to the place $T_C$. The transition $C_T$ adds information about the created Table to the transferred tokens. The transition $I$ becomes active.
Figure B.5.: State, after triggering $L_T$, $L_C$, $C_T$, and $D_T$

This figure shows the states after an additional firing of transition $D_T$. The tokens from $O_L_2$ are copied to the place $O_D$. The transition $D_T$ adds information about the dropped Table to the transferred tokens. Transition $A$ does not become active because place $R_I$ does not contain a token, yet.

Figure B.6.: State, after triggering $L_T$, $L_C$, $C_T$, $D_T$, and $I$

This figure shows the states after an additional firing of transition $I$. The tokens from $T_C$ that match the Table name are transferred to the place $R_I$. The transition $I$ adds information about inserted message to the transferred tokens. Transition $A$ becomes active and fires immediately until $O_D$ does not contain tokens anymore. The tokens values of the tokens from $R_I$ are merged with the token values of the tokens from $O_D$.
# C. Auditable Notifications

<table>
<thead>
<tr>
<th>Class</th>
<th>Audit Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>MYSQL_AUDIT_GENERAL</td>
<td>GENERAL_LOG&lt;br&gt;GENERAL_ERROR&lt;br&gt;GENERAL_RESULT&lt;br&gt;GENERAL_STATUS</td>
</tr>
<tr>
<td>MYSQL_AUDIT_CONNECTION</td>
<td>CONNECTION_CONNECT&lt;br&gt;CONNECTION_DISCONNECT&lt;br&gt;CONNECTION_CHANGE_USER&lt;br&gt;CONNECTION_PRE_AUTHENTICATE</td>
</tr>
<tr>
<td>MYSQL_AUDIT_PARSE</td>
<td>PARSE_PREPARSE&lt;br&gt;PARSE_POSTPARSE</td>
</tr>
<tr>
<td>MYSQL_AUDIT_AUTHORIZATION</td>
<td>AUTHORIZATION_USER&lt;br&gt;AUTHORIZATION_DB&lt;br&gt;AUTHORIZATION_TABLE&lt;br&gt;AUTHORIZATION_COLUMN&lt;br&gt;AUTHORIZATION_PROCEDURE&lt;br&gt;AUTHORIZATION_PROXY</td>
</tr>
<tr>
<td>MYSQL_AUDIT_TABLE_ACCESS</td>
<td>TABLE_ACCESS_READ&lt;br&gt;TABLE_ACCESS_INSERT&lt;br&gt;TABLE_ACCESS_UPDATE&lt;br&gt;TABLE_ACCESS_DELETE</td>
</tr>
<tr>
<td>MYSQL_AUDIT_GLOBAL_VARIABLE</td>
<td>GLOBAL_VARIABLE_GET&lt;br&gt;GLOBAL_VARIABLE_SET</td>
</tr>
<tr>
<td>MYSQL_AUDIT_SERVER_STARTUP</td>
<td>SERVER_STARTUP_STARTUP</td>
</tr>
<tr>
<td>MYSQL_AUDIT_SERVER_SHUTDOWN</td>
<td>SERVER_SHUTDOWN_SHUTDOWN</td>
</tr>
<tr>
<td>MYSQL_AUDIT_COMMAND</td>
<td>COMMAND_START&lt;br&gt;COMMAND_END</td>
</tr>
<tr>
<td>MYSQL_AUDIT_QUERY</td>
<td>QUERY_START&lt;br&gt;QUERY_NESTED_START&lt;br&gt;QUERY_STATUS_END&lt;br&gt;QUERY_STATUS_NESTED_END</td>
</tr>
<tr>
<td>MYSQL_AUDIT_STORED_PROGRAM</td>
<td>STORED_PROGRAM_EXECUTE</td>
</tr>
</tbody>
</table>

Table C.1.: Audible events provided by the MySQL auditing interface
Bibliography


