Detecting SQL Injection Attacks using Machine Learning

Béatrice Moissinac[1,*], Elie Saad[1], Miranda Clay[1] and Maialen Berrondo[1]

[1]*Okta*

Abstract

Injection attacks such as SQL injection attacks (SQLia) are commonly used against systems. The consequences of those attacks range from financial, data, and reputational loss or worse. SQLia can be detected by analyzing the HyperText Transfer Protocol (HTTP) request data from which the SQLia is transmitted into the target resource. Various statistical and analytical tools exist today to detect SQLia, however, they are prone to false positives, which make their usage in production environment limited.

In this paper, we propose (1) a method of feature engineering to generate SQL and HTTP language mixtures, (2) these mixtures are used to significantly reduce the time and effort needed by Subject Matter Experts (SMEs) to label, and (3) evaluate supervised Machine Learning models using this feature engineering method. Furthermore, a major contribution of this paper is that our proposed solution is developed and evaluated using real-world HTTP request data sampled from authentication transactions served by a major Identity & Access Management (IAM) company. Thus, we believe that our results are a strong representation of the real-world effect of this detection method. Finally, we also show that this technique can be trivially extended to other types of injection attacks.

Keywords

SQL injection, Machine Learning, Language mixture

1. Introduction

Injection attacks such as SQL injection attacks (SQLia) are commonly used against platforms to extract, delete, or otherwise corrupt valuable resources [1]. The consequences of those attacks range from financial, data, and reputation loss or worse [2]. Technically, SQLia are carried out by ”injecting” (or inserting) SQL queries in the HyperText Transfer Protocol (HTTP) request data sent between the client and the server[1]. Once the attacker has sent the request containing the nefarious SQL query, she expects the server to read the SQL query and perchance, a vulnerable system would execute the query and either return, delete, or alter sensitive data. Furthermore, the risk of SQLia has recently increased with the introduction of Large Language Models (e.g., ChatGPT), to the general public, lowering the barrier of entry for potential new threat actors [3].

CAMLIS'23: Conference on Applied Machine Learning for Information Security, October 19–20, 2023, Arlington, VA

*Corresponding author.

beatrice.moissinac@okta.com (B. Moissinac); elie.saad@okta.com (E. Saad); miranda.clay@okta.com (M. Clay); maialen.berrondo@okta.com (M. Berrondo)

[1]This paper is situated at the intersection of applied Machine Learning research and Threat Intelligence research. Thus, throughout this paper, we aim at joining both expertise together, by including definition and explanation for concepts to improve the general understanding of the reader, no matter their background.

A diverse landscape of analytical and statistical tools exists to detect SQLia. In section 2.1, we review Threat Intelligence techniques and libraries. These techniques focus on creating rule sets, which are prone to false positives and restrict their usage in real-world settings. In section 2.2, we review Machine Learning (ML) techniques to detect SQLia. These data-based approaches are usually developed using "synthetic data", that is, data generated by a Subject Matter Expert (SME) rather than from the real-world.

On one hand, generating rule sets is time consuming, prone to false positives, and potentially not exhaustive enough. On the other hand, Machine Learning techniques have been limited to synthetic data and weak statistical modeling. In this paper, we propose to address those issues with the following contributions:

1. Propose a novel method of feature engineering to generate SQL and HTTP language mixtures inspired by topic modeling [4];
2. These mixtures are used to significantly reduce the time and effort needed by Subject Matter Experts (SME) to label;
3. Evaluate supervised Machine Learning models using this feature engineering method.

Furthermore, a major contribution of this paper is that our proposed solution is developed and evaluated using real-world HTTP request data sampled from authentication transactions served by a major IAM company. Thus, we believe that our results are representative of how the method would perform in the real-world.

Finally, the novel feature engineering approach presented in this paper can be trivially extended to the parent attack class of injection [5]. Thus, this model is more useful than current existing techniques and covers a wider range of attack classes than what is available today.

2. Related Work on Detection of SQLia

2.1. Detection using Threat Intelligence Techniques

Released in 2012, the Libinjection project [6] proposed a novel way to detect SQLia. Most SQLia detectors were based on rule-sets and regular expressions, while Libinjection developed attack vector identification based on digesting previous patterns and generating an algorithm based on them. Libinjection was published as a library to be integrated on application layer defenses. It is commonly used by Open Source Web Application Firewalls (WAF), Intrusion Detection Systems (IDS), and Open Source Security software, such as ModSecurity, an Apache module [7, 8], which in turn, is used by other tools [9]. Libinjection has been extended to support a number of languages (i.e., C, Python, PHP, JavaScript, Go, Ruby, and Java).

2.2. Detection using Machine Learning

Many industry tenants have also focused on developing and improving signature-based models using ML, for instance, Fortinet [10] or CloudFlare [11]. Some other companies (such as F5 [12] and Imperva [13]) implemented ML models to enable a wider set of signatures, and suppress or trigger alerts based on the confidence of the ML model. However, their algorithms are not publicly available for comparison. On the other hand, Academic research has published multiple ML-based approaches to SQLia detection [14], whose details are described below.

Feature Sets ML models require a feature set, that is, a set of signals (i.e., presence/absence, counts, etc) to be correlated with the desired output (i.e., is/isn't SQLia). Thus, ML-based SQLia detection heavily relies on SQL language markers for detection. For instance, in [15], the authors used the presence of any comment character, the number of semicolons, the presence of a tautology (i.e., a statement that is always true, such as $1 = 1$), the number of commands per statement, and the presence of abnormal command or special keywords. Similarly, in [16], the features included single-line and multi-line comments, SQL operators, punctuation, logical operators, keywords, etc. Virtually all prior work relied on some variation of the SQL language marker, but also *only* the SQL language markers.

Algorithms Many ML-based SQLia detection models have been developed in recent years, using Naive Bayes [16, 17, 18, 19], SVM [18, 20, 21], or an Ensemble method [15, 18, 19, 21]. However, we do note that Naive Bayes approaches may not be statistically robust. Naive Bayes assumes the independence of features, however, programming language markers are not independent from each other. For example, ‘SELECT‘ is very correlated with ‘FROM‘ in SQL.

Data Within the Threat Intelligence research on SQLia detection, models are developed from data collected from ”red teams”, a teams of security SMEs, which generates injection attacks for the purpose of testing a platform vulnerability. This type of data collection is omnipresent in ML research on SQLia as well [15, 16, 18, 19]. From an ML point of view, this type of data is called ’synthetic’ and presents a major risk of not being representative of real-world data, as well as being too small. In [15], the authors trained their models on 105 SQL statements and [16] used 178 examples. In [18], the authors collected 4,000 rows of plain text sentences from HTML forms collected “from user input” via a “web application”. Overall, data collection and labeling is the most expensive problem to solve in ML-based SQLia detection.

3. Data

HTTP URL Request The data is a set of 1 million HTTP URL requests from identity-centric traffic. It was sampled from a large Customer Identity and Access Management (CIAM) platform between 2021 and 2023, at the network edge[2]. The volume of traffic from which this is sampled is substantial enough to be representative of US customer Internet traffic, and the platform may be considered a giant Honey Pot[3].

In this paper, the proposed solution focuses only on the URL request data. We do not consider the IP or any other signals within the transaction, because we want to specifically evaluate

[2]While the dataset may have been filtered before entering our line of vision, our methodology and results still represent a real-world use-case.

[3]In Threat Intelligence research, a Honey Pot is a system mimicking real-world vulnerabilities to attract the attacker and collect useful data about the attacker and the attacker pattern.

the statistical robustness of language mixtures as signals for SQLia detection. Furthermore, methods such as this one are not meant to be a silver bullet, but could be integrated into a layered security architecture.

4. Building a Language Mixture

Intuition behind Language Mixtures The idea described below is similar in spirit to the Latent Dirichlet Allocation (LDA) [4] approach for topic modeling. The traditional LDA model estimates a mixture of topics per document (i.e., what the document is about), and a mixture of words per topic. LDA aims at answering the question: ”What topics are present in this document, and how much are those topics discussed in this document?”. Similarly, we want to calculate ”how much” HTTP and ”how much” SQL is in an URL request. In this sense, HTTP and SQL are the ”topics” of the URL request (document). However, LDA is based on word count[4] and relies on repetition of the same words within a document to estimate the mixture of topics. Thus, LDA did not work well for this problem, because obfuscated SQLia have only very few (and unique) SQL markers in the URL request.

Instead, we propose a ”language mixture”, which is not affected by word count. For each URL request, we score ”how much” SQL-like and ”how much” HTTP-like the request is. We want to automatically estimate a dictionary of language markers for each language (SQL and HTTP). Each marker is associated with a weight based on how important (or common) the marker is to this language. For instance ’SELECT’ is very representative of SQL. Using real-world data is crucial to guarantee that the markers and their weights are representative of real-world usage.

Building a Language Mixture To build a language mixture for SQL, we used 1 million SQL queries from open source SQL repositories from GitHub. For the HTTP language mixture, we used 1 million URL requests[5] from the same provider described in Section 3. We did not start with a known dictionary of SQL or HTTP operator, but rather extracted everything present in the data and sorted it into three categories of token, in order to be representative of the real-world. For each data set separately, we extracted three types of markers:

- Keywords (any character chain of length 2 or more);
- Delimiters (parenthesis, brackets, comma, etc.);
- Operators ($+$, $-$, $\star$, etc.).

The weight of each token is the percentage of ”documents” (URL requests or SQL queries) which contained that token at least once. For instance, the token ‘FROM‘ has a weight of 0.47 because it was present in 47% of the SQL queries. We kept tokens with a weight greater than 0.10.

[4]”Word count” is a featurization in ML which count the occurrence of a word in an instance.

[5]While it is possible that those URL requests contain injections and other ”impurities”, we assume that the low volume of those attacks on this type of traffic sufficiently guarantees that the HTTP tokens extracted are correct and representative.

Table 1
Calculation of SQL and HTTP mixture scores

Token	SQL Mixture	HTTP Mixture
and	0.286	0.000
select	0.385	0.000
–	0.508	0.000
=	0.484	0.732
.	0.573	0.727
/	0.208	0.999
(	0.818	0.000
,	0.747	0.000
)	0.818	0.000
?	0.000	0.738
Total	**4.827**	**3.196**

Estimating the Language Mixture Consider the following SQL injection attack found in the data set.

```
/yyoa/ext/trafaxserver/downloadAtt.jsp?attach_ids=
(1)%20and%201=2%20union%20select%201,2,3,4,5,md5(203735726),7--
```

From this string, we extracted the tokens listed in Table 1. Each mixture is the sum of the weights of the tokens present in the URL request. A weight is summed only once, even if the token appears multiple times. that is, the mixture score is *not* the sum of the weights multiplied by the number of occurrence of the token in the string. This is because it would make this method too insensitive to obfuscation of short SQL queries within a long HTTP string. We also don't normalize the score, because it is not usual for SQL queries or HTTP strings to have all their markers. Thus, the score is an absolute representation of the SQL-likeness or HTTP-likeness rather than a relative percentage of completeness.

5. How to use language mixtures to detect SQLia

Conveniently, the example presented in the previous section has an SQL mixture greater than its HTTP mixture. Unfortunately, comparing the language mixtures is generally not sufficient to make a decision as to whether an URL request contains an SQLia. For instance, we found that highly obfuscated SQLia will have a low SQL mixture and a high HTTP mixture. Nevertheless, we can use the language mixtures to (1) label more efficiently the data set (2) build an ML model using the mixture tokens and weights as features to learn to classify within the non-linear space of SQL/HTTP mixtures.

Language Mixture as Labeling Heuristic We computed the SQL/HTTP mixtures for 1 millions URL requests from identity-centric authentication transactions. The distribution of mixtures over the data set is not linear, as shown in Figure (1). Labeling such a large dataset is not realistic, but the language mixtures can be used as a heuristic to efficiently select batches of URL requests ”of interest”. From a ML point of view, we want to label examples near the

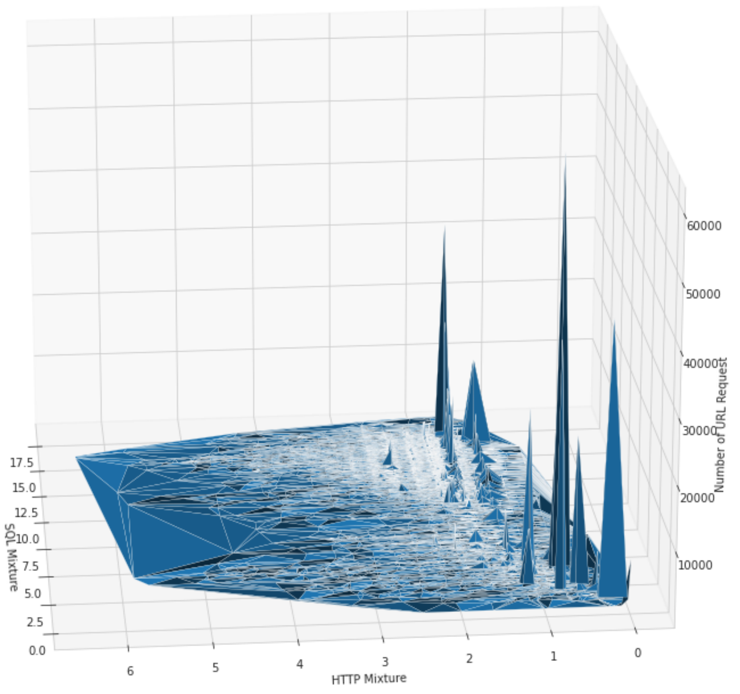

Figure 1: Number of URL request per SQL/HTTP mixture in the 1M row sample

boundaries (or thresholds) between SQL/HTTP. Those are the most ambiguous examples from the ML model's perspective. To find those, we sampled batches of URL request to be labeled, with the following language mixtures characteristics:

(A) Highest SQL mixture : This yielded obvious SQLia such as:

```
/upload/mobile/index.php?c=category&a=asynclist&price_max=1.0
%20AND%20(SELECT%201%20FROM(SELECT%20COUNT(*),CONCAT(0x7e,md5
(1),0x7e,FLOOR(RAND(0)*2))x%20FROM%20INFORMATION_SCHEMA.
CHARACTER_SETS%20GROUP%20BY%20x)a)''
```

(B) Lowest HTTP mixture : Not "HTTP"-enough, (and also not "SQL-enough") revealed the ability of this technique to discover other type of command injections, such as this XSS injection[6]:

```
/?q=%27%3E%22%3Csvg%2Fonload=confirm%28%27testing-
xss1%27%29%3E&s=%27%3E%22%3Csvg%2Fonload=confirm%28%27testing-
xss2%27%29%3E&search=%27%3E%22%3Csvg%2Fonload=confirm%28%27tes
ting-xss3%27%29%3E&id=%27%3E%22%3Csvg%2Fonload=confirm%28%27te
sting-[...]
```

[6]This example was truncated due to space limitation.

(C) Random sample across the entire set : A sample of 1,000 instances across the entire set was labeled to explore other areas of the search space, and increase chances to label diverse types of SQLia (in terms of SQL/HTTP mixtures). Then, using the SQLia found in this batch, we selected more URL request to be labeled by sampling URL requests whose mixture scores were within $+/-x$, with x varying from 0.05 to 1 from those SQLia examples.

Inter-Rater Reliability The data was labeled by threat intelligence and security engineer SMEs. We reached an inter-rater reliability rate of 94.9% , with 1,705 innocuous URL requests (labeled 'HTTP') and 114 SQLia (labeled 'SQL').

6. ML-based SQLia Detection

6.1. Experimental Setup

Features In this paper, we evaluated a feature engineering method based on creating language mixture scores for each URL request. Thus, for each URL request, the feature vector has two parameters: the mixture for SQL and HTTP respectively.

Benchmark To benchmark our proposed feature vector, we compare it with previously proposed feature vectors using word counts[7] and presence/absence flags[8] of SQL tokens (see Section 2.2). We used the 61 SQL tokens generated by the method presented in Section 4.

Algorithm In order to fairly compare the efficacy of the feature vectors described above, we needed to use the same algorithm. Furthermore, the benchmark features have some statistical particularities that restrict which algorithm to use. The features are correlated with each other due to the nature of programming languages (e.g., 'SELECT' and 'FROM' in SQL are likely to go together in a query). Thus, we used Decision Tree[9], an algorithm family which is not sensitive to the correlation between features, and can optimize a solution within a non-linear search space[10].

Training & Testing Sets We used the entire 1,819 labeled examples for the training without hold-out, because the selection of training set example was biased by our goal to find more SQLia examples to train the model. Thus, we did not test and validate on the training set. Instead, we randomly selected 638 examples from the remaining 1 million URL requests, and applied the fitted model to predict an 'http' or 'sql' label. In parallel, our security SMEs also labeled the testing set for groundtruth. This way, the model is evaluated fairly, without biases that may have been inputted from Section 5.

[7]A "word count" feature vector is a vector where each word is a feature, and the value of each feature is the number of times the word appeared in the instance (i.e., the URL request)

[8]'Presence/absence flags' is a feature vector of boolean, with a token is a feature, and the parametrization is a boolean flag set to 1 if the token is present in the instance (i.e., URL request), and 0 otherwise.

[9]We used the *sklearn* Python package, which implements the CART algorithm.

[10]In ML, the search space is the set of all possible solutions to an optimization problem.

7. Results & Discussion

In this paper, we proposed a feature engineering method to calculate how much ”HTTP-like” and ”SQL-like” are URL requests, in order to detect SQLia. These features are used as heuristic to reduce the time and effort needed to create a data set to train an ML model for detecting SQLia. We implemented a supervised ML model using Decision Tree to compare this feature set with traditional feature sets (i.e., boolean flags and word counts). The notation of those feature vectors is listed below, and used in the rest of this section.

(A) HTTP & SQL language mixture scores;

(B) Word Count of SQL tokens;

(C) Presence/Absence of SQL tokens.

Evaluation Metrics From the ML point of view, a lot of the difficulty in evaluating ML methods for SQLia detection is in the strong imbalance in the data set. The testing set contains 17 SQLia for 614 innocuous HTTP URL requests[11], thus accuracy is not a good measure, because even if we mislabeled all the SQL injections, we would still have 97% accuracy. Instead, we focused on False Positive Rate (FPR) and False Negative Rate (FNR). In the rest of this paper, we consider a ’Positive’ to be an SQL injection, and a ’Negative’ to be an innocuous HTTP URL request, and the rest of this section will be referring to the results presented in Tables 2, 3, and 4. We observed a trade-off between model (A), (B), and (C), where model (A) is less likely to falsely identify a legitimate HTTP URL request as an SQLia compared to model (B) and (C) (i.e., FPR 0.16%), while model (C) is the best at identifying SQLia (FNR 0.0%). Additionally, those results are to be nuanced. The inspection of the HTTP URL request marked as SQLia by model (A) revealed that the request did contain an SQL command. The SQL command is expected by that customer’s CIAM implementation. The SQLia sample is not large enough to extrapolate an updated FPR. Overall, model (A) had fewer misclassifications.

The Lesser of Two Evils In a real-world production environment, the minimization of FPR vs FNR will depend on the use-case. On one hand, letting through SQL command injections may be dangerous, although we may also assume that this model would be part of a layered approach to security. On the other hand, the friction caused to legitimate users by a large quantity[12] of false positives might become very undesirable. A preference for each feature vector will depend on the use-case.

Real-World Cost A great advantage of Decisions trees is the full explainability and coverage of the rule set generated (if the tree is allowed to go to its full depth). Hence, the model could be trained off-line (Computation takes less than 1 second), for nearly free, and the rules generated by the Decision Tree could be added to the current rules of any systems. Thus, we argue that the cost of this method is comparable to the cost of current rule-based methods.

[11]and 7 XSS command injection, see our discussion below. Those XSS command injections are removed from the metrics calculation in order to strictly evaluate the model on SQLia detection

[12]Extrapolating the FPR of 1.63% on our original 1M URL requests would caused 16,300 requests to fail or be delayed.

Table 2
Feature Vector (A) Confusion Matrix (Language Mixture Scores)

Prediction/True label	**HTTP**	**SQL**	XSS		
HTTP	613	3	4	**FPR**	0.16%
SQL	1	14	3	**FNR**	17.65%

Table 3
Feature Vector (B) Confusion Matrix (Word count of SQL tokens)

Prediction/True label	**HTTP**	**SQL**	XSS		
HTTP	604	1	4	**FPR**	1.63%
SQL	10	16	3	**FNR**	5.88%

Table 4
Feature Vector (C) Confusion Matrix (Presence/Absence of SQL tokens)

Prediction/True label	**HTTP**	**SQL**	XSS		
HTTP	603	0	4	**FPR**	1.79%
SQL	11	17	3	**FNR**	0.00%

Table 5
Overlapping SQL & HTTP tokens and their language weights.

Token	**SQL**	**HTTP**
=	0.480	0.730
_	0.780	0.660
-	0.350	0.540
.	0.570	0.730
/	0.210	1.000

XSS and other type of injections. While labeling the training and testing sets, our SMEs found that a URL request whose mixtures are not 'HTTP-enough' and not 'SQL-enough' is likely to be some other sort of command injection (e.g., template, code, os, xxe etc.). While those other type of command injections were *removed from the training set*, we decided *to include XSS examples in the testing set* to highlight an avenue for future work: the expansion of language mixtures to other types of injections. From a production perspective, it is desirable to develop one ML model capable of detection/classifying various type of injections. However, the more injection types are added, the more confusion is introduced. For example, in Table 5, SQL and HTTP have overlapping tokens, that is, they "share a feature". When using a presence/absence or word count type of featurization, overlapping features may create ambiguity that makes the problem more difficult for an ML model. Intuitively, a language mixture approach may help alleviate the overlapping of markers between languages, by biasing them with their weights (i.e., their importance within that language).

Future work will focus on 'unknown unknowns' and previously unidentified vulnerabilities. This will include research on the language mixtures with behavioral analysis of the URL request's response. This will deepen the model's understanding to identify potential zero-days[13] before they are known by correlating request and response, and their effect. For instance, this may help organizations identify attacks such as data exfiltration, and reduce the false positive rate on benign requests.

Acknowledgement

We thank Kim Berry for initial discussion on this approach. We thank Mathew Woodyard and George Vauter for their initial work on labels.

References

[1] Cve, 2014. URL: http://cve.mitre.org/.

[2] A. Hern, TalkTalk hit with record £400k fine over cyber-attack, The Guardian (2016). https://www.theguardian.com/business/2016/oct/05/talktalk-hit-with-record-400k-fine-over-cyber-attack.

[3] OWASP Top 10 for Large Language Model Applications | OWASP Foundation, 2023. URL: https://owasp.org/www-project-top-10-for-large-language-model-applications.

[4] D. M. Blei, A. Y. Ng, M. I. Jordan, Latent dirichlet allocation, Journal of machine Learning research 3 (2003) 993–1022.

[5] Shadowd Zecure, 2023. URL: https://capec.mitre.org/data/definitions/248.html.

[6] LibInjection, 2012. URL: https://github.com/client9/libinjection/blob/master/README.md.

[7] ModSecurity, 2002. URL: https://coreruleset.org/faq.

[8] LibModSecurity, 2002. URL: https://github.com/SpiderLabs/ModSecurity/blob/ec86b242e15f9df1d143c1b7f86a27889658b4cb/README.md.

[9] Naxsi, 2014. URL: https://github.com/nbs-system/naxsi/blob/master/README.md.

[10] Fortinet, 2023. URL: https://docs.fortinet.com/document/fortiweb/6.3.7/administration-guide/193258/machine-learning.

[11] CloudFlare, 2023. URL: https://blog.cloudflare.com/waf-ml/.

[12] F5, 2022. URL: https://community.f5.com/t5/technical-articles/f5-distributed-cloud-waf-ai-ml-model-to-suppress-false-positives/ta-p/299946.

[13] Imperva, 2004. URL: https://www.imperva.com/products/attack-analytics/.

[14] M. Al Rubaiei, T. Al Yarubi, M. Al Saadi, B. Kumar, SQLIA Detection and Prevention Techniques, in: 2020 9th International Conference System Modeling and Advancement in Research Trends (SMART), 2020, pp. 115–121. doi:10.1109/SMART50582.2020.9336795.

[15] M. Hasan, Z. Balbahaith, M. Tarique, Detection of SQL injection attacks: A machine learning approach, in: 2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA), IEEE, 2019, pp. 1–6.

[16] I. Jemal, O. Cheikhrouhou, H. Hamam, A. Mahfoudhi, Sql injection attack detection and prevention techniques using machine learning, International Journal of Applied Engineering Research 15 (2020) 569–580.

[13] A Zero-day is a vulnerability that is not yet known, and that can be exploited.

[17] A. Makiou, Y. Begriche, A. Serhrouchni, Improving Web Application Firewalls to detect advanced SQL injection attacks, in: 2014 10th International Conference on Information Assurance and Security, IEEE, 2014, pp. 35–40.

[18] S. Mishra, SQL injection detection using machine learning (2019).

[19] K. Ross, M. Moh, T.-S. Moh, J. Yao, Multi-source data analysis and evaluation of machine learning techniques for SQL injection detection, in: Proceedings of the ACMSE 2018 Conference, 2018, pp. 1–8.

[20] S. O. Uwagbole, W. J. Buchanan, L. Fan, Applied machine learning predictive analytics to SQL injection attack detection and prevention, in: 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), IEEE, 2017, pp. 1087–1090.

[21] K. Ross, SQL Injection Detection Using Machine Learning Techniques and Multiple Data Sources, Master of Science, San Jose State University, San Jose, CA, USA, 2018. URL: https://scholarworks.sjsu.edu/etd_projects/650. doi:10.31979/etd.zknb-4z36.