CODDLE: COde-injection Detection with Deep LEarning

Stanislav Abaimov, *University of Rome Tor Vergata*
Giuseppe Bianchi, *University of Rome Tor Vergata*

***Abstract*—Code Injection attacks such as SQL Injection and Cross-Site Scripting (XSS) are among the major threats for today's web applications and systems. This paper proposes CODDLE, a deep learning-based intrusion detection systems against web-based code injection attacks. CODDLE's main novelty consists in adopting a Convolutional Deep Neural Network and in improving its effectiveness via a tailored pre-processing stage which encodes SQL/XSS-related symbols into type/value pairs. Numerical experiments performed on real-world datasets for both SQL and XSS attacks show that, with an identical training and with a same neural network shape, CODDLE's type/value encoding improves the detection rate from a baseline of about 75% up to 95% accuracy, 99% precision, and a 92% recall value.**

***Index Terms*—Deep Learning, Code Injection, Intrusion Detection, Supervised Learning, SQL Injection, XSS, JavaScript.**

I. Introduction

In its yearly Top Ten List of Web Based Application Exploitation, the Open Web Application Security Project (OWASP) has often placed Code injection as the number one vulnerability [1], i.e., as the main threat for web applications which need to interact with back-end databases for their operation. As a matter of fact, SQL Injection [2] and Cross-Site Scripting (XSS) vulnerabilities were found in more than half of the web applications during a 2015-2016 scan conducted by Acunetix [3]. Several notable breaches were based on code injection techniques. An SQL Injection Attack (SQLIA) was at the basis of the 2013 theft of 160 million credit card credentials from NASDAQ, Dow Jones, and 14 other high-level organizations, resulting in a 300 million USD financial damage [4] [5]. The 2017 Equifax data breach [6] exploited a vulnerability in Apache Struts and exposed over 143 million American consumers sensitive personal information. In 2017 [7] and 2018 [8], the hosting provider Hetzner was targeted by two cyber attacks, one of which was an SQLIA with private customer information exposed. SQL injections can even be found in monitoring software and become publicly known vulnerabilities [9].

Public accessibility of web-based applications makes them a popular target and one of the main channels to compromise security of systems and networks. In addition, the wide installation base makes both web applications and host systems (virtual environments, PCs, servers, IoT devices, ICS) a valuable target for computer and ICS malware that exploits web-related vulnerabilities to spread itself across external and internal networks. This applies to a variety of contexts, from public websites to corporate networks to ICS/SCADA which store command and control information in their databases, and hence where an intrusion via a web interface may even lead to the overall system control by the attacker [10].

Defending against a code injection attack is a challenging task due to a number of stealth techniques that attackers have developed over time. Evasion techniques include, for instance, script and code encapsulation to bypass static filters (e.g., <scr<scr<script>ipt>ipt>), truncation [11], encoding and decoding (base64, URL, etc.), usage of escape and comment symbols which may evade processing, and in more generality blind SQL injection with non-trivial syntax to exploit web applications with hidden database feedback. Moreover, detection is further complicated by syntax differences in SQL implementations (MySQL, MSSQL, PostgreSQL, Oracle, etc.). As a consequence, the effectiveness of *statically pre-programmed detection and filtering countermeasures* such as signatures or policies deployed into web application firewalls [12] or intrusion detection systems further supporting web traffic analysis (e.g. suitably programmed/configured SIEMs - security information and event management systems) is at stake against constantly evolving attack vectors and techniques. At the same time, finding a static, *one-size-fits-all*, configuration, e.g. in the form of a-priori defined primitives or filters for processing HTTP/HTTPS requests, headers, and input field queries, is challenging due to the configuration/implementation differences in the underlying platforms deployed by multiple third-party vendors.

Given the above, machine learning techniques have recently emerged as a natural way to accommodate the diversity of the specific deployments and the highly dynamic nature of the threats. While their motivation - learn from each specific deployment scenario - is indeed straightforward, techniques and architectural details are not. As discussed in the related work section, many state-of-the-art machine learning solutions (deep neural networks, Bayesian networks, decision forests, etc) have been duly adapted to cyber security scenarios and integrated into Web Application Firewalls [13]. Still, performance largely depends on the amount of training, and to the best of our knowledge, detection rate is often well below the ideal 100 percent target for detection of code injection attacks.

In this paper we present an approach which relies on Deep Neural Networks (more specifically, Convolutional Neural Networks - CNN) for detection of code injection attacks. We address the usually large training needs that characterize such types of networks, with a new data pre-processing technique which allows to significantly reduce the time (i.e. amount of

data traffic) needed to configure and train the CNN. During the pre-processing phase, we detect and classify specific Code Injection *sub-patterns*. Based on this, we remove the part of the original (noisy) data and encode the remaining data comprising the query under analysis. Specifically, for each SQL code instruction, we further add a label indicating the *type* of command or symbol (e.g. Operator, Expression, Escape) rather than just delivering to the CNN the command or symbol itself. We then input the pre-processed traffic pattern into the CNN (in both training and detection phases). The proposed approach can be applied in generic web applications, IoT, ICS/SCADA, and CBRNe equipment that uses remote connection to webservices with the deployed databases or web application frameworks.

In more details, the specific contribution of the paper is the following:

- We propose and design CODDLE, a new approach for code injection detection which combines i) dataset pre-processing, which classifies and encodes sub-patterns, with ii) a convolutional neural network. CODDLE's key idea is to use a tailored pre-processing phase to remove *unnecessary* randomness in the data and add *supplementary* semantic labels, thus improving both training efficiency and detection accuracy.
- We further integrate in CODDLE local search techniques to automate the optimization of the shape and parameters of the employed Neural Network.
- We conduct experiments over two real world datasets (SQL and XSS), which show that CODDLE's original pre-processing approach is capable of significantly improving detection accuracy: for a same small-size training dataset, and with no changes in the CNN configuration, we outperform our initial indicators of about 75% accuracy up to performance figures in the order of 95% and more.

Finally, it is fair to remark that CODDLE's improved detection performance, despite the very limited training requirements, does not come for free, but is achieved at the expense of supplementary domain-specific semantic knowledge, employed in the pre-processing stage. For this purpose, we support the implementation of the CODDLE's pre-processing policies as plug-ins, which can then be upgraded and/or replaced when dealing with different attack flavors.

The paper is organised as follows. Section II discusses related work. The necessary background information on code injection detection using deep learning is provided in section III. Section IV, the core of this paper, presents and discusses the proposed approach. The evaluation methodology is then described in section V, while numerical results are discussed in section VI. Finally, conclusions are drawn in section VII.

II. Related work

Traditionally, code injection detection is done via static analysis and signature-based detection as well as design of the web application, so as to [14]:

- Analyse and validate queries before transmitting them to the back-end servers;
- Send users' input to the server with minimal permission privileges;
- Configure SQL Server software with the least necessary privileges.

Recently, we have witnessed a boost in the proposal of machine learning approaches for cyber security, mainly fostered by the recent progresses in *Deep Learning*. Such techniques are certainly not new, and have been applied for long time in the fields of intrusion/anomaly detection. In what follows we restrict our attention to works specifically related to SQL/web attack detection, with special focus on those employing machine learning approaches. For a more comprehensive survey of code vulnerability types, SQL injection attack types and their detection methods and tools, including detection of vulnerable code in applications, the reader is encouraged to refer to one of the many recent surveys on related topics, for instance the 2017 surveys [15] and [16].

A. Machine Learning for web attack detection

The early publications on SQL-specific research of Deep Learning date back to 2005. William [17] outlined AMNESIA tool, a method to detect SQLIA using Neural Networks was. Valeur et al. [18] proposed a method to identify queries that did not match multiple models of typical queries at runtime, including string model and data type-independent model. Though it did not reach high accuracy, the method proved high potential of deep learning in malicious query detection.

Cova et al. (2007) [19] presented Swaddler, an approach for detection of attacks against web applications, specifically against those developed using PHP. The proof-of-concept analyses the internal state of a web application and learns the relationships between the applications critical execution points and the applications internal state.

As opposed to supervised learning, Bockermann et al. (2009) [20] proposed an approach of using clustering ("internal self organizing maps") for modelling SQL statements to parse tree structure of SQL queries as features, e.g., for correlating SQL queries with applications and distinguishing malicious and non-malicious queries.

Cheon (2013) [21] introduced a method for SQLIA detection using Bayesian classifies as well as data set randomisation method for training purposes.

For evaluation and verification purposes Pan et al. (2018) [22] presented a way to classify the feasibility of using machine learning for web application intrusion detection, outlined limitations and challenges for the deep learning implementations, as well as described RSMT, a tool that uses DNN and autoencoders for semi-supervised and unsupervised learning for web attack detection, including SQLIA.

Gurina and Eliseev [23] focus on two common types of attacks: denial of service and code injection. A new lightweight approach to detect attacks as anomalies is proposed, by using an autoencoder for dynamic response anomaly recognition, with the detection rate against flood attacks ranging between 61.9% and 98.5%

B. Machine Learning with dataset pre-processing

The recent were published by Cai et al (2017) [24] and Edalat et al (2018) [25]. Cai et al outlined a method of conversion of Natural language into SQL queries using CNN and RNN, as well as provide techniques for pre-processing and post-processing of input data and queries (encoding and decoding). Edalat et al research SQL injection detection for Android application using concolic execution method and present a tool ConsiDroid. The paper also outlines the dataset generation approach to multiply the number of malicious samples.

Selected authors use empty spaces for pre-processing classification. A pre-processing approach, presented by Valeur [18], replaces variables with "empty space" placeholders, generating a "skeleton query" and creating a set of query "profiles". Cheon (2013) [21] presented a pre-processing algorithm ("converter"), that dissects SQL query into keywords and identifies them by the position of blank spaces: right side, both, or none.

In the survey [15] Alwan et al. (2017) enumerate existing tools and methods of detection of SQL injection attacks and claim that none of the listed tools address the issues of modern types of SQLIA, such as fast flux SQLIA.

Figure 1 presents a selected list of previous methods and tools.

Year	Method	Learning	Language
2005 [17]	AMNESIA	NDFA	SQL
2007 [19]	Swaddler	libAnomaly	PHP
2008 [26]	*Unnamed*	OC-SVM	PHP, SQL
2009 [20]	*Unnamed*	Clustering	SQL
2013 [21]	*Unnamed*	Bayesian	SQL
2017 [27]	HDLN	Hybrid	JavaScript
2017 [24]	*Unnamed*	CNN, RNN	SQL
2017 [28]	AMODS	SVM	SQL, XSS
2018 [25]	ConsiDroid	Concolic Execution	SQL
2018 [29]	DeepXSS	LSTM	XSS
2019 [23]	*Unnamed*	Autoencoder	SQL

Fig. 1. Code injection detection methods based on machine learning

C. Detection rates of existing systems

Dussel et al. (2008) [26] introduced a payload-based anomaly detection method through incorporating structural information obtained from a protocol analyser, with the detection accuracy of 49% for SQL and PHP code injection attacks. Li (2015) [30] proposed a method based on auto-encoders and deep belief network, using KDDCUP99 dataset, with the final accuracy up to 92,10%. Dong (2017) [28], claimed to achieve 94.79% accuracy. In 2018, Yan et al. [27] presented a deep learning method to detect code injection attacks on hybrid applications. The detection rate of the method was 97,55-97,60%.

The theoretical overview has shown that the existing methods are sufficient to detect conventional SQL injection attacks, and have potential to detect sophisticated and innovative ones. However, the implementations and configuration of the neural networks, as well as the training time, tend to be sub-optimal.

Our proposed approach, COODLE, attempts to confirm the existing SQLIA detection capabilities, but with an improved learning time and consequent need for large datasets and relevant availability/processing costs.

III. Background

Code injection vulnerabilities (injection flaws) occur when an application sends untrusted data to an interpreter. Injection flaws are most often found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, program arguments, etc. Injection flaws tend to be easier to discover when examining source code than via testing [31]. Traditionally, scanners and fuzzers can detect injection flaws. However, they fail in sophisticated attacks, in particular with unknown vectors [32].

A. SQL injection

SQL injection is a code injection technique, used to attack data-driven applications, in which intentionally malformed SQL statements are inserted into an entry field for execution (e.g., to reveal database structure or contents, manipulate the database, etc.) [33]. It must exploit the user input with either incorrectly filtered for string literal escape characters embedded in SQL statements, or not strongly typed and unexpectedly executed. SQL injection is a typical attack vector for web applications, but can be used to attack any type of SQL database software.

To gain access or make changes to data, the attacker adds Structured Query Language code to the input field of a web form or an HTTP/S request header. SQL injection vulnerability allows the attacker to flow commands directly to the web application's underlying database and affect both functionality and confidentiality.

For example, a legitimate administrator will be authenticated after typing: employee id=112 and password=admin. Figure 2 describes a login attempt by an attacker exploiting SQL Injection vulnerability [34]. It is carried out in three steps: 1) an attacker sends the malicious HTTP request to the web application; 2) creates the SQL statement; and 3) submits the SQL statement to the back-end database.

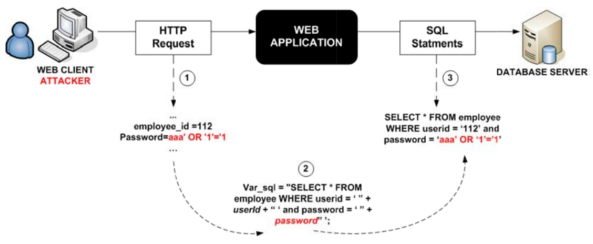

Fig. 2. Example of a SQL Injection Attack [35]

The above SQL statement is algorithmically always true due to the Boolean tautology (OR 1=1). Thus, the application allows access to the restricted resource as an administrator without verifying if the password is correct [34].

SQL Injection attacks can be classified according to different criteria [36], [37], [34], such as attacker's *intent* (extracting data, determining database schema, adding data, evading detection, performing denial of service, executing remote commands) and/or technical *methods* (Tautologies, Illegal/Logically Incorrect Queries, Union Query, Piggy-backed Queries, Stored Procedure, Alternate Encodings, Blind Injection, Timing Attacks).

B. JavaScript injection and Cross-Site Scripting

Cross-site scripting (XSS) is a type of vulnerability that enables attackers to inject client-side scripts into web pages viewed by other users and to bypass access controls such as the same-origin policy [38]. In 2017, a bug bounty company HackerOne reported XSS to be a major threat vector [39]. XSS effects vary in range from information disclosure to significant security risk, based on the sensitivity of the data being handled.

XSS attacks can be stored, reflected, and DOM based [40]. Stored attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when the server requests the stored information. Reflected attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. They target victims via another route, such as in an e-mail message, or on some other website. When a user is tricked into clicking on a malicious link, submitting a specially crafted form, or even just browsing to a malicious site, the injected code travels to the vulnerable web site, which reflects the attack back to the users browser. The browser then executes the code as it comes from a "trusted" server.

C. Deep Learning

Deep learning is based on learning data representations, as opposed to task-specific algorithms. Though with a vast potential for detecting evolving threats, it has a number of challenges in application due to semantic gap, difficulties of results evaluation, and high cost of misclassification [41].

Deep learning can be supervised, semi-supervised or unsupervised. As per [42] unsupervised learning is unstable, it detects any types of network anomalies, creating many false positives, thus requiring significant post-processing of the output events, like in DarkTrace [43]. This technique does not require any datasets with examples, as it learns from the existing flow of data. It has been thoroughly researched for application in detection of inconsistencies and anomalies in network flow behaviour. Meanwhile, semi-supervised learning implies a small amount of known correlations, before proceeding to the clustering of the unknown data, and can be implemented in network intrusion detection systems for large networks, using pre-trained models, that improve their performance the longer they are active inside a specific network.

Supervised learning is a task of devising a function that maps an input to a desired output based on provided input vs desired output pairs [44]. It is used for the detection of similarities of inputs with known patterns of symbols, words, operators, events, and combinations, solving classification problem. This type of predictive analysis yields less false events (i.e., the sum of false positives and false negatives) as compared to unsupervised machine learning, yet requires a large dataset of existing examples with known features.

Signature-based intrusion detection is typically done using static analysis, while anomaly-based intrusion detection is developed using supervised or unsupervised machine learning techniques [45]. However, anomaly detection further implies unsupervised learning without known outcomes during training cycle, while the supervised learning framework usually matches the known "malicious" output. Gurina and Eliseev [23] provide an overview of anomaly-based detection methods for web attack detection, specifically against denial of service attacks and SQLIA.

Convolutional neural network as a set of multiple perceptrons has multiple inputs and multiple outputs. A line or an encoded group of symbols can be forwarded into the neural network and output will provide a set of decisions based on the correlation algorithm. Convolutional neural network was selected for the experiment due to low computational requirements for this specific task, as it does not require sequential output for this specific implementation, and the input dataset is multi-dimensional. CNN are special feedforward networks with layers having a reduced parameter set due to the training of translation-invariant filters with a limited locally-receptive field.

There are still constraints in deep learning application, such as retraining a neural network when the dataset is changed, inconsistency with pattern prediction, finding the optimal shape of a deep neural network. However we hypothesise to be efficient for the set goal.

The notions of *batch size* and *epoch* are used in the present paper. The former defines the subset of samples used for each parameter set update. The latter defines the number of times the whole training set is processed (i.e. one training set pass = one epoch).

The batch size defines the number of samples that will be propagated through the network in a single training step. The use of batch size smaller than the number of training samples in the dataset requires less memory for training and makes the network train faster, while reducing the accuracy of the estimate.

Epoch is a one full pass over the entire training dataset, used to separate training into distinct phases for evaluation. Thus, a number of epochs shows how many times the neural network learns from the same training set.

For the attack simulation Damele and Stampar (2012) [46] presented a method of automatic SQL injection and developed a tool SQLmap, which is now a part of most security testing toolkits and operating systems.

IV. Approach

A. Overview and Rationale

As shown in Figure 3 and Figure 4, and as discussed in more details later on in each dedicated subsection, CODDLE

relies on a *supervised* approach. During the training phase, CODDLE is fed with a dataset comprising labeled entries (input queries, injected code payloads, etc). Rather than using a specific (e.g. pre-configured) Convolutional Deep Neural Network (CNN), CODDLE optimizes the CNN shape and parameters through a local search technique, i.e. by comparatively testing different model alternatives and selecting the best performing set of parameters. The selected trained Deep Neural Network is then used during the online phase to take decisions on online queries. Moreover, as shown on Figure 4, during the online operation an optional module enforcing static filtering via signature checks may be added to the processing pipeline. As it is a classical approach, we will not discuss it further in this paper - numerical results have been purposefully obtained without any static check, i.e., without accounting for such an optional filtering stage.

The main novelty of CODDLE consists in the introduction of a pre-processing module, used in either training as well as in the online phases. The role of such a pre-processing module is to accelerate the training and improve the detection rate by means of two complementary strategies:

- filter out "noisy" information from the input queries (such as column or table names), by this removing duplicates from the dataset, and
- enrich the original queries with semantic labels (operator/symbol types).

Indeed, while the rationale for the first item is quite obvious, the intuition behind such a second item is more subtle. Roughly speaking, the goal of a well-trained neural network is to "understand" the role of a specific symbol or operator. To accelerate such process, our idea is to transform (encode) the original query into a different pattern, by adding domain-specific knowledge in the form of "type" labels, which will therefore recur more often in the traces and will hence be more readily "learned" by the CNN. Concretely, we have tested in this paper an approach which replaces an otherwise atomic expression (such as an escape symbol, a separator, or an operator e.g., the string "SELECT") with a *code pair*. This is a pair of entries (C, T) where T (a numeric value) is the *Type* of the considered symbol, i.e., a label meant to identify the class to which the atomic symbol belongs, and C, again a numeric value, it an encoding of the symbol itself in that class. So far, as discussed in the next section, we have limited to three types, but in principle the approach can be obviously generalized to any alternative pre-classification. The original query is therefore transformed in a sequence of numeric codes, which is then fed to the Neural Network.

In terms of implementation, as shown in the flowchart, the pre-processing module relies on encoding rules which can be externally provided as a ruleset, thus permitting to change the encoding strategy, and hence make CODDLE a very flexible and re-programmable tool.

In Figure 3 Dataset and Encoding are text files with lines of code, that are used by the pre-processing algorithm to convert Dataset into a CNN-readable format using values from the Encoding file. After pre-processing the queries from the dataset, and before padding every transformed array to a fixed length, CODDLE compares the configured maximum initial

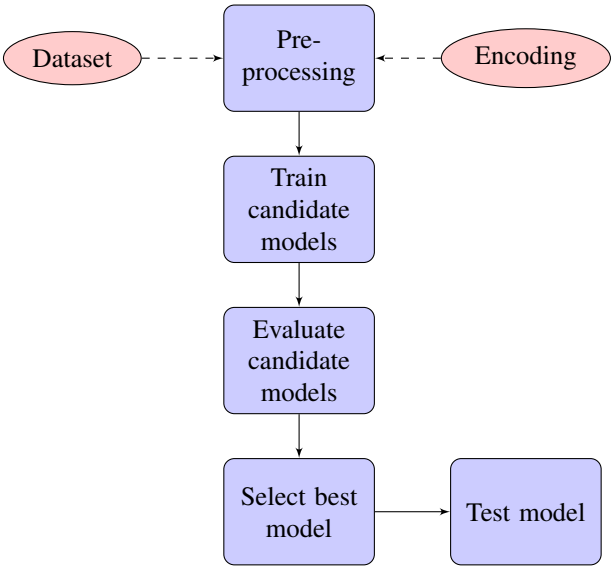

Fig. 3. Training algorithm's workflow

length to the length of the longest pre-processed query in the dataset. CNN is trained repeatably until *loss* value is satisfactory or stops improving. For a small dataset, such as ours, the loss should be below 0.05, or even 0.02. The limitation was added to CODDLE to stop training the model with one set of configurations, and move to another set of configurations, due to the fact that in many cases the loss value does not improve below 0.08. For example, the use of *charcode* preprocessing and CNN with an SQL dataset, cleared of duplicated, does not reduce the loss value below 0.20. Thus, corrections have to be added to the dataset,

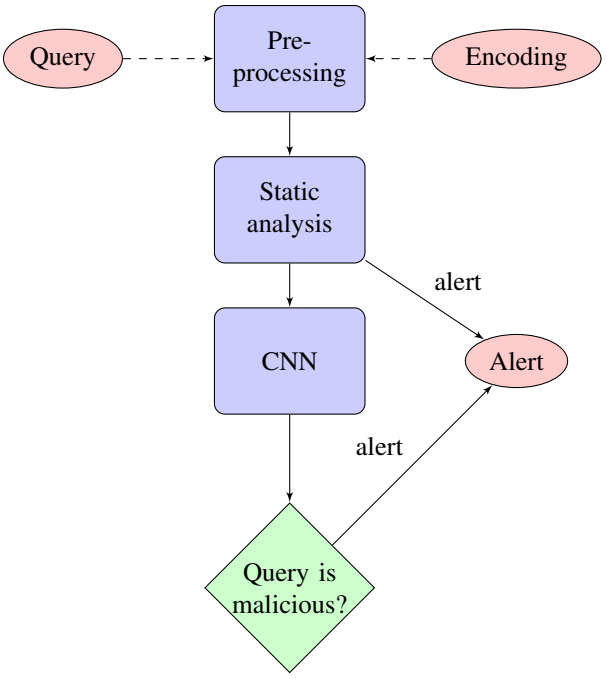

Fig. 4. Testing algorithm's workflow

In Figure 4 The input query is pre-processed with the same

encoding as the dataset in Figure 3, and compared to the "signature dataset" using the traditional static method. If the "signature" is not in the dataset, it is analysed using the trained model. If the probability of the query being malicious is more than 50%, CODDLE generates an notification (alert).

B. Dataset pre-processing

We use SQL injection attacks and XSS attacks as examples of code injection attacks, for evaluation purposes. The input features used for the analysis include operators, expressions, and escape symbols, as well as the classification identifier.

As compared to methods of pre-processing in [18] and [21], we present pre-processing based on the type of the keyword or symbol, completely ignoring blank spaces, as they can be masqueraded by attackers or otherwise irrelevant for some programming languages. The controversy of removing variables and values from statements like *1=1* and *2=0*, leaving only equation symbol and removing some potential trigger conditions, was addressed in [47].

For the experimental purposes, we tested two methods. Our initial approach, which we later abandoned in favour of the next one, was *only* based on removal of randomness from the trace. All the remaining operators, escape symbols, and expressions where then coded as a single value. For example, in the SQL case, every escape symbol was encoded as a value between 0 and 9, expressions were encoded with values between 10 and 19, and programming language operators were mapped to values above 20, based on the language complexity and a syntax file provided.

We then realized that a better encoding mechanism consists in mapping each command/symbol not as a *single* value, but as a *pair* of values, where one of such values (the *type* of command/symbol/expression), serving as a simple *semantic label*. More specifically, the encoding algorithm converts words and symbols into numeric tokens and pairs them with an identifier (0 for operators, 1 for expressions and 2 for escape symbols) and processes them as a pattern. Note that such an approach, which we quantitatively found to give superior performance than our initial one, does *not* require modifications in the CNN: the neural network remains unaware about the pairwise structure of the encoded query, and still processes it as a single unstructured sequence of bytes.

Since any queries or lines of code have variable lengths, pre-processing the dataset for the neural network training requires padding for the lines that do not have enough symbols to match a fixed width of the sequence. Figure 5 presents an example of query pre-processing before the training or classification by the neural network (removing variables and values, classifying and encoding).

For completeness and visualisation, additional examples are presented in Fig. 6. The "Merge" list is padded with zeroes, and an additional 1 or 0 to provide a supervised knowledge if the query is an injection attempt or not. The final training data set is formed with a list of those padded queries.

Language syntax file contains the exhaustive set of programming language "words" and operators, while the algorithm strips variables and values, thus preventing the system from encountering unknown symbols or issues with encoding.

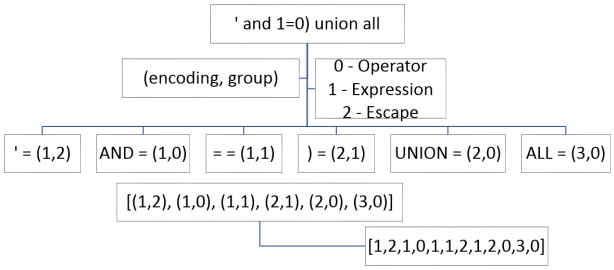

Fig. 5. Dissection of the SQL query using the presented method

```
SELECT column1, column2, column3 FROM
    tablename
Remove noise: SELECT , , FROM
Encode: (5,0) (3,1) (3,1) (10,0)
Merge: [5,0,3,1,3,1,10,0]

" or "1"="1
Remove noise: " OR " " = "
Encode: (2,2) (7,0) (2,2) (2,2) (1,1)
    (2,2)
Merge: [2,2,7,0,2,2,2,2,1,1,2,2]

<img src=1 href=1 onerror="javascript:
    alert(1)"></img>
Remove noise: < img src = href =
    onerror = " javascript : alert ( ) "
     > < / img >
Encode: (1,0) (4,1) (4,1) (4,1) (5,0)
    (4,1) (2,1) (10,0) (3,1) (2,1) (3,1)
     (3,1) (6,1)
Merge: [1,0,4,1,4,1,4,1,5,0,4,1,
    2,1,10,0,3,1,2,1,3,1,3,1,6,1]
```

Fig. 6. Pre-processing examples for SQL and JavaScript

C. Machine learning approach

CODDLE detects malicious queries using a model-based approach. Supervised learning was selected for the current research as the optimal solution to the problem, that has many existing examples (SQL injection and XSS) and requires a non-linear decision-making.

After the neural model is trained, it has a fixed number of input neurons. It remains unchanged after the full configuration and is never updated. After the successful review of event history by the evaluation function, the system can select the best performing neural model and trains next set of neural models with the set best parameters from the previous cycle, changing the next variable parameter in the list. For the neural model to be fully updated, it has to be retrained with the new dataset.

When the test input query is longer than the number of input neurons, it is analysed again, whether it can be split into multiple smaller queries using new line symbols and escape symbols as split points.

The presented method also addresses problems of detection

of the multi-level code encapsulation without any additional pre-processing.

With the given dimensionality of the data set we used Convolutional 1D layers (as the input is a single pattern of values) instead of 2D.

D. Optimisation

Optimisation is a critical component to our method. Neural models of different shape under the same conditions (same dataset, same number of training cycles, etc.), once evaluated, show different detection rates ranging from 40% to 94%. Manual selection of the optimal neural network shape and training settings is a stochastic method that requires significant time of trial and error by re-evaluating the program with numerous parameters, and does not always result in the most optimal outcomes. Optimisation algorithms allow to automate this process and select the highest performing configuration of parameters, without manual re-configuration. Local search is used in the experiment for the optimisation in CODDLE, by changing parameters of the neural network, such as the number of hidden layers, number of neurons in a hidden layer, training batch size, number of epochs. The output of CODDLE yields the optimal configuration for future retraining in future implementations.

In the neural network for our specific application the first deep layer should be wider than the input layer, while all the following deep neural layers have to be shorter than the preceding ones, gradually reducing the number of neurons to one in the output layer.

For optimisation we change the training parameters only, the mathematical model of the neural network remains the same. The features that might affect optimisation process include a number of training cycles, a number of neurons in a hidden layer (or layers) of a neural network, number of hidden layers, batch size of the patterns sent from the dataset to the input of the neural network, types of the optimisers.

In addition to the external parameters (number of neurons, training cycles, etc.), Le (2011) [48], provides insight into the optimisation methods of the machine learning mathematical models.

V. Evaluation Methodology

A. Dataset

For the analysis, before the initial neural network training, the script converts input query (or injected code) into a sequence and pads with zeroes to reach the maximum query length, forming a training dataset out of provided files [49]. Dataset of XSS payloads was taken from the GitHub repository [50]. The syntax files are formed from the publicly available lists of programming language files.

Convolutional Neural Network requires patterns of fixed length for input data. Minimal sufficient width of the dataset is equal to the longest pre-processed query and defines the number of input neurons.

The system uploads and converts all queries from the dataset of the input line from the text file into a pattern of pairs (value and category) and pads it with 0 to match the length of the longest query in the dataset. The non-malicious patterns are then padded with an extra 0, while lines from the malicious dataset are padded with 1.

In unique cases, when a malicious query attempts to masquerade the injection splitting the operator with escape characters, the pre-processing algorithm converts letter to a charcode and classifies it as a symbol. This approach also solves the challenge when multiple programming languages are chained for sophisticated code injections (e.g., non-web application code execution through code injection in web application).

B. Metrics

CODDLE uses the dataset of queries to train multiple deep neural networks. As they are trained, the part (20%/30%/50%) of the dataset, that has already been marked with expected outputs (1 or 0) and has not been used for training, is used for testing of the trained neural models instead.

As soon as several neural models with changing parameters are trained, the system selects the model with minimal loss value and maximum acc value. The best model is then evaluated with the test dataset, and the predictions (outputs) of the neural model are compared to the expected results, calculating True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). Following this step, the system calculates accuracy, precision, and recall at the end of the evaluation process.

The optimisation is automated and CODDLE will process data aiming for minimal resource usage for future training, allowing the output models to be tested and deployed on low-computational-power devices, even as an ad-hoc hardware.

The method of machine learning is justified by the potential to detect new attacks, when the signature database was not been updated yet and by the ability to operate with less knowledge about the attack.

The detection rate is high due to the small size of the dataset. And the adversarial dataset composition is limited due to the limited number of SQL query combinations. Thus, minimising the number of training cycles and batch size is detrimental to reducing the overtraining.

Key indicators for CODDLE performance evaluation are calculated as follows:

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

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

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

Those key indicators are only statistical, and can fluctuate based on the size and composition of the training dataset, as well as if the samples in the testing dataset differ from the samples in the training dataset or are the same.

VI. Results

The experiment was performed in multiple steps:

1) Pre-process the dataset detect minimal sufficient size of the patter to avoid excessive padding
2) Identify optimal number of hidden layers for the neural network
3) Identify maximum batch size with minimal number of epochs (training cycles)
4) Evaluate the model with testing dataset and statistical analysis

To assess viability on commodity hardware, experiments were conducted on a standard PC with an Inter Core i7-6500U (4 CPU cores). The training runtime of CODDLE ranges between 2 and 10 minutes for the basic training for a single neural network, and up to 1 hour if it needs to find the best shape of the neural network with given settings.

A. Neural model training

The output layer is set to 1 neuron to output a single variable, ideally 1 or 0, or percentage in between: potentially non-malicious (0.00-0.49) or potentially malicious (0.50-1.00). After reaching good accuracy and minimal loss, none of the values are close to 0.50, and they gravitate heavily to 0 or 1. Same logic is applied to a second output neuron for programming language attribution.

The number of neurons in the input layer is defined by the largest item in the dataset (in our dataset any SQL query had no more than 500 values in a single pattern, while any JavaScript line had no more than 3000 values). The triangular shape of the neural network ensures that a pattern of 500-3000 numbers is processed to have only one value in the output layer. Thus, every next layer has gradually less neurons than the previous one. For example, the input layers was set to 500 neurons, the first hidden layers - 600 neurons, and the second (if any) to 100 neurons. The output neuron remained 1 for all of the experiments.

After pre-processing and removal of duplicates, dataset is split 80%/20%, where 80% of the dataset (both malicious and non-malicious) is used to train the Deep Neural Network, and 20% of the dataset is used to evaluate the accuracy, precision, and recall of the dataset. Accuracy between 92% and 94% on average. Alternatively, the dateset is split 70/30% and 50/50% for evaluation.

For additional evaluation the dataset was shuffled, to validate detection rate over multiple experiments with similar settings.

B. Evaluation of the detection method

Unbalanced dataset with duplicates introduces a bias into the neural model, and the model detects all queries as malicious or as non-malicious. Before the initial tests, we ran CODDLE without pre-processing function (see Figure 7), simply encoding each symbol of the query into a *charcode*. This type of encoding has limitation against attacks that change sequences of symbols in the injection query. The *loss* parameter has never gone below 0.30 under the same conditions.

Dataset split	Hidden layers	Batch size	Accuracy	Precision	Recall
80%/20%	2	2000	74.0%	84.2%	82.9%
70%/30%	2	2000	71.7%	85.3%	81.7%
50%/50%	2	2000	57.8%	80.0%	54.9%
80%/20%	1	1000	73.4%	87.6%	80.7%
50%/50%	1	1000	63.9%	78.2%	75.6%

Fig. 7. Performance of the NN on dataset for SQL injection detection [49] without pre-processing

As experiments show, with loss value of 0.10-0.07 it is possible to achieve relatively high accuracy of attack prediction with a small dataset. Neural network shape in this particular example is as follows: 500 neurons in the input layers, 600 neurons in the first hidden layer, 400 neurons in the second hidden layers, and 1 neuron in the output layer.

With no pre-processing the detection results are sub-optimal for a sophisticated threat detection system. The implementation of pre-processing aids the neural network with pattern recognition, providing it additional knowledge about the query and the language, thus, improving the detection rate for selected language.

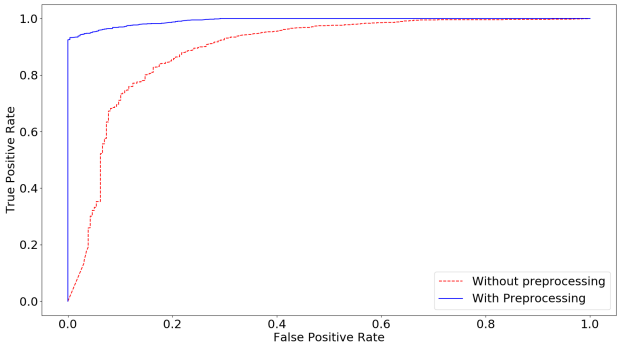

Fig. 8. ROC curve of CODDLE against SQL injection dataset [49]

Figure 8 is a receiver operating characteristic (ROC curve) of a sample output of a neural model with 500 input neurons and two hidden layers (600 and 200 neurons) (first sample in Figure 9) compared to the ROC curve of the same neural network without pre-processing. The accuracy is 95.7%, Precision 99.0, Recall 91.2, F1 Score 0.949.

Dataset split	Hidden layers	Batch size	Accuracy	Precision	Recall
80%/20%	2	2000	95.7%	99.0%	91.2%
70%/30%	2	2000	90.1%	90.1%	90.1%
50%/50%	2	2000	85.0%	93.3%	89.0%
80%/20%	1	1000	94.0%	97.8%	96.2%
50%/50%	1	1000	93.8%	96.5%	95.1%

Fig. 9. Performance of the NN on dataset for SQL injection [49] detection with pre-processing

In Figures 9 and 10 the split of the dataset reduces the number of training samples and increases the number of testing samples. As training dataset is different from the testing

Dataset split	Hidden layers	Batch size	Accuracy	Precision	Recall
80%/20%	2	2000	90.2%	99.0%	90.0%
70%/30%	2	2000	88.0%	97.9%	88.8%
50%/50%	2	2000	84.5%	94.5%	87.1%
80%/20%	1	1000	90.0%	98.6%	90.5%
50%/50%	1	1000	87.3%	95.4%	92.1%

Fig. 10. Performance of the NN on dataset for XSS [50] detection with pre-processing

dataset, the complexity increases for the neural model and the accuracy of the output reduces. However, this issue can be partially resolved by increasing the number of training cycles. For example, training of an 80/20 split model may take 20 epochs, which training of a 50/50 split model may take up to 50 epochs.

Unbalanced SQL dataset contains 13000 malicious entries and 1020 legitimate entries and yields detection accuracy of 92.2-94.0%. Those results are achieved with 20-22 epochs training, without overtraining the system. Bigger data set will require more time to train, and potentially less epochs, as it will have more samples.

C. Discussion and Extensions

Overfitting - Overfitting [51] is "the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably". A pragmatic approach to control overfitting consists in avoiding an excessive training. We experimentally found out that for the type and size of datasets used in the experimental results, a neural network training not exceeding 20 epochs (training cycles) was adequate.

Randomness removal and Boolean statements - Pre-processing removes variables and values from the analysed dataset. By removing this randomness from the queries, the method simplifies the equation. Ignoring Boolean statements (*True* and *False*) may cause the loss of "trigger" conditions. E.g., for CODDLE the statement *1=1* means the same as *2=0*, as only equation symbol is processed. The implementation of the additional category of values and variables in the method may improve the efficiency and detection rate, while significantly increasing the complexity of the code.

Batch size - Our CNN is fed with sequences of the same size. We therefore selected a sufficiently large batch size so as to accommodate the worst case query in each dataset, and used padding (after pre-processing encoding). Obviously, in an online, dynamic, deployment, even if we set a "large" batch size, an attacker may always generate a malicious injection pattern that can be longer than the neural network input capacity. In this case a pragmatic mitigation technique consists in making sure that the system will not process a query longer than a given maximum size, but will simply reject it.

Extension to other attacks beyond SQL and XSS - Even if, for the sake of experimental evaluation, we have restricted our quantitative analysis to two datasets/attack types (SQL and XSS), we remark that the proposed code injection detection method is general, and can be applied to any other programming language. For every new language, it suffices to change the encoding module, and use a specific encoder customized to the specificities of the considered language (i.e., supply a file that contains programming language syntax).

Data preparation - Our approach requires queries to be expressed in terms of native code. In the case of encoded queries (e.g., Base64), such data has to be additionally pre-processed and converted into the code to be forwarded to the system. Also, during training, dataset should be filtered and duplicates should be removed as they may affect the precision and accuracy of the system.

Additional static / signature-based analysis - although in this paper we focused only on the neural network part, our CODDLE prototype is designed with the capability to integrate further processing modules, devised to support static method and analyses on whether the query fully matches with the existing signatures in a known database of examples.

VII. Conclusion

This paper has proposed CODDLE, a system devised to detect code injection attacks using Convolutional (Deep) Neural Network.

In order to circumvent the demanding training requirements that Convolutional Neural Networks bring about, CODDLE introduces an original pre-processing technique.

Rather than processing the raw code injection data, CODDLE *transforms* the original data into an encoded pattern. On one side, such an encoding removes randomness in the original data. On the other side, and more significantly, CODDLE encodes each originally atomic symbol, command or instruction into a <code,type> pair. This permits not only to retain the information about the original symbol, but also to add a simple semantic label which helps the neural network to "understand" the role of the specific symbol or operator itself, thus significantly reducing the training needs.

Numerical results have been produced on two real-world datasets encompassing both SQL and XSS attacks. The results show that CODDLE provides excellent detection detection performance, up to 94% accuracy, 99% precision, and a 93% recall value.

Finally, the CODDLE system is being developed as a highly modular framework, which does not rely on "hardcoded" syntax of programming or scripting languages, but permits to change the type of pre-processing via a configuration module, thus allowing the method to be used for any programming language that allows code injection attacks.

We believe that the flexibility of the programming language syntax encoding, in conjunction with the possibility to add supplementary modules devised to static (e.g. signature-based) analysis makes of CODDLE a versatile tool, which may find compelling application also in different intrusion detection scenarios, such as critical infrastructure and CBRNe. Indeed, We plan to release a first version of CODDLE as public-domain software once this paper will appear.

REFERENCES

[1] "Owasp top 10 2017," accessed: 2019-01-30. [Online]. Available: https://www.owasp.org/index.php/Top_10-2017_Top_10/

[2] J. P. Singh, "Analysis of SQL Injection Detection Techniques," *Theoretical and Applied Informatics*, vol. 28, no. 1&2, pp. 37–55, 2017.

[3] Acunetix, "Acunetix web application vulnerability report 2016," Tech. Rep., 2016. [Online]. Available: https://www.acunetix.com/acunetix-web-application-vulnerability-report-2016/

[4] "Five indicted in new jersey for largest known data breach conspiracy, department of justice, 2013," accessed: 2019-04-03. [Online]. Available: https://www.justice.gov/usao-nj/pr/five-indicted-new-jersey-largest-known-data-breach-conspiracy

[5] "Us prosecutors launch largest ever hacking fraud case," accessed: 2019-04-03. [Online]. Available: https://www.bbc.co.uk/news/technology-23448639

[6] "Equifax, apache struts, and cve-2017-5638 vulnerability," accessed: 2019-04-03. [Online]. Available: https://www.synopsys.com/blogs/software-security/equifax-apache-struts-cve-2017-5638-vulnerability/

[7] "Konsoleh database compromise, hetzner," accessed: 2019-04-03. [Online]. Available: https://hetzner.co.za/insights/konsoleh-database-compromise/

[8] "Web hosting provider have been hacked for the second time in the past year by hackers, hitechwiki," accessed: 2019-04-03. [Online]. Available: https://hitechwiki.com/2018/10/web-hosting-provider-have-been-hacked-for-the-second-time-in-the-past-year-by-hackers/

[9] "Nist cve-2018-18550 details," accessed: 2019-04-03. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2018-18550

[10] "Plcsql tia installation manual v1.04," accessed: 2019-02-13. [Online]. Available: http://www.plcsql-link.com/pdf/PLCSQL_PLC_TIA_Installation_Manual_V1_04.pdf

[11] "Bala neerumalla, sql injections by truncation, microsoft," accessed: 2019-04-03. [Online]. Available: https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Neerumalla.pdf

[12] "Web application firewall, owasp," accessed: 2019-04-05. [Online]. Available: https://www.owasp.org/index.php/Web_Application_Firewall

[13] "The oracle dyn web application firewall (waf), oracle," accessed: 2019-04-05. [Online]. Available: https://dyn.com/waf/

[14] K. Elshazly, Y. Fouad, M. Saleh, and A. Sewisy, "A Survey of SQL Injection Attack Detection and Prevention," *Journal of Computer and Communications*, vol. 02, no. 08, pp. 1–9, 2014.

[15] Z. S. Alwan and M. F. Younis, "Detection and Prevention of SQL Injection Attack: A Survey," *International Journal of Computer Science and Mobile Computing*, vol. 6, no. 8, pp. 5–17, 2017. [Online]. Available: https://www.ijcsmc.com/docs/papers/August2017/V6I8201701.pdf

[16] B. Nagpal, N. Chauhan, and N. Singh, "A survey on the detection of SQL injection attacks and their countermeasures," *Journal of Information Processing Systems*, vol. 13, no. 4, pp. 689–702, 2017.

[17] W. G. J. Halfond, A. Orso, D. A. Kindy, and A. S. K. Pathan, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks," Tech. Rep., 2005. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101935

[18] F. Valeur, D. Mutz, and G. Vigna, "A Learning-Based Approach to the Detection of SQL Attacks," pp. 123–140, 2005.

[19] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An Approach for the Anomaly-Based Detection of State Violations in Web Applications," *Recent Advances in Intrusion Detection*, pp. 63–86, 2007.

[20] C. Bockermann, M. Apel, and M. Meier, "Learning SQL for database intrusion detection using context-sensitive modelling (extended abstract)," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5587 LNCS, pp. 196–205, 2009.

[21] E. H. Cheon, Z. Huang, and Y. S. Lee, "Preventing SQL Injection Attack Based on Machine Learning," *International Journal of Advancements in Computing Technology*, vol. 5, no. 9, pp. 967–974, 2013.

[22] Y. Pan, F. Sun, J. White, D. C. Schmidt, J. Staples, and L. Krause, "Detecting Web Attacks with End-to-End Deep Learning," *Acm*, pp. 1–14, 2019. [Online]. Available: https://www.dre.vanderbilt.edu/{~}schmidt/PDF/machine-learning-feasibility-study.pdf

[23] A. Gurina and V. Eliseev, "Anomaly-based method for detecting multiple classes of network attacks," *Information (Switzerland)*, vol. 10, no. 3, 2019.

[24] R. Cai, B. Xu, Z. Zhang, X. Yang, Z. Li, and Z. Liang, "An encoder-decoder framework translating natural language to database queries," *IJCAI International Joint Conference on Artificial Intelligence*, vol. 2018-July, pp. 3977–3983, 2018.

[25] E. Edalat, B. Sadeghiyan, and F. Ghassemi, "ConsiDroid: A Concolic-based Tool for Detecting SQL Injection Vulnerability in Android Apps," pp. 1–10, 2018. [Online]. Available: http://arxiv.org/abs/1811.10448

[26] H. Jahankhani, S. Fernando, M. Z. Nkhoma, and H. Mouratidis, "Information Systems Security," *International Journal of Information Security and Privacy*, vol. 1, no. 3, pp. 13–25, 2011.

[27] R. Yan, X. Xiao, G. Hu, S. Peng, and Y. Jiang, "New deep learning method to detect code injection attacks on hybrid applications," *Journal of Systems and Software*, vol. 137, pp. 67–77, mar 2018.

[28] Y. Dong, Y. Zhang, H. Ma, Q. Wu, Q. Liu, K. Wang, and W. Wang, "An adaptive system for detecting malicious queries in web attacks," *Science China Information Sciences*, vol. 61, no. 3, Feb 2018. [Online]. Available: https://doi.org/10.1007/s11432-017-9288-4

[29] Y. Fang, Y. Li, L. Liu, and C. Huang, "DeepXSS." Association for Computing Machinery (ACM), may 2018, pp. 47–51.

[30] Y. Li, R. Ma, and R. Jiao, "A hybrid malicious code detection method based on deep learning," *International Journal of Security and its Applications*, vol. 9, no. 5, pp. 205–216, 2015.

[31] G. S. M. Muth, "Top 10 web application security vulnerabilities, penn computing. university of pennsylvania." [Online]. Available: www.upenn.edu/computing/group/npc/pending/draft-20180418-webappsec.html

[32] "Owasp top 10 2013 a1: Injection flaws, owasp." [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-A1-Injection

[33] "Sql injection, microsoft," 2012. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms161953(v=sql.105)

[34] "P. grazie, phd sqlprevent thesis. university of british columbia (ubc) vancouver, canada," 2008.

[35] "SQL Injection Detection and Prevention Techniques," *International Journal of Advancements in Computing Technology*, vol. 3, no. 7, pp. 82–91, 2012.

[36] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations," *Ccs*, pp. 12–24, 2007.

[37] W. G. J. Halfond, J. Viegas, and A. Orso, "A Classication of SQL Injection Attacks.pdf," 2006.

[38] Symantec, *Symantec Internet Security Threat Report: Trends for July-December 2007 (Executive Summary)*, 2008, vol. XIII, no. April.

[39] S. Ranger, "At $30,000 for a flaw, bug bounties are big and getting bigger, zdnet," 2017. [Online]. Available: https://www.zdnet.com/article/at-30000-for-a-flaw-bug-bounties-are-big-and-getting-bigger/

[40] A. Klein, "Dom based cross site scripting or xss of the third kind, a look at an overlooked flavor of xss," 2005. [Online]. Available: http://www.webappsec.org/projects/articles/071105.txt

[41] O. Shaya, "Using machine learning in networks intrusion detection," no. August 2008, pp. 0–13, 2008.

[42] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015. [Online]. Available: https://www.cs.toronto.edu/~hinton/absps/NatureDeepReview.pdf

[43] "Darktrace." [Online]. Available: https://www.darktrace.com/en/resources/

[44] P. N. Stuart J. Russell, "Artificial intelligence: A modern approach," 2010.

[45] H. Kaur, G. Singh, and J. Minhas, "A Review of Machine Learning based Anomaly Detection Techniques," *International Journal of Computer Applications Technology and Research*, vol. 2, no. 2, pp. 185–187, 2013. [Online]. Available: https://arxiv.org/pdf/1307.7286.pdf

[46] "Bernardo damele, a.g., stampar, m.: Sqlmap: automatic sql injection and database takeover tool," accessed: 2019-04-05. [Online]. Available: http://sqlmap.sourceforge.net/

[47] I. Lee, S. Jeong, S. Yeo, and J. Moon, "A novel method for SQL injection attack detection based on removing SQL query attribute values," *Mathematical and Computer Modelling*, vol. 55, no. 1-2, pp. 58–68, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.mcm.2011.01.050

[48] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng, "On optimization methods for deep learning," in *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ser. ICML'11. USA: Omnipress, 2011, pp. 265–272. [Online]. Available: http://dl.acm.org/citation.cfm?id=3104482.3104516

[49] "Sql payload dataset, github," accessed: 2019-01-02. [Online]. Available: https://github.com/SuperCowPowers/data_hacking/tree/master/sql_injection/data

[50] "Xss payload dataset, github," accessed: 2019-01-02. [Online]. Available: https://github.com/ismailtasdelen/xss-payload-list

[51] "Definition of overfitting, oxford dictionaries," accessed: 2019-02-14. [Online]. Available: https://en.oxforddictionaries.com/definition/overfitting