DeepSQLi: Deep Semantic Learning for Testing SQL Injection

Muyang Liu
University of Electronic Science and Technology of China
Chengdu, China
muyangl@foxmail.com

Ke Li[*]
University of Exeter
Exeter, UK
k.li@exeter.ac.uk

Tao Chen
Loughborough University
Loughborough, UK
t.t.chen@lboro.ac.uk

ABSTRACT

Security is unarguably the most serious concern for Web applications, to which SQL injection (SQLi) attack is one of the most devastating attacks. Automatically testing SQLi vulnerabilities is of ultimate importance, yet is unfortunately far from trivial to implement. This is because the existence of a huge, or potentially infinite, number of variants and semantic possibilities of SQL leading to SQLi attacks on various Web applications. In this paper, we propose a deep natural language processing based tool, dubbed `DeepSQLi`, to generate test cases for detecting SQLi vulnerabilities. Through adopting deep learning based neural language model and sequence of words prediction, `DeepSQLi` is equipped with the ability to learn the semantic knowledge embedded in SQLi attacks, allowing it to translate user inputs (or a test case) into a new test case, which is semantically related and potentially more sophisticated. Experiments are conducted to compare `DeepSQLi` with `SQLmap`, a state-of-the-art SQLi testing automation tool, on six real-world Web applications that are of different scales, characteristics and domains. Empirical results demonstrate the effectiveness and the remarkable superiority of `DeepSQLi` over `SQLmap`, such that more SQLi vulnerabilities can be identified by using a less number of test cases, whilst running much faster.

CCS CONCEPTS

• Security and privacy → Web application security; **• Software and its engineering → Software testing and debugging**.

KEYWORDS

Web security, SQL injection, test case generation, natural language processing, deep learning

ACM Reference Format:
Muyang Liu, Ke Li, and Tao Chen. 2020. DeepSQLi: Deep Semantic Learning for Testing SQL Injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3395363.3397375

[*]Li is the corresponding author of this paper. All authors made commensurate contributions to this paper. Li designed the research. Liu built the system and carried out experiments. Li and Chen supervised the research and interpreted experimental data and wrote the manuscript.

1 INTRODUCTION

Web applications have become increasingly ubiquitous and important since the ever prevalence of distributed computing paradigms, such as Cyber-Physical Systems and Internet-of-Things. Yet, they are unfortunately vulnerable to a variety of security threats, among which SQL injection (SQLi) has been widely recognised as one of the most devastating threats. Generally speaking, SQLi is an injection attack that embeds scripts in user inputs to execute malicious SQL statements over the relational database management system (RDBMS) running behind a Web application. As stated in the Akamai report[1], SQLi attacks constituted 65.1% of the cyber-attacks on Web applications during November 2017 to March 2019. It also shows that the number of different types of Web attacks (e.g., XSS, LFI and PHPi) has ever increased, but none of them have been growing as fast as SQLi attacks. Therefore, detecting and preventing SQLi vulnerabilities are of ultimate importance to improve the reliability and trustworthiness of modern Web applications.

There are two common approaches to protect Web applications from SQLi attacks. The first one is customised negative validation, also known as input validation. Its basic idea is to protect Web applications from attacks by forbidden patterns or keywords manually crafted by software engineers. Unfortunately, it is difficult, if not impossible, to enumerate a comprehensive set of validation rules that is able to cover all types of attacks. The second approach is prepared statement that allow embedding user inputs as parameters, also known as placeholders. By doing so, attackers are difficult to embed SQLi code in user inputs since they are treated as value for the parameter. However, as discussed in [23] and [24], prepared statement is difficult to design given the sophistication of defensive coding guideline. In addition, there are many other terms, such as dynamic SQL of DDL statement (e.g., `create`, `drop` and `alter`) and table structure (e.g, names of columns, tables and schema) cannot be parameterised.

Test case generation, which build test suites for detecting errors of the system under test (SUT), is the most important and fundamental process of software testing. This is a widely used practice to detect SQLi vulnerabilities where test suites come up with a set of malicious user inputs that mimic various successful SQLi attacks, each of which forms a test case. However, enumerating a comprehensive set of semantically related test cases to fully test the SUT is extremely challenging, if not impossible. This is because there are a variety of SQLi attacks, many complex variants of which share similar SQL semantic. For example, the same attack can be

[1] https://www.akamai.com/

diversified by changing the encoding form, which appears to be different but is semantically equivalent, in order to evade detection.

Just like human natural language, malicious SQL statements have their unique semantic. Therefore, test case generation for detecting SQLi vulnerabilities can take great advantages by exploiting the knowledge from such semantic naturalness. For example, given a Web form with two user input fields, i.e., `username` and `password`, the following SQL statement conforms to a SQLi attack:

```
SELECT * FROM members WHERE
username='admin'+OR+'1'='1' AND password=''--'
```

where the underlined parts are input by a user and constitute a test case that leads to an attack to the SUT. Given this SQLi attack, we are able interpret some semantic knowledge as follows.

- This is a tautology attack that is able to use any tautological clause, e.g., `OR 1=1`, to alter the statement in a semantically equivalent and syntactically correct manner without compromising its maliciousness.
- To meet the SQL syntax, an injection needs to have an appropriate placement of single quotation to complete a SQL statement. Therefore, the attack should be written as `admin'+OR+'1'='1`. In addition, the unnecessary part of the original statement can be commented by `--`.
- In practice, due to the use of some input filters like firewalls, blank characters will highly likely be trimmed by modern Web applications thus leading to the failure of `admin' OR 1=1` to form a tautology attack. By replacing those blank characters with `+`, which is semantically equivalent, the attacker is able to disguise the attack in a more sophisticated manner.

Although semantic knowledge can be interpreted by software engineers, it is far from trivial to leverage such knowledge to automate the test case generation process.

Traditional test case generation techniques mainly rely on software engineers to specify rules to create a set of semantically tailored test cases, either in a manual [9, 19] or *semi*-automatic manner [3, 5, 31]. Such process is of limited flexibility due to the restriction of human crafted rules. Furthermore, it is expensive in practice or even be computationally infeasible for modern complex Web applications.

Recently, there has been a growing interest of applying machine learning algorithms to develop artificial intelligence (AI) tools that automate the test case generation process [11, 20, 27, 29] and [22]. This type of methods requires limited human intervention and do not assume any fixed set of SQL syntax. However, they are mainly implemented as a classifier that is used to diagnose whether a SQL statement (or part of it) is a valid statement or a malicious injection. To the best of our knowledge, none of those existing AI based tools are able to proactively generate semantically related SQLi attacks during the testing phase. There have been some attempts that take semantic knowledge into consideration. For example, [27] developed a classifier that considers the semantic abnormality of the `OR` phrase (e.g., `OR 1=1` or `OR 'i' in ('g', 'i')`) in a tautology attack. Unfortunately, this method ignores other alternatives, which might be important when semantically generating SQLi attacks, to create tautology (e.g., we can use `--` to comment out other code) .

Bearing the above considerations in mind, this paper proposes a deep natural language processing (NLP) based tool[2], dubbed `DeepSQLi`, which learns and exploits the semantic knowledge and naturalness of SQLi attacks, to automatically generate various semantically meaningful and maliciously effective test cases. Similar to the machine translation between dialects of the same language, `DeepSQLi` takes a set of normal user inputs or existing test case for a SQL statement (one dialect) and *translates* it into another test case (another dialect), which is semantically related but potentially more sophisticated, to form a new SQLi attack.

Contributions. The major contributions of this paper are:

- `DeepSQLi` is a fully automatic, end-to-end tool empowered by a tailored neural language model trained under the `Transformer` [32] architecture. To the best of our knowledge, this work is the first of its kind to adopt `Transformer` to solve problems in the context of software testing.
- To facilitate the semantic knowledge learning from SQL statement, five mutation operators are developed to help enrich the training dataset. Unlike the classic machine translation where only the sentence with the most probable semantic match would be of interest, in `DeepSQLi`, we extend the neural language model with `Beam search` [26], in order to generate more than one semantically related test case based on the given test case/normal inputs that needs translation. This helps to generate a much more diverse set of test cases, and thus providing larger chance to find more vulnerabilities.
- The effectiveness of `DeepSQLi` is validated on six real-world Web applications selected from various domains. They are with various scales and have a variety of characteristics. The results show that `DeepSQLi` is better than `SQLMap`, a state-of-the-art SQLi testing automation tool, in terms of the number of vulnerabilities detected and exploitation rate whilst running up to 6× faster.

Novelty. What make `DeepSQLi` *unique* are:

- It is able to translate any normal user inputs into some malicious inputs, which constitute to a test case. Further, it is capable of translating an existing test cases into another semantically related, but potentially more sophisticated test case to form a new SQLi attack.
- If the generated test case cannot achieve a successful SQLi attack, it would be fed back to the neural language model as an input. By doing so, `DeepSQLi` is continually adapted to create more sophisticated test cases, thus improving the chance to find previously unknown and deeply hidden vulnerabilities.

The remaining paper is organised as follows. Section 2 provides a pragmatic tutorial of neural language model used for SQLi in this paper. Section 3 delineates the implementation detail of our proposed `DeepSQLi`. Section 4 presents and discusses the empirical results on six real-world Web applications. Section 5 exploits the threats to validity and related works are overviewed in Section 6. Section 7 summarises the contributions of this paper and provides some thoughts on future directions.

[2] Source code and experimental data can be accessed at our project repository: https://github.com/COLA-Laboratory/issta2020.

2 DEEP NATURAL LANGUAGE PROCESSING FOR SQLI

In this section, we elaborate on the concepts and algorithms that underpin `DeepSQLi` and discuss how they were tailored to the problem of translating user inputs into test cases.

2.1 Neural Language Model for SQLi

Given a sequence of user inputs $I_u = \{w_1, \ldots, w_N\}$ where w_i is the i-th token, a language model aims to estimate a joint conditional probability of tokens of I_u. Since a direct calculation of this multi-dimensional probability is far from trivial, it is usually approximated by n-gram models [7] as:

$$\begin{aligned} P(w_1, \cdots, w_N) &= \prod_{i=1}^{N} P(w_i | w_1, \cdots, w_{i-1}) \\ &\approx \prod_{i=1}^{N} P(w_i | w_{i-n+1}, \cdots, w_{i-1}) \end{aligned}, \tag{1}$$

where N is the number of consecutive tokens. According to equation (1), we can see that the prediction made by the n-gram model is conditioned on its $n-1$ predecessor tokens. However, as discussed in [12], n-gram models are suffered from a sparsity issue where it does not work when predicting the first token in a sequence that has no predecessor.

To mitigate such issue, `DeepSQLi` makes use of neural language model as its fundamental building block. Generally speaking, it is a language model based on neural networks along with a probability distribution over sequences of tokens. In our context, such probability distribution indicates the likelihood to which a sequence of user inputs conform to a SQLi attack. For example, a sequence of inputs `admin'+OR+'1'='1` will have a higher probability since it is able to conform to a SQLi attack; whereas another sequence of inputs `OR SELECT AND 1`, which is semantically invalid from the injection point of view, will have a lower probability to become a SQLi attack.

Comparing to the n-gram model, which merely depends on the probability, the neural language model represents the tokens of an input sequence in a vectorised format, as known as word embedding which is an integral part in neural language model training. Empowered by deep neural networks, a neural language model is more flexibility with predecessor tokens having longer distances thus is resilient to data sparsity.

In `DeepSQLi`, we adopt a neural language model for token-level sequence modeling, given that a token is the most basic element in SQL syntax. In other words, given a sequence of user inputs I_u, the neural language model aims to estimate the joint probability of the inclusive vectorised tokens.

2.2 Multi-head Self-Attention in Neural Language Model

Attention mechanisms, which allow modelling of dependencies without regarding to their distance in sequences, have recently become an integral part of sequence generation tasks [25]. Among them, self-attention is an attention mechanism that has the ability to represent relationship between tokens in a sequence. For example, we can better understand the token `"1"` in the sequence `"OR 2 > 1"` by answering three questions: "*why to compare (i.e., what kind of attack)*", "*how to compare*" and "*who to compare with*". Self-attention have been successfully applied to deal with various NLP tasks, such as machine translation [32], speech recognition [10] and music generation [17]. In `DeepSQLi`, the self-attention is calculated by the scaled dot-product attention proposed by Vaswani et al [32]:

$$\begin{aligned} \mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \mathbf{K} = \mathbf{X}\mathbf{W}_K, \mathbf{V} = \mathbf{X}\mathbf{W}_V \\ A(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\mathbf{V}, \end{aligned} \tag{2}$$

where $\mathbf{X}$ is the word embedding, i.e., the vector representation, of the input sequence, $\mathbf{Q}$ is a matrix consists of a set of packed queries, $\mathbf{K}$ and $\mathbf{V}$ are keys and values matrices whilst d is the dimension of the key. In particular, $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$ are obtained by multiplying $\mathbf{X}$ by three weight matrices $\mathbf{W}_Q$, $\mathbf{W}_K$ and $\mathbf{W}_V$.

In order to learn more diverse representations, we apply a multi-head self-attention in `DeepSQLi` given that it has the ability to obtain more information from the input sequence by concatenating multiple independent self-attentions. Specifically, a multi-head self-attention can be formulated as:

$$MA(\mathbf{Q}_X, \mathbf{K}_X, \mathbf{V}_X) = [A_1(\mathbf{Q}_1, \mathbf{K}_1, \mathbf{V}_1) \otimes \cdots \otimes A_h(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h)] \cdot \mathbf{W}_a, \tag{3}$$

where $\otimes$ is a concatenation operation, $\mathbf{W}_a$ is a weight matrix and h is the length of parallel attention layers, also known as heads. Since each head is a unique linear transformation of the input sequence representation as queries, keys and values, the concatenation of multiple independent heads enables the information extraction from different subspaces thus leading to more diverse representations.

2.3 Encoder-Decoder (Seq2Seq) Model

To train the neural language model, we adopt `Seq2Seq`—a general framework consists of an encoder and a decoder—in `DeepSQLi`. In particular, `Transformer` [32] is used to build the `Seq2Seq` model in `DeepSQLi` instead of those traditional recurrent neural network (RNN) [33] and convolutional neural network (CNN) [18], given its state-of-the-art performance reported in many `Seq2Seq` tasks.

Figure 1 shows an illustrative flowchart of the `Seq2Seq` model in `DeepSQLi`. The encoder takes a sequence of vector representations of N tokens, denoted as $\mathbf{X} = \{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$, which embed semantic information of tokens; whilst the input of the decoder is another sequence of vector representations of $\overline{N}$ tokens, denoted as $\mathbf{Y} = \{\mathbf{y}_1, \cdots, \mathbf{y}_{\overline{N}}\}$. Note that N is not necessary to be equal to $\overline{N}$. The purpose of this `Seq2Seq` model is to learn a conditional probability distribution over the output sequence conditioned on the input sequence, denoted as $P(\mathbf{y}_1, \cdots, \mathbf{y}_{\overline{N}} | \mathbf{x}_1, \cdots, \mathbf{x}_N)$. As for the example shown in Figure 1 where the input sequence is `OR 2 > 1` and the output sequence is `|| True`, the conditional probability is $P($`|| True`$|$`OR 2 > 1`$)$.

More specifically, the encoder in the left hand side of Figure 1 consists of N identical layers, each of which has a multi-head self-attention mechanism sub-layer and a deep feed-forward neural

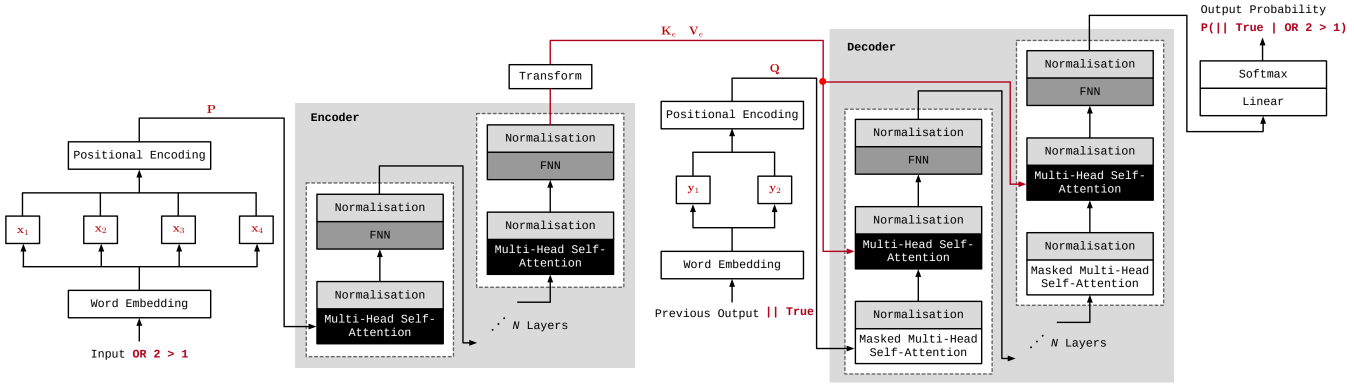

Figure 1: An illustrative working flowchart of the encoder-decoder (Seq2Seq) model in DeepSQLi.

network (FFN) sub-layer. The first sub-layer is the multi-head self-attention. As show in Figure 1, the output of the multi-head self-attention, calculated by equation (3), is denoted as $\mathbf{Z}_1$. It is supplemented with a residual connection $\varepsilon_{\mathbf{Z}_1}$ to come up with the output N_1 after a layer-normalisation. This process can be formulated as:

$$\mathrm{N}_1 = \texttt{layer-normalisation}(\mathrm{Z}_1 + \varepsilon_{\mathrm{Z}_1}). \tag{4}$$

Afterwards, $\mathbf{N}_1$ is fed to the second sub-layer, i.e., a FNN, to carry out a non-linear transformation. Specifically, the basic mechanism of the FNN is formulated as:

$$\mathrm{z}_2 = \mathrm{FFN}(\mathbf{n}_1) = \max(0, \mathbf{n}_1 W_1 + b_1) W_2 + b_2, \tag{5}$$

where $\mathbf{n}_1$ is a vector of $\mathbf{N}_1$. Thereafter, the output of the FNN, i.e., $\mathbf{Z}_2$, will be transformed to two matrices $\mathbf{K}_e$ and $\mathbf{V}_e$ after being normalized to N_2. It is worth noting that $\mathbf{K}_e$ and $\mathbf{V}_e$ are the output of the encoder whilst they embed all information of the input sequence `"OR 2 > 1"`.

As for the decoder shown in the right hand side of Figure 1, it takes $\mathbf{K}_e$ and $\mathbf{V}_e$ output from the encoder as a part of inputs for predicting a sequence of semantically translated vector representation $\mathbf{Y}$. The decoder is also composed of a stack of N identical layers, each of which consists of a masked multi-head self-attention, a multi-head self-attention and a FNN sub-layers. In particular, the computational process of the multi-head self-attention and the FNN sub-layers is similar to that of the encoder, except that $\mathbf{K}_e$ and $\mathbf{V}_e$ are used as the $\mathbf{K}$ and $\mathbf{V}$ of equation (2) in the multi-head self-attention sub-layer. As for the masked multi-head self-attention sub-layer, it is used to avoid looking into tokens after the one under prediction. For example, the multi-head self-attention masks the second token `"True"` when predicting the first one `"||"`.

In principle, the `Transformer` used to do `Seq2Seq` allows for significantly more parallel processing and has been reported as a new state-of-the-art. Unlike the RNN, which takes tokens in a sequential manner, the multi-head attention computes the output of each token independently, without considering the order of words. Since the SQLi inputs used in `DeepSQLi` are sequences with determined semantics and syntax, it may leads to a less effective modelling of the sequence information without taking any order of tokens into consideration. To take such information into account, the `Transformer` supplements each input embedding with a vector called positional encoding (PE). Specifically, PE is calculated by sine and cosine functions with various frequencies:

$$\mathbf{PE}_i = \begin{cases} \{\sin(\frac{i}{10000^{\frac{2}{d_e}}}), \cdots, \sin(\frac{i}{10000^{\frac{2d_e}{d_e}}})\}, & \text{if } i\%2 == 0 \\ \{\cos(\frac{i}{10000^{\frac{2}{d_e}}}), \cdots, \cos(\frac{i}{10000^{\frac{2d_e}{d_e}}})\}, & \text{if } i\%2 == 1 \end{cases}, \tag{6}$$

where d_e is the dimension of the vector representation, i represents the index of the token in the sequence. $\mathbf{PE}_i$ indicates that the sine variable is added to the even position of the token vector whist the cosine variable is added to the odd position. Thereafter, the output token vector $\mathbf{x}_i$ is updated by supplementing $\mathbf{PE}_i$, i.e., $\mathbf{p}_i = \mathbf{x}_i + \mathbf{PE}_i$. By doing so, the relative position between different embedding can be inferred without demanding costs .

3 END-TO-END TESTING WITH DEEPSQLI

`DeepSQLi` is designed as a end-to-end tool, covering all stages in the penetration testing [13]. As shown in Figure 2, the main workflow of `DeepSQLi` consists of four steps: *training*, *crawler*, *test case generation & diversification* and *evaluation*, each of which is outlined as follows:

Step 1: *Training:* This is the first step of `DeepSQLi` where a neural language model is trained by the `Transformer`. Agnostic to the Web application, the training dataset can either be summarised from historical testing repository or, as what we have done in this work, mined from publicly available SQLi test cases/data. The collected test cases/data would then be paired, mutated and preprocessed to constitute the training dataset. We will discuss this in detail in Section 3.1.

Step 2: *Crawler:* Once the model is trained, `DeepSQLi` uses a crawler (e.g., the crawler of the Burp Suite project[3]) to automatically parse the Web links of the SUT (as provided by software engineers). The major purpose of the crawler is to extract the fields for user inputs in the HTML elements, e.g., `<input>` and `<textarea>`, which are regarded as the injection points for a SQLi attack. These injection points, along with their default values, serve as the starting point for the neural language model to generate SQLi test cases.

[3] https://portswigger.net/burp

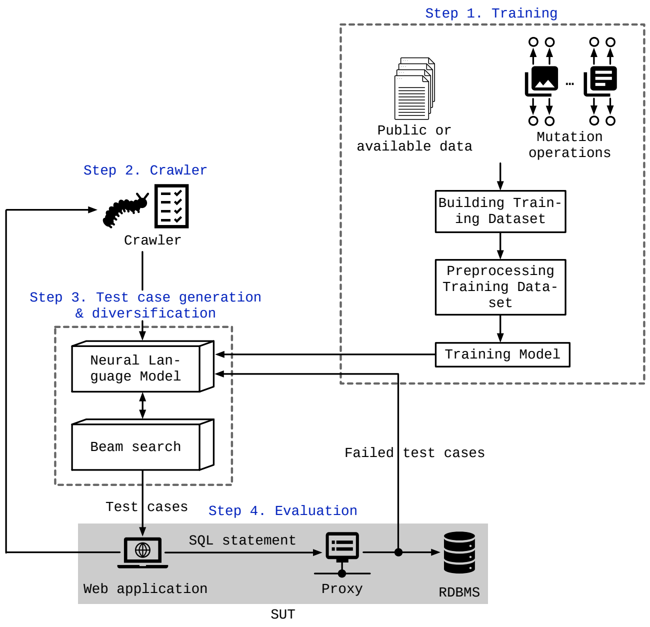

Figure 2: The architecture and workflow of DeepSQLi.

Step 3: *Test Case Generation & Diversification:* The neural language model of DeepSQLi is able to generate tokens with different probabilities leading to a test case. To fully exploit such advantage for exploring a diverse set of alternative test cases, in the translation phase, we let the neural language model to generate and explore tokens with the top *m* probability instead of merely using only the highest one. This is achieved by Beam search [26], a heuristic that guides the neural language model to generate *m* test cases based on the ranked probabilities. This will be elaborated in detail in Section 3.2.

Step 4: *Evaluation:* Based on the test cases generated at Step 3, we randomly choose one and fed it into the SUT for evaluation. In particular, to avoid compromising the SUT, it is equipped with a proxy, i.e., SQL Parser[4], before the RDBMS to identify whether or not a malicious SQL statement can achieve a successful attack. To improve the chance of detecting different vulnerabilities, DeepSQLi stops exploring a specific vulnerability once it has been found through a test case.

It is worth noting that DeepSQLi does not discard unsuccessful test cases which fail to achieve attacks as identified by the proxy. Instead, they are fed back to the neural language model to serve as a starting point for a re-translation (i.e., go through Step 3 again). This comes up with a closed-loop until the test case successfully injects the SUT or the maximum number of attempts is reached. By this means, DeepSQLi grants the ability to generate not only the standard SQLi attacks, but also those more sophisticated ones which would otherwise be difficult to create.

[4] http://www.sqlparser.com.

3.1 Training of Neural Language Model

DeepSQLi is agnostic to the SUT since we train a neural language model to learn the semantic knowledge of SQLi attack that is independent to an actual Web application. Therefore, DeepSQLi, once being trained sufficiently, can be applied to a wide range of SUT as the semantic knowledge of SQLi is easily generalisable. The overall training procedure is illustrated in Figure 2. In the following subsections, we further elaborate some key procedures in detail.

3.1.1 Building Training Dataset. In practice, it is possible that the SUT has accumulated a good amount of test cases from previous testing runs, which can serve as the training dataset. Otherwise, since we are only interested to learn the SQL semantics for injections, the neural language model of DeepSQLi can be trained with any publicly available test cases for SQLi testing regardless to the Web applications, as what we have done in this paper.

Since our purpose is to translate and generate a semantically related test case based on either a normal user inputs or another test case, the test cases in the training dataset, which work on the same injection point(s), need to appear in input-output pairs for training. In particular, an input-output pair (A, B) is valid if any of the following conditions is met:

- A is a known normal user input and B is a test case. For example, `https://xxx/id=7` can be paired with `https://xxx/id=7 union select database()`.
- A is a test case whilst B is another one, which is different but still conform to the same type of attack. For example, A is `' OR 1=1 --` and B is `' OR 5<7 --` . It is clear that both of them lead to a semantically related tautology attack.
- A is a test case whilst B is an extended one based on A, thereby we can expect that B is more sophisticated but in a different attack type. For example, A: `' OR 1=1; --` can be paired with B: `' or 1=1; select group_concat(schema_name) from information_schema. schema" ta; --`, which belongs to a different type of attack, i.e., a piggybacked queries attack extended from A.

In this work, we manually create input-output test case pairs to build the training dataset based on publicly available SQLi test cases, such as those from public repositories, according to the aforementioned three conditions. More specifically, the training dataset is built according to the following two steps.

Step 1: We mined the repositories of fuzzing test or brute force tools from various GitHub projects[5], given that they often host a large amount of test case data in their library and these data are usually arranged according to the types of attacks (along with normal user inputs). This makes us easier to constitute input-output pairs according to the aforementioned conditions. In particular, it is inevitable to devote non-trivial manual efforts to classifying and arranging some more cluttered data.

Step 2: When analyzing the mined dataset, we found it is difficult to constitute input-output pairs for the disguise attack. For example, a test case containing `' OR 1=1 --` may fail to

[5] https://tinyurl.com/wh94b8t

Table 1: Description of five mutation operators used in DeepSQLi to enrich the training dataset.

Operators	Explanation	Example	
		Input	Output
`Predicate`	Mutation by using relational predicates without changing the expression's logical results. In particular, the relational predicates are $\{<, >, \leq, \geq, between, in, like\}$.	`and 8>= 56`	`and‘l’in (‘m’,‘y’)`
`Unicode`	Mutation from a character to its equivalent Unicode format.	`#`	`%23`
`ASCII`	Mutation from a character to its equivalent ASCII encoding format.	`a`	`CHAR(97)`
`Keywords`	Confusion of the capital and small letters of keywords in a test case.	`select`	`seLeCt`
`Blank`	Replace the blank character in a test case with an equivalent symbol.	`or 1`	`or/**/1`

inject the SUT due to the existence of the standard input validation. Whereas a successful SQLi attack can be formulated by simply change `' OR 1=1 --` to `%27%20OR%201=1%20--`, which is semantically the same but in a different encoding format. This is caused by the rare existence of semantically similar SQLi attacks based on manipulating synonyms and encoding formats from those public repositories. To tackle this issue, five mutation operators, as shown in Table 1, are developed to further exploit the test cases obtained from Step 1. By doing so, we can expect a semantically similar test case that finally conforms to a disguise attack. In principle, these mutation operators are able to enrich the test case samples in the training dataset.

3.1.2 Preprocessing Training Dataset. After building the training dataset, we then need to preprocess the data samples by generalisation and word segmentation to eliminate unnecessary noise in the training data. Notably, unlike classic machine learning approaches [4] that generalise all the words in a data sample, we only generalise the user inputs, the table name and column name to unify tokens `"[normal]"`, `"[table]"` and `"[column]"`. This is because other words and characters, including text-, encoding-, blank characters-, quotes-transforms, are specifically tailored in a SQLi attack, thereby they should not be generalised. For example, considering a test case `"admin'%20or%203<7;--"` in the training dataset, it is converted into a sequence as `"['[normal]', ''', '%20', 'or', '%2', '3', '<', '7', ';', '--']"` after the generalisation.

3.1.3 Training the Model. In DeepSQLi, the neural language model is trained under the Transformer architecture. As suggested by Vaswani et al. [32], a stochastic optimization algorithm called Adam [21], with the recommended settings of $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$, is used as the training algorithm.

To prevent overfitting, a 10-fold cross validation is applied in the training process with Adam to optimize the setting of some hyper-parameters, including the number of layers in the encoder and the decoder, the number of hidden layers and neurons in FNN, as well as the number of heads used in the self-attention mechanism of the Transformer. In particular, the following loss function is used in the training process.

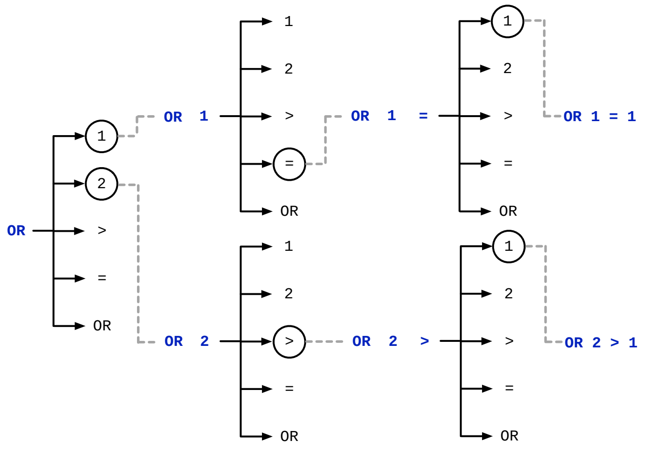

Figure 3: An illustrative example of Beam search with the beam width is 2 and the corpus size is 5, i.e., the possible SQL tokens to be chosen are "1" , "2" , ">" , "=" and "OR" .

$$L(Y, P(Y|X)) = -\log P(y_1, \ldots, y_p | x_1, \ldots, x_s)$$
$$= -\sum_{t=1}^{p} \log P(y_t | y_1, \ldots, y_{t-1}, x_1, \ldots, x_s) \quad (7)$$

The hyper-parameter setting leading to the minimum of the above loss function is chosen to train the model.

3.2 Test Case Generation & Diversification

Since the goal of the classic machine translation is to identify the most accurate sentence that matches the semantics, only the tokens with the highest probabilities in the context is of interest. In contrast, the major purpose of the test case generation in DeepSQLi is to generate as diverse test cases as possible so that more bugs or vulnerabilities can be identified. In this case, any semantically related test cases are of equal importance as long as they are likely to find new vulnerabilities. By this justification, the classic neural language model, which only outputs the test case having the largest matching probability, is not suitable for DeepSQLi.

To make the generated test cases be more diversified, `Beam search` [26] is used to extend and guide the neural language model to generate a set of semantically related test cases. In a nutshell, instead of only focusing on the most probable tokens, `Beam search` selects the m most probable tokens at each time step given all previously selected tokens, where m is the beam width. Afterwards, by leveraging the neural language model, it merely carries on the search from these m tokens and discard the others. Figure 3 shows an example when $m = 2$ and the corpus size is 5. According to the first token `" OR"` , 2 sequences `OR 1` and `OR 2` with the highest probability are selected from 5 candidate sequences at the first time step. Then, the 2 sequences with the highest probability from the 10 possible output sequences are selected at each subsequent step until the last token in the sequence is predicted. From the above search process, we can also see that `Beam search` does not only improve the diversity, but also amplify the exploration of search space thus improve the accuracy. In `DeepSQLi`, we set the beam width as 5, which means that the neural language model creates 5 different test cases for each input. For example, considering the case when the input is `' OR 1=1`, then `DeepSQLi` could generate the following outputs: `' OR 5<7`, `' || 5<7`, `'+OR+1=1`, `' OR 1=1`, and `' OR 1=1--`.

In principle, this diversification procedure enables `DeepSQLi` to find more vulnerabilities since the output test cases translated from a given test case (or normal user inputs) are diversified.

4 EVALUATION

In this section, the effectiveness of `DeepSQLi` is evaluated by comparing with `SQLmap`[6], the state-of-the-art SQLi testing automation tool [28], on six real-world Web applications. Note that `SQLmap` was not designed with automated crawling, thus it is not directly comparable with `DeepSQLi` under our experimental setup. To make a fair comparison and to mitigate the potential bias, we extend `SQLmap` with a crawler, i.e., the Burp Suite project used in `DeepSQLi`.

Our empirical study aims to address the following three research questions (RQs):

- **RQ1:** ***Is* `DeepSQLi` *effective for detecting SQLi vulnerabilities?***
- **RQ2:** ***Can* `DeepSQLi` *outperform* `SQLmap` *for detecting SQLi vulnerabilities?***
- **RQ3:** ***How does* `DeepSQLi` *perform on SUT with advanced input validation in contrast to* `SQLmap`*?***

All experiments were carried out on a desktop with Intel i7-8700 3.20GHz CPU, 32GB memory and 64bit Ubuntu 18.04.2.

4.1 Experiment Setup

4.1.1 Subject SUT. Our experiments were conducted on six SUT[7] written in Java and with MySQL as the back-end database system. All these SUT are real-world commercial Web applications used by many researchers in this literature, e.g., [16]. In particular, there are two levels of input validation equipped with these SUT:

- *Essential:* This level filters the most commonly used keywords in SQLi attacks, e.g., '`AND`' and '`OR`'.

[6] http://SQLmap.org/.

[7] More detailed information can be found at http://examples.codecharge.com.

Table 2: Real-world SUT used in our experiments.

SUT	LOC	Servlets*	DBIs	KV
Employee	5,658	7 (10)	23	25
Classifieds	10,949	6 (14)	34	18
Portal	16,453	3 (28)	67	39
Office Talk	4,543	7 (64)	40	14
Events	7,575	7 (13)	31	26
Checkers	5,421	18 (61)	5	44

* The # of accessible Servlets (the total # of Servlets).

- *Advanced:* This is an enhanced level[8] that additionally filters some special characters, which are rarely used but can still be part of a SQLi attack, e.g., '`&&`' and '`||`'.

Table 2 provides a briefing of the SUT considered in our experiments. In particular, these SUT cover a wide spectrum of Web applications under real-world settings. They are chosen from various application domains with different scales in terms of the line-of-code (LOC) and they involve various database interactions (DBIs)[9]. Furthermore, the number of Servlets and the number of known SQLi vulnerabilities (dubbed as KV in Table 2) are set the same as the existing study [16]. It is worth noting that both the number of accessible Servlets and their total amount are shown in Table 2 since not all Servlets are directly accessible.

To mitigate any potentially biased conclusion drawn from the stochastic algorithm, each experiment is repeated 20 times for both `DeepSQLi` and `SQLmap` under every SUT.

4.1.2 Training Dataset. In our experiments, the dataset used to train `DeepSQLi` is constituted by SQLi test cases, consisting of a diverse type of SQLi attacks, collected from various projects in GitHub. We make the training dataset publicly accessible to ensure that our results are reproducible[10]. Afterwards, we preprocess the training dataset by pairing and mutation according to the steps and conditions discussed in Section 3.1.1. In particular, the number of paired SQLi test case instances is 19,220, all of which can be directly used for training `DeepSQLi`. After using the mutation operators, the training dataset is significantly diversified and the number of training data instances is increased to 56,841.

4.1.3 Quality Metrics. The following three quality metrics are used in our empirical evaluations.

- **Number of vulnerabilities found:** We use the number of SQLi vulnerabilities identified by either `DeepSQLi` or `SQLmap` as a criterion to evaluate their ability for improving the security of the underlying SUT.
- **Number of test cases and exploitation rate (ER):** In order to evaluate the ability for utilising computational resources, we keep a record of the total number of test cases generated by either `DeepSQLi` or `SQLmap`, denoted as $T_{\texttt{total}}$.

[8] This level is usually switched off, because more advanced security mechanism can often make the performance of Web application worse off.

[9] The number of SQL statements that can access user inputs.

[10] https://tinyurl.com/wh94b8t

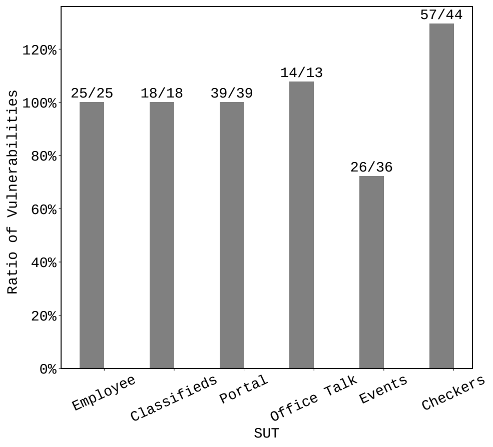

Figure 4: Ratio of # of vulnerabilities identified by `DeepSQLi` to that of `SQLmap`.

In addition, we also chase the number of test cases that successfully lead to SQLi attacks, denoted as $T_{\texttt{success}}$. Thereafter, ER is the ratio of $T_{\texttt{success}}$ to $T_{\texttt{total}}$, i.e., $\frac{T_{\texttt{success}}}{T_{\texttt{total}}}$.

- **CPU wall clock time:** In order to evaluate the computational cost required by either `DeepSQLi` or `SQLmap`, we keep a record of the CPU wall clock time used by them for testing the underlying SUT.

4.2 The Effectiveness of `DeepSQLi`

In this section, we firstly examine `DeepSQLi` on SUT with the *essential* input validation. Figure 4 presents the ratio of the average number of vulnerabilities identified by `DeepSQLi` (over 20 runs) to that of known vulnerabilities. From this result, we can clearly see that `DeepSQLi` is able to identify all known SQLi vulnerabilities for 5 out 6 SUT, except the *Events*. This might be caused by the crawler which fails to capture all injection points in *Events*. In contrast, it is worth noting that `DeepSQLi` is able to identify more SQLi vulnerabilities than those reported in [14] for *Office Talk* and *Checker*. This is a remarkable result that demonstrates the ability of `DeepSQLi` for identifying previously unknown and deeply hidden vulnerabilities of the underlying black-box SUT.

To further understand why `DeepSQLi` is effective for revealing the SQLi vulnerabilities, Table 3 shows the injectable SQL statements and the related test cases generated in our experiments. Specifically, in Example 1, `DeepSQLi` is able to learn that the input test case, which is an unsuccessful SQLi attack, has failed due to a missing quotation mark. Thereby, it generates another semantically related test case which did find a vulnerability. In Example 2, we see more semantically sophisticated amendments though exploiting the learned semantic knowledge of SQL: the initially failed test case `and 3=4` is translated into another semantically related one, i.e., `and‘‘9`, which failed to achieve an attack again. Subsequently, in the next round, `DeepSQLi` then translates it into a more sophisticated and successful test case `and%208=9`, which eventually leads to the discovery of a vulnerability. Likewise, for Examples 3 to 5, the input test cases have been translated into another semantically related and more sophisticated test cases.

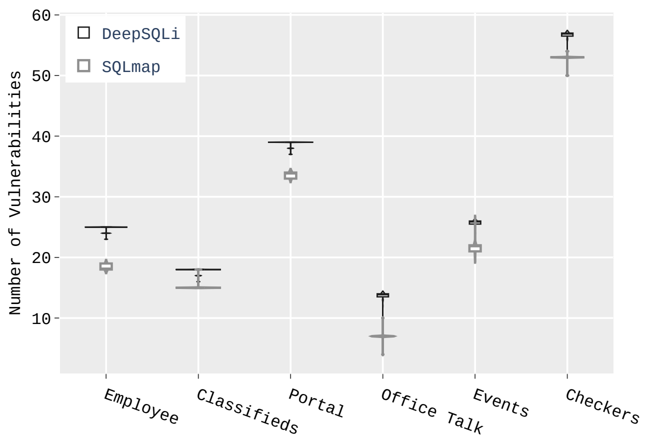

Figure 5: Violin charts of the number of SQLi vulnerabilities identified by `DeepSQLi` (black lines) and `SQLmap` (gray lines) on six SUT with *essential* input validation across 20 runs.

Answer to RQ1: ***Because of the semantic knowledge learned from previous SQLi attacks, `DeepSQLi` has shown its effectiveness in detecting SQLi vulnerabilities. It is worth noting that `DeepSQLi` is able to uncover more vulnerabilities that are deeply hidden and previously unknown in the SUT.***

4.3 Performance Comparison Between `DeepSQLi` and `SQLmap`

Under the SUT with the *essential* input validation, Table 4 shows the comparison results of the total number of test cases generated by `DeepSQLi` and `SQLmap` (dubbed as #total) versus the amount of test cases leading to successful attacks (dubbed as #success). From these results, it is clear that `DeepSQLi` is able to fully test the SUT with fewer test cases than `SQLmap`. In addition, as demonstrated by the better ER values achieved by `DeepSQLi`, we can conclude that `DeepSQLi` is able to better utilise test resources.

To have an in-depth analysis, Figure 5 uses violin charts to visualise the distribution of the number of SQLi vulnerabilities identified by `DeepSQLi` and `SQLmap` on all six SUT across 20 runs. From this comparison result, it is clear that `DeepSQLi` is able to find more vulnerabilities than `SQLmap` at all instances. In particular, as shown in Figure 5, the violin charts of `DeepSQLi` experienced much less variance than that of `SQLmap`. This observation implies that it is capable of producing more robust results by learning and leveraging the semantic knowledge embedded in the previous SQLi test cases.

Table 5 shows the comparison results of the CPU wall clock time required for running `DeepSQLi` and `SQLmap`. From this comparison result, we find that `DeepSQLi` runs much faster than `SQLmap`. In particular, it achieves up to 6× faster running time at the SUT *Portal*. By cross referencing with the results shown in Table 4, we can see that `SQLmap` generates much more test cases than `DeepSQLi`. It is worth noting that more test cases do not indicate any better

Table 3: Examples of inputs and outputs of DeepSQLi

Example 1	*Input*	**SELECT** card_type_id **FROM** card_types **WHERE** card_type_name=‘ and+1=(select count(*) from);-- ';
	Output	**SELECT** card_type_id **FROM** card_types **WHERE** card_type_name=‘ 'and+1=(select count(*) from); -- ';
Example 2	*Input*	**SELECT** category_id, name, par_category_id **FROM** categories **WHERE** category_id= and 3=4 ;
	Output	**SELECT** category_id, name, par_category_id **FROM** categories **WHERE** category_id= and ‘‘7 ;
	Input	**SELECT** category_id, name, par_category_id **FROM** categories **WHERE** category_id= and ‘‘9 ;
	Output	**SELECT** category_id, name, par_category_id **FROM** categories **WHERE** category_id= and%208=9 ;
Example 3	*Input*	**SELECT** category_id, name, par_category_id **FROM** categories **WHERE** category_id= 2;
	Output	**SELECT** category_id, name, par_category_id **FROM** categories **WHERE** category_id= 2;delete from members;
Example 4	*Input*	**SELECT** id, level **FROM** members **WHERE** member_login =‘'%20||%20‘h '=‘h '**AND** member_password=‘'# ';
	Output	**SELECT** id, level **FROM** members **WHERE** member_login =‘'%20Or%20‘h'=‘h '**AND** member_password=‘'-- ';
Example 5	*Input*	**INSERT INTO** members (member_login,member_password,name,email,location,work_phone,home_phone) **VALUES** ("select database()",""test",""test",""test",""test",""1","1");
	Output	**INSERT INTO** members (member_login,member_password,name,email,location,work_phone,home_phone) **VALUES** ("sElEct* database()","test","test","test","test","1","1");

Table 4: Comparison results of the # of total/successful test cases generated by DeepSQLi and SQLmap, and the ER values.

SUT	DeepSQLi		SQLmap	
	#total/#success	ER	#total/#success	ER
Employee	5563/473	8.50%	34851/1534	4.40%
Classifieds	4512/340	7.54%	25954/1046	4.03%
Portal	8657/740	8.55%	63001/2244	3.56%
Office Talk	2998/260	8.67%	9451/462	4.89%
Events	5331/487	9.14%	32136/1440	4.48%
Checkers	13463/1077	8.00%	67919/3352	4.94%

Table 5: Comparison of the CPU wall clock time (in second) used to run DeepSQLi and SQLmap over all 20 runs.

SUT	DeepSQLi	SQLmap
Employee	355	1177
Classifieds	236	931
Portal	357	2105
Office Talk	166	384
Events	259	1094
Checkers	519	2228

contribution for revealing SQLi vulnerabilities, because those test cases might be either unsuccessful or redundant as reflected by the lower ER values achieved by SQLmap. In addition, generating a much higher number of (useless) test cases makes SQLmap much slower than DeepSQLi.

Answer to RQ2: ***DeepSQLi is able to find significantly more SQLi vulnerabilities than SQLmap, with a better utilization of the testing resource as evidenced by the better exploitation rates. DeepSQLi also runs much faster than SQLmap on up to 6× better.***

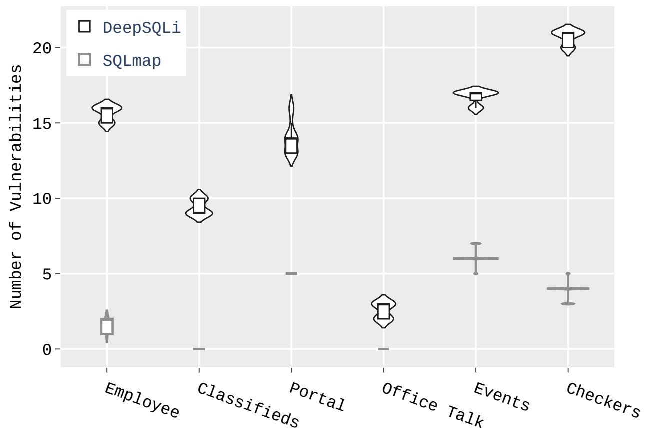

Figure 6: Violin charts of the number of SQLi vulnerabilities identified by DeepSQLi (black lines) and SQLmap (gray lines) on SUT with *advanced* input validation across 20 runs.

4.4 Performance Comparison Between DeepSQLi and SQLmap on SUT with Advanced Input Validation

The previous subsections have validated the effectiveness and performance of DeepSQLi on SUT with *essential* input validation. In this subsection, we switch on the *advanced* input validation in the SUT, aiming to assess the performance of DeepSQLi against SQLmap under more complicated and challenging scenarios.

Figure 6 presents the comparison results for the number of SQLi vulnerabilities found by DeepSQLi and SQLmap on SUT under *advanced* input validation. As can be seen, DeepSQLi finds remarkably more SQLi vulnerabilities than SQLmap. It is worth noting that there are a few runs where SQLmap failed to detect any vulnerability at *Employee*, *Classifieds* and *Office Talk*. In contrast, DeepSQLi shows consistently better performance for finding considerably more SQLi vulnerabilities.

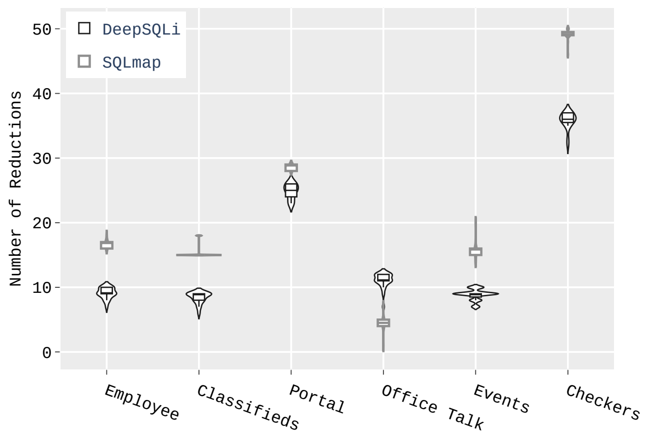

Figure 7: Violin charts of the number of reductions of SQLi vulnerabilities identified by DeepSQLi (black lines) and SQLmap (gray lines) on SUT with *advanced* input validation across 20 runs.

In order to evaluate the sensitivity of DeepSQLi and SQLmap to the strength of input validation, we compare the number of vulnerabilities found on SUT under *advanced* input validation with that on SUT under *essential* input validation. By cross referencing Figure 4, we can observe some reductions on the number of vulnerabilities identified by both tools, as shown in Figure 7. However, in contrast to SQLmap, it is clear that DeepSQLi is much less affected by the strengthened input validation in 5 out of 6 SUT, demonstrating its superior capability of revealing SQLi vulnerabilities in more complicated scenarios. This better result achieved by DeepSQLi can be attributed to the effective exploitation of the semantic knowledge learned from previous SQLi test cases.

Answer to RQ3: ***Under the advanced input validation, DeepSQLi leads to much better results than that of SQLmap, which can hardly find any vulnerability at all in a considerable number of runs. In general, DeepSQLi is much less affected by the Web applications with strengthened input validation.***

5 THREATS TO VALIDITY

As with any empirical study, the biases from the experiments can affect the conclusion drawn. As a result, we study and conclude this work with the following threats to validity in mind.

The metrics and evaluation method used is a typical example of the construct threats, which concern whether the metrics/evaluation method can reflect what we intend to measure. In our work, the metrics studied are widely used in this field of research [6, 30] and they serves as quality indicator for different aspects of SQLi testing. To mitigate the randomness introduced by the training, we repeat 20 experiment runs for each tool under a SUT. To thoroughly report our results without losing information, the distributions about the number of SQLi vulnerability found, which is the most important metric, have also been plotted in violin charts.

Internal threats are concerned with the degree of control on the studies, particularly related to the settings of the deep learning algorithm. In our work, the hyperparameters of Transformer are automatically tuned by using Adam and 10-fold cross validation, which is part of the training. The internal parameters of Adam itself were configured according to the suggestions from Vaswani et al. [32]. The crawler and proxy DeepSQLi are also selected based on their popularity, usefulness and simplicity.

External threats can be linked to the generalisation of our findings. To mitigate such, firstly, we compare DeepSQLi with a state-of-the-art tool, i.e., SQLmap. This is because SQLmap is the most cost-effective tool and has been widely used as a baseline benchmark [1, 3, 30]. Secondly, we study six real-world SUT that are widely used as standard benchmarks for SQLi testing research [14–16]. Despite that all the SUT are based on Java, they come with different scales, number of vulnerabilities and the characteristics, thus they are representatives of a wide spectrum of Web applications. In future work, we aim to evaluate DeepSQLi on other Web applications developed in different programming languages.

6 RELATED WORK

In order to detect and prevent SQLi attacks, different approaches have been proposed over the last decade, including anomalous SQL query matching, static analysis, dynamic analysis, *etc.*

Several approaches aim to parse or generate SQLi statements based on specific SQL syntax. For example, Halfond and Orso [14] proposed AMNESIA, a combination of dynamic and static analysis approach. At the static analysis stage, models of the legitimate queries that applications can generate is automatically built. In the dynamic analysis phase, AMNESIA uses runtime monitoring to check whether dynamically generated queries match the model. Mao et al. [8] presented an intention-oriented detection approach that converted SQL statement into a deterministic finite automaton and detect SQL statement to determine if the statement contains an attack. These aforementioned approaches, unlike DeepSQLi, heavily rely on fixed syntax or source code to estimate unknown attacks. In addition, they are not capable of learning the semantic knowledge from the SQL syntax.

Among other tools, BIOFUZZ[30] and μ4SQLi[2] are black-box and automated testing tools that bear some similarities to DeepSQLi. However, they have not made the working source code publicly available or the accessible code is severely out-of-date, thus we cannot compare them with DeepSQLi in our experiments. Instead, here we qualitatively compare them in light with the contributions of DeepSQLi:

- BIOFUZZ [30] is a search-based tool that generates test cases using context-free grammars based fitness function. However, as the fitness function is artificially designed based solely on prior knowledge, it is difficult to fully capture all possible semantic knowledge of SQLi attacks. This is what we seek to overcome with DeepSQLi. Further, the fact that BIOFUZZ relies on fixed grammars may also restrict its ability to generate semantically sophisticated test cases.
- μ4SQLi [2] is an automated testing tool that uses mutation operators to modify the test cases, with a hope of finding

more SQLi vulnerabilities. However, these mutation operators are designed with a set of fixed patterns, thus it is difficult to generate new attacks that have not been captured in the patterns. In DeepSQLi, we also design a few mutation operators, but they are solely used to enrich the training data, which would then be learned by the neural language model. In this way, DeepSQLi is able to create attacks that have not been captured by patterns in the training samples.

SQLMap is used as state-of-the-art in the experiments because it is a popular and actively maintained penetration SQLi testing tool, which has been extensively used in both academia [1, 3, 30] and industry [28]. Here we also make a qualitative comparison of differences between SQLMap and DeepSQLi.

- SQLMap relies on predefined syntax to generate test cases. Such practice, as discussed in the paper, cannot actively learn and search for new SQLi attacks, as the effectiveness entirely depends on the manually crafted rules, which may involve errors or negligence. On the other hand, DeepSQLi learns the semantics from SQL statements and test cases. Such a self-learning process allows it to generalize to previously unforeseen forms of attacks. Our experiments have revealed the superiority of DeepSQLi in detecting the SQLi vulnerabilities.
- SQLMap generates new test cases from scratch. DeepSQLi, in contrast, allows intermediately unsuccessful, yet more malicious test cases to be reused as the inputs to generate new one. This enables it to build more sophisticated test cases incrementally and is also one of the reasons that leads to a faster process of DeepSQLi over SQLMap.

Recently, the combination of machine learning and injection-based vulnerability prevention has become popular [4][27][27][3][29]. Among others, Kim et al. [20] used internal query trees from the log to train a SVM to classify whether an input is malicious. Sheykhkanloo et al. [27] trained a neural network with vectors which assigned for attacks to classify SQLi attacks. Appelt et al. [3] presented ML-Driven, an approach generates test cases with context-free grammars and train a random forest to detect SQLi vulnerability as the software runs. Jaroslaw et al. [29] applied neural networks to detect SQLi attacks. Their purpose is to build a model that learns the normal input and predicts whether the next input of user is malicious or not.

Unlike DeepSQLi, none of the work aims to generate SQLi test cases for conducting end-to-end testing on the Web application. Moreover, DeepSQLi leverages deep NLP to explicitly learn the semantic knowledge of the SQL for generating the whole sequence of SQLi test case. In particular, DeepSQLi translates the normal user inputs into test cases, which, when fail, would then be re-entered into DeepSQLi to generate more sophisticated SQLi attacks.

7 CONCLUSION AND FUTURE WORK

SQLi attack is one of the most devastating cyber-attacks, the detection of which is of high importance. This paper proposes DeepSQLi, a SQLi vulnerability detection tool that leverages on the semantic knowledge of SQL to generate test cases. In particular, DeepSQLi relies on the Transformer to build a neural language model under the Seq2Seq framework, which can be trained to explicitly learn the semantic knowledge of SQL statements. By comparing DeepSQLi with SQLmap on six real-world SUT, the results demonstrate the effectiveness of DeepSQLi and its remarkable improvement over the state-of-the-art tool, whilst still being effective on Web applications with *advanced* input validation.

In future work, we will extend DeepSQLi with incrementally updated neural language model by using the generated test cases as the testing runs. Moreover, expanding DeepSQLi to handle other vulnerabilities, e.g., Cross-site Scripting, is also within our ongoing research agenda.

ACKNOWLEDGMENT

Li was supported by UKRI Future Leaders Fellowship (Grant No. MR/S017062/1).

REFERENCES

[1] Dennis Appelt, Nadia Alshahwan, and Lionel C. Briand. 2013. Assessing the Impact of Firewalls and Database Proxies on SQL Injection Testing. In *FITTEST'13: Proc. Workshop of the 2013 Future Internet Testing - First International.* 32–47.
[2] Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. 2014. Automated testing for SQL injection vulnerabilities: an input mutation approach. In *ISSTA'14: Proc. of the 2014 International Symposium on Software Testing and Analysis.* 259–269.
[3] Dennis Appelt, Cu D. Nguyen, Annibale Panichella, and Lionel C. Briand. 2018. A Machine-Learning-Driven Evolutionary Approach for Testing Web Application Firewalls. *IEEE Trans. Reliability* 67, 3 (2018), 733–757.
[4] Davide Ariu, Igino Corona, Roberto Tronci, and Giorgio Giacinto. 2015. Machine Learning in Security Applications. *Trans. MLDM* 8, 1 (2015), 3–39.
[5] Ilies Benikhlef, Chenghong Wang, and Sangirov Gulomjon. 2016. Mutation based SQL injection test cases generation for the web based application vulnerability testing. In *ICENCE'16: Proc. of the 2nd International Conference on Electronics, Network and Computer Engineering.*
[6] Josip Bozic, Bernhard Garn, Dimitris E. Simos, and Franz Wotawa. 2015. Evaluation of the IPO-Family algorithms for test case generation in web security testing. In *ICST'15 Workshops: Proc. Workshop of the 2015 Eighth IEEE International Conference on Software Testing, Verification and Validation.* 1–10.
[7] Peter F. Brown, Stephen Della Pietra, Vincent J. Della Pietra, Jennifer C. Lai, and Robert L. Mercer. 1992. An Estimate of an Upper Bound for the Entropy of English. *Computational Linguistics* 18, 1 (1992), 31–40.
[8] Chenyu, Mao, Fan, and Guo. 2016. Defending SQL Injection Attacks based-on Intention-Oriented Detection. In *ICCSE'16: Proc. of the 11th International Conference on Computer Science & Education.* IEEE, 939–944.
[9] Mark Curphey and Rudolph Arawo. 2006. Web application security assessment tools. *IEEE Security & Privacy* 4, 4 (2006), 32–41.
[10] Linhao Dong, Shuang Xu, and Bo Xu. [n.d.]. Speech-Transformer: A No-Recurrence Sequence-to-Sequence Model for Speech Recognition. In *ICASSP'18: Proc. of the 2018 IEEE International Conference on Acoustics, Speech and Signal Processing.*
[11] Rohan Doshi, Noah Apthorpe, and Nick Feamster. 2018. Machine Learning DDoS Detection for Consumer Internet of Things Devices. In *SP Workshop'18: Proc. of the 2018 IEEE Security and Privacy.* 29–35.
[12] David Guthrie, Ben Allison, Wei Liu, Louise Guthrie, and Yorick Wilks. 2006. A Closer Look at Skip-gram Modelling. In *LREC'06: Proc. of the 5th International Conference on Language Resources and Evaluation.* 1222–1225.
[13] Halfond, William GJ, Choudhary, Shauvik Roy, Orso, and Alessandro. 2009. Penetration testing with improved input vector identification. In *ICST'09: Proc. of the 2nd International Conference on Software Testing Verification and Validation.* 346–355.
[14] William G. J. Halfond and Alessandro Orso. 2005. AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks. In *ASE'05: Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering.* 174–183.
[15] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. 2006. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *SIGSOFT'06: Proc. of the 14th ACM International Symposium on Foundations of Software Engineering.* 175–185.
[16] William G. J. Halfond, Alessandro Orso, and Pete Manolios. 2008. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Trans. Software Eng.* 34, 1 (2008), 65–81.
[17] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Ian Simon, Curtis Hawthorne, Noam Shazeer, Andrew M. Dai, Matthew D. Hoffman, Monica Dinculescu, and Douglas Eck. 2019. Music Transformer: Generating Music with

Long-Term Structure. In *ICLR'19: Proc. of the 7th International Conference on Learning Representations*.

[18] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. 2014. A Convolutional Neural Network for Modelling Sentences. In *ACL'14: Proc. of the 52nd Association for Computational Linguistics*. 655–665.

[19] Adam Kiezun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. 2009. Automatic creation of SQL Injection and cross-site scripting attacks. In *ICSE'09: Proc. of the 31st International Conference on Software Engineering*. 199–209.

[20] Mi-Yeon Kim and Dong Hoon Lee. 2014. Data-mining based SQL injection attack detection using internal query trees. *Expert Syst. Appl.* 41, 11 (2014), 5416–5430.

[21] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR'15: Proc. of the 52nd Association for Computational Linguistics*.

[22] Huichen Li, Xiaojun Xu, Chang Liu, Teng Ren, Kun Wu, Xuezhi Cao, Weinan Zhang, Yong Yu, and Dawn Song. 2018. A Machine Learning Approach to Prevent Malicious Calls over Telephony Networks. In *SP'18: Proc. of the 2018 IEEE Symposium on Security and Privacy*. 53–69.

[23] Ofer Maor and Amichai Shulman. 2004. SQL injection signatures evasion. *Imperva, Inc., Apr* (2004).

[24] Stuart McDonald. 2002. SQL Injection: Modes of attack, defense, and why it matters. *White paper, GovernmentSecurity. org* (2002).

[25] Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. 2014. Recurrent Models of Visual Attention. In *NIPS'14: Proc. of the 2014 Neural Information Processing Systems*. 2204–2212.

[26] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *PLDI'14: Proc. of the 2014 Programming Language Design and Implementation*. 419–428.

[27] Naghmeh Moradpoor Sheykhkanloo. 2017. A Learning-based Neural Network Model for the Detection and Classification of SQL Injection Attacks. *IJCWT* 7, 2 (2017), 16–41.

[28] Sanjib Sinha. 2018. SQL Mapping. In *Beginning Ethical Hacking with Kali Linux*. Springer, 221–258.

[29] Jaroslaw Skaruz and Franciszek Seredynski. 2007. Recurrent neural networks towards detection of SQL attacks. In *IPDPS'07: Proc. of the 21th International Parallel and Distributed Processing Symposium*. 1–8.

[30] Julian Thomé, Alessandra Gorla, and Andreas Zeller. 2014. Search-based security testing of web applications. In *SBST'14: Proc. of the 7th International Workshop on Search-Based Software Testing*. 5–14.

[31] Wei Tian, Jufeng Yang, Jing Xu, and Guannan Si. 2012. Attack Model Based Penetration Test for SQL Injection Vulnerability. In *COMPSAC'12: Proc. Workshops of the 36th Annual IEEE Computer Software and Applications*. 589–594.

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS'17: Proc. of the 2017 Neural Information Processing Systems*. 5998–6008.

[33] Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey E. Hinton. 2015. Grammar as a Foreign Language. In *NIPS'15: Proc. of the 2015 Neural Information Processing Systems*. 2773–2781.