DaNuoYi: Evolutionary Multi-Task Injection Testing on Web Application Firewalls *

Ke Li[1], Heng Yang[1] and Willem Visser[2]

[1]Department of Computer Science, University of Exeter, EX4 4QF, Exeter, UK
[2]Amazon Web Service, USA
*Email: k.li@exeter.ac.uk

Abstract: Web application firewall (WAF) plays an integral role nowadays to protect web applications from various malicious injection attacks such as SQL injection, XML injection, and PHP injection, to name a few. However, given the evolving sophistication of injection attacks and the increasing complexity of tuning a WAF, it is challenging to ensure that the WAF is free of injection vulnerabilities such that it will block all malicious injection attacks without wrongly affecting the legitimate message. Automatically testing the WAF is, therefore, a timely and essential task. In this paper, we propose DaNuoYi, an automatic injection testing tool that simultaneously generates test inputs for multiple types of injection attacks on a WAF. Our basic idea derives from the cross-lingual translation in the natural language processing domain. In particular, test inputs for different types of injection attacks are syntactically different but may be semantically similar. Sharing semantic knowledge across multiple programming languages can thus stimulate the generation of more sophisticated test inputs and discovering injection vulnerabilities of the WAF that are otherwise difficult to find. To this end, in DaNuoYi, we train several injection translation models by using multi-task learning that translates the test inputs between any pair of injection attacks. The model is then used by a novel multi-task evolutionary algorithm to co-evolve test inputs for different types of injection attacks facilitated by a shared mating pool and domain-specific mutation operators at each generation. We conduct experiments on three real-world open-source WAFs and six types of injection attacks, the results reveal that DaNuoYi generates up to 3.8× and 5.78× more valid test inputs (i.e., bypassing the underlying WAF) than its state-of-the-art single-task counterparts and the context-free grammar-based injection construction.

Keywords: Web application firewall, security testing, injection testing, multi-tasking, search-based software engineering.

1 Introduction

Due to the maturity of the state-of-the-art web technologies and advancements of Internet of things, web applications have become increasingly ubiquitous and important for enterprises and individuals from various sectors, such as online shopping, e-banking, healthcare, e-governance and social media. Yet, the prevalence of web applications inevitably make them one of the main targets of malicious attacks. For example, Beery and Niv [1] reported that each web application worldwide can experience around 173 injection attacks per month on average. According to a recent report published by Open Web Application Security Project (OWASP)[1], injection attack is one of the most common way to compromise a web application and its data.

*This manuscript is accepted for publication in the IEEE Transactions on Software Engineering.
[1]https://owasp.org/Top10/

Table 1: Examples of three semantically similar but syntactically different types of injection attacks

SQLi	vulnerable code	`SELECT * FROM users WHERE username='$name' and password='$pass';`
	injection attack	`SELECT * FROM users WHERE username='' OR '1'='1'; --' and password='abc';`
XMLi (XPathi)	vulnerable code	`users[username/text()='$name' and password/text()='$pass']`
	injection attack	`users[username/text()='' OR '1'='1 (:' and password/text()=':) ']`
PHPi	vulnerable code	`if (eval("return '$storedName' === '$name' && '$storedPass' === '$pass';"))`
	injection attack	`return $storedName === '' || '1'=='1'; //' && $storedPass === 'abc';`

The blue/bold texts are the inputs from the user, where **$name** and **$pass** denote the variable of the username and password given by an user, respectively. The highlighted texts are what will be commented out by the injection attacks.

An unified and state-of-the-practice solution to injection attacks is the use of a web application firewall (WAF) [2], which is a special type of application firewall that has been widely adopted to provide protection of web applications from various malicious injection attacks. It is usually deployed as a facade between the client and web application server aiming to analyze all HTTP messages, which contain potentially malicious user inputs, sent to the web application — detecting, filtering and blocking anything malicious through a set of rules. As such, a WAF is application-independent and is designed to prevent any type of injection attack in mind. By a type of injection attack, we refer to the attack that specifically seeks to inject malicious code into a particular programming language used in a web application, such as SQL injection (SQLi), XML injection (XMLi), and PHP injection (PHPi).

Given the fast-moving nature of web applications, injection attacks relentlessly emerge all the time and are grown with an evolving sophistication. As a result, fine-tuning the rules in a WAF is a complex, labor-intensive and costly task especially in the presence of multiple types of injection attack. Note that a reliable WAF not only needs to be able to detect the increasingly sophisticated injection attacks but also to avoid mistakenly blocking legitimate HTTP messages. As such, this leaves the WAF with a great chance to suffer from various injection vulnerabilities in practice.

To verify whether a WAF is tuned to be sufficiently secured before its production deployment, a variety of testing techniques has been proposed for generating test inputs to the WAF, including white-box testing [3], static analysis [4], model-based testing [5] and black-box testing [6]. However, none of these techniques are perfect because they are either less applicable in practice or inadequate for vulnerability detection. For example, both white-box testing methods and static analysis tools require full control of source codes, the access of which is difficult, if not impossible, in web applications and their WAFs due to the heterogeneous programming environments [7]. Henceforth, it is not difficult to understand that the detection capability of white-box testing is limited. As for the model-based testing techniques, neither developing models expressing the security policies nor constructing the implementation of WAFs and the web applications is easily accessible. Although black-box testing tools [2, 8, 9], mostly fuzz testing, do not require the access of source code, they often focus on the syntax of attacks yet ignoring the semantic information, which could restrict their testing capability.

A major bottleneck of the existing black-box testing tools for WAF is the lack of support for testing more than one injection attack simultaneously. This essentially contradicts the design principle of a WAF, which aims to serve as a universal filter that works independently of the programming language(s) that underpins a web application. Beside the impact on the applicability of the tools, such a limitation also shuts the gate towards achieving more effective injection testing. This is because, similar to human natural languages, malicious injection attacks of different programming languages may impose different syntax but do have some unique semantics in common. As such, analogous to cross-lingual translation, they can share certain common ground between them, which could generate more sophisticated test inputs to discover injection vulnerabilities that are otherwise difficult to find. Let us consider the examples shown in Table 1 which gives three types of injection attack, i.e., SQLi, XMLi (XPath injection) and PHPi. Clearly, all these injection attacks contain malicious and syntactically different inputs, i.e., the `'1'='1'`,

`OR`, and `--` for SQLi; the `'1'='1'`, `OR`, and `(::)` for XMLi; and the `'1'=='1'`, `||`, and `//` for PHPi. However, the three injection attacks are also semantically similar in the sense that they all aim to 'fool' the WAF and the web application by creating a tautology (i.e., the string `'1'` is always the same) and commenting out parts of the original command fragments. This constitutes the key motivation behind this work.

To overcome the aforementioned limitations, this paper proposes an evolutionary multi-task injection testing fuzzer for WAF, dubbed DaNuoYi[2], which automatically and simultaneously tests multiple types of injection attacks. Specifically, DaNuoYi uses the language models (i.e., Word2Vec, see Section 3.2.1) to capture the semantic information of each type of injection attack. In particular, such semantic information is shared across different injection types during the training process thus to facilitate the translation of a test input from one type (e.g., SQLi) into another (e.g., PHPi) in a multi-task learning manner. Note that the learned translation models thereafter serve as the 'bridges' between different injection testing tasks during the test input generation driven by a novel multi-task evolutionary algorithm. By doing so, DaNuoYi is empowered to improve the test input generation of one injection type by borrowing the promising test inputs generated for all other types considered.

Contributions. In a nutshell, our contributions include:

- DaNuoYi is a fully automatic, end-to-end tool that can simultaneously test any type of injection attack on a variety of WAFs. To the best of our knowledge, this is the first tool of its kind that can automatically and simultaneously generate test inputs from multiple types of injection attacks for testing a WAF.
- Surrogate classifiers based on neural networks are developed to embed the semantic information of injection contexts into a vector representation that predicts the likelihood of bypassing the underlying WAF.
- A multi-task translation (i.e., multi-task injection translation) framework trained by an encoder-decoder architecture from the natural language processing (NLP) domain. Based on this framework, a test input for one type of injection attack can be translated into a syntactically different, but semantically similar one for another type.
- A multi-task evolutionary algorithm (MTEA), empowered by the surrogate classifiers, the multi-task translation module, and six tailored mutation operators, co-evolves test inputs for different types of injection attacks.
- A quantitative and qualitative analysis[3] on three real-world WAFs (i.e., `ModSecurity`, `Ngx-Lua-WAF`, `Lua-Resty-WAF`, see 4.1.2 for details), three alternative classifiers, and six types of injection attacks including SQLi, XMLi, PHPi, HTML injection (HTMLi), OS shell script injection (OSi), cross-site script injection (XSSi). Empirical results fully demonstrate the superiority of DaNuoYi for disclosing more injection vulnerabilities for WAF (up to 3.8× more bypassing injection cases) comparing to the single-task counterparts.

Novelty. What makes DaNuoYi *unique* are:

- It is able to learn the common semantic information from syntactically different test inputs for different types of injection attacks. This is achieved by the translation model that mimics the cross-lingual translation between natural languages.

[2] DaNuoYi is originally from *Heavenly Sword and Dragon Slaying Sabre*, a famous knights-errant novel by Jin Yong (Louis Cha). Its full name is `Qiankun Danuoyi` and it one of the most premium Kongfu which is able to leverage and mimic different types of Kongfu power from the enemies to attack them back harder — this is similar to what our approach can do in imitating different types of injection attack to test the WAF.

[3] The source code and data related to this paper can be found from our lab repository: https://github.com/COLA-Laboratory/DaNuoYi

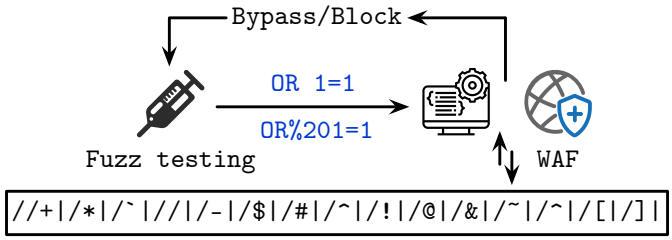


Figure 1: Illustrative example of injection testing for WAF.

- It is capable of generating test inputs for any type of injection attacks with different syntax and exploit the most promising test inputs interchangeably. This is realized by a MTEA that co-evolves multiple test input generation processes simultaneously.

The rest of this paper is organized as follows. Section 2 provides some background knowledge related to the injection testing on WAF and the neural language model for injection testing. Section 3 delineates the technical details of DaNuoYi while its effectiveness is quantitatively and qualitatively analyzed on three open-source real-world WAFs as in Section 4. At the end, Sections 5 and 6 discuss the threats to validity and related work respectively while Section 7 concludes this paper and sheds some lights on future directions.

2 Preliminaries

2.1 Injection Testing on WAF

In the development of modern web applications, a WAF serves as the first entity to filter malicious injection attacks coming to the underlying application. Taking the most commonly used signature-based WAFs as an example, this is achieved by setting some rules, e.g., the regular expression, that detects critical words in the user inputs embedded in a HTTP request it receives.

Fig. 1 gives an intuitive example of a typical testing process on a WAF. To test a particular type of injection, say SQLi, one usually relies on fuzzing that randomly generates test inputs for the WAF. In practice, a test input is a failed attack in case it is blocked by the WAF; otherwise it bypasses the WAF thus indicating a SQLi vulnerability under the current WAF setting. For example, the expression shown in Fig. 1 can block any input that contains `+*‘/-$#^!@&~`. Therefore, the test input `OR 1=1` will be blocked. However, some other ones, e.g., `OR%201=1`, can bypass the WAF. Note that, according to Demetrio et al. [10] and Appelt et al. [2], the test inputs that bypass the WAF are always considered to be malicious, as they can either successfully inject the web application behind or would provide necessary information that help to eventually achieve so (e.g., allowing the attackers to know which inputs have been blocked or not — a typical blind attack).

Although a HTTP message may contain multiple user inputs (e.g., a login form needs both username and password), the entire message is compromised as long as one of the malicious user inputs can bypass the WAF. Therefore, automatically testing WAF often focus on generating a single test input for the WAF [2]. Further, it can be easily adopted to other cases where multiple user inputs are required by combining different generated test inputs together.

2.2 Feature Embedding for Injection Testing

A successful test input that discovers injection vulnerabilities needs to comply with the syntax of the underlying injection type. Our recent study [11] demonstrated that a programming language bears many

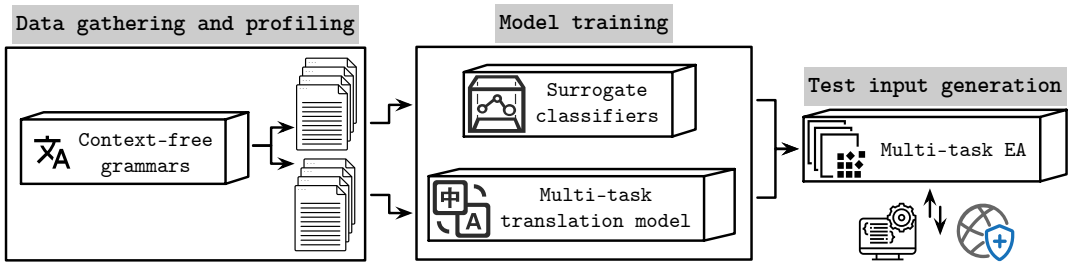


Figure 2: System architecture and workflow of DaNuoYi.

similarities with natural languages. Henceforth, leveraging the semantic information embedded in injection attacks can facilitate the automated test case generation accordingly.

To exploit the semantic information of a test input, the first step is to embed its words into a measurable representation. Its basic idea is to encode each word separated by blank, through a vocabulary, into a fixed length vector with the same dimension as the number of words in the test input. Taking SQLi as an example, the test input `' OR '1'='1'` will be tokenized into {`'`, `OR`, `'`, `1`, `'`, `=`, `'`, `1`,`'`}. After distinguished encoding, `'`$= (1,0,0,0)$, `OR`$= (0,1,0,0)$, `1` $= (0,0,1,0)$, and $== (0,0,0,1)$. These vectors are then further embedded by a neural network, namely a neural language model (NLM), into another $d \geq 1$ dimensional vector (d is a predefined hyper-parameter) to take additional contextual information into account (see Section 3.2.1). In particular, each dimension partially contributes to the meaning of a word. For example, when $d = 5$, we may have `OR`$= (32, 19, 3, 81, 22)$. The word vectors would then be further extracted, when needed, to better understand their semantic information in the translation, for which we will elaborate in Section 3.3.2.

3 Multi-task Injection Testing with DaNuoYi

DaNuoYi is designed as an end-to-end fuzzing tool to automate the test input generation for detecting multiple types of injection vulnerabilities. Its overarching hypothesis is that test inputs from syntactically different types of injection attacks share certain latent semantic similarities that can be useful to generate sophisticated test inputs for each other. As shown in Fig. 2, the main workflow in DaNuoYi consists of four components, each of which is outlined as follows.

- *Data gathering and profiling:* DaNuoYi assumes a context-free grammar (CFG) for each type of injection attack, based on which test inputs can be generated to profile the WAF.
- *Surrogate classifiers:* To better distinguish the 'good' test inputs from the 'bad' ones for a WAF, in DaNuoYi we build a classifier for each type of injection attack, by using the data generated by the CFG, to estimate the likelihood of a test input bypassing the WAF. Note that this likelihood is used to evaluate the fitness of a candidate test input generated by the MTEA.
- *Multi-task translation model:* We develop a multi-task injection translation paradigm to bridge the test input generation across different types of injection attack. For any pair of injection types, we build a translation model that translates the test input from one type of injection attack into a semantically related one. Note that all these translation models are trained using the data also generated by the CFG and their parameters are shared during the training process, thus we can expect to improve the effectiveness of the translated test inputs.
- *Multi-task evolutionary algorithm:* To generate test inputs for multiple types of injection attack simultaneously, we develop a MTEA to continuously evolve test inputs towards successful injections.

Table 2: Characteristics of various CFGs used in DaNuoYi including the number of productions (Prod.), terminals (Term.), nonterminals (Nonterm.), recursive productions (Rec.), unproductive symbols (Unprod.), and inaccessible symbols (Inacc.).

Injection	Prod.	Term.	Nonterm.	Rec.	Unprod.	Inacc.
SQLi	49	102	26	0	0	3
XSSi	58	404	47	0	0	1
PHPi	51	144	49	0	0	26
OSi	7	41	7	0	0	2
XMLi	49	102	26	0	0	3
HTMLi	66	141	55	0	0	19

The initial test inputs are seeded according to the CFG. During each evolutionary iteration, promising test inputs generated for one type of injection attack are shared across all other types of injection attack by using the multi-task translation models. Only the most promising test inputs, which are highly likely to achieve successful injections, can survive to the next iteration.

In the following paragraphs, we elaborate the implementation detail of each component.

3.1 Data Gathering and Profiling

3.1.1 Context-free Grammar for Injection Searching

Data is arguably the *fountain of life* for any machine learning task. In DaNuoYi, we take advantage of the CFG for each type of injection attack to automatically generate some initial test inputs, called rule-based injection search method. Specifically, we define a CFG as a tuple $\mathcal{G} = \langle \mathcal{V}, \Sigma, \mathcal{R}, \mathcal{S} \rangle$ while the meaning of each element is delineated as follows.

- $\mathcal{V}$ is a set of symbols used to represent non-terminal entities in the language, also known as variables.
- Σ consists of a finite number of terminal symbols that are separate from the set $\mathcal{V}$ and collectively form the content of the strings that belong to the language.
- The set of production rules $\mathcal{R}$ consists of a finite number of rules, each comprising a single variable and a string of variables and terminals. These rules define the way in which variables can be replaced by other variables or terminals. A grammar may have multiple production rules for a given variable, and the entire set of rules is known as the grammar's production system.
- The start symbol $\mathcal{S}$ is a special non-terminal symbol that represents the entire string generated by the grammar.

Table 2 summarizes the statistics of the selected characteristics of the CFG used in this work, while Algorithm ?? provides the pseudo-code for the CFG-based injection generation. Note that for the types of injection attack considered in this work, the grammar for SQLi is derived from those defined in [2] while the others are developed by ourselves according to a systematic summary of the literature [12–14] and several well known open-source payload[4]. In practice, the CFG for any arbitrary type of injection attack can be created in a similar way or there are readily available ones to use, such as those generated in this work. We reported the numbers of unproductive symbols and unreachable symbols. These symbols do not work in the test inputs generation process. The context-free grammars used to generate different types of test inputs are for proof-of-concept purposes only currently.

[4] MCIR, Commodity-Injection-Signatures, xxe-injection-payload-list

Algorithm 1: Injection Generation Algorithm Using Context-Free Grammar

Input: A CFG $\mathcal{G} = \langle \mathcal{V}, \Sigma, \mathcal{R}, \mathcal{S} \rangle$ for the target injection task
Output: $\mathcal{I}$: An injection instance

```
1  if s == Null then
2      s ← S
3   /* Initialize the result injection instance                    */
4  I ← ∅
5   /* Set the start symbol as the current symbol                  */
6  while s ∈ V do
7      /* Randomly select a production rule for s                  */
8      rule ← SELECTRANDOMRULE(s, R)
9      foreach symbol in rule do
10         if symbol ∈ Σ then
11             /* Append the terminal symbol to the injection instance */
12             I ← I + symbol
13         else if symbol ∈ V then
14             /* Recursively expand non-terminal symbols          */
15             I ← I+GENEARTETESTINPUT(G, symbol)
16 return I
```

To have an intuitive illustration of the CFG and its corresponding test input generation mechanism, Fig. 3 gives a sample grammar of XSSi while the complete grammar can be found from our supplementary document[5]. In Fig. 3, '→' represents production; ',' represents connector; '|' represents replacement sign. In order to generate a test input, we start from the root and the branches of the grammar tree that are generated according to a some predefined rules of the grammar in a random manner till the leaf node is reached. At the end, the combination of leaf nodes constitutes a test input. It is worth noting that the grammar is generic. Let us look at the test input example shown in Fig. 3 again. `%0A%53r%43=javascript:alert(1)%09` gives a popup attack by using `alert(1)`. In particular, `alert(1)` is derived from a non-leaf node `jsString`. Therefore, the other variants of `alert(1)` can also be included in the generator of `jsString`. Note that a simple alternation of content like `alert(2)` does not come up to a feasible variant while only some semantically similar variants (e.g., `%61%6c%65%72%74%28%31%29,&#x61`) count.

3.1.2 Datasets Construction

Based on the test inputs generated from the CFG of all the types of injection attack considered in DaNuoYi, we can build the dataset for training our surrogate classifiers and the multi-task translation model as well as the initial seeds for the test input generation in MTEA thereafter. Note that it is not uncommon that there is a readily available dataset for injection testing of a given type of injection attack from past releases[6], which can be of great help to enrich the dataset. To demonstrate a wide applicability of DaNuoYi, this paper assumes that there is no readily available dataset.

Note that although using the CFG alone can search for injection cases, its capability is rather limited. According to our preliminary experiments, the number of bypassing injection cases, as well as the diversity, generated by the CFG alone is not sufficient to train a capable surrogate classifier and neural

[5] https://tinyurl.com/3cu4ahrb
[6] https://github.com/payloadbox

```
'root': (('directContext')|('attributeContext')|('eventContext')
'attributeContext': ('terDQuote', 'wsp', 'srcAttr', 'terEqual', 'jsStatement',
'wsp')|
                    ('pre', 'terLess', 'aTag', 'wsp', 'herfAttr',
'terEqual', 'jsStatement','wsp', 'terGreater', 'rightALabel')|
                    (‘pre', 'terLess', 'iframeTag', 'wsp', 'srcAttr', 'terEqual',
'jsStatement','wsp', 'terGreater')|
                    ('pre', 'terLess', 'scriptTag', 'wsp', 'srcAttr', 'terEqual',
'jsFile','wsp', 'terGreater', 'rightSciptLabel')
'jsString': ('alert(1)')|
            ('%61%6c%65%72%74%28%31%29')|
            ('&#x61;&#6c;&#x65;&#x72;&#x74;&#x28;&#x31;&#x29;')|
            ('&#97;&#108;&#101;&#114;&#116&#40;&#57;&#41;&#59')
```

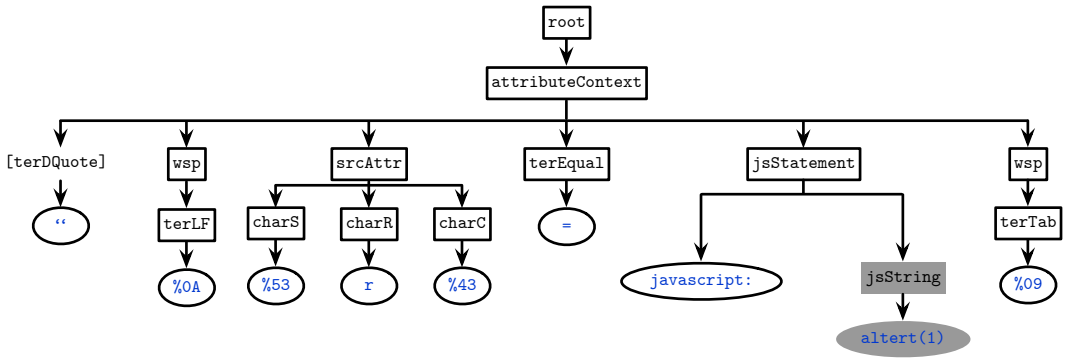


Figure 3: An illustrative example of the grammar tree for XSSi.

translation models, even when being allocated with a sufficiently large amount of computational budget. In addition, it highly depends on the type of injection attack. In contrast, DaNuoYi has shown an outstanding performance for injection case generation in terms of both successful rate and the diversity.

3.2 Surrogate Classifiers Learning with Word Embedding

For most existing WAFs, a test input can only result in a binary outcome, i.e., either pass or fail. This does not provide sufficient information to evaluate the effectiveness of a test input and can, even worse, mislead the search-based test input generation in DaNuoYi. To address this problem, we train a surrogate classifier for each type of injection attack by using the data collected in Section 3.1.1 as a priori. As reported in a recent study [11], semantic information of test inputs can significantly improve the injection testing. In this work, we treat the test inputs of a type of injection attack akin to sentences from a natural language. Then, we leverage techniques from the NLP domain to build the language model for each type of injection attack. In the following paragraphs, we first introduce the language model used in DaNuoYi and then describe the mechanism of the surrogate classifier.

3.2.1 Word Embedding

As mentioned in Section 2.2, to better handle the semantic knowledge in the test inputs, the first step is to convert a sequence of words belonged to a test input into a word vector and this is also critical for DaNuoYi to understand the semantics embedded in test inputs. For example, the language model needs to be able to understand that ‘+’, ‘/**/’, ‘%20’, ‘%09’ are synonyms for the blank character in a test input. They merely represent different forms that disguise the attacks.

More specifically, we apply the classic Word2vec in DaNuoYi to train a NLM that converts a test input into the corresponding word vectors. In particular, we use the continuous bag of words (CBOW) model [15], which is a context aware version of Word2vec, to identify similar words in test inputs based on the contextual information, as the meaning of each single word in a test input vary depending on the context. Suppose that w_i is the i-th word in a test input, the CBOW model correlates a target word w_i

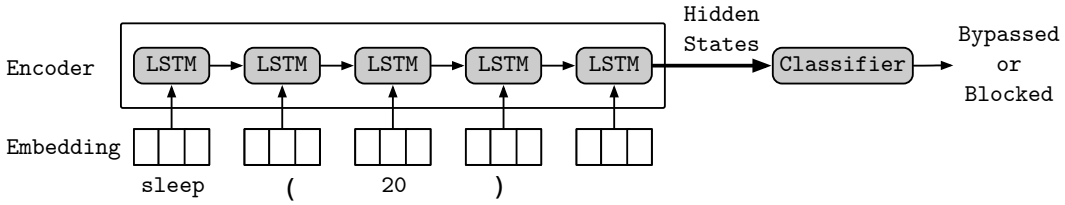


Figure 4: The neural architecture of the surrogate classifier based on LSTM. Note that LSTM can be replaced by GRU and RNN.

and its context words under a given window size. For example, if the window size is two, the context words are $(w_{i-2}, w_{i-1}, w_{i+1}, w_{i+2})$. Note that the context information have shown to be promising on providing more accurate vector embedding of the test inputs [11].

3.2.2 Surrogate Classifier

There are many classifiers [16, 17] available for predicting whether an injection test input can bypass the WAF or not. In this work, we choose three neural network models, i.e., long-short term memory (LSTM) [18] network, recurrent neural networks (RNN) [19] and gated recurrent unit (GRU) [20] to serve our purpose. There are four major reasons.

- They have been reported to be highly effective in handling the semantics of languages, i.e., they work well with the vector embedding of words [21, 22].
- They have been widely used in prior works [23–25].
- It is plausible to build a large set of data samples since the possible number of test inputs for injection testing is enormously high.
- They only need to be trained once at the beginning and thus the training overhead is acceptable.

Using LSTM as an example, the neural architecture of our surrogate classifier is given in Fig. 4. In practice, the classifier[7] takes the vector embedding of a test input as input and predicts a probability that decides whether this test input can bypass the WAF or not. In DaNuoYi, we leverage this output probability as the fitness function to guide the search-based test input generation.

Remark 1. *Our proposed fitness assignment approach utilizing surrogate classifiers offers multiple advantages:*

- *First, it allows the EA to allocate a fitness value to every input, even if the input is blocked by the WAF.*
- *Second, the employment of surrogate classifiers decreases the amount of queries made to the SUT, thereby enhancing the efficiency of the EA.*
- *Last but not the least, surrogate classifiers are capable of revealing the distinct characteristics of successful injection attacks, which can contribute to the development of more effective test input designs.*

[7] It is worth noting that the validation accuracy of surrogate classifier is approximately range from 90 to 99% in our offline experiments. Based on context-free grammar developed in Section 3.1.1, our training dataset consists of diverse types of test inputs. These are adequate to support the effectiveness of surrogate classifier.

Remark 2. *In the literature of search-based test input generation, there have been various fitness assignment approaches proposed to guide the search process. For example, Poulding et al. [26] proposed a coverage probability-based fitness assignment that estimates the minimum coverage probability induced by a candidate input profile. Unfortunately, this approach may not accurately represent the behavior of the system under test (SUT) due to its reliance on a finite set of inputs. In [27], Kifetew et al. proposed a fine-grained fitness measurement based on branch distances [28] that span multiple classes of the SUT. Havrikov et al. [29] introduce k-path coverage as a measure of input coverage that considers the coverage of individual syntactic elements and their combinations up to a given depth k. Compared to coverage probability-based fitness assignment, surrogate classifiers offer a more efficient and precise method for predicting if an injection can bypass WAFs by executing a set of inputs on the SUT and observing the coverage achieved. Besides, another idea [30] is to use a CFG to parse a set of sample input files representing common program usage and determine probabilities for individual grammar productions during parsing. Although this method aims to generate uncommon test inputs, it is not suitable for use as a fitness measurement for EAs.*

3.3 Multi-task Translation Framework

In DaNuoYi, the translation between the test inputs for different types of injection attack is conceptually similar to the cross-lingual translation in NLP. This is the foundation of knowledge transfer in DaNuoYi. It aims to translate a high quality test input for one type of injection attack to a semantically related and meaningful test input for another type. Such translation enables DaNuoYi to share and unify knowledge among different types of injection attack. More specifically, there are two main steps as follows.

3.3.1 Data Preprocessing

Generally speaking, the input of our multi-task translation model is a test input for one type of injection attack, while the output is the 'translated' test input for another type. In this case, a data instance is a pair of semantically related test inputs from two types of injection attack, and the translation is asymmetric. Henceforth, when there are, say, six different types of injection attack, we need to prepare $\binom{6}{2} = 15$ pairs of datasets and models to cover all bidirectional translations.

Given a pair of datasets for two types of injection attack, it is challenging to evaluate the semantic similarity between two test inputs. In DaNuoYi, we use the latent semantic indexing model (provided by Gensim[8]) to measure the similarity of semantic correlation between test inputs from two different types of injection attack. By doing so, any test input for one type of injection attack is paired with a (most similar) test input for another type. Thereafter, the paired data is stored into a dataset.

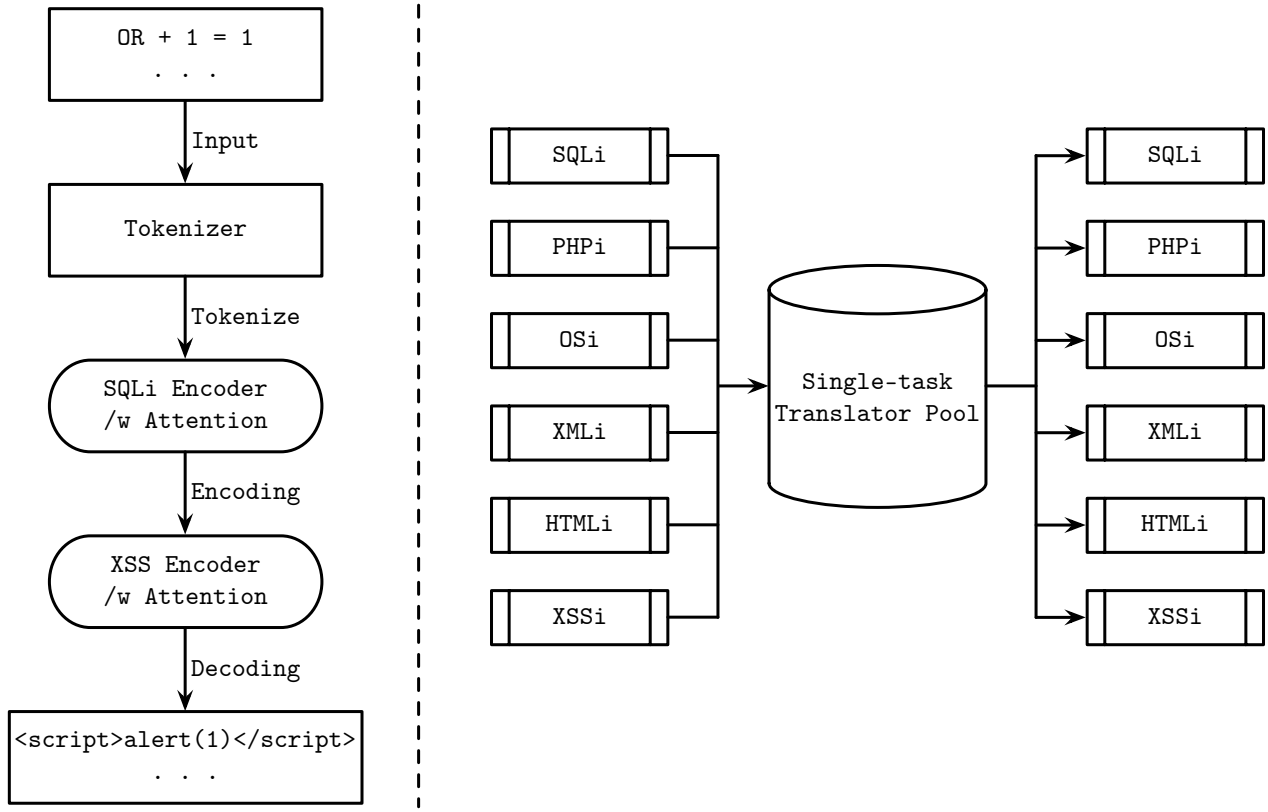
3.3.2 Multi-Task Injection Translation

To capture the semantic information of a type of injection attack, DaNuoYi chooses Seq2Seq [31], a LSTM-based Seq2Seq translation model[9] to serve the purpose. Note that Seq2Seq has been widely used in many downstream NLP tasks including machine translation [31], text summarization [32], and question answering [33,34], to name a few. Seq2Seq is able to transform a sequence of words into another relevant sequence, thus enabling the translation of test inputs from one type of injection attack into another. Let us consider the example shown in Fig. 5(a), the incoming test input is `OR+1=1` while its corresponding outcome could be `<scrpit>alert(1)</script>`.

As shown in Fig. 5(a), Seq2Seq model consists of two key components, i.e., attention-based `encoder` and `decoder`, each of which can be regarded as an independent LSTM model [19]. The inputs of the

[8] https://radimrehurek.com/gensim

[9] Our translation models are built upon OpenNMT-Py, a popular open source neural machine translation system.



(a) Single-task injection translator based on Seq2Seq

(b) Multi-task injection translator architecture

Figure 5: Illustrative example of the Seq2Seq framework for test input translations.

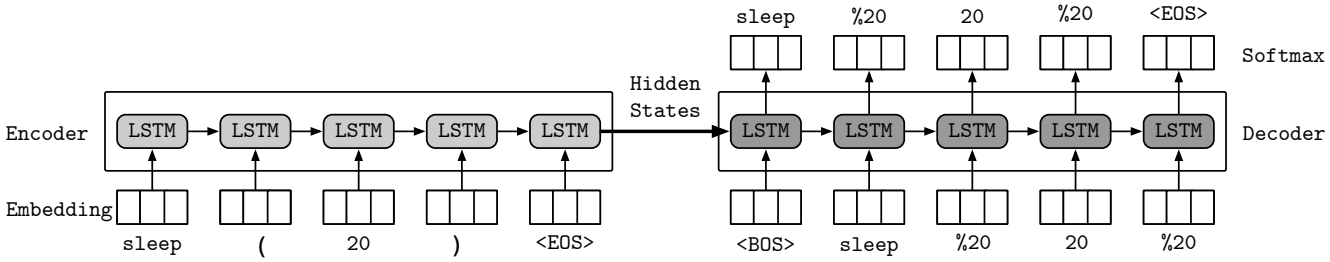


Figure 6: An illustration of the architecture of our Seq2Seq model for multi-task injection translation. ⟨BOS⟩ and ⟨EOS⟩ are used to indicate the beginning and end of a sequence, respectively. Note that the decoder uses the previously decoded token and hidden state as supplementary inputs for the next time step. Please refer to https://opennmt.net for more details about translation model.

encoder are a sequence of word vectors $X = \{\mathbf{x}_1, \cdots, \mathbf{x}_s\}$ while the outputs of the decoder are another sequence of word vectors $Y = \{\mathbf{y}_1, \cdots, \mathbf{y}_p\}$. In particular, s and p respectively indicates the number of characters of the input and output sequences.

In a nutshell, Seq2Seq model aims to learn the following conditional distribution based on which the prediction of a sequence can be made:

$$p\left(\mathbf{y_1}, \cdots, \mathbf{y_p} | \mathbf{x_1}, \cdots, \mathbf{x_s}\right). \tag{1}$$

3.4 Multi-Task Evolutionary Test Input Generation

Specifically, each word of the target sequence is conditioned on the following conditional probability:

$$p\left(\mathbf{y}_{t+1}|\mathbf{y}_1,\cdots,\mathbf{y}_t\right)=g\left(\mathbf{h}_{y_t},\mathbf{c}_{t+1},\mathbf{y}_t\right). \tag{2}$$

Given the input sequence $\{\mathbf{x_1},\cdots,\mathbf{x_s}\}$, we can calculate and update the hidden states of the `encoder` as $\{\mathbf{h_{x_1}},\cdots,\mathbf{h_{x_s}}\}$. Then, we use the attention mechanism [31] to calculate the context vector $C=\{\mathbf{c}_1,\cdots,\mathbf{c}_p\}$, where $\mathbf{c}_i=\sum_{t=1}^{s}a_{i_t}\mathbf{h}_t$ and weight a_{i_t} enables the `Seq2Seq` to focus on different parts of the input sequence when predicting the word vector $\mathbf{y}_i$. At the end, the `decoder` sequentially predicts the target word by using the hidden states and the context vector C which also serves as the initial hidden state of the `decoder`. Note that the hidden states will be updated as words being generated. This process stops when the last word in the sequence is predicted. To help understand the multi-task injection translation model, we also provide the architecture of the `Seq2Seq` model in Fig. 6.

In `DaNuoYi`, a test input from one type of injection attack can in principle be translated into any other type. This cross-lingual translation is implemented by a multi-task learning paradigm, as shown in Fig. 5(b), where we train the pair-wise translators (i.e., the single-task translator based on `Seq2Seq` as shown in Fig. 5(a)) for any two of the injection types. Our preliminary experiments show that the single-task translator outperforms the parallel multi-task translation architecture. This structure enables the semantic information to be shared between the test input pairs for any two types of injection attack during the training process. Thus, it can produce more effective translation models.

Remark 3. *In our context, the term "semantic" refers to the intended meaning or purpose of a payload or query in a web application. Note that injection attacks that modify the semantics of a query or command can have serious consequences, such as allowing an attacker to access sensitive data or perform unauthorised actions.*

3.4 Multi-Task Evolutionary Test Input Generation

In `DaNuoYi`, we develop an MTEA, extended from the classic single-task $(\mu+\lambda)$[10] evolutionary algorithm where $\mu=\lambda$, to evolve and automate the test input generation across multiple types of injection attacks. The reasons for using evolutionary algorithms to facilitate the test input generation are justified as follows.

- As discussed in [11], the number of potentially feasible test inputs for a specific type of injection attack is typically too large to efficiently enumerate. Furthermore, the corresponding search space is complex and unknown a priori. These characteristics make evolutionary algorithms well-suited for generating effective test inputs due to their ability to explore a wide range of potential solutions in a black-box search space.
- The iterative and population-based nature of evolutionary algorithms also allows for a more thorough and diverse exploration of the search space. This can lead to the discovery of novel and effective test inputs [2, 9, 11].

As the pseudo-code shown in Algorithm 2, each type of injection attack has its own population of test inputs, which can be shared with each others during the search. To comply with the syntactical correctness, the initial population (lines 1 to 4) of each injection type is fully seeded by the test inputs generated according to Algorithm **??** (line 2) introduced in Section 3.1.1. To improve the evolution efficiency, the bypassing cases are removed before evaluation in every generation.

Note that at each generation, we run the test inputs in all the populations against the WAF, after which those can successfully bypass are removed from its population but stored in the corresponding

[10] In our implementation, it's worth noting that the choice of the evolution strategy is not a critical factor and does not impact our experimental conclusions.

Algorithm 2: Multi-Task Evolutionary Algorithm.

Input:
$\mathcal{G}$: context-free grammars for all injection tasks
$\mathcal{C}$: surrogate classifiers for fitness assignment
$\mathcal{T}$: multi-task translation models
Output:
$\mathcal{A}$: bypassing cases for all injection tasks

1 **foreach** $\mathcal{P}_i \in \mathcal{P}$ **do**
2 $\quad \mathcal{P}_i \leftarrow$ GENEARTETESTINPUT($\mathcal{G}_i \in \mathcal{G}$)
3 $\quad$ /* Shared mating pool composing of all populations */
4 $\quad \mathcal{M}_i \leftarrow \mathcal{P}_i$, $\mathcal{M}_i \in \mathcal{M}$
5 **while** *The computational budget is not exhausted* **do**
6 $\quad$ **foreach** $\mathcal{P}_i \in \mathcal{P}$ **do**
7 $\quad\quad \mathcal{P}' \leftarrow \emptyset$ /* Offspring injections in $\mathcal{P}_i$ */
8 $\quad\quad \mathcal{B} \leftarrow \emptyset$ /* Bypassed injections in $\mathcal{P}_i$ */
9 $\quad\quad$ **foreach** $\mathcal{S} \in \mathcal{P}_i$ **do**
10 $\quad\quad\quad S_x \leftarrow$ GETFROMMATINGPOOL($\mathcal{M}_x \in \mathcal{M}$)
11 $\quad\quad\quad \mathcal{S}_t \leftarrow$TRANSLATE($\mathcal{S}_x, \mathcal{T}_k \in \mathcal{T}$)
12 $\quad\quad\quad$ **if** BYPASSINGWAF*($\mathcal{S}_t$)* == *True* **then**
13 $\quad\quad\quad\quad \mathcal{B} \leftarrow \mathcal{S}_x$
14 $\quad\quad\quad$ **else**
15 $\quad\quad\quad\quad \mathcal{S}_m \leftarrow$MUTATE($\mathcal{S}$)
16 $\quad\quad\quad\quad$ **if** BYPASSINGWAF*($\mathcal{S}_m$)*==*True* **then**
17 $\quad\quad\quad\quad\quad \mathcal{B} \leftarrow \mathcal{S}_t$
18 $\quad\quad\quad \mathcal{P}' \leftarrow \mathcal{P}' \bigcup (\mathcal{S}_x | \mathcal{S}_m)$
19 $\quad\quad \mathcal{A}_i \leftarrow \mathcal{A}_i \bigcup \mathcal{B}$
20 $\quad\quad \mathcal{P}' \leftarrow \mathcal{P}' - \mathcal{B}$
21 $\quad\quad$ FITNESSEVALUATE($\mathcal{P}', \mathcal{C}_i \in \mathcal{C}$)
22 $\quad\quad \mathcal{U} \leftarrow$SORTBYFITNESS($\mathcal{P}_i \bigcup \mathcal{P}'$)
23 $\quad\quad \mathcal{P}_i \leftarrow$top m test inputs from $\mathcal{U}$
24 $\quad$ **foreach** $\mathcal{P}_i \in \mathcal{P}$ **do**
25 $\quad\quad$ /* Update the sharing mating pool */
26 $\quad\quad \mathcal{M}_i \leftarrow \mathcal{P}_i$, $\mathcal{M}_i \in \mathcal{M}$
27 **return** $\mathcal{A}$

archive for later evaluation (lines 19 to 23). Moreover, the individual injection case whose fitness is below the average fitness of the population is replaced by randomly generated injections in order to maintain a promising balance between convergence and diversity at the early stage of evolution. This can help avoid the search being trapped at some patterns of injection attacks that have already discovered the relevant vulnerabilities. This is important for injection testing [2].

Remark 4. *In Algorithm 2,* GENERATETESTINPUT BYPASSINGWAF *returns the result of the bypassing check.* FITNESSEVALUATE*: Assigns fitness using the corresponding surrogate classifiers.* SORTBYFITNESS *returns the population sorted according to each individual's fitness.*

3.4.1 Representation

Given the support of the word embedding in the surrogate classifiers and translation models for each type of injection attack, our encoding for the MTEA in `DaNuoYi` is a string of words, representing a particular test input that can be of a different length. As such, the evolved test inputs can be directly attached to the HTTP request sent to the WAF for performance evaluation.

3.4.2 Objective Function

As discussed in Section 3.2.2, the trained surrogate classifiers serve as the fitness function that evaluates the likelihood of a test input bypassing the WAF (line 21). Since such a classifier is empowered by the word embedding, no additional processes are required for evaluating a test input.

3.4.3 Mutation Operators

To better maintain syntactically correct test inputs, DaNuoYi only uses mutation operators for offspring reproduction (line 15). In particular, we develop the following six word-level mutation operators to mutate a test input into another different yet semantically related test input. In practice, at least one mutation operator will be randomly selected with the same probability for offspring reproduction:

1. *Grammar tree transformation:* Drawing upon the CFGs described in Section 3.1.1, this method involves converting a subtree of the grammar tree into a new subtree, which is randomly generated using the same production rules. It is important to ensure that the resulting subtree adheres to the syntax constraints of the production rules in the original subtree's position within the tree. For example, `jsString-> alert(1)` to `%61%6c%65%72%74%28%31%29`

2. *Text transformation:* It confuses the capital and small letters in a test input. For example, `SCRIPT` to `sCrIPT`.

3. *Blank replacement:* It replaces the blank character in a test input with an equivalent symbol. For example, `"+alert('XSSi')+"` to `"%20alert('XSSi')%20"`.

4. *Comment concatenation:* It randomly adds comments between two words. For example, `<table background=''></table>` to `<table/*inj- ection*/background=''></table>`.

5. *ASCII mutation:* It mutates a word to its equivalent ASCII encoding format.

6. *Unicode mutation:* It mutates a word to its equivalent Unicode format.

The first mutation operator, i.e., *Grammar tree transformation*, depends on the underlying injection type, as it is based on the corresponding CFG which is language specific; while the other five mutation operators are generic. In particular, all mutation operators are syntax-compliant since they either make change in a way that complies with the corresponding grammar as the first operator or perform straightforward encoding amendment and comment insertion, etc, as the other five operators.

Remark 5. *Certain evolutionary algorithms, including evolutionary strategies, predominantly employ mutation operators while excluding recombination operators such as crossover. In the context of* DaNuoYi*, we have observed that mutation-centric methods are better suited for preserving input semantic information. Conversely, recombination operators have a higher tendency to generate solutions that fail to retain the required semantic characteristics. To be more specific, traditional crossover operators are not atomic and involve global recombination based on randomly selected node positions. When we cut the context-free grammar (especially for complex CFGs) from random positions, the recombination operations may lead to test inputs with potential code syntax errors. In other words, crossover operations can be more challenging to control in terms of adhering to the syntax of the target programming language, particularly when combining parts from parents with complex context-free grammars. In contrast, mutation operators are atomic and utilize nodes with the same types, which often simplifies the task of ensuring that the resulting structure remains valid within the CFG. In conclusion, defining and implementing crossover operators for context-free grammars used in test input generation presents significant difficulties. We will explore the development of reliable crossover operators in future works.*

3.4.4 Knowledge Sharing and Multi-Task Evolution

As shown in Algorithm 2, to facilitate the evolution of test input generation across multiple types of injection attack from different populations, the MTEA in DaNuoYi carries out a multi-task evolution in which each task is a standard evolutionary algorithm that evolves the test inputs with respect to a given type of injection attack (lines 5 to 26). Different from the traditional single-task evolutionary algorithm, in which the mating parents are merely from the same parent population, the offspring generation in MTEA aims to take advantages of elite information from all tasks. In particular, MTEA maintains a sharing mating pool, in which the mating parents are collected from the populations of any randomly chosen tasks by using a *top-k* (k equals the population size) fitness ranking selection mechanism (line 22). Such a mating pool is updated at the end of every generation (lines 24 to 26). Since the test input generation for one type of injection attack can exploit the semantic knowledge of the promising test inputs from other relevant types, we can expect to have a better chance to find more sophisticated test inputs for injection. In particular, the translation of a test input from one type of injection attack to another is realized by a corresponding translation model as introduced in Section 3.3. Note that if the translation fails, a mutation operation will be conducted to amend this translated test input. Note that since the initial population is seeded by test inputs generated according to different CFGs, which are malicious by themselves, we can expect that the test inputs generated by MTEA are still malicious.

4 Experiments and Evaluation

In this section, we evaluate and analyze the effectiveness of DaNuoYi through answering the following research questions (RQs) in the presence of multiple types of injection attack.

- **RQ1:** Does DaNuoYi find more valid test inputs that bypass the WAF than the baselines including the open-source SQLMap and its single-task counterparts respectively designed for each type of injection attack? Note that the corresponding single-task counterpart does not apply the translation between different types of injection attacks.
- **RQ2:** Does DaNuoYi find more valid test inputs than a random search (based on grammar only) with translation but without the mutation operators?
- **RQ3:** Does DaNuoYi find as many bypassed injection attacks as the variant without fitness ranking? Note that this variant applies the translation but it does not have an elitism.
- **RQ4:** What happens if we merely rely on the use of the dedicated CFG to generate test inputs?

All experiments were carried out on a server equipped with 64-bit Cent OS7, which has two Intel Xeon Platinum 8160 CPU (48 Cores 2.10GHz), 256GB RAM and four RTX 2080 Ti GPUs. The target WAFs were deployed on a virtual cloud server with 64-bit Ubuntu 18.04, which is based on the AMD EPYC 7K62 48-Core CPU and has 1GB RAM.

Remark 6. ***RQ1*** *seeks to evaluate the effectiveness and added value of our proposed multi-task test input generation paradigm.* ***RQ2*** *aims to evaluate the effectiveness of mutation operators in evolutionary algorithms.* ***RQ3*** *aims to evaluate the importance of the selection pressure in evolutionary algorithms.* ***RQ4*** *investigates the potential drawbacks of merely using the CFG-based method for the test input generation, compared to employing an evolutionary algorithm, and aims to understand the bottlenecks of* DaNuoYi.

4.1 Experiment Setup

4.1.1 Types of Injection Attack

In theory, DaNuoYi can generate any type of injection attacks to test the vulnerabilities of the system under test as long as the computational budget permits. In this work, we examine six types of injection attack. They are chosen according to their prevalence and severity reported by OWASP[11]. We summarize the characteristics of these injection attacks as follows.

- SQLi: It targets relational database management systems. This attack involves an attacker inserting malicious SQL statements into an entry field of the application, which can cause the application to unintentionally execute those statements, resulting in unauthorized access to or modification of sensitive data. SQLi is widely recognized as the most prevalent injection attack and is a significant threat to data security, often resulting in severe data breaches.
- XSSi: It targets web applications using JavaScript to dynamically update or modify web page content. In an XSSi, an attacker inserts malicious code, typically in the form of a script, into a web page. When other users visit the page, the code can execute, potentially resulting in various types of attacks such as session cookie theft or malware injection.
- XMLi: It targets applications utilizing XML or XPath to process data. Attackers can manipulate data encoded in XML format and exploit vulnerabilities in XPath queries to gain unauthorized access to sensitive information. This attack can result in severe data breaches.
- HTMLi: It targets web applications and can result in the modification of web pages rendered by the application, potentially affecting all visitors. Attackers insert malicious HTML code into a web page, which can be executed by other users who visit the page. This attack can lead to various types of consequences, including phishing or the theft of session cookies.
- OSi: It targets applications executing shell commands. Attackers can exploit OSi vulnerabilities to inject malicious system-level commands, such as port listening or a fork bomb, and execute them on the server-side. This attack can lead to serious security breaches and unauthorized access to sensitive data.
- PHPi: It targets applications utilizing the PHP programming language. Attackers exploit PHPi vulnerabilities to insert malicious PHP code into the application, potentially leading to various types of attacks depending on the context, such as path traversal or denial-of-service attacks.

4.1.2 Subject WAFs

To evaluate the practicality and improve the external validity, we evaluate DaNuoYi on the following three widely used real-world WAFs.

- `ModSecurity`[12]: This is a cross-platform WAF and it serves as the fundamental security component for `Apache HTTP Server`, `Microsoft IIS`, and `Nginx`, which underpin millions of the web applications worldwide. This WAF maintains a large amount of rule sets that provide defense mechanisms for various types of injection attack.
- `Ngx-lua-WAF`[13]: This is a scalable WAF designed for high-performance web applications. It also supports the defense on different types of injection attack with rule sets complement those covered by `ModSecurity`.

[11] https://owasp.org/www-community/attacks
[12] https://www.modsecurity.org
[13] https://github.com/loveshell/ngx_lua_waf

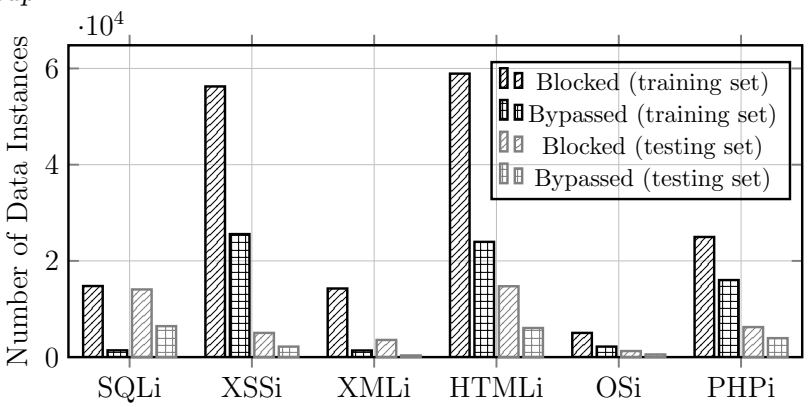


Figure 7: Bar charts of the distribution of both the blocked and the bypassed data in the training and testing sets, respectively, of six types of injection attack.

- `Lua-resty-WAF`[14]: This is another popular and scalable open-source WAF based on `OpenResty`[15]. It supports the patterns extended from `ModSecurity`.

We deploy a dummy web application behind the WAFs. Note that in this work, the testing target is the WAF; the web application merely serves as the destination of the attacks.

4.1.3 Dataset

As discussed in Section 3.1.1, we use the CFGs to generate the initial test inputs which constitute the datasets for training both the surrogate classifiers and the multi-task translation framework of `DaNuoYi` shown in Fig. 5.

- To train the classifier for each type of injection attack, we generate $20,000$ test inputs for three underlying WAFs, respectively. These test inputs are labeled as either *blocked* or *bypassed* according to the results when they are fed into the WAF. The same mechanism is applied to sample $20,000$ test inputs, exclusive from the training set, to constitute the testing set. Note that the identical injections are filtered in all datasets. The distribution of the blocked and the bypassed data in both training and testing sets are shown as bar charts in Fig. 7 while the corresponding numbers are listed in Table 8 of the Appendix.
- To train the models for the bidirectional translation between any two types of injection attacks as introduced in Section 3.3.1, we need $\binom{6}{2} = 15$ pairs of datasets given six types of injection attack. To that end, we generate $30,000$ pairs of translatable test inputs for each of these 15 pairs of datasets. To avoid data overlapping, we also filter identical injections in all datasets and the testing sets are sampled as extra $20,000$ test inputs exclusive from the training sets. As before, Fig. 8 plots the distribution of the number of instances in the translation datasets for six types of injection attacks used in our experiments. Besides, the corresponding numbers are listed in Table 9 of the Appendix.

Remark 7. *From our preliminary experiments, we determined that* $20,000$ *test inputs are sufficient to evaluate the performance of different algorithms while remaining computationally efficient in terms of time and memory usage. We acknowledge that the appropriate number of test inputs can vary depending on the specific problem and research objectives; however, exploring this aspect is beyond the scope of this paper.*

[14] https://github.com/p0pr0ck5/lua-resty-waf
[15] https://openresty.org

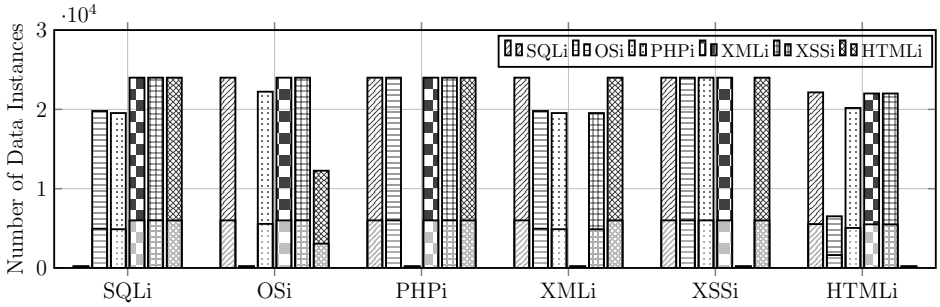


Figure 8: Bar charts of the distribution of the number of instances in the translation datasets for six types of injection attack. Note that the training set is highlighted in black while the testing set is in gray color.

4.1.4 Settings

DaNuoYi synergies several key techniques of NLP and search-based software testing, each of which has some hyper-parameters. In our empirical study, the relevant hyper-parameters are set according to the results of offline parameter tuning.

- Word2Vec: The dimension of the embedding vector of each token w.r.t. a test injection in DaNuoYi as 128.
- Classifier: For each type of injection attack, we apply three different neural networks (i.e., RNN, LSTM and GRU) as the alternative classifiers. They are all set to have one hidden layer. All the classifiers share the same pretrained `Word2Vec` embedding.
- Translation Model: The LSTM under the `Seq2Seq` framework is set with 128 hidden units, which are consistent to the surrogate classifiers.
- MTEA: Each task maintains a population of 100 solutions and the number of generations is set as 50. Note that we find that these settings can strike a good balance between performance and efficiency.

4.1.5 Metric and Statistical Test

To evaluate the ability for generating injection instances, we consider the following two metrics in our experiments.

- The first metric is to the number of different bypassing test inputs. If a distinct test input bypasses the WAF, it means the identification of a potentially new vulnerability [2].
- We also propose another metric to assess the number of test inputs based on the semantic diversity clustering i.e., the number of test input clusters. To calculate this metric, we apply Code-BERT [35] to encode the generated test inputs into feature vectors. Then, we apply DBSCAN [36] to categorize these feature vectors as different clusters, with the number of clusters serving as an indicator of semantic diversity.

Remark 8. *We have experimented with two alternative methods to measure test input similarity: Jaccard distance and Levenshtein distance, in addition to semantic similarity. However, we found that these two distance metrics are not sufficiently powerful to capture deep semantic similarity. Consequently, we opted for a more recent approach to compare deep semantics, namely pretrained language models. Pretrained language models have been proven to be highly suitable for programming language modeling. For instance, CodeBERT [35] is a language model pretrained on a large corpus of code, including code related to injection*

tasks. Therefore, we decided to extract the semantics of the test inputs using CodeBERT and cluster the semantics based on DBSCAN. The DBSCAN algorithm is unsupervised and well-suited for identifying clusters. In contrast, the other two distance metrics are inefficient and semantically irrelevant, which means they cannot accurately capture semantic diversity.

To mitigate potential bias, each experiment is repeated 21 independent runs with different random seeds. To have a statistical interpretation of the significance of comparison results, we use the following three statistical tests in our empirical study.

- Wilcoxon signed-rank test [37]: This is a non-parametric statistical test that makes no assumption about the underlying distribution of the data. In particular, the significance level is set to $p = 0.05$ in our experiments.
- Scott-Knott test [38]: Instead of merely comparing the raw metric values, we apply the Scott-Knott test to rank the performance of different peer techniques over 21 runs on each test scenario. In a nutshell, the Scott-Knott test uses a statistical test and effect size to divide the performance of peer algorithms into several clusters. The performance of peer algorithms within the same cluster is statistically equivalent. Note that the clustering process terminates until no split can be made. Finally, each cluster can be assigned a rank according to the mean metric values achieved by the peer algorithms within the cluster. In particular, the larger the rank is, the better performance of the algorithm achieves.
- A_{12} effect size [39]: To ensure the resulted differences are not generated from a trivial effect, we apply A_{12} as the effect size measure to evaluate the probability that one algorithm is better than another. Specifically, given a pair of peer algorithms, $A_{12} = 0.5$ means they are *equivalent*. $A_{12} > 0.5$ denotes that one is better for more than 50% of the times. $0.56 \leq A_{12} < 0.64$ indicates a *small* effect size while $0.64 \leq A_{12} < 0.71$ and $A_{12} \geq 0.71$ mean a *medium* and a *large* effect size, respectively.

Note that both Wilcoxon signed-rank test and A_{12} effect size are also used in the Scott-Knott test for generating clusters.

4.2 Performance Comparison of DaNuoYi with the Corresponding Single-Task Counterparts

4.2.1 Methods

To the best of our knowledge, DaNuoYi is the first of its kind tool to generate test inputs for more than one type of injection attack simultaneously. To answer **RQ1**, we plan to compare our proposed DaNuoYi with its single-task counterparts which can only generate test inputs for a given type of injection attack. There have been some tools to serve this purpose, such as [2] for SQLi, [9] for XMLi, and [40] for XSSi. Unfortunately, these tools are neither open-source projects nor readily available. In our experiments, we use SQLMap[16], an open-source penetration testing tool for detecting and exploiting SQLi flaws, as a peer method. In addition, we extract each task of the MTEA in DaNuoYi as the peer method for automatic test input generation for a given type of injection attack (generally denoted as STEA). They serve as resemblances to the existing single-task tools given that they share similar learning and evolutionary search techniques. Since we use three different neural networks, i.e., RNN, LSTM, and GRU, as the surrogate classifiers, there are three different groups in the comparisons.

[16] https://sqlmap.org/

4.2 Performance Comparison of DaNuoYi *with the Corresponding Single-Task Counterparts*

Table 3: The total number of bypassed test inputs (over 21 runs) obtained by SQLMap, CFG-based method and STEA versus DaNuoYi with different surrogate classifiers under the same computational budget.

WAF	Injection	RNN		LSTM		GRU		SQLMap
		STEA	DaNuoYi	STEA	DaNuoYi	STEA	DaNuoYi	
ModSecurity	SQLi	1798 (171)†	**2232 (118)**	1922 (93)†	**2309 (102)**	1844 (86)†	**2127 (54)**	663 (59) †
	OSi	2622 (36)†	**3424 (18)**	2555 (35)†	**3375 (20)**	2578 (25)†	**3375 (30)**	\
	PHPi	4015 (105)†	**4790 (13)**	4111 (53)†	**4815 (14)**	2578 (25)†	**4800 (16)**	\
	XMLi	2010 (150)†	**2337 (110)**	1959 (155)†	**2340 (103)**	1876 (78)†	**2188 (45)**	\
	XSSi	1836 (99)†	**2830 (125)**	2435 (201)†	**3100 (284)**	1786 (169)†	**2546 (95)**	\
	HTMLi	997 (693)†	**2937 (102)**	742 (234)†	**2852 (306)**	1122 (561)†	**2773 (232)**	\
Ngx-Lua-WAF	SQLi	4610 (18)†	**4984 (5)**	4620 (16)†	**4985 (6)**	4602 (22)†	**4984 (5)**	153 (31) †
	OSi	5000 (0)	5000 (0)	5000 (0)	5000 (0)	5000 (0)	5000 (0)	\
	PHPi	2434 (969)†	**3471 (468)**	2170 (716)†	**3433 (114)**	2156 (318)†	**3617 (311)**	\
	XMLi	4630 (26)†	**4982 (6)**	4625 (25)†	**4984 (2)**	4610 (12)†	**4983 (4)**	\
	XSSi	947 (37)†	**1762 (46)**	715 (63)†	**1631 (50)**	952 (72)†	**1808 (60)**	\
	HTMLi	1614 (180)†	**3523 (56)**	1744 (997)†	**3599 (85)**	2241 (166)†	**3907 (105)**	\
Lua-Resty-WAF	SQLi	2231 (124)†	**2901 (118)**	2338 (49)†	**2965 (38)**	2286 (94)†	**2880 (50)**	2334 (21) †
	OSi	4514 (17)†	**4942 (9)**	4469 (66)†	**4936 (14)**	4322 (128)†	**4907 (9)**	\
	PHPi	2789 (706)†	**3967 (116)**	2651 (736)†	**4005 (53)**	2372 (537)†	**3918 (161)**	\
	XMLi	2373 (81)†	**3034 (110)**	2380 (138)†	**2996 (78)**	2288 (67))†	**2935 (43)**	\
	XSSi	1542 (85)†	**3048 (219)**	2560 (529)†	**3717 (115)**	1893 (233)†	**3296 (111)**	\
	HTMLi	1072 (843)†	**3011 (136)**	779 (256)†	**2952 (221)**	1158 (539)†	**2771 (320)**	\

Each cell shows the median value of the number of bypassing test inputs with the IQR value in the parentheses.

† denotes that DaNuoYi is significantly better than the peer algorithm according to the Wilcoxon rank-sum test at a 0.5 significance level.

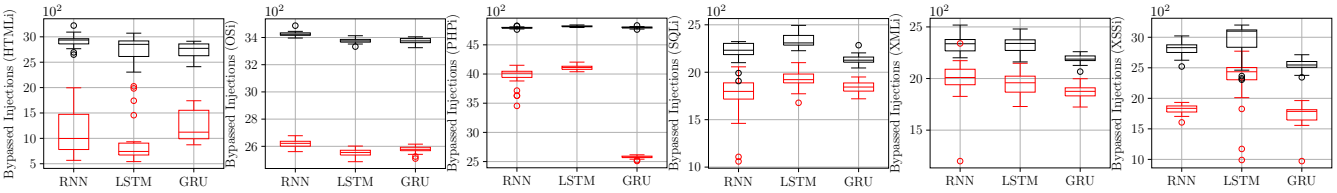


Figure 9: Box plots of the distributions of the bypassed test inputs generated by DaNuoYi (denoted as the black boxes) compared against the corresponding single-task counterparts (denoted as the red boxes) with RNN, LSTM, GRU as the surrogate classifier, respectively, on ModSecurity over 21 runs under the same computational budget. The numbers of bypassed injections are scaled by 10^2.

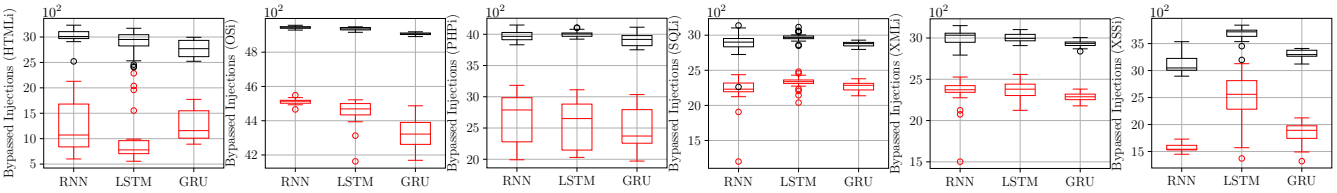


Figure 10: Box plots of the distributions of the bypassed test inputs generated by DaNuoYi (denoted as the black boxes) compared against the corresponding single-task counterparts (denoted as the red boxes) with RNN, LSTM, GRU as the surrogate classifier, respectively, on Lua-Resty-WAF over 21 runs under the same computational budget. The numbers of bypassed injections are scaled by 10^2.

4.2 Performance Comparison of DaNuoYi *with the Corresponding Single-Task Counterparts*

Table 4: The total number of clusters of bypassing test inputs (over 21 runs) obtained by SQLMap and STEA versus DaNuoYi with different surrogate classifiers under the same computational budget.

WAF	Injection	RNN		LSTM		GRU		SQLMap
		STEA	DaNuoYi	STEA	DaNuoYi	STEA	DaNuoYi	
ModSecurity	SQLi	1611 (131)†	**1669 (119)**	1621 (26)†	**1745 (84)**	1441 (246)†	**1520 (209)**	461 (50) †
	OSi	1721 (180)†	**2238 (142)**	1661 (26)†	**2143 (113)**	1629 (47)†	**2158 (112)**	\
	PHPi	2940 (206)†	**3779 (114)**	3438 (109)†	**3813 (142)**	3494 (82)†	**3792 (142)**	\
	XMLi	1803 (199)†	**2032 (117)**	1609 (88)†	**1821 (46)**	1593 (123)†	**1696 (93)**	\
	XSSi	825 (296)†	**1714 (82)**	737 (187)†	**1468 (82)**	900 (38)†	**1540 (82)**	\
	HTMLi	988 (91)†	**1177 (178)**	995 (266)†	**1038 (178)**	1108 (188)†	**1228 (214)**	\
Ngx-Lua-WAF	SQLi	2950 (167)†	**3264 (52)**	3157 (117)†	**3277 (61)**	3121 (29)†	**3265 (57)**	68 (28) †
	OSi	3188 (207)†	**3328 (112)**	3029 (29)†	**3234 (86)**	3149 (79)†	**3304 (30)**	\
	PHPi	1802 (159)†	**2579 (185)**	1728 (100)†	**2679 (124)**	1848 (146)†	**2894 (93)**	\
	XMLi	**3577 (183)**	3240 (72)	3214 (80)†	**3328 (91)**	3308 (9)	**3324 (12)**	\
	XSSi	871 (82)†	**1707 (210)**	750 (82)†	**1556 (212)**	878 (82)†	**1767 (221)**	\
	HTMLi	1576 (296)†	**2570 (230)**	1384 (196)†	**2595 (276)**	1868 (235)†	**2849 (100)**	\
Lua-Resty-WAF	SQLi	1781 (57)†	**2309 (237)**	1726 (95)†	**2210 (102)**	1668 (164)†	**2016 (213)**	1378 (76)†
	OSi	2864 (218)†	**3248 (35)**	2861 (104)†	**3206 (179)**	2817 (86)†	**3196 (199)**	\
	PHPi	1707 (100)†	**2954 (16)**	1509 (280)†	**2927 (146)**	1602 (218)†	**2914 (24)**	\
	XMLi	1925 (202)†	**2249 (238)**	1790 (134)†	**2452 (33)**	1893 (177)†	**2238 (98)**	\
	XSSi	1463 (244)†	**2698 (60)**	1272 (201)†	**2406 (180)**	1473 (147)†	**2651 (168)**	\
	HTMLi	924 (200)†	**1272 (239)**	479 (248)†	**1036 (45)**	803 (90)†	**1272 (196)**	\

Each cell shows the median value of the number of bypassing test inputs with the IQR value in the parentheses.

† denotes that DaNuoYi is significantly better than the peer algorithm according to the Wilcoxon rank-sum test at a 0.5 significance level.

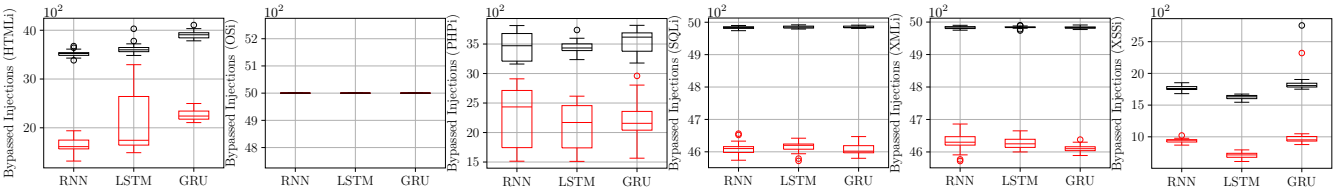


Figure 11: Box plots of the distributions of the bypassed test inputs generated by DaNuoYi (denoted as the black boxes) compared against the corresponding single-task counterparts (denoted as the red boxes) with RNN, LSTM, GRU as the surrogate classifier, respectively, on Ngx-Lua-WAF over 21 runs under the same computational budget. The numbers of bypassed injections are scaled by 10^2.

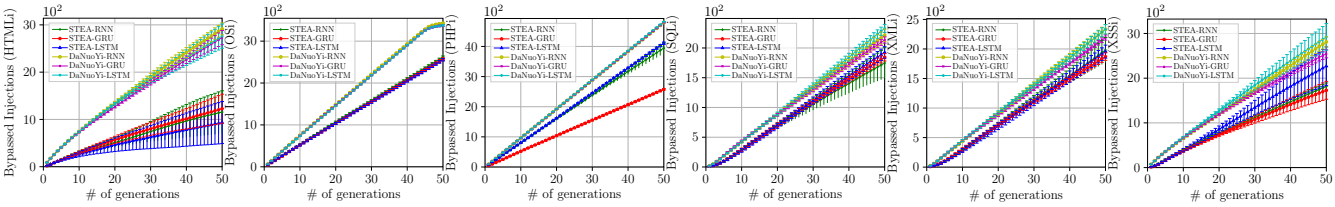


Figure 12: The number of valid test inputs generated by DaNuoYi compared against the corresponding single-task counterparts with RNN, LSTM, GRU as the surrogate classifier, respectively, during the evolutionary process on ModSecurity over 21 runs under the same computational budget (shown as error bars). The numbers of bypassed injections are scaled by 10^2.

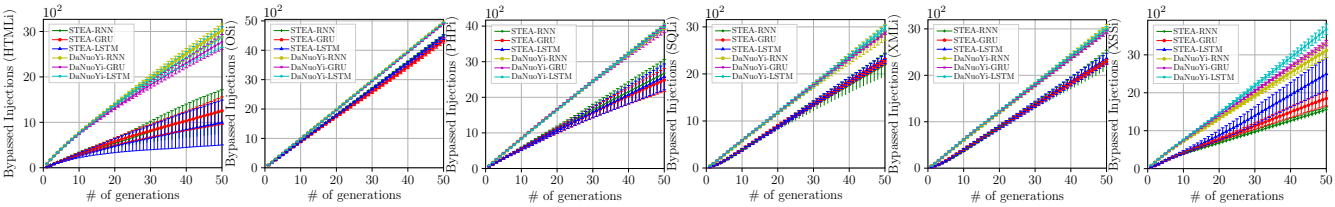


Figure 13: The number of valid test inputs generated by DaNuoYi compared against the corresponding single-task counterparts with RNN, LSTM, GRU as the surrogate classifier, respectively, during the evolutionary process on Lua-Resty-WAF over 21 runs under the same computational budget (shown as error bars). The numbers of bypassed injections are scaled by 10^2.

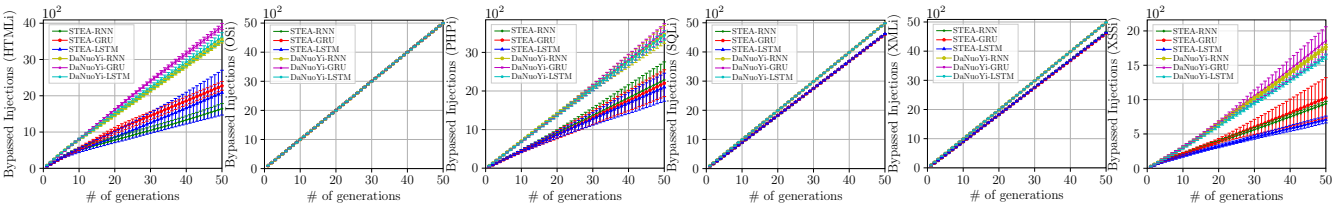


Figure 14: The number of valid test inputs generated by DaNuoYi compared against the corresponding single-task counterparts with RNN, LSTM, GRU as the surrogate classifier, respectively, during the evolutionary process on Ngx-Lua-WAF over 21 runs under the same computational budget (shown as error bars). The numbers of bypassed injections are scaled by 10^2.

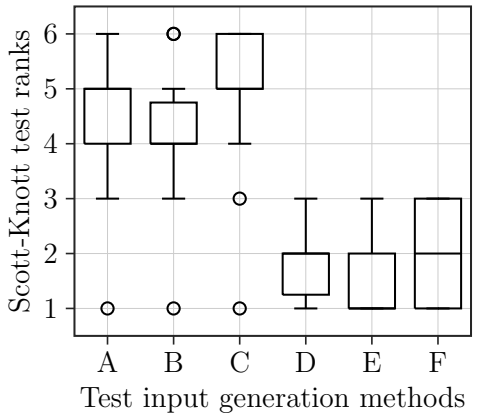


Figure 15: Box plots of Scott-Knott test ranks of the number of bypassed test inputs achieved by DaNuoYi with RNN, LSTM, and GRU as the surrogate classifier, respectively, on all WAFs and injection tasks compared against the corresponding STEAs. In particular, the x label from A to F represents DaNuoYi-RNN, DaNuoYi-GRU, DaNuoYi-LSTM, STEA-RNN, STEA-GRU and STEA-LSTM, respectively.

4.2.2 Results

Table 3 gives the comparison results of the total number of distinct test inputs, generated by different test input generation methods, bypassing a given WAF under the same computational budget. Let us first look into the comparison with SQLMap. The experimental results show that DaNuoYi and its six single-task variants significantly outperform SQLMap on the subject WAFs, ModSecurity and Ngx-Lua-WAF. Specifically, the number of test inputs found by SQLMap is only around 26% of those found by DaNuoYi and around 40% of those obtained by the corresponding single-task variant of DaNuoYi for SQLi. For the WAF Lua-Resty-WAF, although SQLMap's performance improves, it is still outperformed by DaNuoYi.

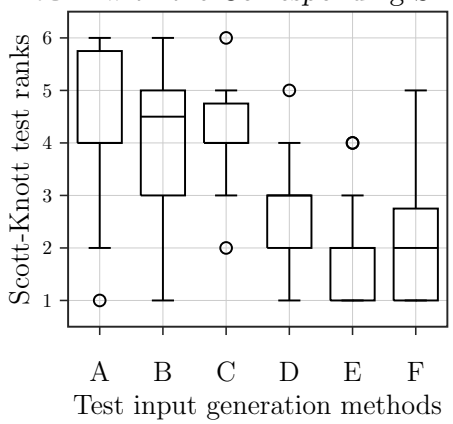


Figure 16: Box plots of Scott-Knott test ranks achieved by DaNuoYi based on the number of injection clusters, respectively, on all WAFs and Injection tasks compared against the corresponding STEAs. In particular, the x label from A to F represents DaNuoYi-RNN, DaNuoYi-GRU, DaNuoYi-LSTM, STEA-RNN, STEA-GRU and STEA-LSTM, respectively.

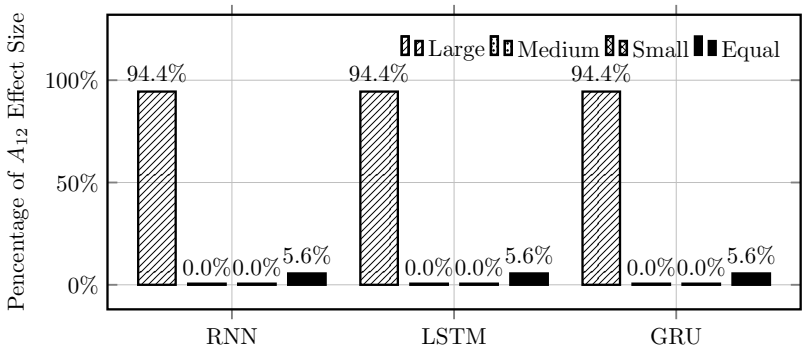


Figure 17: Percentage of the large, medium, small, and equal A_{12} effect size, respectively, when comparing DaNuoYi with the corresponding STEA that uses same surrogate classifier on the number of bypassed test inputs.

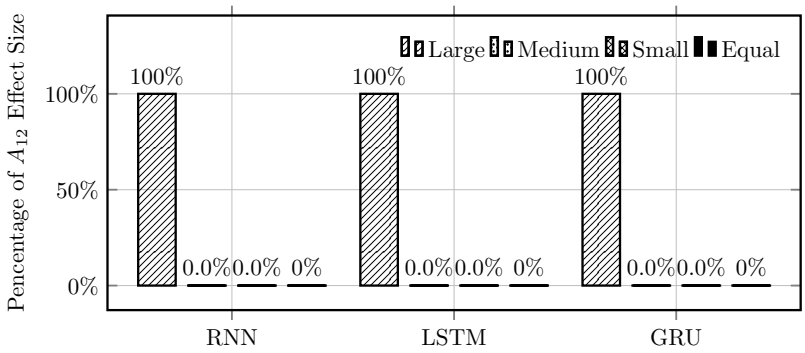


Figure 18: Percentage of the large, medium, small, and equal A_{12} effect size, respectively, when comparing DaNuoYi with the corresponding STEA that uses same surrogate classifier based on the number injection clusters.

As for the comparison results between DaNuoYi and its corresponding single-task counterparts, it is clear to see that DaNuoYi is able to find more bypassed test inputs (nearly up to 3.8×) for each type of injection attack on all three WAFs. The only exception is the generation of test inputs for the OSi on

Ngx-Lua-WAF where all methods have shown constantly the same performance. This can be explained as the small search space in this scenario, which does not pose any challenge to the search of valid test inputs. As shown in Fig. 9 to Fig. 11, we apply the box plots to give a better statistical view upon the comparison results. From these figures, we can clearly see that the number of valid test inputs identified by DaNuoYi is consistently larger than its single-task counterparts.

In addition, to investigate the performance of DaNuoYi against its corresponding single-task counterparts, we keep a record of the number of valid test inputs generated during the evolutionary process. Note that SQLMap does not involve an evolutionary search process, it is thus not considered in this study. As the trajectories are shown in Fig. 12 to Fig. 14, it is clear to see that DaNuoYi is able to generate more valid test inputs all the time. It is also interesting to note that these single-task test input generation methods can easily get stuck at the early stage of evolution; whereas the number of test inputs generated by DaNuoYi steadily increases with the evolutionary process.

To have an overall comparison of DaNuoYi against the corresponding single-task counterparts for all different types of injection attack w.r.t. all WAFs, we apply the Scott-Knott test and A_{12} upon the collected comparison results. From the box plots of Scott-Knott test results shown in Figs. 15 and 16 along with the bar charts of A_{12} shown in Figs. 17 and 18, we find that DaNuoYi have shown overwhelmingly better performance compared to the corresponding single-task counterparts.

Response to RQ1: *From the empirical results discussed in this subsection, we confirm the effectiveness of* DaNuoYi*. Specifically, by leveraging the similarity of semantic knowledge across different injection attacks,* DaNuoYi *can generate more valid test inputs compared to the corresponding single-task counterparts, which easily get stuck at the early stage of evolution. In addition,* DaNuoYi *serves the purpose of generating multiple types of injection attacks in a multi-task manner.*

4.3 Investigation of the Impacts of the Mutation Operators in DaNuoYi

4.3.1 Methods

According to the experiments in Section 4.2.2, we have witnessed the effectiveness of our proposed DaNuoYi for generating test inputs for different types of injection attacks. We argue that the six mutation operators developed in Section 3.4.3 are the driving force to create valid test inputs for the underlying WAF. To understand the usefulness of these mutation operators in DaNuoYi, we develop a variant, dubbed CFG-DaNuoYi, that uses a random sampling method based on the CFG w.r.t. the underlying injection attack to replace the mutation operators in DaNuoYi. In particular, we investigate CFG-DaNuoYi for all three surrogate classifiers considered in this paper.

4.3.2 Results

Table 5 and Table 6 show the performance of CFG-DaNuoYi under different WAFs by using three surrogate classifiers. As we expected, the six mutation operators play a major role in generating test cases. It is clear to see that the performance of DaNuoYi, without using the mutation operators, is degraded in all scenarios. In particular, this variant even generates fewer valid test inputs than the STEA methods in some cases, e.g., on SQLi, PHPi and XMLi. As shown in Table 5, CFG-DaNuoYi suffers more than 50% performance loss on SQLi and XMLi in most of the scenarios. This indicates that the corresponding WAFs are more vulnerable than those mutated SQLi and XMLi test inputs. On the contrary, CFG-DaNuoYi generally outperforms the corresponding STEA methods on XSSi and HTMLi, which are featured in a longer length of the injection string. This indicates that the bidirectional translation can lead to more diversified test inputs so as to make up the missing mutation operators to a certain extent.

4.4 Investigation of the Impacts of the Surrogate Classifier in DaNuoYi

Table 5: The total number of bypassed test inputs (over 21 runs) obtained by CFG-DaNuoYi, random search (RAN) and rule-based injection search (RIS) versus DaNuoYi with different surrogate classifiers under the same computational budget.

WAF	Injection	RNN		LSTM		GRU		RIS	RAN
		CFG-DaNuoYi	DaNuoYi	CFG-DaNuoYi	DaNuoYi	CFG-DaNuoYi	DaNuoYi		
ModSecurity	SQLi	492 (24)†	**2232 (118)**	510 (17)†	**2309 (102)**	501 (16)†	**2127 (54)**	405 (23)†	808 (65)†
	OSi	2167 (20)†	**3424 (18)**	2154 (33)†	**3375 (20)**	2168 (14)†	**3375 (30)**	2187 (10)†	3182 (23)†
	PHPi	3820 (30)†	**4790 (13)**	3815 (24)†	**4815 (14)**	3801 (13)†	**4800 (16)**	3774 (23) †	4662 (29)†
	XMLi	519 (20)†	**2337 (110)**	524 (13)†	**2340 (103)**	528 (12)†	**2188 (45)**	405 (23) †	781 (75)†
	XSSi	1160 (35)†	**2830 (125)**	1202 (24)†	**3100 (284)**	1254 (36)†	**2546 (95)**	849 (38)†	1710 (88)†
	HTMLi	1501 (37)†	**2937 (102)**	1517 (19)†	**2852 (306)**	1445 (31)†	**2773 (232)**	967 (9)†	1926 (67)†
Ngx-Lua-WAF	SQLi	4628 (11)†	**4984 (5)**	4618 (12)†	**4985 (6)**	4634 (29)†	**4984 (5)**	4456 (25)†	4960 (11)†
	OSi	5000 (0)	5000 (0)	5000 (0)	5000 (0)	5000 (0)	5000 (0)	5000 (0)	5000 (0)
	PHPi	2056 (19)†	**3471 (468)**	2066 (15)†	**3433 (114)**	2061 (29)†	**3617 (311)**	2074 (22)†	3297 (47)†
	XMLi	4628 (6)†	**4982 (6)**	4621 (15)†	**4984 (2)**	4619 (25)†	**4983 (4)**	4456 (25)†	4953 (13)†
	XSSi	1012 (26)†	**1762 (46)**	1010 (25)†	**1631 (50)**	1017 (26)†	**1808 (60)**	1175 (18)†	2120 (86)
	HTMLi	2694 (36)†	**3523 (56)**	2667 (31)†	**3599 (85)**	2680 (15)†	**3907 (105)**	2746 (50)†	4010 (67)
Lua-Resty-WAF	SQLi	1164 (25)†	**2901 (118)**	1131 (39)†	**2965 (38)**	1132 (48)†	**2880 (50)**	1013 (19)†	1864 (50)†
	OSi	4422 (7)†	**4942 (9)**	4423 (11)†	**4936 (14)**	4423 (16)†	**4907 (9)**	4434 (8)†	4916 (12)†
	PHPi	2766 (22)†	**3967 (116)**	2773 (33)†	**4005 (53)**	2759 (40)†	**3918 (161)**	2767 (38)†	3911 (54)†
	XMLi	1136 (25)†	**3034 (110)**	1157 (49)†	**2996 (78)**	1135 (39)†	**2935 (43)**	1013 (19)†	1854 (60)†
	XSSi	2067 (75)†	**3048 (219)**	2027 (46)†	**3717 (115)**	2063 (33)†	**3296 (111)**	1817 (56)†	3072 (49)†
	HTMLi	1530 (25)†	**3011 (136)**	1529 (35.5)†	**2952 (221)**	1509 (46)†	**2771 (320)**	1022 (17)†	1994 (76)†

Each cell shows the median value of the number of bypassing test inputs with the IQR value in the parentheses.
† denotes that DaNuoYi is significantly better than the peer algorithm according to the Wilcoxon rank-sum test at a 0.05 significance level.

Response to RQ2: *From the empirical results discussed in this subsection, we confirm the importance of our proposed six mutation operators. They can bring more diversity into the MTEA thus leading to an increased number of valid test inputs.*

4.4 Investigation of the Impacts of the Surrogate Classifier in DaNuoYi

4.4.1 Methods

According to the experiments in Section 4.3.2, we have already validated the effectiveness of those mutation operators for offspring reproduction in the MTEA of DaNuoYi. How to select the elite solutions to either survive to the next generation or to construct the mating pool for offspring reproduction is another important component of an evolutionary algorithm. In DaNuoYi, the selection process is guided by the surrogate classifier that predicts the chance of a candidate test input for bypassing the underlying WAF. To address **RQ3**, we replace this mechanism with a random selection, dubbed RAN. More specifically, it randomly picks up the mating parents from the mating pool, so as the survival of parents and offspring. Note that we still apply the bidirectional translation between different types of injection attack within the mating pool.

4.4.2 Results

From the comparison results shown in Table 5 and Table 6, it is clear to see that the performance of RAN is outperformed by our proposed DaNuoYi in all scenarios. In addition, STEA can even find more valid test inputs than RAN in some cases such as the XMLi on ModSecurity and Lua-Resty-WAF. This observation confirms the importance of elitism in an evolutionary algorithm. In other words, without the guidance of an appropriate fitness function (i.e., the surrogate classifier in DaNuoYi), the evolutionary

4.5 Investigation of the Impacts of CFG in DaNuoYi

Table 6: The total number of clusters of bypassed test inputs (over 21 runs) obtained by CFG-DaNuoYi, random search (RAN) and rule-based injection search (RIS) versus DaNuoYi with different surrogate classifiers under the same computational budget.

WAF	Injection	RNN		LSTM		GRU		RIS	RAN
		CFG-DaNuoYi	DaNuoYi	CFG-DaNuoYi	DaNuoYi	CFG-DaNuoYi	DaNuoYi		
ModSecurity	SQLi	390 (69)†	**1669 (153)**	405 (29)†	**1745 (144)**	380 (60)†	**1520 (94)**	357 (54)†	628 (95)†
	OSi	1544 (65)†	**2238 (53)**	1499 (80)†	**2143 (36)**	1511 (34)†	**2158 (71)**	1383 (32)†	2078 (68)†
	PHPi	3093 (66)†	**3779 (56)**	3117 (63)†	**3813 (41)**	3078 (39)†	**3792 (65)**	2319 (35) †	3449 (74)†
	XMLi	380 (62)†	**2032 (149)**	405 (25)†	**1821 (130)**	397 (37)†	**1696 (85)**	310 (41) †	718 (110)†
	XSSi	799 (82)†	**1714 (169)**	841 (73)†	**1468 (309)**	792 (46)†	**1540 (138)**	573 (58)†	1312 (135)†
	HTMLi	650 (84)†	**1177 (142)**	667 (57)†	**1038 (320)**	702 (79)†	**1228 (273)**	499 (26)†	943 (116)†
Ngx-Lua-WAF	SQLi	867 (38)†	**3264 (34)**	874 (41)†	**3277 (17)**	859 (73)†	**3265 (42)**	2764 (38)†	3070 (56)†
	OSi	2881 (48)†	**3328 (25)**	2913 (16)†	**3234 (30)**	2936 (11)†	**3304 (16)**	2096 (9)	3277 (29)
	PHPi	2019 (29)†	**2579 (505)**	1985 (35)†	**2679(155)**	1941 (45)†	**2894 (350)**	2074 (39)†	1881 (84)†
	XMLi	875 (51)†	**3240 (36)**	913 (44)†	**3328 (22)**	873 (48)†	**3324 (16)**	732 (40)†	3154 (23)†
	XSSi	983 (43)†	**1707 (88)**	992 (37)†	**1556 (95)**	926 (60)†	**1767 (120)**	795 (30)†	1906 (155)
	HTMLi	728 (80)†	**2570 (89)**	711 (59)†	**2595 (132)**	750 (63)†	**2849 (144)**	1484 (97)†	2267 (103)
Lua-Resty-WAF	SQLi	867 (33)†	**2309 (160)**	874 (82)†	**2210 (65)**	859 (89)†	**2016 (99)**	755 (38)†	1514 (98)†
	OSi	2881 (12)†	**3248 (35)**	2913 (36)†	**3206 (47)**	2936 (44)†	**3196 (27)**	2578 (27)†	3025 (45)†
	PHPi	2119 (46)†	**2954 (151)**	2085 (54)†	**2927 (88)**	2041 (57)†	**2914 (208)**	1698 (72)†	2843 (99)†
	XMLi	875 (42)†	**2249 (155)**	913 (95)†	**2452 (126)**	873 (75)†	**2238 (78)**	819 (35)†	1490 (114)†
	XSSi	1683 (130)†	**2698 (260)**	1692 (81)†	**2406 (192)**	1626 (71)†	**2651 (157)**	1628 (101)†	2429 (96)†
	HTMLi	728 (65)†	**1272 (179)**	711 (77)†	**1036 (266)**	750 (96)†	**1272 (347)**	693 (33)†	1080 (141)†

Each cell shows the median value of the number of bypassing test inputs with the IQR value in the parentheses.

† denotes that DaNuoYi is significantly better than the peer algorithm according to the Wilcoxon rank-sum test at a 0.05 significance level.

search suffers from a lack of selection pressure thus is less effective. Another interesting observation from our experiments is that the capability of generating valid test inputs is partially related to the length of the characters w.r.t. the corresponding injection. More specifically, if an injection string is long such as XSSi and HTMLi, a sufficient selection pressure becomes important to guide the evolutionary search. This explains the better performance achieved by STEA w.r.t. RAN on XSSi and HTMLi. On the other hand, if the length of an injection string is short such as OSi, the translation between different types of injection attack can be highly beneficial to the generation of valid test inputs.

Response to RQ3: *From the comparison results discussed in this subsection, we confirm the importance of the surrogate classifier as an alternative of the fitness function. It provides a sufficient selection pressure to guide the evolutionary search process. This is critical for problems with a large search space.*

4.5 Investigation of the Impacts of CFG in DaNuoYi

4.5.1 Methods

As discussed in Section 3.1.1, the CFG used in the data gathering and profiling step is derived from some existing injection attack examples. According the corresponding CFG for an injection type, the rule-based injection search (RIS) method introduced in Section 3.1.1 plays as the source to seed some initial test inputs for training the surrogate classifier. In this case, a natural question is whether this RIS method guided by the CFG adequate to serve the purpose of test input generation? To address **RQ4**, we use the CFG as the template to generate 100,000 test inputs for each injection type. In particular, we are mainly interested in the statistics of the number of non-duplicated test inputs and those bypassing the underlying WAFs.

Table 7: The validity statistics of injection cases generated by rule-based injection search according to a given CFG.

Injection type	# of non-duplicated test inputs (percentage)	# of bypassed test inputs (success rate)		
		ModSecurity	Ngx-Lua-WAF	Lua-Resty-WAF
SQLi	25079 (25.1%)	2082 (8.3%)	24287 (96.8%)	5502 (21.9%)
OSi	6534 (6.5%)	2826 (43.3%)	6534 (100%)	5766 (88.2%)
PHPi	48093 (48.1%)	37593 (78.2%)	20366 (89.0%)	22880 (47.6%)
XMLi	25042 (25.0%)	2110 (8.4%)	24250 (96.8%)	4997 (20.1%)
XSSi	98931 (98.9%)	16978 (17.2%)	23419 (23.7%)	35824 (36.2%)
HTMLi	99535 (99.5%)	18371 (18.6%)	55439 (55.7%)	19510 (19.6%)

4.5.2 Results

According to the statistical results shown in Table 7, we can see that the percentage of the non-duplicated test inputs is less than 50% for SQLi, XMLi, OSi, and PHPi. Especially for OSi, the corresponding percentage is merely around 6.5%. However, it is surprising to see that the success rates to bypass all three WAFs for OSi are very promising. This can be explained as the relatively small search space of OSi as discussed in Section 4.2 that renders the test inputs generated by the corresponding CFG duplicated. In contrast, the success rates of bypassing the WAFs for SQLi, XMLi and PHPi vary significantly case by case. For example, both SQLi and PHPi experience a hard time in ModSecurity of which the success rates of the bypassed test inputs generated by the corresponding CFGs are less than 10%; whereas the success rates for Ngx-Lua-WAF are always high. On the other hand, we can see that the percentage of non-duplicated test inputs generated by the RIS methods for XSSi and HTMLi is extremely high with 98.9% and 99.5%, respectively. This can be explained as the fact that both HTMLi and XSSi tend to be long strings leading to a large search space, i.e., an exponentially increased number of potential combinations of different strings. Partially due to this reason, the success rates to bypass all three WAFs for those generated test inputs are unfortunately low. This suggests that the RIS can hardly find effective injection cases in a large search space.

In addition, we compare the RIS method with DaNuoYi under the same amount of computational search budget for generating test inputs. Since the RIS method cannot work in a multi-task manner, we run a test input generation routine for one injection type at a time, as done in STEA. From the comparison results shown in Table 5, we find that the performance of the RIS method is significantly worse than that of DaNuoYi in all cases, especially for ModSecurity and Lua-Resty-WAF, the relatively more challenging WAFs. In particular, it is worth noting that the RIS method is comparable with STEA in many cases. This also supports the importance of our multi-task mechanism that complements each other when generating different types of injection attacks. Let us look at the evolutionary trajectories shown in Fig. 12 to Fig. 14, we can see that DaNuoYi keeps on generating more effective test inputs after being seeded by the RIS as the initial test inputs. This observation confirms the importance and usefulness of the follow-up evolution and translation in DaNuoYi.

Response to RQ4: *There are two takeways from the experiments in this subsection. First, the CFG itself can be used to generate test inputs for a given injection type. However, its effectiveness is hardly guaranteed especially when handling a large search space. Second, by leveraging the semantic information associated with each injection attack along with the multi-task paradigm,* DaNuoYi *can develop more effective test inputs for different types of injection attacks simultaneously.*

5 Threats to Validity

Threats to construct validity may raised from the way we measure the effectiveness of DaNuoYi. In this work, we use the number of generated test inputs that bypass the WAF as the main metric, following the suggestion from prior work [2]. To improve reliability of the conclusions, we have also applied Wilcoxon rank-sum test to evaluate the statistical significance.

The parameter settings in the experiments, e.g., the dimension of the Word2vec and the number of generations in MTEA, may cause threats to internal validity. Indeed, we acknowledge that a different set of settings may lead to other results. However, we set these parameter values based on the results of both automatic and manual hyper-parameter tuning.

To ensure external validity, we evaluate DaNuoYi on three widely used real-world WAFs and six most prevalent types of injection attack. We have also run DaNuoYi under three alternative classifiers, i.e., LSTM, RNN, and GRU. To mitigate evaluation bias, we repeat each experiment run 10 times. Nonetheless, we do agree that additional subject WAFs are useful for future work.

6 Related Works

Over the past decade, several white-box, static and model-based approaches have been proposed for testing injection vulnerability based on specific syntax, mainly target exclusively for SQLi. Among others, Halfond and Orso [41] propose AMNESIA, which generates test input for SQLi based on static code analysis. Similarly, Mao et al. [42] also seek to generate SQLi test inputs based on a pre-defined finite automata. However, these approaches are restricted by the given rules and require full access to the source code, which is what limits their testing ability as already shown in prior work [2, 9, 11].

In the industry, SQLMap is a well-known rule-based testing tool for SQLi, which is also applicable to WAF. However, it is fundamentally different from DaNuoYi and difficult to be experimentally compared in a fair manner, because:

- SQLMap focuses on SQLi only. DaNuoYi, in contrast, works on any type of injection attack.
- SQLMap stops as soon as it finds a test input that can bypass the WAF while DaNuoYi would continue to evolve till the pre-defined resources are exhausted.
- Prior work (e.g., Liu et al. [11]) has shown that SQLMap is significantly inferior to single-task learning and search-based testing methods, which are the peer methods that we have quantitatively compared for **RQ1** and **RQ2**.

Other black-box, learning and search-based fuzzers for injection testing also exist. For example, to discover SQLi vulnerabilities on WAF, Demetrio et al. [10] combine a random search fuzzer with strongly-typed syntax for testing. Similarly, Appelt et al. [2, 8] leverage an evolutionary algorithm supported by a learned surrogate model to generate test inputs. Liu et al. [11] propose DeepSQLi, a method that uses techniques from NLP to learn and test SQLi vulnerabilities in web applications, including the WAFs. Edalat et al. [43] develops an Android-oriented SQL injection detection tool to protect Android applications, while this tool aims at function-level attacks instead of multiple types of injection attacks. Uwagbole et al. [44] propose to generate SQL injections by leveraging attack patterns derived from existing SQL injections. However, this method relies on the reserved keywords of SQL and is hard to adapt to other injection types. Eassa et al. [45] argues that the importance of defending injection attack on NoSQL is underestimated and try to detect injections by developing an independent module based on PHP. They share some of the same motivations as DaNuoYi such that every test input has its own semantic information. For other injection types, Jan et al. [9] exploit an evolutionary algorithm, empowered by a specifically

designed distance function, to generate test inputs for XMLi on WAF. Evolutionary fuzzing for testing XSS has also been explored [40].

On the other hand, many recent works focus on injection detection (e.g., code injection attack and SQL injection detection). Zuech et al. [46] argues that using ensemble classifiers to predict potential SQL injections. However, the test injections used in work are prepared as a dataset that is not designed to find new injections. Thang [47] also adopts three existing dataset to predict the malicious HTTP requests. Jahanshahi et al. [48] presents SQLBlock, a plugin for PHP & MySQL-based Web applications, to prevent SQL injection without any modification on existing Web applications. And the test case preparation method is not reusable for other injection types.

Several previous works have used CFGs to generate test inputs, but they have not specifically focused on efficient selection and exploitation of injection vulnerabilities like DaNuoYi. For example, Purdom's algorithm [49] generates short sentences from a CFG but does not address injection attack vulnerabilities. Havrikov et al. [29] introduced k-path coverage for test generation, but their approach is not suitable for fitness evaluation due to efficiency limitations. Kifetew et al. [27] proposed using grammar annotations for system-level test generation, achieving similar levels of coverage and fault detection as learned probabilities, but their approach does not focus on detecting injection vulnerabilities. Soremekun et al. [50] developed strategies for learning input distributions for grammar-based test generation, but their method may not precisely generate and identify test inputs that can expose vulnerabilities. Poulding et al. [26] proposed to optimize probability distributions for test input generation, but do not directly address the detection and prevention of injection attacks. While the aforementioned works contributed to testing and grammar-based systems, DaNuoYi's focus on injection vulnerabilities sets it apart and offers a promising approach for future research in this area. DaNuoYi specifically addresses the efficient selection and exploitation of injection vulnerabilities through its use of an MTEA and its focus on generating effective injection attacks for multiple types of vulnerabilities. Although the literature focusing on context-free grammars usually share similar processes with Algorithm **??**, the details of the context-free grammars are quite different for various purposes, particularly the aim of injection generation for multiple tasks, such as SQLi. Due to different application domains, the CFGs in the previous literature cannot be easily adapted to the test input generation, to the best of our knowledge. As a result, we collect simple but effective CFGs for six injection tasks to generate test inputs. Nevertheless, we believe the insights from existing works on context-free grammar can certainly improve our future works

As mentioned, the above work neither publishes the source code (or the code contains severe errors) nor has readily available tools for direct quantitative comparisons. However, given the common techniques that underpin those tools, they are the resemblances of the single-task counterparts we evaluated for **RQ1**. It is clear that DaNuoYi differs from all the above work in the sense that:

- It works on any type of injection attack as opposed to one.
- It learns, translates and exploits the common semantic information across different types of injection attack. This, as we have shown in Section 4, allows DaNuoYi to find more bypassing test inputs on the WAF.

7 Conclusion

In this paper, we propose DaNuoYi, a multi-task end-to-end fuzzing tool that simultaneously generates test inputs for any type of injection attacks on WAF. DaNuoYi trains a classifier to predict the likelihood of a test input bypassing the WAF, and a pair of translation models between any two types of injection attack. These then equip the proposed multi-task evolutionary algorithm, which is the key component that realizes the multi-tasking in DaNuoYi, with the ability to share the most promising test inputs for different injection types. Through experimenting on three real-world open-source WAFs, three classifiers,

and six types of injection attack, we show that, in DaNuoYi, both the multi-task translation and multi-task search are effective in handling and transferring the common semantic information for different injection types, allowing it to produce up to 3.8× more bypassing test inputs than its single-task counterparts.

Future opportunities from this work are fruitful, including extending DaNuoYi beyond injection testing and a better synergy with the concept from reinforcement leaning.

Acknowledgment

This work was supported in part by the UKRI Future Leaders Fellowship under Grant MR/S017062/1 and MR/X011135/1; in part by NSFC under Grant 62376056 and 62076056; in part by the Royal Society under Grant IES/R2/212077; in part by the EPSRC under Grant 2404317; in part by the Kan Tong Po Fellowship (KTP\R1\231017); and in part by the Amazon Research Award and Alan Turing Fellowship. **Author contributions**: Li leads the research idea and writes the manuscript, Yang implements the system and designs the experiments and Visser participates in the supervision of the research.

References

[1] T. Beery and N. Niv, "Web application attack report," *Imperva, Annual Report*, 2013.

[2] D. Appelt, C. D. Nguyen, A. Panichella, and L. C. Briand, "A machine-learning-driven evolutionary approach for testing web application firewalls," *IEEE Trans. Reliab.*, vol. 67, no. 3, pp. 733–757, 2018.

[3] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *PLDI'08: Proc. of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, 2008, pp. 206–215.

[4] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao, "A static analysis framework for detecting SQL injection vulnerabilities," in *COMPSAC'07: Proc. of the 31st Annual International Computer Software and Applications Conference*, 2007, pp. 87–96.

[5] J. Jürjens and G. Wimmel, "Specification-based testing of firewalls," in *PSI'01: Proc. of the 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, vol. 2244, 2001, pp. 308–316.

[6] A. Doupé, M. Cova, and G. Vigna, "Why johnny can't pentest: An analysis of black-box web vulnerability scanners," in *DIMVA'10: Proc. of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2010, pp. 111–131.

[7] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Software Eng.*, vol. 37, no. 6, pp. 772–787, 2011.

[8] D. Appelt, A. Panichella, and L. C. Briand, "Automatically repairing web application firewalls based on successful SQL injection attacks," in *ISSRE:'17: Proc. of the 28th IEEE International Symposium on Software Reliability Engineering*. IEEE Computer Society, 2017, pp. 339–350.

[9] S. Jan, A. Panichella, A. Arcuri, and L. C. Briand, "Automatic generation of tests to exploit XML injection vulnerabilities in web applications," *IEEE Trans. Software Eng.*, vol. 45, no. 4, pp. 335–362, 2019.

[10] L. Demetrio, A. Valenza, G. Costa, and G. Lagorio, "WAF-A-MoLE: evading web application firewalls through adversarial machine learning," in *SAC'20: The 35th ACM/SIGAPP Symposium on Applied Computing*. ACM, 2020, pp. 1745–1752.

[11] M. Liu, K. Li, and T. Chen, "Deepsqli: deep semantic learning for testing SQL injection," in *ISSTA'20: Proc. of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 286–297.

[12] M. I. P. Salas, P. L. de Geus, and E. Martins, "Security testing methodology for evaluation of web services robustness - case: XML injection," in *SERVICES'15: Proc. of the 2015 IEEE World Congress on Services*. IEEE Computer Society, 2015, pp. 303–310.

[13] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *POPL'06: Proc. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2006, pp. 372–382.

[14] I. Hydara, A. B. M. Sultan, H. Zulzalil, and N. Admodisastro, "Current state of research on cross-site scripting (XSS) - A systematic literature review," *Inf. Softw. Technol.*, vol. 58, pp. 170–186, 2015.

[15] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS'13: Proc. of the 27th Annual Conference on Neural Information Processing Systems*, 2013, pp. 3111–3119.

[16] C. Pinzón, J. F. de Paz, Á. Herrero, E. Corchado, J. Bajo, and J. M. Corchado, "idmas-sql: Intrusion detection based on MAS to detect and block SQL injection through data mining," *Inf. Sci.*, vol. 231, pp. 15–31, 2013.

[17] A. Makiou, Y. Begriche, and A. Serhrouchni, "Improving web application firewalls to detect advanced SQL injection attacks," in *IAS'14: Proc. of the 10th International Conference on Information Assurance and Security*. IEEE, 2014, pp. 35–40.

[18] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[19] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A search space odyssey," *IEEE Trans. Neural Networks Learn. Syst.*, vol. 28, no. 10, pp. 2222–2232, 2017.

[20] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *EMNLP'14: Proc. of the 2014 Conference on Empirical Methods in Natural Language Processing*. ACL, 2014, pp. 1724–1734.

[21] R. Collobert and J. Weston, "A unified architecture for natural language processing: deep neural networks with multitask learning," in *ICML'08: Proc. of the 25th International Conference on Machine Learning*, vol. 307. ACM, 2008, pp. 160–167.

[22] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *ICLR'13: Proc. of the 1st International Conference on Learning Representations (Workshop Poster)*, 2013.

[23] Y. Kim, "Convolutional neural networks for sentence classification," in *EMNL'14: Proc. of the 2014 Conference on Empirical Methods in Natural Language Processing*. ACL, 2014, pp. 1746–1751.

[24] R. Socher, C. C. Lin, A. Y. Ng, and C. D. Manning, "Parsing natural scenes and natural language with recursive neural networks," in *ICML'11: Proc. of the 28th International Conference on Machine Learning*. Omnipress, 2011, pp. 129–136.

[25] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *EMNLP'13: Proc. of the 2013 Conference on Empirical Methods in Natural Language Processing*. ACL, 2013, pp. 1631–1642.

[26] S. M. Poulding, R. Alexander, J. A. Clark, and M. J. Hadley, "The optimisation of stochastic grammars to enable cost-effective probabilistic structural testing," *J. Syst. Softw.*, vol. 103, pp. 296–310, 2015.

[27] F. M. Kifetew, R. Tiella, and P. Tonella, "Generating valid grammar-based test inputs by means of genetic programming and annotated grammars," *Empir. Softw. Eng.*, vol. 22, no. 2, pp. 928–961, 2017.

[28] P. McMinn, "Search-based software test data generation: a survey," *Softw. Test. Verification Reliab.*, vol. 14, no. 2, pp. 105–156, 2004.

[29] N. Havrikov and A. Zeller, "Systematically covering input structure," in *ASE'19: Proc. of the 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2019, pp. 189–199.

[30] E. Pavese, E. O. Soremekun, N. Havrikov, L. Grunske, and A. Zeller, "Inputs from hell: Generating uncommon inputs from common samples," *CoRR*, vol. abs/1812.07525, 2018.

[31] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *ICLR'15: Proc. of the 3rd International Conference on Learning Representations*, 2015.

[32] R. Paulus, C. Xiong, and R. Socher, "A deep reinforced model for abstractive summarization," in *ICLR'18: Proc. of the 6th International Conference on Learning Representations*, 2018.

[33] R. Al-Rfou, M. Pickett, J. Snaider, Y. Sung, B. Strope, and R. Kurzweil, "Conversational contextual cues: The case of personalization and history for response ranking," *CoRR*, vol. abs/1606.00372, 2016. [Online]. Available: http://arxiv.org/abs/1606.00372

[34] L. Shang, Z. Lu, and H. Li, "Neural responding machine for short-text conversation," in *ACL'15: Proc. of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing*, 2015, pp. 1577–1586.

[35] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *EMNLP'20: Findings of the Association for Computational Linguistics*, ser. Findings of ACL, vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547.

[36] M. Ester, H. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *KDD'96: Proc. of the 2nd International Conference on Knowledge Discovery and Data Mining*. AAAI Press, 1996, pp. 226–231.

[37] W. Haynes, *Wilcoxon Rank Sum Test*. New York, NY: Springer New York, 2013, pp. 2354–2355.

[38] N. Mittas and L. Angelis, "Ranking and clustering software cost estimation models through a multiple comparisons algorithm," *IEEE Trans. Software Eng.*, vol. 39, no. 4, pp. 537–551, 2013.

REFERENCES

[39] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *J. Educ. Behav. Stat.*, vol. 25, no. 2, pp. 101–132, 2000.

[40] F. Duchene, R. Groz, S. Rawat, and J. Richier, "XSS vulnerability detection using model inference assisted evolutionary fuzzing," in *ICST'12: Proc. of the 5th IEEE International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2012, pp. 815–817.

[41] W. G. J. Halfond and A. Orso, "AMNESIA: analysis and monitoring for neutralizing sql-injection attacks," in *ASE'05: Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2005, pp. 174–183.

[42] C. Mao and F. Guo, "Defending SQL injection attacks based-on intention-oriented detection," in *ICCSE'16: Proc. of the 11th International Conference on Computer Science & Education*. IEEE, 2016, pp. 939–944.

[43] E. Edalat, B. Sadeghiyan, and F. Ghassemi, "Considroid: A concolic-based tool for detecting SQL injection vulnerability in android apps," *CoRR*, vol. abs/1811.10448, 2018. [Online]. Available: http://arxiv.org/abs/1811.10448

[44] S. O. Uwagbole, W. J. Buchanan, and L. Fan, "Applied machine learning predictive analytics to SQL injection attack detection and prevention," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), Lisbon, Portugal, May 8-12, 2017*. IEEE, 2017, pp. 1087–1090.

[45] A. M. Eassa, M. Elhoseny, H. M. El-Bakry, and A. S. Salama, "Nosql injection attack detection in web applications using restful service," *Program. Comput. Softw.*, vol. 44, no. 6, pp. 435–444, 2018.

[46] R. Zuech, J. T. Hancock, and T. M. Khoshgoftaar, "Detecting SQL injection web attacks using ensemble learners and data sampling," in *CSR'21: Proc. of the 2021 IEEE International Conference on Cyber Security and Resilience*. IEEE, 2021, pp. 27–34.

[47] N. M. Thang, "Improving efficiency of web application firewall to detect code injection attacks with random forest method and analysis attributes HTTP request," *Program. Comput. Softw.*, vol. 46, no. 5, pp. 351–361, 2020.

[48] R. Jahanshahi, A. Doupé, and M. Egele, "You shall not pass: Mitigating SQL injection attacks on legacy web applications," in *ASIA CCS'20: The 15th ACM Asia Conference on Computer and Communications Security*. ACM, 2020, pp. 445–457.

[49] P. Purdom, "A sentence generator for testing parsers," *BIT*, vol. 12, no. 3, p. 366–375, sep 1972. [Online]. Available: https://doi.org/10.1007/BF01932308

[50] E. O. Soremekun, E. Pavese, N. Havrikov, L. Grunske, and A. Zeller, "Inputs from hell," *IEEE Trans. Software Eng.*, vol. 48, no. 4, pp. 1138–1153, 2022.

8 Additional Experiments

8.1 An Qualitative Study of the Translation in DaNuoYi

8.1.1 Methods

From the discussion in Section 4.2, we have confirmed the effectiveness and outstanding performance of DaNuoYi for generating valid test inputs for different types of injection attacks. This subsection aims to step further and to understand whether the semantic meanings between different types of injection attacks have indeed been learned and exploited. In this section, we plan to qualitatively analyze some examples selected from our experiments, as well as some that go beyond our expectations.

8.1.2 Results

By investigating the intermediately translated and evolved test inputs together with their parent inputs in DaNuoYi, we found many examples similar to those listed in Table 1. This confirms that creating tautology is the most common way in injection attacks. Nevertheless, we have also identified some 'unusual' translations and evolution. In the following paragraphs, we will discuss some selected examples.

***Example 1*:** from SQLi (`'/**/or/**/'1'='1'--`) to XSS (`><s%43%72%49pt>%61%6c%65%72%74%28%31%29</s%43ri%70%74><!--`).

In the ***Example 1***, the parent input for SQLi is translated and evolved to another test input for XSSi equivalent to `><script>alert(1)</script><!--`, some of the characters are in ASCII encoding. We notice that DaNuoYi did not pick up the semantic meaning of tautology attack (as in the SQLi), but it does take the other piece of semantic information from the SQLi. In particular, the previous statement is closed by `'` in SQLi, so that `or '1'='1'` can be injected successfully. The same operation is used in XSSi which using `>` to close the previous statement.

***Example 2*:** from OSi (`0 ; sleep %20 1 |`) to SQLi (`)/**/;select sleep (2) #`).

In the ***Example 2***, DaNuoYi learned that the function `sleep %20 1`[17] in OSi is similar to the function `sleep (2)` in SQLi. In particular, it conducted two things when translating and evolving such test input from SS to SQLi. First, it changes the separator from `|` to `#`. Then, it takes a different syntax into account, i.e., the sleep function of SQLi is `sleep()` but not for the OSi. This is a typical example in which some semantic knowledge is captured by DaNuoYi while being aware of their syntactical differences.

***Example 3*:** from SQLi (`')/**/&& not/**/false or('`) to PHPi (`O:7:''Example":1:{s:3:''var";s:10:''phpinfo();";}`).

The ***Example 3*** is one of the most surprising examples we found from the results of DaNuoYi. At the first glance, the tautology in SQLi (due to the `not false` clause) may seem to be irrelevant to the serialized code injection in PHPi. Yet, if we inspect closely, it is clear that the PHPi aims to access some information through the `phpinfo()` call somewhere in the `eval` function. This matches with the ultimate purpose of creating a tautology, i.e., accessing or revealing some unauthorized information by creating something that is always true. This is beyond our expectation, as it suggests that DaNuoYi is not only able to interpret the literal meaning in the test inputs, but also some of the semantics related to the ultimate goal of an attack.

[17] `%20` is a *blank space* in ASCII encoding.

Table 8: The dataset summary for different injections used in three surrogate classifiers training(i.e., $20,000$ injections for each surrogate classifier). We denote the number of instances(**# of Inst.**), number of bypassed instances (**# of By. Inst.**), and number of blocked instances (**# of Bl. Inst.**) for each injection type across different dataset types (training and testing).

Injection	Dataset	# of Inst.	# of By. Inst.	# of Bl. Inst.
SQLi	Training	14776	1404	13372
	Testing	14065	6438	7627
XSSi	Training	56264	25571	30693
	Testing	5032	2184	2848
XMLi	Training	14260	1375	12885
	Testing	3564	346	3218
HTMLi	Training	58925	23956	34969
	Testing	14731	6049	8682
OSi	Training	5032	2184	2848
	Testing	1258	553	705
PHPi	Training	24973	16017	8956
	Testing	6243	3942	2301

Table 9: The number of instances in the injection translation datasets for SQLi, OSi, PHPi, XMLi, XSSi, and HTMLi. The source and target injection indicate translation direction.

Source Injection	Target Injection	Number of instances		
		Total	Train	Test
SQLi	OSi	24745	19795	4950
	PHPi	24421	19536	4885
	XMLi	30000	24000	6000
	XSSi	30000	24000	6000
	HTMLi	30000	24000	6000
OSi	SQLi	30000	24000	6000
	PHPi	27786	22228	5558
	XMLi	30000	24000	6000
	XSSi	30000	24000	6000
	HTMLi	15331	12264	3067
PHPi	SQLi	30000	24000	6000
	OSi	30000	24000	6000
	XMLi	30000	24000	6000
	XSSi	30000	24000	6000
	HTMLi	30000	24000	6000
XMLi	SQLi	30000	24000	6000
	OSi	24745	19795	4950
	PHPi	24421	19536	4885
	XSSi	24421	19536	4885
	HTMLi	30000	24000	6000
XSSi	SQLi	30000	24000	6000
	OSi	30000	24000	6000
	PHPi	30000	24000	6000
	XMLi	30000	24000	6000
	HTMLi	30000	24000	6000
HTMLi	SQLi	27678	22142	5536
	OSi	8146	6516	1630
	PHPi	25225	20179	5046
	XMLi	27506	22004	5502
	XSSi	27506	22004	5502