# ART4SQLi: The ART of SQL Injection Vulnerability Discovery

Long Zhang, Donghong Zhang, Chenghong Wang, Jing Zhao, and Zhenyu Zhang

***Abstract*—SQL injection (SQLi) is one of the chief threats to the security of database-driven Web applications. It can cause serious security issues such as authentication bypassing, privacy leakage, and arbitrary code execution. Dynamic testing techniques are used in SQLi vulnerability discovery, which de-facto approach is to maintain a collection of elaborately designed user inputs (aka. attack payloads) and based on it to compose malicious SQL queries to Web applications. Such techniques are effective to reveal SQLi threats before an application is released, thus reducing the cost of manual analysis, monitoring or postdeployment of other defensive mechanisms. However, because of the diversity of SQLi attacks and the difficulty of SQLi discovery, the process to execute payloads can be costly, time-consuming, and even risky. In this paper, we approach from a test case prioritization perspective to give a more effective SQLi discovery proposal, which is based on adaptive random testing with the aim to successfully trigger an SQLi within limited attempts. To evaluate our method, we conduct an experiment using three extensively adopted open source vulnerable benchmarks. The experiment results indicate that our method ART4SQLi can effectively improve the conventional random testing approach on three common benchmarks by more than 26% in reducing the number of SQLi attempts before accomplishing a successful injection.**

***Index Terms*—Adaptive random testing (ART), attack payload, SQL injection (SQLi), test case prioritization.**

## I. Introduction

DATABASE-DRIVEN Web applications are rapidly applied in a wide range of areas including online stores, e-commerce, social network services, etc. At the same time, the popularity makes them more attractive to attackers. The number of reported Web attacks is growing sharply [76]. For instance, a Web application attack report has observed an average increment of around 17% in different types of Web attacks over a nine-month period [35]. It was also reported that Web attacks had become more sophisticated and dramatically longer in length (44% longer than they were in previous reports), and a typical Web application may suffer more than 26 attacks in 1 min. Another security report indicated that at least 8% of the Web services of companies such as Microsoft and Google contained multiple types of security vulnerabilities [72].

```
1 <?php
2   $user_name = $_GET['user_name'];
3
4   $sql = "select * from users where "
5         . "user_name='$user_name'";
6   return $db->query($sql);
7 ?>
```

Fig. 1. Example of an SQLi vulnerability.

Within the class of Web-based vulnerabilities, SQL injection (SQLi) is labeled as one of the most serious threats by the Open Web Application Security Project (OWASP).[1] SQLi refers to a class of code injection attacks, which are carried out by composing a malicious user input into an SQL query to alter the behavior of Web applications [25]. Fig. 1 shows an example of login module, which parses user input (line 2), synthesizes a query (lines 4–5), and retrieves matched items from database (line 6). An attacker can fabricate a username "`' or 'a'='a`", and use the login form to submit an SQLi. The login module will synthesize the query "`select * from users where user_name='' or 'a'='a'`" as expected (lines 4–5), which will unconditionally retrieve all items from the database, exposing sensitive data to the attacker.

In this example, the input "`'/**/or/**/'1'='1`" is called an *attack payload*, and the exploited login vulnerability belongs to a kind of *SQLi vulnerability*. A lot of such vulnerabilities are ultimately caused by insufficient validation of user inputs [25], [26], [28]. To exploit such vulnerabilities, attackers make use of elaborately designed attack payloads to synthesize malicious requests, and submit such requests to database-driven Web applications.[2] Once the crafty SQL queries are executed by the application, attackers can obtain the privilege of accessing

Manuscript received March 31, 2018; revised November 20, 2018 and March 19, 2019; accepted March 21, 2019. This work was supported in part by the National Key Basic Research Program of China (project no. 2014CB340702) and by the National Natural Science Foundation of China under Grant 61572150. Associate Editor: W. K. Chan. *(Corresponding author: Zhenyu Zhang.)*

L. Zhang is with the University of Chinese Academy of Sciences, Beijing 100190, China, and also with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China (e-mail: zlong@ios.ac.cn).

D. Zhang is with the First Research Institute, Ministry of Public Security, Beijing 100733, China (e-mail: zhangdh15@gmail.com).

C. Wang is with the Department of Computer Science, Duke University, Durham, NC 27708 USA (e-mail: chw148@ucsd.edu).

J. Zhao is with the School of Software Technology, Dalian University of Technology, Dalian 116023, China (e-mail: zhaoj9988@dlut.edu.cn).

Z. Zhang is with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China (e-mail: zhangzy@ios.ac.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2019.2910285

[1]An open community dedicated to enable organizations to conceive, develop, acquire, operate, and maintain trustable applications.

[2]The relationship of "payload" and "vulnerability" is similar to that of "test case" and "bug" in conventional software testing.

the underlying database (in this example) or even take control of the system that hosts the Web applications. In the past years, SQLi vulnerabilities have become the top threat for database-driven Web applications, and SQLi attack is known as one of the most popular attacks.

Although the cause of SQLi is easy to understand, SQLi vulnerabilities are hard to detect and SQLi attacking events never disappear. In consequence, the approach to discover SQLi vulnerabilities is of great importance. Static analysis is a conventional security mechanism to address SQLi issues. For example, Fu *et al.* [22] described a compile-time static SQLi analysis framework, based on symbolic execution. When an SQL query is submitted, a hybrid constraint solver is used to determine corresponding user inputs that could lead to breach of information security. However, since constraint solving is often related to substantial computational complexity, static approaches come with intrinsic scalability problems such as state explosion. Besides the substantial computational complexity, static approaches also face other realistic problems. In practice, few companies are willing to share their source code because—"given enough eyeballs, all bugs are shallow" (the Linus' Law [57]). As a result, dynamic methods such as black-box testing become a popular choice. One representative method is mutation-based fuzz testing [3], [64]. This technique is effective to generate test cases covering many types of complicated attack patterns, such as BASE64 encoding attacks and white space obfuscation injections.

An effective testing-based SQLi vulnerability discovery technique should reduce applications from SQLi vulnerabilities before the applications are released [64]. Currently, a de-facto manner is to manipulate a collection of known attack payloads, such as the SQL query string, "`' or 'a'='a`," to carry out a black-box testing. On one hand, a major challenge is that there are many different kinds of SQLi attacks and countless variations on these basic forms. To cope with that, a payload collection is often huge in size and unavoidably contains redundancy. Executing such a payload collection without selection is infeasible. On the other hand, because SQLi vulnerabilities may hide deeply, and effective payloads are often rare. They comprise only a small portion of the entire payload collection, manifesting a sparse distribution. As a result, even if we have a set of attack payloads covering many types of SQLi attacks, it is still difficult to pick out the effective payloads from it. In addition, it may be complicated to statically evaluate or predict the execution result of attack payloads sometimes. Researchers have proposed an automatic mechanism for testing result checking and evaluation, but professional engineers are still necessary to manually complete the checking and judging tasks (e.g., SQLi existing in the HTTP Head Referrer, CVE-2011-3340) [3], [58]. To our best knowledge, such issues of testing-based SQLi vulnerability discovery have not been adequately addressed.

In this paper, we propose a technique ART4SQLi to facilitate SQLi vulnerability discovery. It is based on the observation from our experience that "*effective attack payloads are rare and unevenly distributed in the payload collections*." It uses a context-free grammar (CFG) [31] to describe attack payloads. Initially, it decomposes the original attack payload into tokens [29]. After that, it applies training label cleaning [52] to extract the feature vectors. Then, it makes use of the cosine distance to measure the distance between two vectorized attack payloads. Finally, it schedules the payload execution in an adaptive random testing (ART) manner, by selecting a payload set of a certain size according to the distance metrics. The payloads in such a set will be executed in turn, until a successful SQLi is reached. In such a way, ART4SQLi selects promising attack payloads to discover SQLi vulnerability, with the aim to reduce the number of unsuccessful attack trials. We conduct an experiment to evaluate the effectiveness of ART4SQLi, and compare its effectiveness in discovering SQLi vulnerabilities against the original random testing approach. We carry out the experiment on three open source vulnerability simulation benchmarks. To measure the effectiveness, the F-measure[3] [10] is adopted to estimate the expected number of attack payloads required to accomplish an SQLi. Experiment results show that ART4SQLi achieves 26% effectiveness improvements over its original random testing approach on Web for Pentester, DVWA 2014, and MCIR-SQLol subjects.

The main contributions of the work are as follows.

1) It reports that the effective payloads for detecting SQLis are rare and sparsely distributed, and tend to cluster.
2) It presents the feature vector extraction method and distance metrics quantification process for SQLi attack payloads.
3) It presents an ART-based novel technique ART4SQLi to facilitate SQLi vulnerability discovery.
4) An experimental evaluation shows that ART4SQLi improves the original random testing approach on Web for Pentester, DVWA 2014, and MCIR-SQLol by more than 26%.

The rest of this paper is organized as follows. Section II motivates this paper. In Section III, we propose and elaborate on our method ART4SQLi. Section IV uses three common benchmarks to evaluate the effectiveness of our proposal. Section V introduces related work. Section VI concludes this paper.

## II. Background and Motivation

In this section, we introduce the background of SQLi and use an example to motivate this paper.

### A. Preliminaries

To evaluate the SQLi vulnerability risk of a Web application, a security expert or a third-party service provider often tends to simulate the de-facto SQLi attacks. Fig. 2 shows the process.

Generally, to evaluate whether a Web application comes with SQLi issues is a high-frequency task. A security expert or an attacker (in the rest of this paper, we do not differentiate the two roles since they share the same workflow) often has a prepared collection of attack payloads in hand, as shown in Fig. 2. The role of the payloads is similar to the test cases in a conventional testing task. However, different from a conventional software testing task, security experts usually use a common set

[3]Not the namesake metrics in machine learning (F-measure [47]).

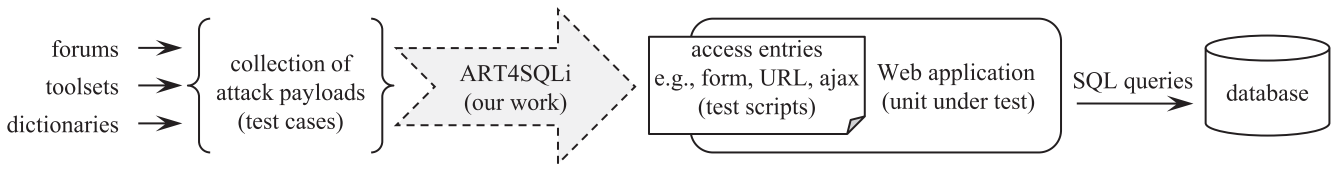
forums
toolsets
dictionaries
collection of attack payloads (test cases)
ART4SQLi (our work)
access entries e.g., form, URL, ajax (test scripts)
Web application (unit under test)
SQL queries
database

Fig. 2. De-facto workflow of SQLi and our work.

of attack payloads to discover SQLi vulnerabilities in various applications, because the targets (e.g., MySQL, Oracle, Microsoft Access, or common operating systems and platforms) often have similar security issues (e.g., single quote injection and relative path injection). In practice, such attack payloads can be collected from testing tools like *sqlmap*, *Pangolin*, *BSQL Hacker*, security forums such as *exploit-db*, *GreySec*, websites like *fuzzdb*, *PentesterLab*, and so on.

At the same time, in order to carry out more types of injection, security experts will usually mutate the original collection by using some mutation tools or plugins such as *sqlmap tamper*. Like a test script used to manipulate test cases to drive a unit under test, a payload is loaded into a Web application from various access entries. The easiest method is to replace some parameters in a request url or directly type malicious input in a page form. Sometimes, people also construct an HTTP request package manually to test more sensitive section, such as referring section, user-agent section, and host section.

Finally, a security expert will judge whether an injection succeeds, according to the response of the target Web application. In this checking phase, some tools are built to help understand the response and determine the result according to predefined features [3].

### B. Observations on Current Approach

We have the following observations on the current SQLi vulnerability discovery approach.

*1) Huge Size of a Payload Collection:* Previous works have summarized some classes of SQLi vulnerabilities, such as Boolean-based blind SQLi, error-based SQLi, union query SQLi, stacked queries SQLi, and time-based blind SQLi [26]. On the basis of these basic types, there are countless variations and many kinds of SQLi attacks. A good payload collection (aka. payload collection) is updated over time to not miss any effective SQLi payload.

The defense to prevent all kinds of SQLi is very hard. For instance, function and keyword filtering is expected to prevent Web applications from being attacked by using a functions and keywords black list. Such keywords may include *sleep*, *or*, *and*, and so on. Although the technique is effective and performs well in practice, applications equipped with such security mechanisms may still be vulnerable to SQLi attacks. A professional attackers might evade the inspection by using case changing, character encoding, or inline comment methods. If he submits an injection code containing a keyword or SQL function that is absent in the black list, such an injection may succeed. Thus, it is important to include enough payloads for testing in order to discover SQLi

```
1  <?php
2    $user_name = $_GET['user_name'];
3
4    $keys = array(" ", "%20", "%25", "(", ")", "'",
5                  "update", "sleep", "insert");
6    $user_name = str_replace($keys, "", $user_name);
7
8    $sql = "select * from users where "
9         . "user_name='$user_name'";
10   return $db->query($sql);
11 ?>
```

Fig. 3. Example of SQLi vulnerability.

vulnerabilities as more as possible. At the same time, mutation-based methods are employed to generate payload instances from payload template [33]. For example, Appelt *et al.* [3] used a set of mutation operators to generate a lot of payloads. Their evaluation demonstrated that it is effective to detect SQLi vulnerability. In both cases, a collection of payloads is huge in size because of the large scope or diversity of attack types, and the attempt to execute the whole collection may not be always practicable.

Furthermore, an attacker in practice seldom performs a brute-force trying to expose SQLi vulnerabilities, since such a manner increases both attacking cost and the risk of exposing the attacker [20]. Modern application servers are often equipped with the mechanism of recognizing suspicious queries. Once the intention of an attack is realized, an application may refuse to serve, slow down response, or throw out misleading answers. In practice, a hacking tool may even switch multiple attacking tactics in a short period. A sequential evaluating of all payloads in a payload collection against a Web application neither simulates the realistic practice nor be feasible to discover SQLi vulnerability given limited resources.

*2) Sparse Distribution of Effective Payloads:* A Web application may have complicated input validation mechanism to prevent some SQLi attempts. At the same time, SQLi attacks have never become extinct, and security experts always learn from smart hackers' unexpected attacking tactics.

Fig. 3 lists out the PHP code of a concrete login module in one of our previous tasks [73]. We find that this code excerpt has an input validation mechanism: the function `str_replace()` located at line 6 will filter whitespace and similar keywords. For example, if we start an attack using a usual payload, "`' or 'a'='a'#`," the special character "`'`" and all whitespaces will be removed. As a result, the synthesized query statement will be "`select * from users where user_name='ora=a#'`," which basically does no harm to the Web application. However, the

TABLE I
EXAMPLE OF PAYLOADS

| Effective Attack Payloads | Intraclass Distance | Interclass Distance |
|---|---|---|
| `%27/**/or/**/%27a%27=%27a%27#` | 18.2 | 64.7 |
| `/**/Or/**/%271%27=%271` | 14.3 | 81.3 |
| `%27/**/Or/**/%271%27=%271` | 17.3 | 71.3 |
| **Ineffective Attack Payloads** | | |
| `/**/or/**/'a'='a` | | |
| ` or 'abbab'='abbab` | | |
| ` or/**/'1'/**/like/**/'1` | | |
| `/**/or 'yes'='yes` | | |
| `/*/**/*/ or/*/**/*/'1'='1` | | |
| `/**/ or 'test'='test` | | |
| ` Or/**/' '/**/like/**/'` | | |
| `'/**/or true--` | | |
| `/**/or/**/'2'/**/like/**/'2` | | |
| `/**/ or 'y'='y` | | |

special character sequences "`/**/`" and "`%27`" are missing in the black list (lines 4–5). A well-designed injection trial can thus beg the validation mechanism. Let us take the payload "`%27/**/or/**/%27a%27=%27a%27#`" to illustrate it. When such a malicious user name is submitted, neither "`/**/`" nor "`%27`" is recognized as keywords and the query statement "`select * from users where user_name='%27/**/or/**/%27a%27=%27a%27#'`" will be synthesized. Such a strange string equals to "`select * from users where user_name='' or 'a'='a'`" in effect, which can unconditionally retrieve all items from the "`users`" table.

Without knowing the implementation detail of a Web application, what we can do is to try each payload candidate in a payload collection database. In practice, a strong payload collection covers many SQLi attack types, and the effective payloads are mostly a very small part of the full collection. This small proportion makes identifying them difficult.

*3) Uneven Distribution of Effective Payloads:* We further realize that effective payloads often resemble one another in syntax and grammar since they are products of similar tactics. Table I lists out some effective payloads used to attack the login module in Fig. 3. The table has two parts, i.e., the upper part and the lower part, which list out three effective payloads and ten ineffective payloads. Let us focus on the upper part first. The first column shows the payload strings. The second column "intraclass distance" calculates an intra-class distance, which is the average edit distance similarity [59] between an effective payload to the other effective payloads. The third column "interclass distance" calculates an interclass distance, which is the average edit distance similarity between an effective payload to all the ineffective payloads. For example, the first one is the payload we demonstrated in the last section. Its average edit distance to the other two effective payloads is 18.2, while the average edit distance between it and the ten ineffective payloads is 64.7. According to [59], it means that the first one payload is very similar to the other two effective payloads showed in Table I, and there is a great difference between this payload and the ineffective payloads.

From Table I, we have the observation that the effective payloads for the sample vulnerable page are similar to one another. At the same time, they look different from the ineffective payloads. The fact is related to the validation mechanism, which filters characters or character sequences according to the black list (lines 4–5 of Fig. 3). As a result, the surviving keywords are rare and the corresponding effective payloads often have centralized distribution or cluster together in the space of the payload collection. For the same reason, when an effective payload is found for some type of SQLi vulnerability, payloads closed to it are considered in practice and they are more likely to reveal the same vulnerability.

*4) High Expense to Evaluate Payloads:* Each time we evaluate a payload on a Web application, we need to obtain the postcondition state of the Web application after it answers the input. We then compare the state to the predefined security state to determine whether an SQLi have been triggered. Researchers have proposed some automatic mechanisms for result checking and evaluation [3], [58].

However, manual intervention is still necessary at times. For instance, in some Web applications, the injectable parameters appear in a particular section, such as the HTTP Head Referrer (CVE-2011-3340) section. Because the response is hard to predefined, and the location is usually ignored by the testing tool, we have to manually check whether the payload is injected successfully. As a result, the most accurate and effective way to check or evaluate the payloads is to have software engineers check every payload input and its response. Furthermore, response from the Web application can be also random or misleading, which makes automatic rules even harder to apply.

During the process, the Web application server sometimes blocks our requests and even responds nothing, we have to take a rest or change the IP, explorer agent or other information, to continue. It can be expensive to evaluate a payload, which also limits the possibility of large-scale or in-parallel testing.

## C. Our Basic Idea

Fuzz testing [23] is extensively used to discover SQLi vulnerability [80]. It is technically a kind of random black-box testing method. A software engineer first collects or designs SQLi payloads, then selects test cases from the payload collection uniformly at random without replacement, and finally uses selected payloads to evaluate the target Web application. Let us recall the goal of discovering SQLi vulnerability, which is to find an effective payload (that can successfully reveal SQLi vulnerability in the Web application) as fast as possible. It is fundamentally a black-box testing process. Considering the observations in the previous section that we have reported, we realize that a practical payload-driven testing process to discover SQLi vulnerability has not been given adequate attention [19], [22], [60].

We thus consider to find an approach to select effective payloads as far as possible, so as to reduce the time of the payload evaluating process and improve the performance of discovering SQLi vulnerability. In our observation, we find that effective payloads tend to have sparse distribution and cluster in the payload collection. As a result, we need to find payloads that are

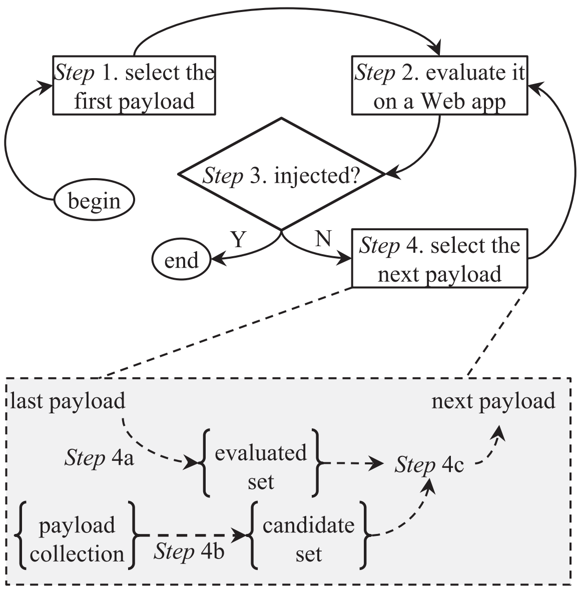

Fig. 4. ART4SQLi testing process overview.

far from those revealing no SQLi vulnerabilities, and evaluate such promising payloads in the next evaluation, with the aim to trigger SQLi vulnerability early. Many methods are proposed to select test cases that lead to failures, and their cost is less than the simple random manner, such as test case prioritization [19], [60], adaptive random test [13], [14], [46], etc. However, no research is reported in the field of testing-based SQL vulnerability.

The main challenge we face to approach in such a direction is how to estimate effective payloads from evaluated ones. This includes at least two aspects:

1) We need to find a suitable distance metrics, in which the distribution of payloads matches our perspective that effective ones cluster in the payload collection.
2) We need to find an effective algorithm to select candidate payloads for evaluation, so as to pick out promising ones without involving in too much computation cost.

In the next section, we will elaborate on our method.

## III. Our Method: ART4SQLi

In this section, we first illustrate the process of our method ART4SQLi and key algorithms, elaborate on a distance metrics needed by the algorithm, and give research questions.

### A. Overview of ART4SQLi

One aim of ART [7], [10] is to find the first test case causing the failure more quickly than the ordinary random testing. The process of ART4SQLi is inspired by ART. Fig. 4[4] depicts an overview of the process of ART4SQLi. As mentioned in the previous sections, we focus on scheduling the prioritization of payloads for evaluation by iteratively selecting more promising payload candidates.

To start such a process, we assume that a collection of payload is ready. We refer to it as the *payload collection*, and use it as

[4]The term “evaluated set” of this paper and the term “executed set” of ART techniques have the same meaning.

**Algorithm 1:** FSCS-Fixed Size Candidate Selection.

```
Input: Payload collection: $PC$
Input: Evaluated set: $ES$
Input: Last payload: $p_{\text{last}}$
Output: Next payload: $p$
1: $ES \leftarrow ES \cup \{p_{\text{last}}\}$          Step 4a ⌝
2: $CS \leftarrow \emptyset$                             Step 4b ⌝
3: for $i = 1$ to FixedSize do
4:     randomly pop $q$ from $PC$
5:     $CS \leftarrow CS \cup \{q\}$
6: end for                                               ⌟
7: $p \leftarrow$ FNC$(CS, ES)$                          Step 4c ⌝
8: $PC \leftarrow PC \setminus p$
9: return $p$                                            ⌟
```

input to ART4SQLi. Such a candidate selection process has four steps, as illustrated in the figure.

*Step 1:* Select the first payload $p$.
*Step 2:* Evaluate the selected payload $p$.
*Step 3:* End if any successful injection, otherwise $Step$ 4.
*Step 4:* Select the next payload $p$, and goto $Step$ 2.

More specifically, $Step$ 1 randomly selects a payload from the payload collection, since we have no information for reference at the very beginning. In $Step$ 2, the selected payload is submitted to test a target Web application through the access entries of that application. We judge whether any SQLi vulnerability has been revealed by checking the response of the Web application. The process completes at $Step$ 3 when an injection succeeds; otherwise we come to $Step$ 4 to select the next payload. The next payload will be selected from the payload collection based on a heuristic algorithm (given as Algorithm 1) with the aim to find a payload far from all the evaluated ones. The selected payload will be forwarded to $Step$ 2 for evaluation. In particular, when there is no available payload in the payload collection (e.g., all the candidates are exhausted, revealing no SQLi vulnerability), the process ends at $Step$ 4.

To ease explanation, we refer to $Step$ 2 as *Payload Evaluation*, $Step$ 3 as *Payload Examination*, and $Step$ 4 as *Payload Selection*, in the rest of this paper.

### B. Step 4: Payload Selection

ART is based on the intuition that an even spread of test cases is more likely to detect failures using fewer test cases than ordinary random testing [10]. It considers the distance between the previous test cases and the candidate set to select the next test case(s). Accordingly, we design the payload selection of ART4SQLi. As illustrated in Fig. 4, $Step$ 4 further consists of three sub-steps.

*Step 4a:* Collect the last payload.
*Step 4b:* Generate a candidate set.
*Step 4c:* Select the next payload $p$.

The pseudo code fixed size candidate selection (FSCS) implementing $Step$ 4 is given in Algorithm 1, where line 1 performs $Step$ 4a, lines 2–6 perform $Step$ 4b, and lines 7–9 perform $Step$ 4c. More specifically, $Step$ 4a maintains a set of all the

**Algorithm 2:** FNC - Farthest Nearest Candidate.

```
Input: Candidate set: $CS$
Input: Evaluated set: $ES$
Output: Next payload: $p$
 1: $p \leftarrow null,\ \ max\_dist \leftarrow -\infty$
 2: for all $u \in CS$ do
 3:     $min\_dist \leftarrow +\infty$
 4:     for all $v \in ES$ do
 5:         $dist \leftarrow \mathcal{D}istance(u, v)$
 6:         $min\_dist \leftarrow \min\{min\_dist, dist\}$
 7:     end for
 8:     if $max\_dist < min\_dist$ then
 9:         $p \leftarrow u,\ \ max\_dist \leftarrow min\_dist$
10:     end if
11: end for
12: return $p$
```

executed payloads. We refer to it as an *evaluated set*. In $Step$ 4b, we randomly generate a fixed-sized subset of the payload collection. It is referred to as a *candidate set*. On line 3, **FixedSize** indicates a global parameter, which is the size of the candidate set. After that, $Step$ 4c selects a payload $p$ from the candidate set, satisfying that $p$ is the one having the maximum distance from the evaluated set. Note that $PC$ and $ES$ are also global parameters, which are initialized to be the set of payload collection and empty, respectively. Every time a payload is selected and evaluated, it is removed from $PC$ and added to $ES$, until any SQLi vulnerability is revealed.

The FSCS algorithm is based on our observation that the effective payloads tend to cluster together. As a result, if a set of previously evaluated payloads have not revealed any SQLi vulnerability, we need to select payloads far from them, with the aim to increase the possibility to reveal failures [12].

A detailed process of selecting $p$ from the candidate set (line 7 of Algorithm 1) is given using the pseudo code FNC in Algorithm 2. Line 2 iterates each candidate in the candidate set, lines 3–7 calculate the minimum distance min_dist of the candidate by visiting every evaluated payloads, and lines 8–10 select the most promising candidate, which maximizes min_dist. Finally, line 12 returns the selected payload that has the maximized minimum distance. Such a distance is also formulated as follows:

$$\max_{u \in CS} \left\{ \min_{v \in ES} \{\text{Distance}(u, v)\} \right\}. \tag{1}$$

Line 5 in Algorithm 2 involves a distance formula Distance. In the next section, we elaborate on how we use it to calculate the distance of payloads.

## C. Payload Distance—$\mathcal{D}$istance

The selection of distance metrics is crucial to our method. First, we will introduce the grammar used to understand a payload string. We next state how we extract feature vector from the payload string. Finally, we employ a cosine distance to calculate the distance between two feature vectors.

$$\begin{aligned}
payload ::=\ & \langle num\_payload\rangle \mid \langle s\_quote\_payload\rangle \\
& \mid \langle d\_quote\_payload\rangle\ ; \\
num\_payload ::=\ & \langle ZERO\rangle,\ \langle wsp\rangle,\ \langle b\_payload\rangle,\ \langle wsp\rangle \\
& \mid \langle ZERO\rangle,\ \langle parR\rangle,\ \langle wsp\rangle,\ \langle b\_payload\rangle,\ \langle wsp\rangle, \\
& \langle opOR\rangle,\ \langle parL\rangle,\ \langle ZERO\rangle \\
& \mid \langle ZERO\rangle,\ \langle wsp\rangle,\ \langle sqli\_payload\rangle,\ \langle cmt\rangle\ ; \\
s\_quote\_payload ::=\ & \langle squote\rangle,\ \langle wsp\rangle,\ \langle b\_payload\rangle,\ \langle wsp\rangle, \\
& \langle opOR\rangle,\ \langle squote\rangle \\
& \mid \langle squote\rangle,\ \langle parR\rangle,\ \langle wsp\rangle, \langle b\_payload\rangle,\ \langle wsp\rangle, \\
& \langle opOR\rangle,\ \langle parL\rangle,\ \langle squote\rangle \\
& \mid \langle squote\rangle,\ [\langle parR\rangle],\ \langle wsp\rangle, \langle sqli\_payload\rangle,\ \langle cmt\rangle; \\
d\_quote\_payload ::=\ & \langle dquote\rangle,\ \langle wsp\rangle,\ \langle b\_payload\rangle,\ \langle wsp\rangle,\ \langle opOR\rangle, \\
& \langle dquote\rangle \\
& \mid \langle dquote\rangle,\ \langle parR\rangle,\ \langle wsp\rangle,\ \langle b\_payload\rangle,\ \langle wsp\rangle, \\
& \langle opOR\rangle,\ \langle parL\rangle,\ \langle dquote\rangle \\
& \mid \langle dquote\rangle,\ \langle parR\rangle,\ \langle wsp\rangle,\ \langle sqli\_payload\rangle,\ \langle cmt\rangle \\
sqli\_payload ::=\ & \langle u\_payload\rangle \mid \langle piggy\_payload\rangle \mid \langle b\_payload\rangle \\
& \mid \langle IF\rangle,\ \langle parL\rangle,\ \langle true\_expr\rangle,\ \langle parC\rangle,\ \langle time\_payload\rangle, \\
& \langle parC\rangle,\ \langle false\_expr\rangle,\ \langle parR\rangle; \\
b\_payload ::=\ & \langle opOR\rangle,\ \langle wsp\rangle,\ \langle b\_expr\rangle \mid \langle opAND\rangle,\ \langle wsp\rangle, \\
& \langle b\_expr\rangle \mid \ldots\ ; \\
time\_payload ::=\ & \langle sleep\_payload\rangle \mid \langle benchmark\_payload\rangle\ ; \\
b\_expr ::=\ & \langle true\_expr\rangle \mid \langle false\_expr\rangle\ ; \\
true\_expr ::=\ & \langle eq\_true\_expr\rangle \mid \langle gt\_true\_expr\rangle \mid \ldots; \\
eq\_true\_expr ::=\ & \langle squote\rangle,\ \langle char\rangle,\ \langle squote\rangle,\ \langle opEQ\rangle,\ \langle squote\rangle, \\
& \langle char\rangle,\ \langle squote\rangle \mid \ldots\ ; \\
sq\_quote ::=\ & ' \mid \%27 \mid \ldots\ ; \\
wsp ::=\ & \sqcup \mid /**/ \mid \ldots\ ; \\
opOR ::=\ & or \mid || \mid \ldots\ ; \\
opEQ ::=\ & =\ ; \\
cmt ::=\ & \# \mid -- \mid \ldots\ ; \\
& \ldots
\end{aligned}$$

Fig. 5. Syntax of payload [4].

*1) Decomposition of a Payload String:* In our method, we base on the CFG for SQLi proposed by Appelt *et al.* [4] to design our own grammar. In order to incorporate payload forms like time-based blind SQLi,[5] we extend their CFG by adding new grammar components. The final specific grammar definition is given in Fig. 5. The SQLi payload grammar is defined by using extended backus normal form ([63]), in which payload is the start symbol, "::=" is the production symbol, "," means concatenation, and "|" represents alternatives. Note that this is an excerpt of the complete grammar.

A parse tree (aka. parsing tree) is an ordered, rooted tree that represents the syntactic structure of a string according to a CFG. In our method, each SQLi payload is a malicious attack string, which can be parsed by an SQLi parser using a corresponding SQLi derivation tree. On the basis of the grammar proposed in Fig. 5, we parse a payload string into such a tree structure [4]. In the derivation tree, the root node is the start variable. All the internal nodes are labeled with variables, and all the leaves are labeled with terminals [63]. We thus use such a derivation tree as a graphical representation of an SQLi payload string. The leaf nodes of the derivation tree are called the token of the parsed payload (i.e., the minimal indecomposable part for the parsed payload).

[5]For time-based attacks, the attacker needs to instruct the database to perform a time-intensive operation. If the website does not return a response immediately, the Web application is vulnerable to SQLi.

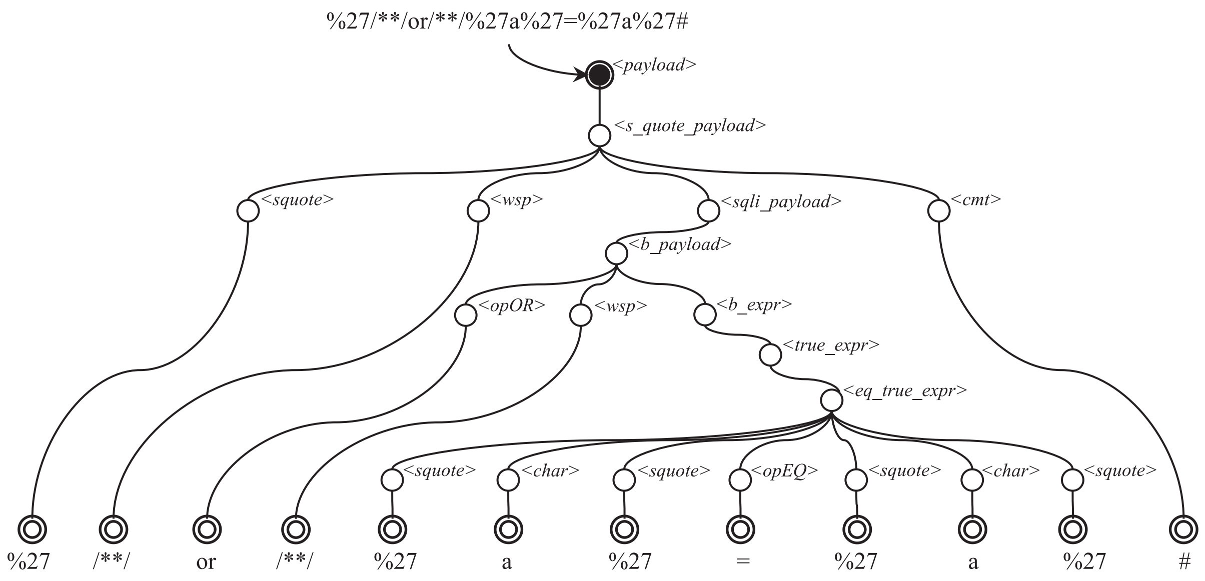

Fig. 6. Decomposition example of a payload string.

We employ the open source tools *flex* and *bison* to compose the parser. The process for parsing specific SQLi payload strings by applying the derivation tree is referred to as the *String Decomposition* process. In this process, payload strings are loaded into the SQLi derivation tree, and the output consists of a set of ordered tokens according to the position where each token is located in the corresponding payload string. The decomposition result for each payload string is called a *decomposition* of that payload, in this paper.

Fig. 6 depicts the parsing process of the single quote attack payload "`%27/**/or/**/%27a%27=%27a%27#`". The parse tree first identifies that the given string is a *single quote payload* ($\langle s\_quote\_payload \rangle$). It then uses the corresponding form to parse it hierarchically until all the terminal nodes are indecomposable. In the next section, we introduce how to vectorize a decomposition of payload.

*2) Feature Extraction:* The feature extraction process is regarded as a vectorization. For each payload $p$, we count the frequency of each token contained in the decomposition of $p$, and generate a feature vector $\mathbf{v}_p$ for $p$.

In the process, we calculate their entropy to extract a feature vector from a decomposition of payload [52]. For each decomposition of a payload string, we apply an abstraction process to generate a quantified vector to represent it. We refer to this output vector as a *feature vector* of the payload. For a payload $p$, its vectorized presentation is denoted as vector $\mathbf{v}_p = \left[w_p^1, w_p^2, \ldots, w_p^\kappa\right]^{\mathrm{T}}$. Here, $\mathbf{v}_p$ is a $\kappa$-dimension vector, where $T$ is the collection of decomposed tokens of all payloads, and $\kappa$ is the size of $T$. Each dimension of the vector $\mathbf{v}_p$ is called a *token weight* (also *weight*). It is calculated using (2), and corresponds to a unique $t$ token in $T$

$$w_p^i = \frac{\log\left(F_p^i + 1.0\right) \times \log\left(\frac{\kappa}{N^i}\right)}{\sqrt{\sum_{i=1}^{\kappa}\left[\log\left(F_p^i + 1.0\right) \times \log\left(\frac{\kappa}{N^i}\right)\right]^2}} \tag{2}$$

where $F_p^i$ is the frequency of the token $t_i$ in the decomposition of the payload $p$, and $N^i$ is the number of payload samples whose decomposition contain the token $t_i$. ART4SQLi thus generate a $k$-dimension vector $\mathbf{v}_p$, which is used as the feature vector for $p$. In the next section, we will introduce how to calculate the distance between two vectors.

*3) Distance Calculation:* The distance metrics used to measure the distance of two payloads are based on their vector presentation proposed in previous section.

Suppose we are given two payloads $p$ and $q$, which have feature vectors $\mathbf{v}_p = [w_p^1, w_p^2, \ldots, w_p^\kappa]^{\mathrm{T}}$ and $\mathbf{v}_q = \left[w_q^1, w_q^2, \ldots, w_q^\kappa\right]^{\mathrm{T}}$, respectively. We define the distance $\mathcal{D}\mathrm{istance}(p, q)$ of $p$ and $q$ by using the cosine similarity measurement [75]. The calculation is given below

$$\begin{aligned}\mathcal{D}\mathrm{istance}(p, q) &= \left(\frac{\mathbf{v}_p \cdot \mathbf{v}_q}{||\mathbf{v}_p|| \cdot ||\mathbf{v}_q||}\right)^{-1} \\ &= \left(\frac{\sum_{i=1}^{\kappa}\left(w_p^i \cdot w_q^i\right)}{\sqrt{\sum_{i=1}^{\kappa}\left(w_p^i\right)^2} \cdot \sqrt{\sum_{i=1}^{\kappa}\left(w_q^i\right)^2}}\right)^{-1}.\end{aligned} \tag{3}$$

The result of (3) is in the range $[1, +\infty)$. A distance calculated as 1 indicates two identical vectors. A distance calculated as $+\infty$ indicates two orthogonal vectors. An in-between value indicates intermediate distances.

## D. Complexity

We first analyze the time complexity of $Step$ 4 in Fig. 4. We suppose that $Step$ 4 is repeated for $F$ times (it will be named as "*F-Measure*" later in Section IV-E) until finding the first effective payload. Line 1 of Algorithm 1 accumulates each evaluated payload to form an evaluated set $ES$. $ES$ thus increases its size over the iterative process, and finally has $F$ payloads. Lines 3–6 of Algorithm 1 show that we select a payload $q$ from the

TABLE II
EXPERIMENTAL SQLI SUBJECTS

| | Tactic Limitations | SQLi01 | SQLi02 | SQLi03 | SQLi04 | SQLi05 | SQLi06 | SQLi07 | SQLi08 |
|---|---|---|---|---|---|---|---|---|---|
| Web for Pentester | whitespace | | + | + | + | + | + | + | |
| | `\s` | | | + | + | + | + | + | |
| | `'` | | | | + | + | + | + | |
| | `^\d+` | | | | | + | + | + | |
| | `\d+$` | | | | | | + | + | |
| | encoding | | | | | | | + | |
| DVWA 2014 | `or` | | + | + | + | + | + | + | |
| | `order` | | | + | + | + | + | + | |
| | `union` | | | | + | + | + | + | |
| | `database()` | | | | | + | + | + | |
| | table name | | | | | | + | + | |
| | encoding | | | | | | | + | |
| MCIR-SQLol | whitespace | | + | + | + | + | + | + | + |
| | `union` | | | + | + | + | + | + | + |
| | `'` | | | | + | + | + | + | + |
| | `limit` | | | | | + | + | + | + |
| | error injection | | | | | | + | + | + |
| | blind injection | | | | | | | + | + |
| | stacked query | | | | | | | | + |
| | | SQLi09 | SQLi10 | SQLi11 | SQLi12 | SQLi13 | SQLi14 | SQLi15 | |
| | `delete` | | + | + | + | + | + | + | |
| | capital | | | + | + | + | + | + | |
| | `update` | | | | + | + | + | + | |
| | `where` | | | | | + | + | + | |
| | xssql | | | | | | + | + | |
| | `like` | | | | | | | + | |

payload collection for each iteration and repeat for **FixedSize** times. Candidate set $CS$ thus has **FixedSize** payloads and the time complexity of lines 3–6 of Algorithm 1 is $O(\mathbf{FixedSize})$. The lines 2 and 4 of Algorithm 2 traverse $CS$ and $ES$, respectively. As a result, the time complexity of Algorithm 2 is $O(F \cdot \mathbf{FixedSize}) \cdot O_D$, where $O_D$ is the complexity of the $\mathcal{D}\mathrm{istance}$ function in (3). We further know that the time complexity of Algorithm 1 is $O(F \cdot \mathbf{FixedSize}) \cdot O_D$.

Since $Step$ 4 is repeated for $F$ times, the time complexity of ART4SQLi is $O(F^2 \cdot \mathbf{FixedSize}) \cdot O_D$.

In the worst case, the whole payload collection is exhausted to find an effective payload. The time complexity in the worst case is $O(N^2 \cdot \mathbf{FixedSize}) \cdot O_D$, where $N$ is the size of payload collection.

### *E. Research Questions*

We want to answer the following questions.

*Q1;* Do the effective payloads cluster together in the payload space described using the vector representation proposed in Section III-C2?

*Q2:* Will the scheduling method proposed in Section III-B more effectively find effective payloads, than the original random manner?

*Q3:* Is the efficiency of the ART4SQLi process proposed in Section III-A acceptable in practice?

$Q1$ will validate our experiences that effective payloads are rare in the payload space. It will also evaluate the soundness of the selection of our vectorization method. Note that with a well-selected vectorization, the effective payloads are expected to cluster together in the payload space. Only in that case, our proposal to select a promising payload by maximizing its distance to the evaluated payloads makes sense. If the answer to $Q1$ is yes, $Q2$ uses an empirical study to evaluate the effectiveness of the proposal. If the answer to $Q2$ is yes, $Q3$ further finds out whether the computation cost is acceptable. We will answer these questions in the following sections.

## IV. EVALUATION

In this section, we will introduce the experiment setup, including payload collection preparation, benchmark selection, experiment steps, and effectiveness measurement. After that, we will give the experiment results and discuss threats to validities of the empirical observations.

### *A. SQLi Benchmarks*

In the experiment, we use three open source vulnerable Web applications as the units under tests. They are *Web for Pentester I*, *Wooyun DVWA 2014*, and *SpiderLab MCIR-SQLol*, which are listed in Table II. They are well known as web vulnerability testbeds [53], [71], which contain many intentionally embedded representative Web issues, including SQLi flaws. We will test the SQLi vulnerable pages within these three testbeds.

*Web for Pentester I* includes seven pages, each of which has SQLi issues of different security levels. For example, the first page named "SQLi01" in this experiment can be easily injected by common attack payloads, while the second page "SQLi02" is

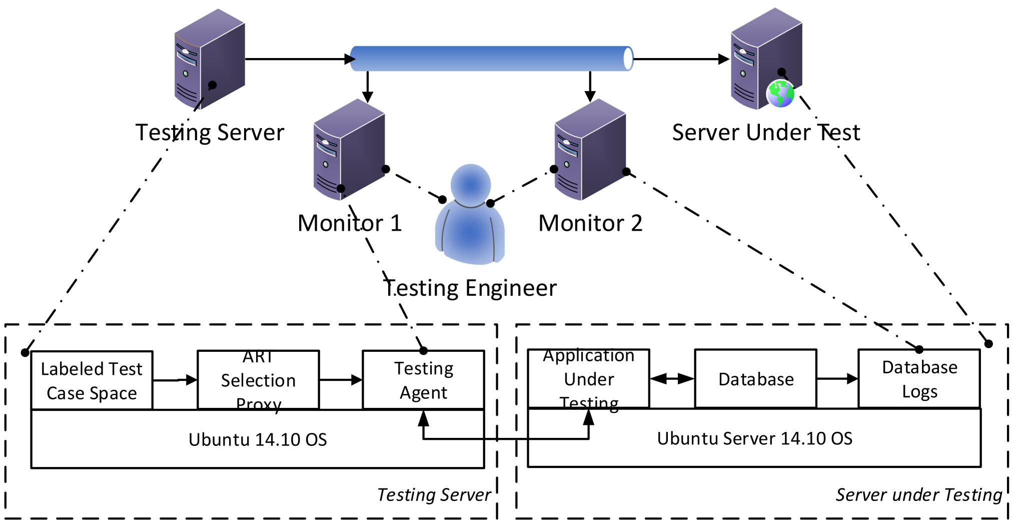

Fig. 7. Testing bed illustration.

equipped with whitespace validation to limit the use of whitespace attacking tactics. The last page "SQLi07" has the most strict checking mechanism. Similarly, *Wooyun DVWA 2014* has seven pages for testing, which are different from those of *Web for Pentester I* in order to simulate different SQLi vulnerabilities. *SpiderLab MCIR-SQLol* has two groups of pages, SQLi01-08 and SQLi09-15, which are dedicated for clause-related SQLi.

## B. Testing Bed

As showed in Fig. 7, the constructed testing bed to carry out the experiment is composed of a testing server, a server under test (SUT), and two monitors. The operating system of the SUT is a Ubuntu Server 14.10, which runs an Apache Web Service and a MySQL database. The testing server is also a Ubuntu 14.10 build host, which runs our method ART4SQLi together with a testing agent.

ART4SQLi performs the candidate selection process, and forwards the selected payloads to the testing agent. We use *burpsuite* as the testing agent, which receives candidate payloads from ART4SQLi, packs them into HTTP packets, and requests SUT by using the packed HTTP packets. When it receives a payload from ART4SQLi, it replaces a parameter in the HTTP request package with the payload. The replaced parameter has been previously defined, and can be located at URL section, refer section, user-agent section, and other sections. For instance, "`tester`" is such a parameter for these benchmarks. To test a page, we use the query in the URL format "`http://test.com/sqli.php?uname=tester`" to submit a payload. We will locate "`tester`" in the URL, and then replace it with a payload string. For example, during the process, when the testing agent *burpsuite* receives the payload "`%27/**/or/**/%27a%27=%27a%27#`", it replaces the predefined parameter "`tester`" with the payload "`tester%27/**/or/**/%27a%27=%27a%27#`". As a result, the new query is "`http://test.com/sqli.php?uname=%27/**/or/**/%27a%27=%27a%27#`".

TABLE III
PREPARATION OF THE PAYLOAD COLLECTION

| Name | Site | Number of Payloads |
|---|---|---|
| fuzzdb | `github.com/fuzzdb-project` | 74595 |
| sqlmap | `www.sqlmap.org` | 3482 |
| wooyun | `wooyun.org` | 21384 |
| OWASP | `www.owasp.org.cn` | 893 |
| exploit-db | `www.exploit-db.com` | 315 |
| GreySec | `www.greysec.net` | 796 |
| Total | | 101465 |

The subjects selected in our experiment have defined the response pages that indicate successful injections. Once an SQLi is triggered successfully, *Web for Pentester I* will reply with a page containing a list of users and passwords in its predefined database. Similarly, *Wooyun DVWA 2014* will reply with a page containing the input and a list of user information, once an SQLi is triggered successfully. And *SpiderLab MCIR-SQLol* will reply with a success notification along with user information using a response page. These predefined page formats of each subject are used as oracles to help determine whether a payload has successfully injected a subject.

The two monitors are responsible for logging and judging the testing results. Monitor 1 is used to record the requesting information from the testing agent, and Monitor 2 is used to log the database operations. Both the two sets of information are processed by our testing scripts to determine whether the requests has successfully discovered an SQLi vulnerability.

## C. Payload Collection Preparation

Like most SQLi discovery technologies, ART4SQLi requires an original payload collection, which is located on the testing server. In our experiment, the sources used to prepare the original payload collection are described in Table III.

First, we extract the injection part payloads from the sources (e.g., *fuzzdb*) into a local file. We then use the tool *sqlmap tamper* to mutate the extracted payloads. After that, we remove duplicate mutations, and obtain a set of distinct payloads. All the remaining payloads form the original payload collection.

In particular, our payload collection also include some time-based blind payloads. They do not trigger any injection response page. Their testing results are judged according to the difference of response time, which is described in the next section.

### D. Experiment Steps

To evaluate the effectiveness of ART4SQLi, we follow previous work [16], [17], [48], [49], [81] to use the standard random manner (where a payload is selected by random in each iteration) as a peer strategy for comparison. Our considerations are as follows:

1) ART4SQLi is proposed in an ART manner, which is naturally against the standard random strategy.
2) Using the random strategy as a baseline reflects the basic capability of a technique.

In the rest of this section, we refer to the standard random manner as Random consistently.

In our experiments, we create testing oracles using the mechanisms described in Section IV-A. More specifically, in the testing process, the monitors will trigger a pause when either of the following situations happens: 1) Monitor 1 receives pages containing a predefined message or characters from a subject. 2) Monitor 2 detects a database SQL execution log, and monitor 1 does not receive a response page after a predefined threshold time length. In the latter case, we deem that a time-based blind SQLi has happened. For a time-based blind SQLi, we instruct the database to perform a time-intensive operation. If there exists a database SQL execution log that is detected by Monitor 2, it means that the payload has injected into the Web application successfully.

In practice, after a testing process is paused, the current request packet and the corresponding payload will be sent to the testing engineer, together with the monitor logs. The engineer judges whether the current test flags an SQLi vulnerability. If the answer is yes, the process completes; otherwise it continues until an effective payload is found (recalling Section III-A). In our experiment, this step is fully automatic.

In the experiment, we let the global parameter **FixedSize** to be a suggested value 10 according to [10]. In addition, we store the feature vectors and their corresponding payload strings into a global hash table for acceleration purpose.

In our experiments, we separately carry out tests on the three subjects. On each vulnerable page of the three subjects, we perform Random and ART4SQLi independently, record their effectiveness, repeat the testing process for ten times, and calculate the average results. The termination criterions of a test session between ART4SQLi and Random are consistent. They will always iterate the next payload until the first effective payload is found or all payloads have been exhausted.

### E. Measurement

This section will introduce the metrics used to answer the three research questions.

*1) Distribution Metric:* To validate the sparse distribution of effective payloads, we are inspired by [10] to use an *E-measure* metrics. In this paper, the metrics *E-measure* returns the number of effective payloads in a payload collection. A smaller *E-measure* means a more sparse distribution of effective payloads.

In this experiment, we make statistics for the effective payloads in the whole payload collection. However, in practice, it cannot be feasible to visit all payloads consecutively, especially for a large payload collection. Furthermore, we use the $\mathcal{D}\mathrm{istance}$ defined in (3) to calculate the average distance among effective payloads (referred to as *Intra-Class Distance*), and the average distance from effective payloads to ineffective payloads (referred to as *Interclass Distance*). The two distances on average are used to validate whether effective payloads cluster together in the payload space. A small *Intra-Class Distance* and a large *Interclass Distance* will mean that the effective payloads tend to cluster together in the payload space.

*2) Effectiveness Metric:* During the process of evaluating each payload returned by ART4SQLi, we know an effective payload is found, and accordingly ART4SQLi is terminated, when an SQLi succeeds. The process of Random is similar. As a result, the less the payloads evaluated, the better the effectiveness of an algorithm is. To measure the effectiveness of ART4SQLi and Random, we follow [10] to use the *F-measure* metrics. *F-measure* calculates how many payload are executed until the first effective payload is found. A lower *F-measure* value means that fewer tests are used to accomplish the task.

The *F-measure* metric also reflects the true/false positive/negative concepts in classification. ART4SQLi searches for payloads to discover SQLi vulnerability, so we mark a payload as *effective* if it reveals any failures in a Web application. The process of ART4SQLi always stops at the first such effective payload. We thus count the first effective payload as a *true-positive* sample. For the same reason, we count all the payloads evaluated ahead of the first effective one as *false-positive* samples, since they are given higher suspiciousness by ART4SQi but evaluated ineffective. In such a way, we let *F-measure* reflect the extents of true positive and false positive properties. Suppose ART4SQLi terminates with the $m$th ($>= 1$) executed payloads (which are evaluated effective), we also say we encounter $m - 1$ false-positive and 1 true-positive. Note that ART4SQLi stops evaluating the payloads after the $m$th. As a result, we get no suggestion for a true negative or false negative.

For tests on each page of each subjects, we calculate the *F-measure* of ART4SQLi and Random separately. Since a testing strategy yielding a lower *F-measure* value is considered to be more effective, we expect a lower *F-measure* value for ART4SQLi than that of Random. We thus calculate $\frac{R-A}{R} \times 100\%$ to evaluate the effectiveness improvement, where $R$ and $A$ stand for the effectiveness of Random and ART4SQLi in *F-Measure*, respectively. Such an equation measures the improvements from Random to ART4SQLi. The greater the value is, the more effective ART4SQLi is than Random.

*3) Efficiency Metric:* To evaluate the practicability of our proposal, we simply record the time used in each phase.

Since we report each test for ten times, we further compute the average time used to complete each testing task. Thus, the result is present as the average time spent to find out the first

TABLE IV
E-MEASURE, INTRA-CLASS, AND INTER CLASS DISTANCE OF PAYLOADS

| | | SQLi01 | SQLi02 | SQLi03 | SQLi04 | SQLi05 | SQLi06 | SQLi07 | SQLi08 | *Average* |
|---|---|---|---|---|---|---|---|---|---|---|
| **E-Measure** (%) | Web for Pentester | 1.58% | 0.53% | 0.48% | 0.59% | 0.04% | 0.42% | 0.48% | - | 0.58% |
| | Wooyung DVWA | 0.39% | 0.37% | 0.46% | 0.53% | 0.49% | 0.51% | 0.06% | - | 0.40% |
| | MCIR-SQLol | 0.56% | 0.59% | 0.53% | 0.67% | 0.49% | 0.47% | 0.48 % | 0.43% | |
| | | SQLi09 | SQLi10 | SQLi11 | SQLi12 | SQLi13 | SQLi14 | SQLi15 | | |
| | | 0.61% | 0.52% | 0.55% | 0.49% | 0.54% | 0.52% | 0.51% | | 0.53% |
| | *Average* | | | | | | | | | 0.51% |
| | | SQLi01 | SQLi02 | SQLi03 | SQLi04 | SQLi05 | SQLi06 | SQLi07 | SQLi08 | *Average* |
| **Intra**-class Distance | Web for Pentester | 14.28 | 14.56 | 15.02 | 17.85 | 21.36 | 16.32 | 19.45 | - | 15.97 |
| | Wooyung DVWA | 16.23 | 15.32 | 18.21 | 15.87 | 17.62 | 15.89 | 20.16 | - | 17.32 |
| | MCIR-SQLol | 14.18 | 12.66 | 13.45 | 14.02 | 14.32 | 13.56 | 12.59 | 13.05 | |
| | | SQLi09 | SQLi10 | SQLi11 | SQLi12 | SQLi13 | SQLi14 | SQLi15 | | |
| | | 13.57 | 14.24 | 13.91 | 12.88 | 13.02 | 14.15 | 13.68 | | 13.55 |
| | *Average* | | | | | | | | | 15.04 |
| | | SQLi01 | SQLi02 | SQLi03 | SQLi04 | SQLi05 | SQLi06 | SQLi07 | SQLi08 | *Average* |
| **Inter**-class Distance | Web for Pentester | 68.13 | 67.22 | 63.17 | 62.58 | 57.64 | 63.03 | 60.16 | - | 64.24 |
| | Wooyung DVWA | 65.24 | 62.78 | 60.12 | 63.17 | 62.74 | 61.53 | 58.62 | - | 62.17 |
| | MCIR-SQLol | 68.57 | 70.23 | 69.02 | 71.25 | 68.19 | 67.94 | 68.81 | 69.05 | |
| | | SQLi09 | SQLi10 | SQLi11 | SQLi12 | SQLi13 | SQLi14 | SQLi15 | | |
| | | 65.23 | 66.92 | 67.28 | 67.48 | 68.42 | 67.77 | 68.56 | | 68.32 |
| | *Average* | | | | | | | | | 65.85 |

effective payload toward a specific vulnerable page. The timing issue consists of two parts, namely, the time spent to generate the candidate payloads, and the time used to execute and reach an effective payload.

All the experiment results are summarized and reported in the next section.

## F. Experiment Results

We report our experiment results in three sections to answer the three research questions, respectively.

*1) Answering Q1:* The distribution of effective payloads on a subject benchmark is evaluated using the *E-measure* metrics, the *intra-class distance*, and the *interclass distance*. These results with respect to the three subjects *Web for Pentester I*, *Wooyun DVWA 2014*, and *SpiderLab MCIR-SQLol* are shown in Table IV.

Table IV consists of three parts, which report the experiment results in the three metrics, respectively. We first come to the "E-measure (%)" part. It shows the results of applying the metrics *E-measure* to calculate the *percentage* of effective payloads with respect to each vulnerability page of each benchmark. To do that, we evaluate each payload in the payload collection, repeat the process on each vulnerability page, and record the data. Take the first cell as example. The number 1.58% indicates that for the vulnerability page SQLi01 of the benchmark *Web for Pentester*, there are 1604 effective payloads in the whole payload collection. The second number 0.53% in the same row indicates that there are 538 (less than 1604, with respect to SQLi01, since the SQLi02 case has more strict validation mechanism) effective payloads for page SQLi02. The last number (in the *Average* column) in the same row shows that on average there are 650 (0.64%) effective payloads for *Web for Pentester I*. The other rows can be similarly interpreted. We find that there are on average 429 (0.40%) and 538 (0.53%) effective payloads for *Wooyun DVWA 2014* and *SpiderLab MCIR-SQLol*, respectively. We further know that on average there are 517 (0.51%) effective payloads for a vulnerability page. Our basic impression is that the effective payloads expose a small portion in the payload space.

The second part "intra-class distance" of Table IV reports the on average distance of two effective payloads, with respect to each vulnerability page. To get the results, we calculate the mean pair-wise distance for the ten effective payloads (recalling that we repeat our tests for ten times and find one effective payload each time) of each vulnerability page, by averaging the distance of $\frac{10\times 9}{2} = 45$ pairs of effective payloads. The third part "interclass distance" of Table IV reports the on average distance of effective payloads to ineffective payloads, with respect to each vulnerability page. To get the results, we calculate a mean pair-wise distance between effective payloads (ten in total) and ineffective payloads (collected from all the ten tests) of each vulnerability page, by averaging the distance with respect to all the effective–ineffective pairs. Take the same page for example, we find that such an intra-class distance and the interclass distance for the page SQLi01 of *Web for Pentester* are 14.28 and 68.13, respectively. It means that the average distance of two effective payloads for SQLi01 is 14.28, and the average distance between an effective payload and an ineffective payload is 68.13. By comparing the two numbers, we say that the effective payloads for SQLi01 tend to cluster together in the payload space.

TABLE V
PAIR-WISE COMPARISON OF ART4SQLI AND RANDOM ON EACH INDIVIDUAL SUBJECTS

| | | SQLi01 | SQLi02 | SQLi03 | SQLi04 | SQLi05 | SQLi06 | SQLi07 | SQLi08 | *Average* |
|---|---|---|---|---|---|---|---|---|---|---|
| **ART4SQLi (A)** | Web for Pentester | 264.2 | 401.3 | 690.2 | 989.8 | 698.7 | 717.6 | 1187.5 | - | 707.04 |
| | Wooyung DVWA | 2022.7 | 690.7 | 829.8 | 647.10 | 736.1 | 888.5 | 501 | - | 902.27 |
| | MCIR-SQLol | 201.10 | 194.50 | 196.80 | 186.40 | 244.20 | 214.50 | 237.10 | 212.70 | |
| | | SQLi09 | SQLi10 | SQLi11 | SQLi12 | SQLi13 | SQLi14 | SQLi15 | | |
| | | 382.10 | 216.70 | 213.20 | 219.50 | 247.20 | 368.30 | 525.60 | | 257.33 |
| | *Average* | | | | | | | | | 521.55 |
| | | SQLi01 | SQLi02 | SQLi03 | SQLi04 | SQLi05 | SQLi06 | SQLi07 | SQLi08 | *Average* |
| **Random (R)** | Web for Pentester | 329.3 | 524.7 | 901.9 | 1216.7 | 806.2 | 978.8 | 1631.4 | - | 912.71 |
| | Wooyung DVWA | 3066.1 | 990.9 | 1148.3 | 890.8 | 1081.2 | 1269 | 605.9 | - | 1293.17 |
| | MCIR-SQLol | 286.00 | 251.20 | 311.40 | 211.40 | 334.30 | 307.50 | 340.40 | 311.00 | |
| | | SQLi09 | SQLi10 | SQLi11 | SQLi12 | SQLi13 | SQLi14 | SQLi15 | | |
| | | 513.60 | 292.90 | 272.70 | 329.10 | 349.20 | 552.80 | 798.30 | | 364.12 |
| | *Average* | | | | | | | | | 720.79 |
| | | SQLi01 | SQLi02 | SQLi03 | SQLi04 | SQLi05 | SQLi06 | SQLi07 | SQLi08 | *Average* |
| $\frac{R-A}{R} \times 100\%$ | Web for Pentester | 19.77% | 23.52% | 23.47% | 18.65% | 13.33% | 26.69% | 27.21% | - | 21.81% |
| | Wooyung DVWA | 34.03% | 30.30% | 27.74% | 27.36% | 31.92% | 29.98% | 17.31% | - | 28.38% |
| | MCIR-SQLol | 29.69% | 22.57% | 36.80% | 11.83% | 26.95% | 30.24% | 30.35% | 31.61% | |
| | | SQLi09 | SQLi10 | SQLi11 | SQLi12 | SQLi13 | SQLi14 | SQLi15 | | |
| | | 25.60% | 26.02% | 21.82% | 33.30% | 29.21% | 33.38% | 34.16% | | 28.23% |
| | *Average* | | | | | | | | | 26.72% |
| | | SQLi01 | SQLi02 | SQLi03 | SQLi04 | SQLi05 | SQLi06 | SQLi07 | SQLi08 | *Average* |
| **std. ev.** | Web for Pentester | 72% | 116% | 97% | 51% | 48% | 22% | 48% | - | 65% |
| | Wooyung DVWA | 72% | 26% | 42% | 29% | 27% | 39% | 37% | - | 39% |
| | MCIR-SQLol | 54% | 27% | 19% | 11% | 6% | 22% | 2% | 14% | |
| | | SQLi09 | SQLi10 | SQLi11 | SQLi12 | SQLi13 | SQLi14 | SQLi15 | | |
| | | 23% | 7% | 5% | 15% | 45% | 25% | 10% | | 19% |
| | *Average* | | | | | | | | | 17% |
| | | SQLi01 | SQLi02 | SQLi03 | SQLi04 | SQLi05 | SQLi06 | SQLi07 | SQLi08 | *Average* |
| **p-value** | Web for Pentester | 0.0037 | 0.0038 | 0.0002 | 0.0011 | 0.0042 | 0.0021 | 0.0005 | - | 0.0020 |
| | Wooyung DVWA | 0.0006 | 0.0001 | 0.0003 | 0.0005 | 0.0017 | 0.0011 | 0.0001 | - | 0.0006 |
| | MCIR-SQLol | 0.0037 | 0.0035 | 0.0029 | 0.0031 | 0.0018 | 0.0031 | 0.0017 | 0.0025 | |
| | | SQL09 | SQL10 | SQL11 | SQL12 | SQL13 | SQL14 | SQL15 | | |
| | | 0.0009 | 0.0015 | 0.0024 | 0.0035 | 0.0005 | 0.0013 | 0.0005 | | 0.0022 |
| | *Average* | | | | | | | | | 0.0017 |

Furthermore, the average value of such a distance for *Wooyun DVWA 2014* and *SpiderLab MCIR-SQLol* are 17.32 and 13.55, respectively. On average, we observe an intra-class distance of 15.04 and an interclass distance of 65.85 for all the benchmarks.

Finally, we answer $Q1$ as follows:

$A1$: The effective payloads, with respect to each subject benchmark, expose a sparse distribution and tend to cluster together in the payload space.

*2) Answering Q2:* The effectiveness of ART4SQLi and Random in revealing SQLi vulnerabilities on a subject benchmark is evaluated using the *F-Measure* metrics. The experiment results of ART4SQLi and Random in *F-Measure* on *Web for Pentester I*, *Wooyun DVWA 2014*, and *SpiderLab MCIR-SQLol* are shown in Table V.

Table V consists of four parts. The first part "ART4SQLi (A)" lists out the effectiveness of ART4SQLi in *F-Measure*, for each page, each benchmark, and the whole experiment. The second part "Random (R)" lists out the effectiveness of Random in *F-Measure*. Let us take the first cell to illustrate to the contents. It shows that the *F-Measure* for ART4SQLi and Random on the SQLi01 page of *Web for Pentester I* are 264.2 and 329.3, respectively. It means that on average (recalling that ten individual tests are conducted for each page to avoid sample bias), 264.2 and 329.3 payloads needed to be evaluated before an SQLi

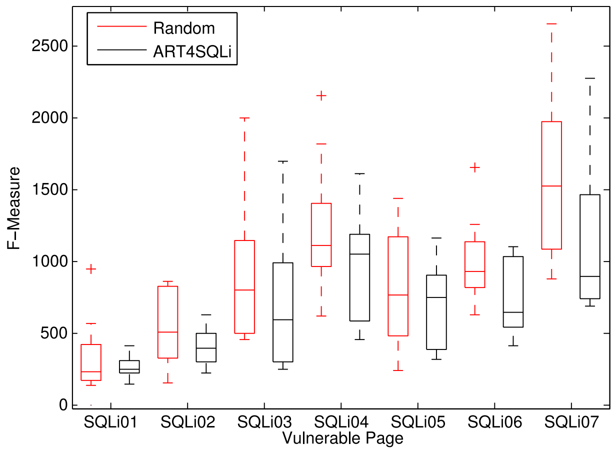

Fig. 8. Web for pentester.

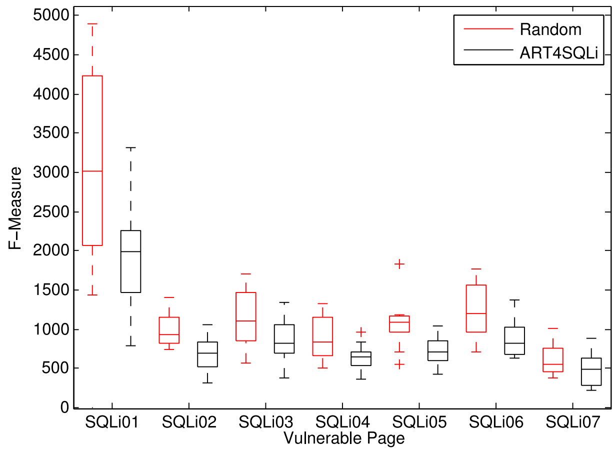

Fig. 9. Wooyung DVWA.

vulnerability is revealed, by ART4SQLi and Random, respectively. We further use the ratio $\frac{R-A}{R} \times 100\%$ to calculate the improvements from Random to ART4SQLi and report them in the third part "$\frac{R-A}{R} \times 100\%$" of the table. We say that ART4SQLi makes 19.77% improvements over Random on SQLi01 of *Web for Pentester I*. on average, such an improvements are 21.81%, 28.38%, and 28.23%, for the three benchmarks, and 26.72% on average. These data shows that ART4SQLi have an effective improvements over Random.

We next calculate the standard deviation of the ten improvements (in $\frac{R-A}{R} \times 100\%$) for each page, since we want to know whether such an improvement supports a stable observation. The results are shown in the "std. ev." part of the table. We find that the values are relatively high (in the range of $[22\%, 116\%]$). It means that though the effectiveness can be observed on average, they are not stably observed on each subject. We further want to know whether such improvements are statistically significant. We adopt the Wilcoxon signed rank test method, which is a nonparametric test for two populations when the observations are paired. The p-value of the Wilcoxon signed rank test can be used to measure the difference level between two populations via evaluating a hypothesis of no significant difference. Here, a p-value less than a chosen significance level (e.g., 0.05) indicates the rejection of the hypothesis, otherwise a failure to reject the hypothesis at the chosen significance level. From the "p-value" part of the table, we observe that the p-values of Wilcoxon signed rank test conducted with each page, each benchmark, and the whole experiment set are always less than 0.05. It means that the hypothesis is always rejected at the significance level of 0.05 and a statistical significance exists.

From Table V, we can find that on average ART4SQLi gives a roughly 26.72% improvement over Random. However, there are still counter-examples on which the improvements are less than 20%. We will further study them in Section IV-F4.

To give an intuitive view, we further use the box-whisker plots in Figs. 8–10 to represent the data. In these figures, the *X*-axis represents pairs of comparison on each page of each benchmark. The *Y*-axis is the effectiveness of ART4SQLi or Random in *F-Measure*, i.e., the number of payloads evaluated before revealing the first SQLi issue. Each column of box-whisker, no matter ART4SQLi or Random, depicts statistics of ten points (since each test is repeated for ten times in our experiment). The top of the upper whisker shows the maximum value in *F-Measure* among ten tests, while the bottom of the lower whisker shows the minimum value. The boundaries of the box correspond to the 75th percentile (upper quartile) and the 25th percentile (lower quartile) of ten points, with the median shown within the box. The difference of upper quartiles and lower quartiles is called IQR. Any results that are either 3IQR above the third quartile or 3IQR below the first quartile are outliers, and are labeled as "+" signs in the plots. Whiskers of each box extend to the minimal and maximum results within the non-outlier data.

From Figs. 8–10, we can see that in most cases, ART4SQLi (represented by the black boxes) outperforms Random (represented by the red boxes). Let us take the first pair of boxes in the first figure as example. It shows that on the page SQLi01 of *Web for Pentester*, ART4SQLi and Random can reveal an SQLi vulnerability by evaluating 135 and 185 payloads in the best case, 414 and 945 payloads in the worst case, and 264.2 and 329.3 payloads in a median level. The pairs of numbers are $\langle 221, 310\rangle$ and $\langle 168, 421\rangle$, by referencing the 25th and 75th percentile points. Such comparison reflects the effectiveness of our proposal and are consistent with our observations in Table V. In other words, ART4SQLi shows observable advantages over Random, in *F-Measure*. Similar results are observed with results of *Wooyun DVWA 2014* and *SpiderLab MCIR-SQLol*. For example, in Fig. 9, on SQLi01, Random needs to evaluate approximately 35% more payloads than ART4SQLi to find a SQLi issue. In Fig. 10, similar improvements are also observed on SQLi03, SQLi12, SQLi14, and SQLi15.

Moreover, for some pages, the maximum ART4SQLi testing results are even equal or less than the median value of Random. It means in the worst case, the performance of ART4SQLi is equal to or even better than the mean performance of Random, such as SQLi02 of *Web for Pentester*, SQLi04 of *Wooyung DVWA*, and SQLi04 of *SpiderLab MCIR-SQLol*.

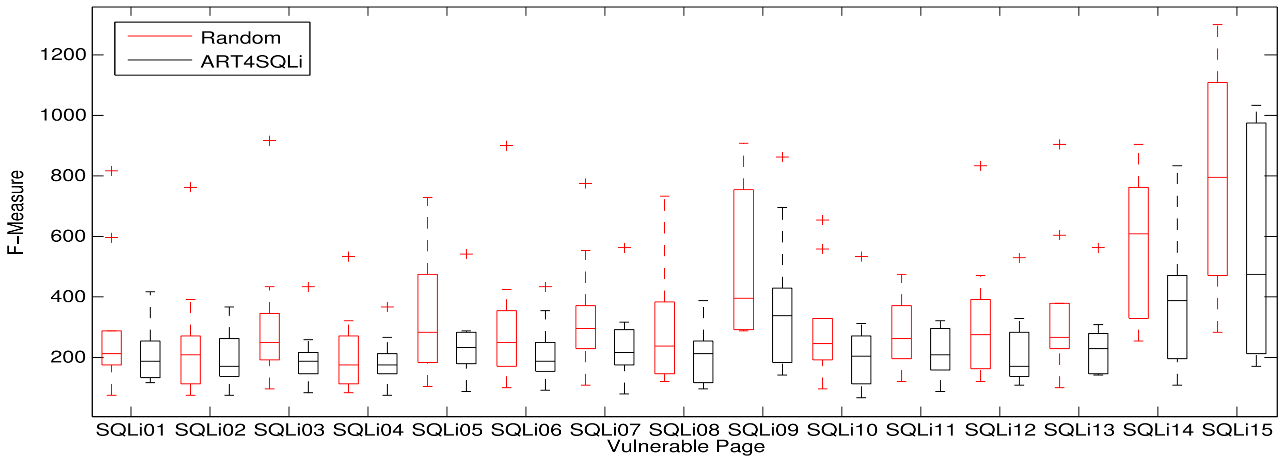

Fig. 10. MCIR-SQLol.

TABLE VI
TIMING OF ART4SQLI AND RANDOM ON EACH INDIVIDUAL SUBJECTS

| | | SQLi01 | SQLi02 | SQLi03 | SQLi04 | SQLi05 | SQLi06 | SQLi07 | SQLi08 | *Average* |
|---|---|---|---|---|---|---|---|---|---|---|
| **ART4SQLi (A)** (seconds) | Web for Pentester | 23.78 | 44.14 | 67.64 | 100.95 | 69.17 | 70.32 | 115.73 | - | 70.25 |
| | Wooyung DVWA | 208.17 | 67.86 | 83.03 | 63.22 | 73.92 | 87.63 | 51.02 | - | 90.69 |
| | MCIR-SQLol | 21.13 | 18.98 | 20.32 | 19.54 | 24.87 | 22.65 | 24.01 | 20.31 | |
| | | SQLi09 | SQLi10 | SQLi11 | SQLi12 | SQLi13 | SQLi14 | SQLi15 | | |
| | | 39.16 | 20.95 | 22.06 | 21.92 | 25.63 | 38.01 | 52.07 | | 26.1 |
| | *Average* | | | | | | | | | 52.35 |
| | | SQLi01 | SQLi02 | SQLi03 | SQLi04 | SQLi05 | SQLi06 | SQLi07 | SQLi08 | *Average* |
| **Random (R)** (seconds) | Web for Pentester | 23.52 | 34.98 | 64.42 | 93.59 | 57.59 | 75.29 | 108.76 | - | 65.45 |
| | Wooyung DVWA | 204.43 | 63.29 | 80.39 | 60.32 | 73.69 | 84.56 | 47.28 | - | 87.71 |
| | MCIR-SQLol | 20.43 | 16.73 | 22.24 | 17.68 | 24.17 | 21.96 | 24.31 | 20.14 | |
| | | SQLi09 | SQLi10 | SQLi11 | SQLi12 | SQLi13 | SQLi14 | SQLi15 | | |
| | | 36.87 | 20.86 | 20.42 | 20.35 | 24.94 | 37.99 | 51.02 | | 25.34 |
| | *Average* | | | | | | | | | 49.93 |
| | | SQLi01 | SQLi02 | SQLi03 | SQLi04 | SQLi05 | SQLi06 | SQLi07 | SQLi08 | *Average* |
| $\frac{A-R}{R} \times 100\%$ | Web for Pentester | 1.09% | 20.75% | 4.76% | 7.29% | 16.74% | -7.07% | 6.02% | | 6.83% |
| | Wooyung DVWA | 1.79% | 6.73% | 3.17% | 4.58% | 0.31% | 3.50% | 7.33% | | 3.28% |
| | MCIR-SQLol | 3.31% | 4.73% | -9.44% | 9.52% | 2.81% | 3.04% | -1.25% | 0.83% | |
| | | SQLi09 | SQLi10 | SQLi11 | SQLi12 | SQLi13 | SQLi14 | SQLi15 | | |
| | | 5.84% | 0.42% | 7.43% | 7.16% | 2.69% | 0.05% | 2.01% | | 2.91% |
| | *Average* | | | | | | | | | 3.94% |

Finally, we answer $Q2$ as follows:

1) In summary, our experiments have demonstrated that ART4SQLi is able to reveal SQLi vulnerability 26% *faster*, on average, than the standard random manner.

*3) Answering Q3:* We use Table VI to give statistics of the time issues of ART4SQLi and Random. It consists of three parts, which shows the time used to execute a payload on each page, each benchmark, and the whole experiment set, by ART4SQLi, by Random, and the different from the latter to the former, respectively.

To achieve that, we perform the following steps. For both Random and ART4SQLi, we count the time elapsed from picking out a payload to finishing the evaluation of that payload, and let the time summing up for each payload evaluated divided by the number of payload evaluated to calculate such a "time per payload." Since payloads are selected in random by Random and adaptively by ART4SQLi, the report timing data are different. From the table, we observe that ART4SQLi always needs more time than Random to process a payload. For example, on page SQLi01 of Web for Pentester, the on average time to process a payload is 23.52 by Random and 23.78 by ART4SQLi. We

TABLE VII
WORST CASE ANALYSIS

| | Vulnerable Page | *E-Measure* | Random (R) | ART4SQLi (A) | $\frac{R-A}{R} \times 100\%$ |
|---|---|---|---|---|---|
| Web for Pentester | SQLi01 | 1.58% | 329.3 | 264.2 | 19.77% |
| Web for Pentester | SQLi05 | 0.04% | 806.2 | 698.7 | 13.33% |
| Wooyung DVWA | SQLi07 | 0.06% | 605.9 | 501 | 17.31% |
| MCIR-SQLol | SQLi04 | 0.67% | 211.40 | 186.4 | 11.83% |

TABLE VIII
TYPE OF EFFECTIVE PAYLOADS [26]

| | Boolean-Based Blind | Error-Based | Union Query | Stacked Queries | Time-Based Blind | Others |
|---|---|---|---|---|---|---|
| Web for Pentester | 16 | 14 | 12 | 8 | 9 | 11 |
| DVWA | 9 | 17 | 6 | 16 | 14 | 8 |
| MCIR-SQLol | 38 | 17 | 8 | 28 | 24 | 35 |
| $Total$ | 63 | 48 | 26 | 52 | 47 | 54 |

further use $\frac{A-R}{R}$ to calculate the relative different from Random to ART4SQLi. It means that 1.09% additional cost is needed by using ART4SQLi to replace the original random testing manner. The other cells and the cell in the "Average" column show similar phenomenon. At the same time, we also observe that these exist some counter-examples, which needs more time to evaluate a payload by Random on average, than by ART4SQLi. It is due to that a long-lasting payload happened to be selected by Random, since we are counting the evaluation time. We do not exclude these samples since ART4SQLi also have the chance to select a long-lasting payload and it makes no harm to a fair comparison. Finally, from Table VI, we find that ART4SQLi needs a roughly 3.94% additional running time than Random on average. We consider such a cost acceptable in practice.

Finally, we answer $Q3$ as follows:

1) The additional cost by ART4SQLi is acceptable in practice.

*4) Worst Case Discussion:* Next, we examine four worst cases, on which ART4SQLi produces only moderate improvements (i.e., less than 20%) over Random. As showed in Table VII, they are SQLi01, SQLi05 of *Web for Pentester*, SQLi07 of *Wooyun DVWA*, and SQLi04 of *MCIR-SQLol*. The corresponding effectiveness improvements of ART4SQLi over Random are 19.77%, 13.33%, 17.31%, and 11.83%, respectively.

We investigate the original payload collection we used in testing, and find that, for these four pages, the set of effective payloads expose either a very high or a very low portion of the original payload space. For example, the vulnerable page SQLi01 in *Web for Pentester* is a simple string-based injection page without any inspection mechanisms, and the vulnerable page of stage SQLi04 in *MCIR-SQLol* is also a string injection page with a very simple validation mechanism, which only checks the space used in request string. From Table IV, we can find that the set of effective payloads for SQLi01 in *Web for Pentester* and SQLi04 in *MCIR-SQLol* forms 1.58% and 0.67% of the original payload space, respectively. It is really a high portion of total payloads compared to other vulnerability pages. When effective payloads are common, both Random and ART4SQLi can easily reveal SQLi vulnerabilities within a small time period, and there may not exist great improvements from Random to ART4SQLi.

For the other two vulnerability pages in *Web for Pentester* and *Wooyun DVWA*, the corresponding vulnerable pages only accept input strings satisfying very complicated predefined structures. Therefore, only a very small part of the payloads can be successfully injected. For example, there are only 40 effective payloads in a total of 101 465 for SQLi05 of *Web for Pentester*. On the contrary, if effective payloads are very rare, the cluster of effective payloads regresses to one or two small portions within the original payload space, and it becomes more difficult for ART4SQLi to reveal an SQLi vulnerability. This may explain the four worst cases. Note that even in these cases, ART4SQLi still produces a more than 13% improvement over Random.

*5) Types of Effective Payloads:* We further analyze the types of the effective payloads found in each benchmark. There are many ways to classify a vulnerability, such as injection type, attack type, submission method, and information retrieval means [26], [61], [67]. Halfond *et al.* [26] has summarized five types of SQLi vulnerabilities for payloads. We show them as the "Boolean-based blind," "Error-based," "Union query," "Stacked queries," and "Time-based blind" columns in Table VIII. The payloads belonging to no group are reported in the "Others" column of the table.

Recall that we perform ART4SQLi on each vulnerable page and repeat the testing process for ten times. We finally record $10 \times (7 + 7 + 15) = 290$ effective payloads, investigate their types, and report their statistics in Table VIII. If a payload can be described by more than one type, we manually choose its dominant type. Let us take the "Web for pentester" row and the "Boolean-based blind" column to Illustrate. The number 16 denotes that there are 16 effective payloads of the "Boolean-based blind" type. The other cells are similarly interpreted.

We further find that the payloads of type "Boolean-based blind" form the smallest class, i.e., 63 out of 290. At the same time, the payloads of type "Union query" form the largest class, i.e., 26 out of 290. This is because that the two types of payloads have the simplest and most complicated structures, respectively. As a result, the two types of payloads are the easiest and the most difficult ones to synthesize, respectively.

### *G. Threats to Validity*

The alternative strategies used to implement ART4SQLi may put threats to the empirical observation on validating the effectiveness of our basic proposal. This may relate to the details of payload grammar and parsing tree, the distance formula adopted, SQLi benchmarks, and the adaptive random strategy chosen. For example, there exist other syntax processors (e.g., SOFIA [6]), other distances (e.g., Jaccard [36]), other effectiveness metrics (e.g., F-score [70]), other benchmarks (e.g., [2], [21]), and other adaptive random strategies (e.g., restricted random testing [8]). Cooperating them may result in different empirical observations. On the other hand, the determination of arguments for the proposed adaptive system may also result in different empirical results. Chen *et al.* [11] observed that the effectiveness of the adaptive system FSCS-ART can be significantly improved by increasing **FixedSize** and suggested that 10 is close to the most optimal setting [12]. Though there is no knowledge to guide the choosing of the argument for different goals, we follow them to use the suggested value 10. Although we find that effective payloads tend to have sparse distribution and be clustered in the payload collection based on limited data observation, the real situation may be different from our empirical observation.

Other experiment settings can result in different empirical observations. For example, in our experiment, we apply the feature vector extraction process on all the payloads from the payload collection, rather than sampling the payload collection to estimate the payload space. We do that to make full use of the payload information to reduce the sample bias. The preparation of the payload collection is another impact factor. Our payload is prepared according to our experiences and suggestions from popular forums. We further equip time-based blind payloads in our payload collection with the following considerations. Modern application servers often have the mechanism of recognizing suspicious queries. Once the intention of an attack trial is realized, an application may refuse to serve, slow down response, or throw out misleading answers. As a result, the cost of executing an attack may be higher than expected, and such a cost should not be ignored in any case. In practice, the manner of brute-force trying increases both attacking cost and the risk of exposing the intention of the attackers. They thus have been abandoned by popular attacking approaches [68]. To conduct an SQLi vulnerability discovery study, we choose to simulate the realistic environment.

The subject benchmarks can have impacts on the experiment results. We select three widely used benchmarks to evaluate the effectiveness of our proposal. However, our proposal shows inconsistent effectiveness on various vulnerability pages included. For example, we have discussed the worst cases found in the experiments. In some empirical studies, both the worst cases and the best cases are excluded from the statistics, however, we keep all of them to reflect the properties of the benchmark. We foresee that in practice such situations are not common. First, real-world vulnerabilities are more sophisticated than the vulnerable pages in the benchmarks. Usually, modern Web applications use some existed filtering expressions, security functions, or third-party modules, and it is quite often to have security issues. Second, the worst case of *Wooyun DVWA 2014*, which defines very complicated input data format but gives no validation mechanism, is not common in practice. In realistic Web applications, such complicated input format means significant functionality loss. If a dynamic Web page is designed for indexing source codes from user input (user input source file name), which satisfy the format of "*.py," such a Web application can only index python sources. Such a designed method is not common, and will be only adopted for special systems. Therefore, the worst cases in the subjects will not stop ART4SQLi from being considerably acceptable in practice. Even if the aforementioned situations indeed exist, the experiments suggest that ART4SQLi can still manifest an observable improvements over Random. Although the three benchmarks are also used to evaluate the effectiveness of ART4SQLi, they represent limited types of vulnerability in practice. Other benchmarks (e.g., [2]) not included in the benchmark may lead to different results.

From the experiment, we conclude that ART4SQLi comes with acceptable additional cost in the Web applications, which have similar characteristics to the benchmarks. In our experiment, we let ART4SQLi store a payload string together with its feature vector into a hash table. The keys of the hash table are the original payload strings, and the value of each key is the corresponding feature vector. We thus speed up the indexing of the retrieving of payload during the payload selection process. The process without such an acceleration may have different timing results. At the same time, we also realize that the finding of a farthest nearest payload may be further optimized by branch-cutting search based on the distance metrics adopted.

## V. Related Work

Many existing techniques, such as input filtering, static analysis, runtime monitoring defensive coding, key words randomization, and security testing, can detect and prevent a subset of the vulnerabilities that lead to SQLi. In this section, we list work related to this paper.

### *A. SQLi Vulnerability Detection and Evaluation*

The root cause of SQLi vulnerabilities is an insufficient input validation. Thus, the most straightforward way is to set up an input filtering proxy in front of Web applications. Howard and LeBlanc [32] proposed a filtering proxy based on a blacklist, which detects predefined SQLi key words in user inputs. If a user input contains such keys, the request is blocked; otherwise, it is considered safe. Although the filtering method is straightforward, it is too simple to block some complicated SQLi attacks, as it can be easily bypassed via obfuscation methods [3].

Static analysis is also a popular security mechanism against SQLi. Fu *et al.* [22] described a compile-time static SQLi analysis framework, which can detect and block several kinds of SQLi attacks, but is limited to the ASP.NET language. Huang *et al.* [34] developed a static analysis platform. It detects input validation related errors that use developer-provided annotations. Though this static framework supports more types of injection attacks, relying on developer-provided annotations limits the accuracy of such methods. Wassermann and Su [74] proposed an approach that used static analysis combined with automated reasoning to verify that the SQL queries generated in the application layer cannot contain a tautology. Their method is effective in finding SQLi vulnerabilities from source code; however, a drawback of their method is the limitation of supporting only tautological type injections.

Due to the lack of runtime information, static analysis may misreport SQLi vulnerabilities. In response, some researchers have proposed *runtime monitor mechanisms*. Halfond and Orso [25] offered an approach that combines static analysis with a runtime monitor. In this approach, a static process builds a legitimate queries model while dynamic part is responsible for checking maliciousess. By combining the two techniques, their method made the static analysis more reliable. However, this combination approach is very complex, and may not scale well to large applications. Halfond *et al.* [27] further proposed a dynamic tainting technique against SQLi, which marks and tracks certain data in a program at runtime, then identifies trusted strings in an application and allows only these trusted strings to be used to create certain parts of an SQL query. Buehrer *et al.* [5] proposed an injection detection framework based on comparing, at run time, the parse tree of the SQL statement before inclusion of user input with that resulting after inclusion of input. Prakash and Saravanan [55] proposed a mechanism based on static and dynamic analysis methods. The proposed approach, SQLi detection (SQLID), is introduced as an intermediate virtual layer or database between the application and the database. Khalid and Yousif [42] proposed a dynamic analysis tool for detecting SQLi. Xiao *et al.* [77] proposed an approach for SQLID based on behavior and the analysis of response and state of the Web application under different attacks. These runtime monitoring techniques perform better than black list based proxies. Because they support advanced input analysis such as data tracing, some SQLi attacks can be detected.

Recently, off-the-shelf runtime monitors such as application-level intrusion detection systems [45], [54], and Web application firewalls such as *ModeSecurity*[6] have been released. However, such advanced input analysis bring additional computing complexity and require additional resources. When the request rate is very high, such intermediate monitors will consume a large amount of system resources [73]. On the other hand, Appelt *et al.* [4] proposed a machine learning driven testing approach, which can generate more effective test cases, in which the training process leads the generation process of more effective tests. Singh [66] analyzed the detection and prevention using the classical methods as well as modern approaches.

ART4SQLi focuses on scheduling given payloads to reveal an SQLi vulnerability issue as early as possible. Some other works such as [1], [38], [41], and [50] also detect SQLi vulnerability. Different from them, ART4SQLi is based on predefined payload grammar. The experiment carried out in this paper evaluates only SQLi issue. Nevertheless, the grammar of ART4SQLi can be extended and has no limit on the types of vulnerability.

There exist popular ways to detect other vulnerabilities such as buffer overflows, null pointer, and subscript out of bounds. For example, Jovanovic *et al.* [38] proposed a static analysis method. Xu *et al.* [78], [79] developed tools Melton and Canalyze, which can detect memory leak for C programs. Godefroid [23] proposed a framework to detect resource leaks in Android applications, and developed the tool Relda. Compared with such static analysis approaches, ART4SQLi is different in its dynamic testing manner and interested vulnerability types.

## B. Payload Grammar, Syntax Parsing, and Payload Mutants

Each SQLi payload can be regarded as a malicious string that follows a specific string structure. Thus, it is feasible to define a CFG for SQLi attacks [69]. A CFG [31] is a set of recursive rewriting rules (or productions) used to generate patterns of strings. For example, a CFG $G$ can be defined by the 4-tuple: $G = (V, \Sigma, R, S)$. Here, $V$ is a finite set, in which each element $v$ is called a nonterminal character or a variable. A variable (aka. syntactic category) represents a different type of phrase or clause in the sentence, defining a sub-language of the language defined by $G$. $\Sigma$ is a finite set of terminals, disjoint from $V$, which makes up the actual content of the sentence. The set of terminals is the alphabet of the language defined by the grammar $G$. $R$ is a finite relation from $V$ to $(V \cup \Sigma)^*$, where the asterisk represents the Kleene star operation. The members of $R$ are called the (rewrite) rules or productions of the grammar, and also commonly symbolized by a $P$. $S$ is the start variable (or start symbol), used to represent the whole sentence (or program). It is also an element of $V$.

Appelt *et al.* [4] proposed a CFG for three specific SQLi attacks: *BOOLEAN*, *UNION*, and *PIGGY*, which aim at manipulating the intended logic, exploiting the family of union queries, or injecting additional statements in the original SQL queries [26]. This type of CFG covers most typical SQLi attacks. Hashemi and Hwa [30] proposed using a parse tree to analyze ungrammatical sentences. Kauchak *et al.* [40] showed how the metrics can be used to understand grammar regularity in a broad range of corpora. In this proposal, we stick to the formulation by Strawson [69] to implement the payload grammar.

The scale of the payload collection makes an important role in such an approaching. It is popular to mutate payloads to generate new ones and the generalization of the method thus relates to the description capability of the payload grammar. Moreover, the grammar of payload in Fig. 5 is easy to generalize. For example, we can extend "$opOR ::= or \mid || \mid \ldots;$" to "$opOR ::= or \mid xor \mid || \mid \ldots;$" to support the "XOR" operator in MYSQL.

## C. Adaptive Random Testing

ART is an improvement based on random testing [15]. It is based on the observation that test cases revealing software failures tend to cluster together in the test case domain. It therefore proposes to have randomly selected test cases being more evenly spread throughout the input domain by employing the location

[6][Online]. Available: https://www.modsecurity.org/

information of the test cases, which have been executed but do not reveal failures [11]. In this method, we approach in such a direction to study the SQLi vulnerability discovery problem.

ART uses the distance among test cases to pick up the test case candidate to quickly find the first test case that leads to failures. Many distance metrics [9], [18], [43] have been proposed to compute the difference between test cases. The Euclidean distance focuses on the spatial distance between the samples rather than the difference in sample characteristics. The cosine distance focuses on the difference in sample characteristics rather than the spatial distance between the samples. In this paper, we adopted the latter since payloads are represented by vectors in a token space, and there is no scientific basis to manipulate different dimensions of such a space. Chen *et al.* [15] proposed an adaptive sequence approach for object-oriented software (OOS) test case prioritization. Qi *et al.* [56] studied the influence of the distance calculation error on the performance of ART. Sinaga *et al.* [65] analyzed the path coverage information for ART.

We realize that there are at least two key points determining the successful adaptation of ART to SQLi vulnerability discovery problem. The first is to find a suitable vectorization method and distance function, in which payload space of the effective payloads expose to cluster together as expected. The second is to design a payload selection process so that a promising payload, which is expected to be far from all the evaluated ones, can be selected. Based on such understanding, we also expect to cooperate other ART-related or ART-inspired optimization and strategies to further boost our proposal.

The computational overhead of ART has been reported in previous studies. For example, Chan *et al.* [7], [9] tried to reduce the overhead of ART to address its scalability issue. We are also interested in integrating such mechanisms in ART4SQLi. On the other hand, potential improvements to decrease the complexity include cutting unpromising payloads in the construction of candidate set and caching payload distances in a candidate selection.

## VI. Conclusion

SQL have been ranked as one of the most dangerous Web application vulnerabilities. At the same time, dynamic methods such as black-box testing are regarded as one of the most effective ways for detecting such vulnerabilities within Web applications before they are ready for servicing. However, their performance may be limited due to the black-box nature and the complexity of the problem, and has not been adequately studied.

In this paper, we designed a grammar for interested SQLi attack payloads, and proposed ART4SQLi to accelerate the testing process of manipulating attack payloads to reveal SQLi vulnerabilities. ART4SQLi first decomposed each payload string into tokens, and characterized each payload as a feature vector. In the next stage, ART4SQLi selected a promising payload for evaluation, by randomly generating a size-fixed candidate set from the payload collection and picking out from it the one farthest to all the evaluated payloads. When an SQLi vulnerability issue was revealed by a payload, we marked the payload as an effective one and let the process complete; otherwise, the evaluated set of payloads was updated.

Experiments using three extensively adopted open-source SQLi benchmarks showed the sparse and clustering distribution of effective payloads, validated the effectiveness of our proposal, and evaluated its practicability. On average, ART4SQLi showed up to a 26% improvements over the original random testing manner on Web for Pentester, DVWA 2014, and MCIR-SQLol subjects, with acceptable additional computation costs.

In the future, we will transfer our methodology to other kinds of testing-based injection vulnerability discovery problem. We are also interested in cooperating other ART-related techniques to further improve the solution to this problem.

## References

[1] N. Antunes and M. Vieira, "Detecting SQL injection vulnerabilities in web services," in *Proc. 4th Latin-Am. Symp. Dependable Comput.*, 2009, pp. 17–24.

[2] N. Antunes and M. Vieira, "Assessing and comparing vulnerability detection tools for web services: Benchmarking approach and examples," *IEEE Trans. Services Comput.*, vol. 8, no. 2, pp. 269–283, Mar./Apr. 2015.

[3] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for SQL injection vulnerabilities: An input mutation approach," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 259–269.

[4] D. Appelt, C. D. Nguyen, and L. Briand, "Behind an application firewall, are we safe from SQL injection attacks?" in *Proc. 8th Int. Conf. Softw. Testing, Verification, Validation*, 2015, pp. 1–10.

[5] G. Buehrer, B. W. Weide, and P. A. Sivilotti, "Using parse tree validation to prevent SQL injection attacks," in *Proc. 5th Int. Workshop Softw. Eng. Middleware*, 2005, pp. 106–113.

[6] M. Ceccato, C. D. Nguyen, D. Appelt, and L. Briand, "SOFIA: An automated security oracle for black-box testing of SQL-injection vulnerabilities," in *Proc. 31st IEEE/ACM Int. Conf. Automat. Softw. Eng.*, 2016, pp. 167–177.

[7] K. P. Chan, T. Y. Chen, and D. Towey, "Adaptive random testing with filtering: An overhead reduction technique," in *Proc. 17th Int. Conf. Softw. Eng. Knowl. Eng.*, 2005, pp. 292–299.

[8] K. P. Chan, T. Y. Chen, and D. Towey, "Restricted random testing: Adaptive random testing by exclusion," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 16, no. 4, pp. 553–584, 2006.

[9] K. P. Chan, T. Y. Chen, and D. Towey, "Forgetting test cases," in *Proc. 30th Annu. Int. Comput. Softw. Appl. Conf.*, 2006, pp. 485–494.

[10] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Proc. Annu. Asian Comput. Sci. Conf.*, 2004, pp. 320–329.

[11] T. Y. Chen, D. H. Huang, and F.-C. Kuo, "Adaptive random testing by balancing," in *Proc. 2nd Int. Workshop Random Testing Co-Located 22nd IEEE/ACM Int. Conf. Automat. Softw. Eng.*, 2007, pp. 2–9.

[12] T. Y. Chen, F.-C. Kuo, and H. Liu, "Adaptive random testing based on distribution metrics," *J. Syst. Softw.*, vol. 82, no. 9, pp. 1419–1433, 2009.

[13] T. Y. Chen, F. C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The ART of test case diversity," *J. Syst. Softw.*, vol. 83, no. 1, pp. 60–66, 2010.

[14] T. Y. Chen, F. C. Kuo, D. Towey, and Z. Q. Zhou, "A revisit of three studies related to random testing," *Sci. China Inf. Sci.*, vol. 58, no. 5, pp. 052104:1–052104:9, 2015.

[15] J. Chen *et al.*, "An adaptive sequence approach for OOS test case prioritization," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops*, 2016, pp. 205–212.

[16] J. Chen, F. C. Kuo, T. Y. Chen, D. Towey, C. Su, and R. Huang, "A similarity metric for the inputs of OO programs and its application in adaptive random testing," *IEEE Trans. Rel.*, vol. 66, no. 2, pp. 373–402, Jun. 2017.

[17] J. Chen *et al.*, "Test case prioritization for object-oriented software: An adaptive random sequence approach based on clustering," *J. Syst. Softw.*, vol. 135, pp. 107–125, 2018.

[18] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "ARTOO: Adaptive random testing for object-oriented software," in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng.*, 2008, pp. 71–80.

[19] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 159–182, Feb. 2002.

[20] I.A. Elia, J. Fonseca, and M. Vieira, "Comparing SQL injection detection tools using attack injection: An experimental study," in *Proc. 21st Int. Symp. Softw. Rel. Eng.*, 2010, pp. 289–298.

[21] J. Fonseca, M. Vieira, and H. Madeira, "Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks," in *Proc. 13th Pacific Rim Int. Symp. Dependable Comput.*, 2007, pp. 365–372.

[22] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao, "A static analysis framework for detecting SQL injection vulnerabilities," in *Proc. 31st Annu. Int. Comput. Softw. Appl. Conf.*, 2007, pp. 87–96.

[23] P. Godefroid, "Random testing for security: Blackbox vs. whitebox fuzzing," in *Proc. 2nd Int. Workshop Random Testing Co-Located 22nd IEEE/ACM Int. Conf. Automat. Softw. Eng.*, 2007, pp 1–1.

[24] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and detecting resource leaks in Android applications," in *Proc. 28th IEEE/ACM Int. Conf. Automat. Softw. Eng.*, 2013, pp. 389–398.

[25] W. G. Halfond and A. Orso, "Amnesia: Analysis and monitoring for neutralizing SQL-injection attacks," in *Proc. 20th IEEE/ACM Int. Conf. Automat. Softw. Eng.*, 2005, pp. 174–183.

[26] W. G. Halfond, J. Viegas, and A. Orso, "A classification of SQL-injection attacks and countermeasures," in *Proc. IEEE Int. Symp. Secure Softw. Eng.*, 2006, pp. 65–81.

[27] W. G. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks," in *Proc. 14th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2006, pp. 175–185.

[28] W. G. Halfond and A. Orso, "Detection and prevention of SQL injection attacks," in *Malware Detection*. New York, NY, USA: Springer, 2007, pp. 85–109.

[29] P. Hanks, *Lexical Analysis: Norms and Exploitations*. Cambridge, MA, USA: MIT Press, 2013.

[30] H. B. Hashemi and R. Hwa, "Parse tree fragmentation of ungrammatical sentences," in *Proc. 25th Int. Joint Conf. Artif. Intell.*, 2016, pp. 2796–2802.

[31] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation," *ACM SIGACT News*, vol. 32, no. 1, pp. 60–65, 2001.

[32] M. Howard and D. LeBlanc, *Writing Secure Code*. London, U.K.: Pearson Education, 2002.

[33] Y. Huang *et al.*, "A mutation approach of detecting SQL injection vulnerabilities," in *Proc. Int. Conf. Cloud Comput. Security*, 2017, pp. 175–188.

[34] Y. W. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo, "Securing web application code by static analysis and runtime protection," in *Proc. 13th Int. Conf. World Wide Web*, 2004, pp. 40–52.

[35] IMPERVA, "Web application attack report (WAAR)," 2015. [Online]. Available: https://www.imperva.com/defensecenter/waar

[36] P. Jaccard, "The distribution of the flora in the alpine zone," *Int. J. Comput., Electr., Autom., Control Inf. Eng.*, vol. 11, no. 2, pp. 37–50, 1912.

[37] B. Jiang and W. K. Chan, "Input-based adaptive randomized test case prioritization: A local beam search approach," *J. Syst. Softw.*, vol. 105, pp. 91–106, 2015.

[38] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Proc. IEEE Symp. Security Privacy*, 2006, pp. 1–6.

[39] D. Kar, S. Panigrahi, and S. Sundararajan, "SQLiDDS: SQL injection detection using document similarity measure," *J. Comput. Security*, vol. 24, no. 4, pp. 507–539, 2016.

[40] D. Kauchak, G. Leroy, and A. Hogue, "Measuring text difficulty using parse-tree frequency," *J. Assoc. Inf. Sci. Technol.*, vol. 68, no. 9, pp. 2088–2100, 2017.

[41] K. Kemalis and T. Tzouramanis, "SQL-IDS: A specification-based approach for SQL-injection detection," in *Proc. ACM Symp. Appl. Comput.*, 2008, pp. 2153–2158.

[42] A. Khalid and M. M. Yousif, "Dynamic analysis tool for detecting SQL injection," *Int. J. Comput. Sci. Inf. Security*, vol. 14, no. 2, pp. 224–232, 2016.

[43] B. Lei *et al.*, *Classification, Parameter Estimation and State Estimation: An Engineering Approach Using MATLAB*. New York, NY, USA: Wiley, 2017.

[44] P. Li *et al.*, "Application of hidden Markov model in SQL injection detection," in *Proc. 41st Annu. Comput. Softw. Appl. Conf.*, 2017, pp. 578–583.

[45] J.-C. Lin and J.-M. Chen, "The automatic defense mechanism for malicious injection attack," in *Proc. 7th IEEE Int. Conf. Comput. Inf. Technol.*, 2007, pp. 709–714.

[46] H. Liu, F. C. Kuo, D. Towey, and T. Y. Chen, "How effectively does metamorphic testing alleviate the oracle problem?" *IEEE Trans. Softw. Eng.*, vol. 40, no. 1, pp. 4–22, Jan. 2014.

[47] J. Makhoul, F. Kubala, R. Schwartz, and R. Weischedel, "Performance measures for information extraction," in *Proc. DARPA Broadcast News Workshop*, 1999, pp. 249–252.

[48] C. Mao, T. Y. Chen, and F. C. Kuo, "Out of sight, out of mind: A distance-aware forgetting strategy for adaptive random testing," *Sci. China Inf. Sci.*, vol. 60, no. 9, 2017, Art. no. 092106.

[49] C. Mao and X. Zhan, "Towards an improvement of bisection-based adaptive random testing," in *Proc. 24th Asia-Pacific Softw. Eng. Conf.*, 2017, pp. 689–694.

[50] C. Mao and X. Zhan, "An approach for SQL injection vulnerability detection," in *Proc. 6th Int. Conf. Int. Technol.*, 2009, pp. 1411–1414.

[51] C. Nie, H. Wu, X. Niu, F. C. Kuo, H. Leung, and C. J. Colbourn, "Combinatorial testing, random testing, and adaptive random testing for detecting interaction triggered failures," *Inf. Softw. Technol.*, vol. 62, pp. 198–213, 2015.

[52] K. Nigam, J. Lafferty, and A. McCallum, "Using maximum entropy for text classification," in *Proc. Workshop Mach. Learn. Inf. Filtering*, 1999, vol. 1, pp. 61–67.

[53] L. Nyffenegger, "Web for pentester," 2015. [Online]. Available: https://www.pentesterlab.com

[54] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2005, pp. 124–145.

[55] J. Prakash and G. Saravanan, "SQLID: SQL injection detection based on static and dynamic analysis," *Transylvanian Rev.*, vol. 1, 2016.

[56] Y. Qi, Z. Wang, and Y. Yao, "Influence of the distance calculation error on the performance of adaptive random testing," in *Proc. Int. Conf. Softw. Quality, Rel. Security Companion*, 2017, pp. 316–319.

[57] E. Raymond, "The cathedral and the bazaar," *Knowl., Technol. Policy*, vol. 12, no. 3, pp. 23–49, 1999.

[58] A. Riancho, "W3AF-web application attack and audit framework," *World Wide Web Electronic Publication*, vol. 21, 2011.

[59] E. S. Ristad and P. N. Yianilos, "Learning string-edit distance," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 20, no. 5, pp. 522–532, May 1998.

[60] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 1999, pp. 179–188.

[61] S. M. S. Sajjadi and B. T. Pour, "Study of SQL injection attacks and countermeasures," *Int. J. Comput. Commun. Eng.*, vol. 2, no. 5, pp. 539–543, 2013.

[62] Y. Sasaki, "The truth of the F-measure," *Teach. Tutor. Mater.*, vol. 1, no. 5, pp. 1–5, 2007.

[63] R. S. Scowen, "Generic base standards," in *Proc. IEEE Software Eng. Standards Symp.*, 1993, pp. 25–34.

[64] H. Shahriar and M. Zulkernine, "Music: Mutation-based SQL injection vulnerability checking," in *Proc. 8th Int. Conf. Quality Softw.*, 2008, pp. 77–86.

[65] A. M. Sinaga, O. D. Hutajulu, R. T. Hutahaean, and I. C. Hutagaol, "Path coverage information for adaptive random testing," in *Proc. Int. Conf. Inf. Technol.*, 2017, pp. 248–252.

[66] J. P. Singh, "Analysis of SQL injection detection techniques," 2016, arXiv preprint arXiv:1605.02796.

[67] S. Som, S. Sinha, and R. Kataria, "Study on SQL injection attacks: Mode detection and prevention," *Int. J. Eng. Appl. Sci. Technol.*, vol. 1, no. 8, pp. 23–29, 2016.

[68] D. Stiawan, S. Sandra, E. Alzahrani, and R. Budiarto, "Comparative analysis of K-means method and Naive Bayes method for brute force attack visualization," in *Proc. IEEE Int. Conf. Anti-Cyber Crimes*, 2017, pp. 177–182.

[69] P. F. Strawson, *Subject and Predicate in Logic and Grammar*. Abingdon, U.K.: Routledge, 2017.

[70] C. J. Van Rijsbergen, *Information Retrieval*. London, U.K.: Butterworths, 1979.

[71] E. A. A. Vega, A. L. S. Orozco, and L. J. G. Villalba, "Benchmarking of pentesting tools," *Int. J. Comput., Electr., Autom., Control Inf. Eng.*, vol. 11, no. 5, pp. 590–593, 2017.

[72] M. Vieira, N. Antunes, and H. Madeira, "Using web security scanners to detect vulnerabilities in web services," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2009, pp. 566–571.

[73] C. Wang, D. Zhang, H. Lu, J. Zhao, Z. Zhang, and Z. Zheng, "An experimental study on firewall performance: Dive into the bottleneck for firewall effectiveness," in *Proc. 10th Int. Conf. Inf. Assurance Security.*, 2014, pp. 71–76.

[74] G. Wassermann and Z. Su, "An analysis framework for security in web applications," in *Proc. FSE Workshop Specification Verification Component-Based Syst.*, 2004, pp. 70–78.

[75] G. Wei, "Some cosine similarity measures for picture fuzzy sets and their applications to strategic decision making," *Informatica*, vol. 28, no. 3, pp. 547–564, 2017.

[76] P. Wood, B. Nahorney, K. Chandrasekar, S. Wallace, and K. Haley, "Symantec global internet security threat report," *White Paper, Symantec Enterprise Security*, vol. 21, 2016.

[77] Z. Xiao, Z. Zhou, W. Yang, and C. Deng, "An approach for SQL injection detection based on behavior and response analysis," in *Proc. 9th Int. Conf. Commun. Softw. Netw.*, 2017, pp. 1437–1442.

[78] Z. Xu, J. Zhang, and Z. Xu, "Melton: A practical and precise memory leak detection tool for C programs," *Frontiers Comput. Sci.*, vol. 9, no. 1, pp. 34–54, 2015.

[79] Z. Xu, J. Zhang, Z. Xu, and J. Wang, "Canalyze: A static bug-finding tool for C programs," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 425–428.

[80] X. Yuan, I. Williams, T. H. Kim, J. Xu, H. Yu, and J. H. Kim, "Evaluating hands-on labs for teaching SQL injection: A comparative study," *J. Comput. Sci. Colleges*, vol. 32, no. 4, pp. 33–39, 2017.

[81] X. Zhang, X. Xie, and T. Y. Chen, "Test case prioritization using adaptive random sequence with category-partition-based distance," in *Proc. IEEE Int. Conf. Softw. Qual.*, 2016, pp. 374–385.

**Long Zhang** received the bachelor's degree in computer science and technology from the Hefei University of Technology, Hefei, China, in 2012. He is currently working toward the Ph.D. degree in computer software and theory with the University of Chinese Academy of Sciences, Beijing, China.

He is also with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China. His research interests include program debugging and program verification.

**Donghong Zhang** received the bachelor's degree in computer science and technology from Harbin Engineering University, Harbin, China, in 2015, and the master's degree in computer software and theory from the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2018.

He is an Engineer with the First Research Institute, Ministry of Public Security, Beijing, China. His research interests include security and software testing.

**Chenghong Wang** received the bachelor's and master's degrees from Harbin Engineering University, Harbin, China, and Syracuse University, Syracuse, NY, USA, respectively.

He is a Research Scholar with the Department of Biomedical Informatics, University of California San Diego, La Jolla, CA, USA. His research interests include secure computation, privacy-preserving computing, applied cryptography, security testing, vulnerability discovery, and security applications in precision medicine and genome wide association studies.

**Jing Zhao** received the Ph.D. degree from the Harbin Institute of Technology, Harbin, China.

She is a Professor with the School of Software Technology, Dalian University of Technology, Dalian, China. She has published research results in venues such as IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, *ACM Journal on Emerging Technologies in Computing*, *Journal of Systems and Software*, and Practice and Experience (PE). Her research interests include dependable software, software testing, and combinatorial testing.

**Zhenyu Zhang** received bachelor's and master's degrees from Tsinghua University, Beijing, China, and the Ph.D. degree from the University of Hong Kong, Hong Kong.

He is an Associate Professor with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China, since 2011. He has published research results in venues such as IEEE TRANSACTIONS ON SOFTWARE ENGINEERING (TSE), IEEE TRANSACTIONS ON SERVICE COMPUTING (TSC), IEEE TRANSACTIONS ON RELIABILITY (TREL), Computer, International Conference on Software Engineering (ICSE), ACM SIGSOFT Symposium on the Foundation of Software Engineering (FSC), International Conference on Automated Software Engineering (ASE), and International World Wide Web (WWW) conferences. His research interests include debugging and testing for software and systems.