# CMM: A Combination-based Mutation Method For SQL Injection

Jing Zhao[1,2], Tianran Dong[3,1], Yang Cheng[3,1], and Yanbin Wang[4]

[1] Software Engineering, Dalian University of Technology, China
[2] Cyberspace Security Research Center, Peng Cheng Laboratory, Shenzhen, China
[3] Department of Computer Scienceand Technology,
Harbin Engineering University, China
[4] Department of Industrial Engineering,
Harbin Institute of Technology, Harbin, China
E-mail:zhaoj9988@dlut.edu.cn, dongtianran@hrbeu.edu.cn,
chengyangheu@hrbeu.edu.cn, wangyb@hit.edu.cn

**Abstract.** With the rapid development of Web applications, SQL injection (SQLi) has been a serious security threat for years. Many systems use superimposed rules to prevent SQLi like backlists filtering rules and filter functions. However, these methods can not completely eliminate SQLi vulnerabilities. Many researchers and security experts hope to find a way to find SQLi vulnerabilities efficiently. Among them, mutation-based fuzzing plays an important role in Web security testing, especially for SQLi. Although this approach expands the space for test cases and improves vulnerability coverage to some extent, there are still some problems such as mutation operators cannot be fully covered, test cases space explosions, etc. In this paper, we present a new technique Combinatorial Mutation Method (CMM) to generate test set for SQLi. The test set applies *t-way* and variable strength Combinatorial Testing. It makes the mutation progress more aggressive and automated and generates test cases with better *F-measure Metric* and *Efficiency Metric*. We apply our approach to three open source benchmarks and compare it with *sqlmap*, *FuzzDB* and *ART4SQLi*. The experiment results demonstrate that the approach is effective in finding SQLi vulnerabilities with multiple filtering rules.

**Keywords:** SQL injection · Mutation method · Combinatorial testing (CT) · *t-way* Combinatorial Testing (*t-way* CT) · Variable Strength Combinatorial Testing (VSCT)

## 1 Introduction

Database-driven Web applications have been rapidly adopted in a wide range of areas including on-line stores, e-commerce, etc. However, this popularity makes them more attractive to attackers. The number of reported Web attacks is growing sharply [8]: for instance, a recent Web application attack report observed an average increase of around 17% in different types of Web attacks over the nine-month period from August 1, 2013 to April 30, 2014.

Within the class of Web based vulnerabilities, SQL injection (SQLi) vulnerabilities have been labeled as one of the most dangerous vulnerabilities by the Open Web Application Security Project (OWASP). Although the cause and mechanism of SQLi is single, the number of SQLi attack events increased continuously. To solving SQLi problems, static analysis is a traditional security mechanism with trivial process, in which hybrid constraint solver would be used when an SQL query is submitted. Therefore, dynamic testing methods such as black-box testing have become a popular choice. For example, mutation-based fuzzy testing[22] can automatically generate mutations and analyze them.

There have been many studies devoted to detecting SQLi vulnerabilities[9][23][12]. Adaptive Random Testing (ART)[1] is an improvement based on random testing. Researchers believe that software bugs are continuous and can be reflected in the input domain. If the test cases are evenly distributed across the input field, they will have strong error detection capabilities. Therefore, ART4SQLi[25] selects the farthest test case from the selected one. ART4SQLi uses this method to trigger SQLi vulnerabilities as quickly as possible, i.e. the smallest possible F-measure (this metric will be detailed later).

Nowadays, multiple filtering rules can pick out many statements with sensitive characters, so a mutated test case will be dropped if the sensitive part remains after the mutation operation. Meanwhile the types of mutation operator increasing is causing the test space explosion. We integrate the other black box testing with the mutation approach to reduce the probability of the test case being filtered and decrease its space. In addition we focus on generating test cases by integrating the Combinatorial testing (CT) with the mutation method and detecting potential SQLi vulnerabilities by generating more effective test cases. The goal of the CT is to select a small number of test cases from the huge combined space, and detect the faults that may be triggered by the interaction of various factors. This method reduces the size of the test cases set and will have a good F-measure and efficiency.

In this paper, we present a new input mutation method CMM aimed at increasing the likelihood of triggering vulnerabilities beyond filtering rules. We import the original mutation methods from *sqlmap*[5], and classify them into five groups according to their mutation behavior. Motivated by CT, we want to find the best combination of mutation operators with the least test cases during the test. Then we build a CT model for these mutation methods. The model is input into a combinatorial test generation tool *ACTS*[24], and a combined object is generated using *t-way* and variable strength CT, which is simply a symbol array. Every row of the generated array is used to output a test case. We use the array to instruct our test cases mutation progress. The experiment result demonstrated that the better efficiency and effectiveness in triggering vulnerabilities of our method when there are multiple filtering rules. This paper is structured as follows. Section 2 provides background information on SQLi vulnerabilities and CT. Section 3 discusses in detail our approach, followed by

[5] http://sqlmap.org/

Section 4 where we introduce our experiments and results. Section 5 concludes this paper and discusses future work.

## 2 Background And Related Work

### 2.1 Background

**SQLi Vulnerabilities** Two major cause fro SQL injections are the ignorance towards filtering out user-input, and framing SQL queries dynamically using string data type by concatenating SQL code and user-input during runtime[6]. For example, a SQL statement is formed as Figure 1 (a simplified example from one of our Web applications): The user's input is stored in the string variable

```
1 $id = $_GET['$id'];
2 $id = tsFilter($id);
3 $getid = '' SELECT first_name, last_name FROM users WHERE
      user_id='$id' '';
4 $result = mysqli_query($getid) or die(mysqli_error());
```

Fig. 1: Example of an SQLi Vulnerability

*$id*. After being filtered by the filter function *tsFilter*, *$id* will be concatenated with the rest of the SQL statement. The SQL statement will be send to database server to be executed. If the filter function is not strong enough, the SQLi Vulnerabilities will be be easily exploited.

**Combinatorial Testing (CT)** Combinatorial testing is motivated by the selection of a few test cases in such a manner that good coverage is still achievable, while for a general treatment of the field of CT we refer the interest reader to the recent survey of [17]. The combinatorial test design process can be briefly described as follows:

(1) Model the input space. The model is expressed in terms of parameter and parameter values.

(2) The model is input into a combinatorial design procedure to generate a combinatorial object that is simply an array of symbols.

(3) Every row of the generated array is used to output a test case for a System Under Test (SUT).

One of the benefits of this approach is that steps 2. and 3. can be completely automated. In particular, we used the *ACTS* combinatorial test generation tool. Currently, *ACTS* supports *t-way* test set generation for $1 \leq t \leq 6$. In addition, *ACTS* also supports VSCT, We refer to two definitions from as follows:

**Definition 1.** *(t-way Covering Arrays): For the SUT, the n-tuple($v_{n_1}$,...,$v_{n_t}$,...) is called a t-value schema ($t > 0$) when some t parameters have fixed values and the others can take on their respective allowable values, represented as "-". When*

*$t = n$, the n-tuple becomes a test case for the SUT as it takes on specific values for each of its parameters. It is based on the fact that a test set covering all possible* t-way *combinations of parameter values for some* t *is very effective in identifying interaction faults.* t *usually takes a value from* 2 *to* 6 *[10] [14].*

**Definition 2.** *(Variable Strength Covering Arrays): $VSCA(N; t, v_1, v_2, ..., v_k, \{C\})$ is a $N \times k$ CA of strength t containing C. Here C is a subset of the k columns having strength $t' >$ t and can be CA. A VSCA can have more than one C, each having same or different strength [5]. For instance, consider a VSCA (*27*;* 2, $3^9 2^2$, $CA(27; 3, 3^3)^3$*). It has total* 11 *parameters:* 9 *parameters each having* 3 *values and* 2 *parameters each having* 2 *values. The strength of the MCA is* 2 *and the number of rows is* 27*. Of these* 11 *parameters, there exists three subsets C having* 3 *parameters each having* 3 *values. The strength of the C is* 3*. Thus, the VSCA should cover all the* 2*-way interactions among* 11 *parameters and* 3*-way interactions among* 3 *parameters of subset C [20].*

### 2.2 Related Work

Researchers have proposed a series of techniques to prevent SQL injection vulnerabilities. Many existing techniques, such as input filtering, static analysis and security testing can detect and prevent a subset of the vulnerabilities that lead to SQLi [21].

The most important reason of SQLi vulnerabilities is insufficient input validation of data and code in dynamic SQL statements [7]. Thus, the most straightforward way is to set up an input filtering proxy in front of Web applications. A novel approach for detection of SQLIA is proposed by Bisht et al [2], for mining programmer-intended queries with dynamic evaluation of candidate input. However, implementation of this method is not always feasible. Specifically, in a highly accessible web-application in which verification of a run-time query needs to retrieve its programmer intent or application constraint.

More and more researchers applied software testing technology to Web application security. Static analysis focus is to validate the user input type in order to reduce the chances of SQLi attacks rather than detect them. While the dynamic analysis method do not need to modify the web applications. However, the vulnerabilities found in the web application pages must be fixed manually and not all of them can be found without predefined attack codes. So the combination of static and dynamic analysis techniques are used as the basis of a preventative solution [15].

Random testing is one of the dynamic analysis and Adaptive Random Testing (ART) is its improvement [4]. ART is based on the observation that test cases revealing software failures tend to cluster together in the test cases domain. Inspired by this, we realize that effective payloads tend to cluster in the payload space and proposed a new method ART4SQLi to accelerate the SQLi vulnerability discovery process [25]. ART4SQLi selects the farthest payload from all the evaluated ones. But in some test scenarios, the test cases would not distributed evenly. So ART may not perform as expected [18].

Mutation testing has been proposed and studied extensively [13] as a method of evaluating the adequacy of test suites where the program under test is mutated to simulate faults. While detecting software vulnerabilities, it can also measure the detection capability of the test set. Because of these unique advantages, mutation testing has been rapidly developed in network security and is widely used. Shahriar and Zulkernine [22] defined SQLi specific mutation operators to evaluate the effectiveness of a test suite in finding SQLi vulnerabilities. Holler et al. [11] proposed an approach LangFuzz to test interpreters for security vulnerabilities, and it has been applied successfully to uncover defects in Mozilla JavaScript and PHP interpreter. However, there are still some problems. First, as the type of mutation increases, the test cases space will produce a combined explosion. Second, the output of some mutation operators could be filtered because of sensitive characters. But CT can detect failures triggered by interactions of parameters in the SUT with a covering array test suite generated by some sampling mechanisms [17].

Kuhn et al. [14] found that the faulty combinations are caused by the combination of no more than 6 parameters. Three or fewer parameters triggered a total of almost 90% of the failures in the application. Nie [3] gave a different coverage definition, which also shows only the coverage definition of the k-value schema and does not reflect the importance of failure-causing schema in the test cases set. Qi [19] indicated that variable strength reachability testing reaches a good tradeoff between the effectiveness and efficiency. It can keep the same capability of fault detection as exhaustive reachability testing while substantially reducing the number of synchronization sequences and decreasing the execution time in most cases. Therefore, to solve the problem of mutation testing mentioned above, we propose the CMM technique in combination with the characteristics of CT and mutation method.

## 3 Approach

### 3.1 A simple example of SQL injections

SQL injections come from a lack of encoding/escaping of user-controlled input when included in SQL queries. For example, a SQL statement is formed as follows:

A simplified example of one of our web applications is shown in Figure 1. The *user_id* is an input provided by users, which is concatenated with the rest of the SQL statement and then stored in the string variable *$id*. For instance, the id value entered by the user is *'OR '1'='1*, then the query generated by passing the value of id into the SQL statement, which is: " SELECT first_name, last_name FROM users WHERE user_id = '' *OR '1'='1*' ". The query logic of the SQL statement has been changed, because the statement after the OR operation of the query will always return true. Then the function *mysqli_query* sends the SQL statement to the database server, and return all the first_name and last_name information in the users table. The information obviously exceeds the query permission of users and the data table has been leaked. But nowdays,

there is always a filter function (such as, $\$id = tsFilter(\$id)$) between line 1 and line 2 of Figure 1 to solve this kind of problems. So from the point of view of penetration testers or attackers, it is necessary to enhance the mutation of the SQL statement to prevent it from being filtered,then further to access the system information for attacking.

### 3.2 Test Case Mutation Methods

*Sqlmap* is an open source penetration testing tool that automates the process of detecting and exploiting SQLi flaws and taking over of database. We import the mutation methods from *sqlmap*, and classify them into five groups according to their behavior:

- Comment : This method attach a SQL comment command at the back of the input test case, there are four comments: "–", "#", "%00", "and '0having'='0having'". These comments can make the SQL statement followed the input be valid or change the structure of the whole SQL. Example like this: the original input may be "1 and 1=1", after the mutation it would be "1 and 1=1 %00".

- Stringtamper : This kind of mutation method is aimed at changing the appearance of original input, it will cheat the target system as a normal input. However, the input is a malicious code. There are eight kinds of methods: "between", "bluecoat", "greatest", "lowercase", "nonrecursivereplacement", "randomcase", "randomcomments", "symboliclogical".

- Space2 : This kind of mutation method is aimed at changing the space in the input, because the target system will not allow a space in the input. There are fourteen kinds of methods: "halfversionedmorekeywords", "modsecurityversioned", "modsecurityzeroversioned", "multiplespaces", "overlongutf8", "space2c", "space2dash", "space2morehash", "space2mssqlblank", "space2randomblank", "space2hash", "space2mssqlhash", "space2mysqlblank", "space2mysqldash", "space2plus".

- Apostle : This kind of mutation method is aimed at changing the apostle in the input. There are two kinds of methods: "apostrophemask", "apostrophenullencode".

- Encoding : This kind of mutation method is aimed at changing the encoding of the original statement. there are five kinds of methods: "base64encode", "chardoubleencode", "charencode", "charunicodeencode", "percentage".

Algorithm 1 introduces the detailed process of the test case mutation methods. Given an valid test case input, the apply_MO function randomly applies one or more mutation operators to mutate this test case. Because the input values are diverse and specific, so the mutation operator may not always be effective. For example, the SQL statement would not be changed if a mutation operator just converts the single quotes to the double quotes. So before adding the mutated SQL statement into the test set TS, checking for repeatability is needed. After using a series of mutation methods, the expanded original test cases TS can be obtained from the input value set $I$, which could cover more vulnerabilities.

**Algorithm 1** Test Case Mutation Method

**Input:** $I$:A set of Legal input.
**Output:** $TS$:Test case space.
$T = \{\ \}$
**for** each input in $I$ **do**
  **while** *not max_tries* **do**
    $t$ = apply_MO(input)
    **if** t not in $T$ **then**
      Add $t$ into $TS$
    **end if**
  **end while**
**end for**
**return** $TS$

Table 1: A input model of CT

| Parameter | Input | Comment | Stringtamper |
|---|---|---|---|
| Value | 1:"1' or 1=1 or "<br>2:"1 or 1=1" | 0:Not be capable<br>1:""<br>2:"#"<br>3:"%00"<br>4:" '0having'='0having' " | 0:Not be capable<br>1:"between"<br>2:"bluecoat"<br>...<br>8:"symboliclogical" |
| Parameter | Space2 | Apostle | Encoding |
| Value | 0:Not be capable<br>1:"halfversionedmorekeywords"<br>2:"modsecurityversioned"<br>...<br>14:"space2randomblank" | 0:Not be capable<br>1:"apostrophemask"<br>2:"apostrophenullencode" | 0:Not be capable<br>1:"base64encode"<br>2:"chardoubleencode"<br>...<br>5:"percentage" |

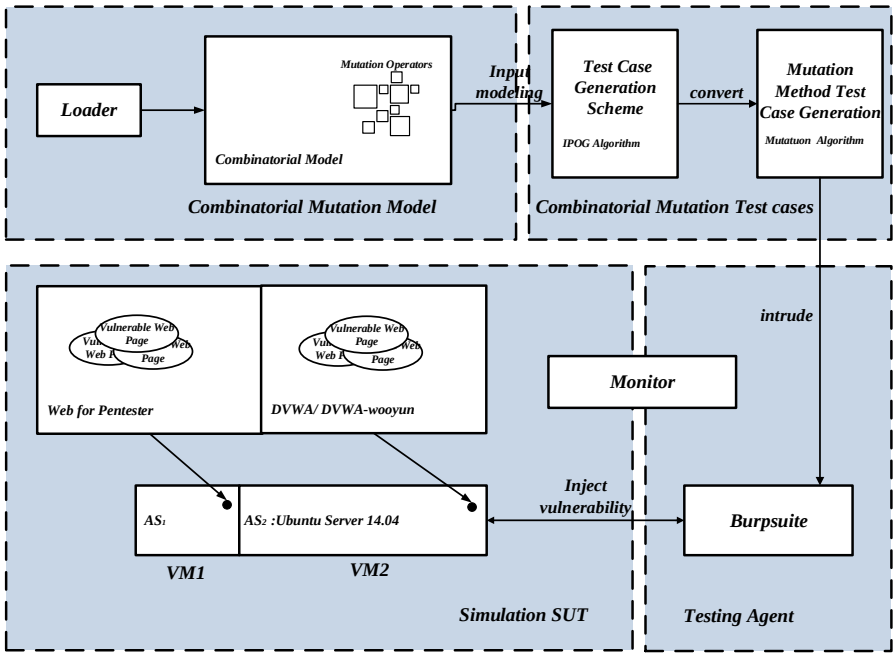


Fig. 2: SQLi vulnerability detection based on the combinatorial mutation testing

### 3.3 Combinatorial Mutation Method

As the number of mutation operators increases, the space explosion problem will be highlighted. The CT could reduce the number of test cases while ensuring coverage. Therefore, we propose a new CT-based mutation method to detect vulnerabilities, which is *Combinatorial Mutation Method*. It contains two main steps: 1) Produce test case generation scheme; 2) Output test set with the given scheme.

**CT-based test case generation scheme** Since the target system may have multiple filtering rules for the input, only one mutation method is not enough for attackers or penetration testers. Firstly, we parameterize the mutation methods to output the test case generation scheme.

Table 1 shows the input model of CT including the Six parameters and their corresponding values. For example, the parameter "Stringtamper" has nine values, of which " 1: "between" " means that the index "1" represents the mutation operator "between", and "1" has no numerical meaning. Particularly, "0:not be capable" is a placeholder for a null operation. If a test case has a value of "0", it means skipping this mutation operator and performing subsequent operations.

For instance, the test case generation scheme is {2, 3, 0, 0, 0, 0}, and the test case after the mutation method is "1 or 1 = 1% 00". In addition, since the parameters "Stringtamper" "Space2" and "Encoding" contain more mutation operators, the values are not all listed, and the omitted part can be seen in Subsection 3.2, and the omitted part is represented by "...".

*ACTS* is a test generation tool for constructing *t-way* (t=1,2,...,6) combinatorial test sets. The model of CT can be passed to *ACTS* to output the specific test case generation schemes. Several test generation algorithms are implemented in *ACTS*. These algorithms include IPOG, IPOG-D, IPOG-F and IPOG-F2. We use the IPOG algorithm to generate the covering array.

Since the strength of combination differs according to the target system, a tester cannot know the proper strength for the mutation. Thus, we generate different strength covering array from two to five. Furthermore, we also use the VSCT to generate test set, but this method need an interaction relationship of parameters in its subset. So we get subset according to the effective payload in the test results of *t-way* (Please see the experimental section for specific steps). With instruction of the specific scheme, test cases can be generated by different strength coverage.

**CMM-based test set generation** A set of test case generation schemes is parsed into test set with specific mutations. Algorithm 2 lists the IPOG-based *t-way* CMM process: the test case generation algorithm needs to pass the model's parameter CM and the coverage strength $k$ to the test case set generation module according to the CT model. Then the IPOG algorithm (line 2-16) [16] is used to generate TS set. Next, for each legal input, the test case mutation algorithm apply_MO is executed according to the index of the cover set TS. Similarly, VSCT also generates CMM test cases based on the IPOG algorithm, as shown in Algorithm 3. The difference between those two algorithms is that VSCT is based on the 2-way CT (line 2-16) test set TS, and the subset C has the *t-way*

**Algorithm 2** IPOG-based t-way CMM

**Input:** The parameter $f_1, f_2, ...f_n$ of the system S
**Input:** $k$: The strength of t-way CT
**Input:** The corresponding value set P $(P_1, P_2, ..., P_n)$ of parameter $f_1, f_2, ...f_n$
**Output:** $TS'$: Test cases space.
1: $T = \{\ \}$
2: $TS$=$\{(v_1, v_2, ..., v_k)|v_1 \in P_1, v_2 \in P_2, ..., v_k \in P_k\}$;
3: **if** n==k **then return end if**
4: **for** $P_i$(i=k+1,...,n) **do**
5: let $\pi$ be the set of $k$-way combinations of values involving parameter $P_i$ and $k-1$ parameters among the first $i-1$ parameters
6: **for** each test (o=$v_1, v_2, ..., v_{i-1}$) in $TS$ **do**
7: choose a value $v_i$ of $P_i$ and replace o with $o'$= $v_1, v_2, ..., v_{i-1}, v_i$ so that $o'$ covers the most number of k-way of values in $P_i$
8: remove from $P_i$ the combinations of values covered by $o'$
9: **end for**
10: **while** $TS$ does not cover the value pairs formed by $P_i$ and $P_1, P_2, ..., P_{i-1}$ **do**
11: Add a new test for the parameters $P_1, P_2, ..., P_i$ to the $TS$, remove from $\pi$ the k-way of values covered by $TS$
12: **end while**
13: **end for**
14: **for** each input in $TS$ **do**
15: **while** *not max_tries* **do**
16: $t$ = apply_MO(input)
17: **if** $t$ not in $T$ **then** Add $t$ into $TS'$ **end if**
18: **end while**
19: **end for**
20: **return** $TS'$

($t > 2$) interaction (line 17-21) relationship sample $S'$. Then we remove the combinations of values covered by $S'$ from $TS$, and the rest combinations are merged with $S'$ to generate a new test set $S$.

After the process, we pass the test cases set to the Web Security Auditing and Scanning tool *Burpsuite*[6], then the intruder mode in *Burpsuite* is applied to inject SQL vulnerabilities into Simulation $SUT$ to get the request statement. The SQLi vulnerability detection process based on the combinatorial mutation testing is shown in Figure 2.

## 4 Experiment And Results

### 4.1 Experiment Framework

In the experiment, Web server is deployed on a virtual machine, the operating system is Ubuntu14.04, as shown in the Fig. 3. We set three open source vulnerable Web applications on the server as the target system: *Web For Pentester* [7], *DVWA* [8], *DVWA-WooYun* [9]. Testing tool is set on the host, the OS is Windows7.

We visit the vulnerable pages on the server and then hijack the Web request by *Burpsuite* which is an integrated platform for performing security testing of

[6] https://portswigger.net/burp
[7] https://www.pentesterlab.com/
[8] http://www.dvwa.co.uk/
[9] http://sourceforge.net/projects/dvwa-wooyun/

**Algorithm 3** IPOG-based variable strength CMM

**Input:** The parameter $f_1, f_2, ...f_n$ of the system S
**Input:** $k$: The variable strength of subset C $\subseteq \{C_1, C_2, C_{n-1}\}$
**Input:** The corresponding value set P $(P_1, P_2, ..., P_n)$ of parameter $f_1, f_2, ...f_n$
**Output:** $TS'$: Test cases space.
1: $T = \{\ \}$
2: $TS$=$\{(v_1, v_2)|v_1 \in P_1, v_2 \in P_2\}$;
3: **if** n==2 **then return end if**
4: **for** $P_i$(i=3,4,...,n) **do**
5: let $\pi$ be the set of $i$-way combinations of values involving parameter $P_i$ and $i-1$ parameters among the first $i-1$ parameters
6: **for** each test (o=$v_1, v_2, ..., v_{i-1}$) in $TS$ **do**
7: choose a value $v_i$ of $P_i$ and replace o with $o'$= $v_1, v_2, ..., v_{i-1}, v_i$ so that $o'$ covers the most number of k-way of values in $\pi$
8: remove from $\pi$ the combinations of values covered by $o'$
9: **end for**
10: **while** $TS$ does not cover the value pairs formed by $P_i$ and $P_1, P_2, ..., P_{i-1}$ **do**
11: Add a new test for the parameters $P_1, P_2, ..., P_i$ to the $TS$, remove from $\pi$ the k-way of values covered by $TS$
12: **end while**
13: **end for**
14: **while** $C_i$(i=1,2,...,n-1) **do**
15: $S' = IPOG(k, C_i)$
16: remove from $TS$ the combinations of values covered by $S'$
17: $S = TS \cup S'$
18: **end while**
19: **for** each input in $S$ **do**
20: **while** *not max_tries* **do**
21: $t$= apply_MO(input)
22: **if** $t$ not in $T$ **then** Add $t$ into $TS'$ **end if**
23: **end while**
24: **end for**
25: **return** $TS'$

Web applications. The mutated input is loaded in intruder mode of *Burpsuite*, some important parameters in the request which are input of users are injected with our mutated input.

When a malicious input is successfully injected, these vulnerable Web applications will reply with a response page containing a list of users and passwords. Then we judge whether the injection point can be injected based on response pages. Here we choose these vulnerable Web applications because they have a high coverage of SQLi types, at the same time, there are different filter functions.

Pentester lab is one of the basics training for Web testing and summary of the most common vulnerabilities. This platform show how to manually find and exploit the vulnerabilities, which contains SQLi. *Web For Pentester* has 9 SQLi pages with different filter functions, but the last two pages are about how to use the vulnerability.

*DVWA* is a PHP/MySQL Web application that is damn vulnerable. Its main goals are to provide a legal environment for security experts to test their technology and tools, to help Web developers better understand the security issues of Web applications. In SQLi part, it contains general injection and blind injection with security level of low, medium and high. Low level has SQLi vulnerability, medium level has filter functions which want to protect the system, while high level is a standard safe programming paradigm.

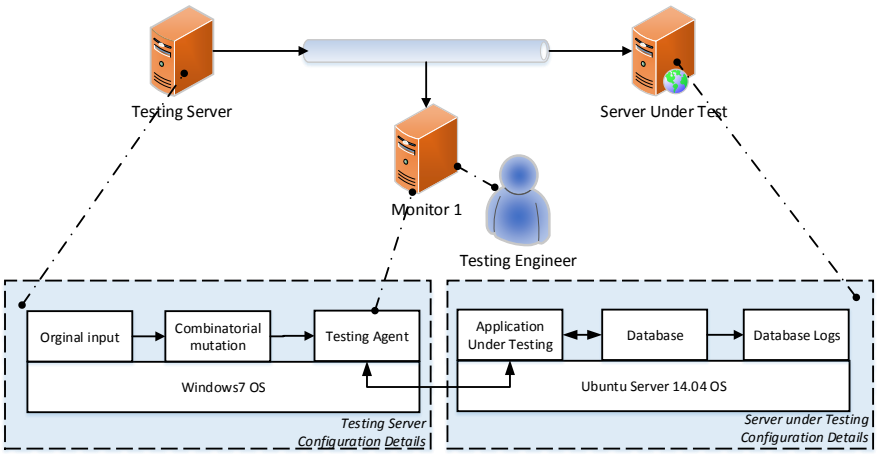


Fig. 3: Experiment Framework

*DVWA-WooYun* is a Wooyun OpenSource project based on DVWA project. It is based on real bug reported on *wooyun.org*. Each vulnerable page has a "View Source" button to show the php source code, which can help people understand the vulnerability.

### 4.2 Metrics

In our experiments, we use the *F-measure* (represented by $F$ in tables), *Efficiency* (represented by $E$ in tables) to demonstrate the capabilities of different approaches.

- *F-measure Metric*: *F-measure* calculates the expected number of payloads required to detect the first SQLi vulnerability. In other words, a lower *F-measure* value means fewer tests are used to accomplish the task. Therefore, if a testing strategy yields a lower *F-measure* value, it is considered to be more effective. Obviously the metric is affected by the order of the test cases, but in our experiment the order is random, so it is not enough to evaluate the CMM. As *sqlmap* would stop after the first effective test case were found, if *sqlmap* cannot stop after.

- *Efficiency Metric*: The *Efficiency* is a very common test case evaluation standard that reflects the percentage of effective test cases. It can effectively evaluate the quality of the test case when detecting vulnerability. The formula is expressed as follows: $Efficiency(\%) = (X/N) * 100\%$, where X is the number of test cases that can trigger vulnerabilities, and N is the size of test set. As *sqlmap* would stop after the first effective test case were found, the *Efficiency* of *sqlmap* is not calculated, and the corresponding results are represented by "-".

### 4.3 Results

As a comparison, we conduct a series of experiment with *sqlmap*, *FuzzDB*[10] and our method. Our method is a mutation method to generate aggressive test cases. The automated SQLi tool *sqlmap* requires manual judgment and implementation, and may need to try many times. *FuzzDB* is an open source database of attack patterns, predictable resource names, regex patterns for identifying interesting server responses and documentation resources, which is hosted at Google Code. We only choose the cross platform SQLi part of the dictionary.

In addition to using *t-way* (1<t<6) combination approach, we also consider combination method of variable strength to generate test sets to trigger more vulnerabilities. We analyze the *Efficiency* of the 3-way, 4-way, 5-way test set of the three benchmarks and the effective payload (effective test cases) in the experimental results, and find three parameter sets: {input, stringtamper, space2}, { comment, space2, apostle}, { input, comment, apostle} , in which the interaction relationships can make the payload more efficient. To balance the effectiveness and flexibility of the mutation-force combination method, the number of test cases should be set between the number of 2-way test cases and the number of 3-way test cases. In order to achieve this goal, the six parameters based on the 2-way CT generate test set. Then respectively add two subset {{ input, stringtamper, space2}, { input, comment, apostle}}, {{ input, stringtamper, space2}, {comment, space2, apostle}}. these subsets have the 3-way ($t > 2$) interaction

[10] https://github.com/fuzzdb-project/fuzzdb

Table 2: Web for Pentester

| | Example 1 | | Example 2 | | Example 3 | | Example 4 | | Example 5 | | Example 6 | | Example 7 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *F* | *E* | *F* | *E* | *F* | *E* | *F* | *E* | *F* | *E* | *F* | *E* | *F* | *E* |
| 1-way | 0 | 0.00% | 0 | 0.00% | 0 | 0.00% | 7 | 13.33% | 7 | 26.67% | 0 | 0.00% | 9 | 6.67% |
| 2-way | 14 | 5.19% | 26 | 3.70% | 0 | 0.00% | 12 | 5.93% | 8 | 20.74% | 19 | 3.70% | 15 | 1.48% |
| 3-way | 14 | 6.38% | 14 | 3.39% | 211 | 0.14% | 16 | 6.68% | 8 | 18.73% | 26 | 3.53% | 26 | 3.26% |
| 4-way | 2 | 6.46% | 10 | 3.44% | 63 | 0.14% | 4 | 5.96% | 4 | 17.06% | 45 | 3.62% | 9 | 1.74% |
| 5-way | 11 | 5.13% | 92 | 2.65% | 2805 | 0.08% | 7 | 4.62% | 1 | 15.60% | 1 | 3.37% | 30 | 1.74% |
| $VSCA_1$ | 20 | 8.14% | 45 | 4.65% | 71 | 0.33% | 15 | 8.14% | 4 | 25.25% | 15 | 1.16% | 13 | 1.83% |
| $VSCA_2$ | 13 | 7.45% | 23 | 5.25% | 129 | 1.03% | 5 | 8.46% | 5 | 26.73% | 12 | 0.85% | 9 | 1.69% |
| *sqlmap* | 40 | – | 44* | – | 58* | – | 40 | – | 41 | – | 80 | – | – | – |
| *FuzzDB* | 25 | 6.64% | 42 | 0.47% | 42 | 0.47% | 17 | 1.42% | 17 | 1.42% | 17 | 1.42% | 0 | 0.00% |

Table 3: DVWA

| | SQLi(low) | | SQLi(blind)(low) | | SQLi(medium) | | SQLi(blind)(medium) | |
|---|---|---|---|---|---|---|---|---|
| | *F* | *E* | *F* | *E* | *F* | *E* | *F* | *E* |
| 1-way | 0 | 0.00% | 0 | 0.00% | 7 | 13.33% | 7 | 13.33% |
| 2-way | 26 | 4.44% | 26 | 4.44% | 12 | 5.93% | 12 | 5.93% |
| 3-way | 14 | 5.29% | 14 | 5.29% | 16 | 6.78% | 16 | 6.78% |
| 4-way | 2 | 5.78% | 2 | 5.78% | 4 | 5.96% | 4 | 5.96% |
| 5-way | 11 | 4.09% | 11 | 4.09% | 7 | 4.62% | 7 | 4.62% |
| $VSCA_1$ | 45 | 5.65% | 45 | 5.65% | 49 | 5.48% | 49 | 5.48% |
| $VSCA_2$ | 46 | 5.41% | 46 | 5.41% | 54 | 5.25% | 54 | 5.25% |
| *sqlmap* | 46 | – | 104 | – | 40 | – | 102 | – |
| *FuzzDB* | 25 | 7.11% | 25 | 7.11% | 17 | 1.42% | 17 | 1.42% |

Table 4: DVWA wooyun-I

| | Sqli QUERY_STRING | | Sqli filter #02-Once | | Sqli Mysql #01 | | No [Comma] Sqli | |
|---|---|---|---|---|---|---|---|---|
| | *F* | *E* | *F* | *E* | *F* | *E* | *F* | *E* |
| 1-way | 0 | 0.00% | 0 | 0.00% | 7 | 20.00% | 0 | 0.00% |
| 2-way | 58 | 2.22% | 26 | 2.96% | 15 | 18.52% | 26 | 4.44% |
| 3-way | 8 | 2.71% | 14 | 4.48% | 8 | 16.82% | 14 | 6.38% |
| 4-way | 143 | 1.83% | 2 | 4.22% | 18 | 13.71% | 2 | 7.29% |
| 5-way | 1 | 1.61% | 11 | 2.87% | 1 | 12.56% | 11 | 5.45% |
| $VSCA_1$ | 6 | 2.82% | 45 | 4.32% | 4 | 19.44% | 45 | 5.65% |
| $VSCA_2$ | 8 | 3.05% | 46 | 4.06% | 8 | 20.30% | 68 | 5.41% |
| *sqlmap* | – | – | 73* | – | 380 | – | – | – |
| *FuzzDB* | 0 | 0.00% | 25 | 6.16% | 17 | 1.90% | 25 | 6.64% |

Table 5: DVWA wooyun-II

| | Sqli using[Slashes] | | Sqli filter #02-80sec | | Sqli filter#01 | | No [Space] Sqli | |
|---|---|---|---|---|---|---|---|---|
| | *F* | *E* | *F* | *E* | *F* | *E* | *F* | *E* |
| 1-way | 0 | 0.00% | 14 | 6.67% | 13 | 6.67% | 0 | 0.00% |
| 2-way | 0 | 0.00% | 15 | 7.41% | 58 | 1.48% | 26 | 3.70% |
| 3-way | 0 | 0.00% | 16 | 6.78% | 16 | 1.63% | 14 | 4.48% |
| 4-way | 0 | 0.00% | 27 | 6.37% | 18 | 1.83% | 5 | 4.81% |
| 5-way | 0 | 0.00% | 1 | 5.71% | 1 | 1.78% | 11 | 3.39% |
| $VSCA_1$ | 0 | 0.00% | 13 | 3.99% | 8 | 1.16% | 45 | 5.65% |
| $VSCA_2$ | 0 | 0.00% | 21 | 4.91% | 45 | 1.18% | 45 | 4.40% |
| *sqlmap* | – | – | – | – | – | – | 2626 | – |
| *FuzzDB* | 0 | 0.00% | 17 | 1.90% | 33 | 0.47% | 25 | 4.74% |

relationship. Using 3-way CT, parameters in the subset generate test cases $S'$. Then we remove the combinations of values covered by $S'$ from $TS$, and the rest combinations are merged with $S'$ to generate a new test set $VSCA_1$, $VSCA_2$,the gray part of these tables are *VSCT*.

To prove the validity of our method, we calculate the average time cost of *t-way*. The number of test cases generated by different methods is different and constant. 1-way takes less than 0.01s to generate 15 test cases. 2-way takes 0.45s to generate 135 test cases. 3-way takes 14.8s to generate 737 test cases. 4-way takes 417.8s to generate 2181 test cases. 5-way takes 2158.4s to generate 5287 test cases. $VSCA_1$ takes 43.0s to generate 602 test cases. $VSCA_2$ takes 42.0s to generate 591 test cases. The evaluation of effectiveness in Table $2-5$, where the total number of test cases for each way is derived from test cases sets given above.

We performe an average of 10 experiments on *t-way* as the test results to prevent the randomness of the conclusion and to verify the method more accurately. As can be seen in Table 2, in most cases there are better *F-measure* and *Efficiency* for 2-way and above than the other two methods, especially Example 4, Example 5 and Example 7. In addition, *sqlmap* can exploit this vulnerability only if it implements manual judgment and specifies mutation methods in Example 2 and Example 3(marked with * in the table). Composite-based test case generation methods perform better than other methods in testing these pages. In Example 7, the other two methods are invalid, but *t-way* works well and has

Table 6: Comparison with ART4SQLi

| | Web for Pentester SQLi Example 1 | | DVWA wooyun sql query string | | DVWA wooyun sql filter 01 | |
|---|---|---|---|---|---|---|
| | *F* | *E* | *F* | *E* | *F* | *E* |
| 1 way | 0 | 0.00% | 0 | 0.00% | 9 | 6.67% |
| 2 way | 12 | 2.22% | 41 | 2.22% | 41 | 1.48% |
| 3 way | 23 | 6.38% | 13 | 2.71% | 93 | 1.63% |
| 4 way | 8 | 6.52% | 62 | 1.82% | 52 | 1.79% |
| 5 way | 15 | 5.12% | 30 | 1.59% | 46 | 1.83% |
| $VSCA_1$ | 13 | 8.14% | 6 | 2.82% | 13 | 1.16% |
| $VSCA_2$ | 15 | 7.29% | 14 | 3.05% | 87 | 1.18% |
| ART4SQLi | 49 | 1.58% | 1359 | 0.04% | 1183 | 0.06% |

a superiority in the discovery of potential vulnerabilities under multiple filtering rules.

As shown in Table 3, *t-way* and *VSCT* can ignore the type of injection, whether it is normal or blind. Although there are some fluctuations, *t-way* still has advantages over *sqlmap* and *FuzzDB* on the whole, especially in *SQLi(medium)* and *SQLi(blind)(medium)* with better *Efficiency*. Although *VSCT*'s *F-measure* and *Efficiency* decrease slightly with difficulty, their *Efficiency* is still higher than *FuzzDB*. It shows that *t-way* is good at exploring vulnerabilities when there are interaction between the various factors.

In Table 4 and 5, with different filtering rules from real bug reports, *t-way* also works better than the other two methods. But the page *Sqli using[Slashes]* may have some problems in the application.

Seen as a whole, Example 2, 4, 5, 6, 7 in Table 2, *t-way t-way* and *VSCT* have a better efficiency than *FuzzDB*, especially in Example 5. *VSCT* is better than *t-way* and *FuzzDB* in terms of *F-measure* and *Efficiency* in Example 1 and 2. For Table 3, 4 and and 5 considering *F-measure*, the number of payload and *Efficiency*, CMM are irreplaceable by *sqlmap* and *FuzzDB*.

Above all, combined with Table $2 - 5$, if we have requirements for both evaluation criteria *F-measure* and *Efficiency*, the high-way ($4 \leq t \leq 5$) covering array is undoubtedly a good choice. But for all Benchmarks, *VSCT* is more efficient than any *t-way*. That is, if we want to improve the quality of test cases and enhance the reliability of Web applications, we need to find the subset relationship between the parameters based on the *t-way*'s effective test cases or professional test experience to further generate the variable strength test set.

In order to compare with our previous work, ART4SQLi, we conduct experiments under the same conditions in the ART4SQLi experiment and showed them in Table 6. In addition, ART4SQLi generated 76105 test cases. As can be seen in Table 6, although 1-way cannot successfully inject in Web for Pentester SQLi Example 1 and DVWA wooyun sql query string, it still performs good in DVWA wooyun sql filter 01 with only 15 test cases. In other pages and *t-ways* and *VSCTs* perform much better than *ART4SQLi*. *ART4SQLi*'s strategy of selecting the farthest test case from the selected onetest is talented. But the test cases set of *ART4SQLi* is too large (76105 test cases) and it lacks of function to filter out invalid mutants, which leads to bad *F-measure* and *Efficiency*.

## 5 Conclusion

SQL injections have been ranked as one of the most dangerous Web application vulnerabilities. Researchers have proposed a wide range of techniques to address the problem of detecting SQL injections. In this paper, we introduce a new technique CMM to detect vulnerabilities for SQLi. CMM first establish a CT modeling base on mutation operators of *sqlmap*. After that, test case generation schemes can be generated by specific methods, such as *t-way* CT and VSCT. Then the schemes are converted into mutation sets. Finally, we use the Web Security Auditing and Scanning tool to inject SQL vulnerabilities into Simulation SUT.

The experiments adopt three open source SQLi benchmarks, and the results show that high-way ($4 \leq t \leq 5$) covering arrays are undoubtedly better than the others, considering the *F-measure* and *Efficiency*. But only for *Efficiency*, variable strength is also a good choice, if we know the constraint relationship between parameters in advance. The CMM approach is effective in finding SQLi vulnerabilities with multiple filtering rules. In the future, we also expect to apply CMM to other Web application vulnerabilities, such as XSS.

## Acknowledgement

We would like to thank anonymous reviewers for their invaluable comments and suggestions on improving this work. This work is supported by National Natural Science Foundation of China (NSFC) (grant No. 61572150), and the Fundamental Research Funds for the Central Universities of DUT (No.DUT17RC(3)097).

## References

1. Appelt, D., Nguyen, C.D., Briand, L.C., Alshahwan, N.: Automated testing for sql injection vulnerabilities: an input mutation approach. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 259–269. ACM (2014)
2. Bisht, P., Madhusudan, P., Venkatakrishnan, V.: Candid: Dynamic candidate evaluations for automatic prevention of sql injection attacks. ACM Transactions on Information and System Security (TISSEC) **13**(2), 14 (2010)
3. Changhai, N., Leung, H.: A survey of combinatorial testing. Acm Computing Surveys **43**(2), 1–29 (2011)
4. Chen, J., Zhu, L., Chen, T.Y., Huang, R., Towey, D., Kuo, F.C., Guo, Y.: An adaptive sequence approach for oos test case prioritization. In: IEEE International Symposium on Software Reliability Engineering Workshops (2016)
5. Cohen, M.B., Gibbons, P.B., Mugridge, W.B., Colbourn, C.J.: Constructing test suites for interaction testing. In: Proceedings of the 25th international conference on Software engineering. pp. 38–48. IEEE Computer Society (2003)
6. Deepa, G., Thilagam, P.S.: Securing web applications from injection and logic vulnerabilities: Approaches and challenges. Information & Software Technology **74**, 160–180 (2016)

7. Deshpande, V.M., Nair, D.M.K., Shah, D.: Major web application threats for data privacy & security–detection, analysis and mitigation strategies. International Journal of Scientific Research in Science and Technology **3**(7), 182–198 (2017)
8. Fossi, M., Turner, D., Johnson, E., Mack, T., Adams, T., Blackbird, J., Entwisle, S., Graveland, B., McKinney, D., Mulcahy, J., et al.: Symantec global internet security threat report. White Paper, Symantec Enterprise Security **1** (2009)
9. Gu, H., Zhang, J., Liu, T., Hu, M., Zhou, J., Wei, T., Chen, M.: Diava: A traffic-based framework for detection of sql injection attacks and vulnerability analysis of leaked data. IEEE Transactions on Reliability (2019)
10. Hagar, J.D., Wissink, T.L., Kuhn, D.R., Kacker, R.N.: Introducing combinatorial testing in a large organization. Computer **48**(4), 64–72 (2015)
11. Holler, C., Herzig, K., Zeller, A.: Fuzzing with code fragments. In: Presented as part of the 21st USENIX Security Symposium (USENIX Security 12). pp. 445–458 (2012)
12. Huang, Y., Fu, C., Xuan, C., Hao, G., He, X., Jin, L., Liu, Z.: A mutation approach of detecting sql injection vulnerabilities. In: International Conference on Cloud Computing & Security (2017)
13. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE transactions on software engineering **37**(5), 649–678 (2011)
14. Kuhn, D.R., Kacker, R.N., Lei, Y.: Introduction to combinatorial testing. CRC press (2013)
15. Lee, I., Jeong, S., Yeo, S., Moon, J.: A novel method for sql injection attack detection based on removing sql query attribute values. Mathematical & Computer Modelling **55**(1), 58–68 (2012)
16. Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J.: Ipog: A general strategy for t-way software testing. In: Engineering of Computer-based Systems, Ecbs 07 IEEE International Conference & Workshops on the (2010)
17. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Computing Surveys (CSUR) **43**(2), 11 (2011)
18. Nie, C., Wu, H., Niu, X., Kuo, F.C., Leung, H., Colbourn, C.J.: Combinatorial testing, random testing, and adaptive random testing for detecting interaction triggered failures. Information & Software Technology **62**, 198–213 (2015)
19. Qi, X., He, J., Wang, P., Zhou, H.: Variable strength combinatorial testing of concurrent programs. Frontiers of Computer Science **10**(4), 631–643 (2016)
20. Sabharwal, S., Aggarwal, M.: Variable strength interaction test set generation using multi objective genetic algorithms. In: 2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI). pp. 2049–2053. IEEE (2015)
21. Sadeghian, A., Zamani, M., Manaf, A.A.: A taxonomy of sql injection detection and prevention techniques. In: Informatics and Creative Multimedia (ICICM), 2013 International Conference on. pp. 53–56. IEEE (2013)
22. Shahriar, H., Zulkernine, M.: Music: Mutation-based sql injection vulnerability checking. In: Quality Software, 2008. QSIC'08. The Eighth International Conference on. pp. 77–86. IEEE (2008)
23. Simos, D.E., Zivanovic, J., Leithner, M.: Automated combinatorial testing for detecting sql vulnerabilities in web applications. In: Proceedings of the 14th International Workshop on Automation of Software Test. pp. 55–61. IEEE Press (2019)
24. Yu, L., Yu, L., Kacker, R.N., Kuhn, D.R.: Acts: A combinatorial test generation tool. In: IEEE Sixth International Conference on Software Testing (2013)
25. Zhang, L., Zhang, D., Wang, C., Zhao, J., Zhang, Z.: Art4sqli: The art of sql injection vulnerability discovery. IEEE Transactions on Reliability (2019)