根据推论的

An Inferential Metamorphic Testing Approach to Reduce False Positives in SQLIV Penetration Test

误报 SQL注射缺陷 渗透

Lei Liu[1], Guoxin Su[2], Jing Xu[1], Biao Zhang[1], Jiehui Kang[1], Sihan Xu[1], Peng Li[1] and Guannan Si[3]
[1]College of Computer and Control Engineering, Nankai University, Tianjin, China
[2]School of Computing and Information Technology, University of Wollongong, Australia
[3]School of Information Science and Electrical Engineering, Shandong Jiaotong University, Jinan, China

注入缺陷

***Abstract*—SQL Injection Vulnerability (SQLIV) has been the top-ranked threat to the Web security consistently for many years. Penetration tests, which are a most widely adopted technique to detect SQLIV, are usually affected by testing inaccuracy. This problem is even worse in inference-based, blind penetration tests for online Web sites, where Web page variations (such as those caused by inbuilt dynamic modules or user interactions) may lead to a large number of False Positives (FP). We present a novel approach called Inferential Metamorphic Testing (IMT) to reduce FP in SQLIV penetration tests. First, we define the notion of Inferential Metamorphic Relations (IMR), which is inherited from Mutational Metamorphic Testing (MMT). Second, we present a set of logic operators and mutation operators for generating IMR and deducting the background testing context. Finally, we present an iterative IMT process, which is based on the heuristic IMR generation and the background testing context deduction. Our empirical study demonstrates the effectiveness of our approach by a comparison to three famous SQLIV penetration test tools.**

演绎

启发式的

Keywords*—web vulnerability; penetration test; metamorphic testing; SQL injection; mutation testing; inference-based testing

严峻的

利用

I. INTRODUCTION

Along with the development of Internet, the Web security issues have become increasingly severe. Among all the Web application vulnerabilities, *SQL Injection Vulnerability* (SQLIV) has been one of the top-most vulnerabilities for a long time [1], because SQLIV can be exploited by attackers through specially designed input on Web application to alter the logic of SQL queries into the background database [2]. Penetration tests, which are one of the most popular SQLIV testing technique, are a kind of dynamic tests that aim to expose SQLIV through the mock attacks of testers [3]. However, one highly disturbing problem of this technique is the testing inaccuracy , especially reflected in high False Positives (FP), which may reduce the credibility of testing tools and cost longer time for manual checking.

Blind SQLIV penetration tests are an inference-based technique that relies on observing the variation of a Web page content caused by the SQL queries with "*True*" or "*False*" logic from a tester [4]. This technique is proven to be effective in finding SQLIV even when database errors are hidden. However, blind SQLIV tests tend to result in a large number of FP, because of the frequent changes of Web page contents caused by user interactions and dynamic modules [5]. Furthermore, because of the increasingly complexity of Web structures (such as a search engine), back-end programs of a Web page may exhibit behaviors that are similar to SQLIV. These dynamic variations of Web page contents actually disable the traditional test oracles to some extent, making the FP problem as an oracle problem essentially.

Many works have been done on improving the detecting capabilities of SQLIV penetration tests. Some focus on combining static and dynamic testing approaches that usually use static code analysis or server proxy to increase the testing capability of SQLIV penetration tests [6][7]. But, the need to access the source code or setting proxy by altering server programs, which are often infeasible for legacy systems or third-party outsource, limits the applicability of those approaches. Other works including test case mutation based methods [8][9] focus on increasing the testing coverage [10] and improving testing procedures [11]. However, few existing work deals with the problem of FP, especially for blind SQLIV penetration tests.

To address the FP problem in (especially blind) SQLIV penetration tests, in this paper we proposes an *Inferential Metamorphic Testing* (IMT) approach. Our IMT approach is based on the technique of Mutational Metamorphic Testing (MMT) [12], which is a variant of the Metamorphic Testing (MT) [13] by employing test case mutations to solve the oracle problem. In our IMT approach, we first define a relation called the *Inferential Metamorphic Relations* (IMR). Then, we introduce a set of *Logic Operators* (LO) and *Mutation Operators* (MO), to generate IMR and deduce *Inferential Testing Context* (ITC). Finally, we present an iterative IMT process based on the heuristic IMR generation and ITC deduction. Our empirical study shows the effectiveness of our approach by a comparison to three famous SQLIV penetration test tools.

The remainder of this paper is organized as follows: Section II describes backgrounds and related works. Section III presents our IMT approach. Section IV presents the empirical study. Finally, Section V concludes the paper.

II. BACKGROUNDS AND RELATED WORKS

SQLIV occurs when specially crafted SQL clause inputs reach the background database of a Web application, and alter the logic of the original SQL queries with the attacker's intentions [3][4], which may cause Serious security issues. Lots of online and legacy Web applications are exposed to the threat of SQLIV. Therefore, researches on discovering SQLIV on these kinds of online systems are very important.

And penetration test [3][6] is one of the most important dynamic testing method for this situation.

Works on SQLIV penetration test include three main aspects: information gathering [11][14], test case generation [9][10] and response analysis [9][14], most of which are focused on increasing testing coverage [7][9][10] or improving testing procedures [8][10][11]. However, the insufficiency of testing accuracy (ie., high FP) in SQLIV penetration test has always been one of the most acute problems because of the lack of background testing context [4][7][12]. Some researching works have been conducted to reduce the impact of insufficient accuracy (ie., high FP), most of which are combined static and dynamic approaches [2][3][5]. However, the needs of accessing the source code or setting proxy by altering server programs is not always be feasible in situations of legacy systems or third-party outsource and thus limits the applicability of those methods. For black-box SQLIV penetration tests, the researches on solving the problem of high FP are still insufficient. Moreover, the FP issue has become increasingly severe after the appearance of inference-based SQLIV [2][5].

Fig. 1 shows an example of the classic decision process of blind SQLIV penetration tests, which can also explain the cause of FP in SQLIV penetration test. Traditional SQLIV can be exposed by database errors, but blind SQLIV can only be revealed by inferences, which database errors are likely hidden [2]. We assume the URL of the targeted HTTP request is "http://WebShop.Example.com/Customers? ID = 001&Session = 123", and the original background SQL query of this request is like the query: "SELECT * FROM Customers WHERE ID = 001 AND Session = 123". If the inputs of the above mentioned request are not properly filtered and "*Session=123*" is the injection point, a classic blind SQLIV penetration test pattern can be described as "*(AND 1=1 → Similar) & (AND 1=2 → Different)*", which can also be described as the route "$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_5$". Other decision paths show the situations of no vulnerabilities.

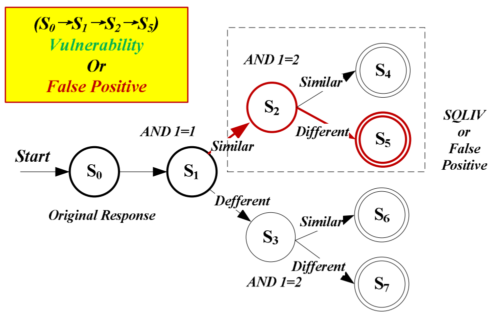

Figure 1. Blind SQL Injection Vulnerability (or FP) decision Process

This kind of SQLIV can be reduced by proper filtration and protection to some extent, but it still has the possibility of the occurrence of FP like the route "$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_5$". The main reasons for this kind of FP can be concluded to three types: (1) random changes of HTTP responses caused by user interaction; (2) variation of Web page contents caused by dynamic modules, such as periodic advertisements; (3) the background functions have the similar behaviors to a SQLIV for some specific purposes, such as basic honey pot system. Such kind of FP are hardly avoided, and too much non-systematic verification may cause overmuch False Negatives (FN) and redundancy.

To address the above problem, this paper proposes a systematic approach to reducing the influence of FP during SQLIV penetration tests based on Mutational Metamorphic Testing (MMT). MMT is a mutation based test approach to tackle the oracle problem and generate test cases [12][13]. Its basic philosophy is to expose errors through the analyzing of related input-output pairs based on Metamorphic Relations (MR). The FP problem in SQLIV penetration tests is similar to the oracle problem, because those tests lack an absolutely accurate procedure to identify FP.

III. THE PROPOSED APPROACH

A. *Inferential Metamorphic Testing*

We propose an *Inferential Metamorphic Testing* (IMT) approach based on Mutational Metamorphic Testing (MMT) [12][13] to test FP in blind SQLIV penetration tests. Here, we define a test case that exposes a SQLIV *alert* as a *seed*. Our target is to verify whether a SQLIV *alert* is FP or not. We first define the *Inferential Metamorphic Relation* (IMR). We use *P* to denote the original program context under test, including the background SQL query and related program modules. Let *I* be the input test cases domain, and *O* be corresponding output domain of *I* that includes the *Response Logic* of "*True*" and "*False*". Let f_n be a test case mutation function on *I* with applicability conditions $C(x_1, x_2, ..., x_n)$.

Definition 1: an *Inferential Metamorphic Relation (IMR)* is represented as follows:

$$R(P(x_1), P(x_2), ..., P(x_n), P(f_1(x_1, x_2, ..., x_n)), P(f_2(x_1, x_2, ..., x_n)), ..., P(f_m(x_1, x_2, ..., x_n))).$$

In other words, an *IMR* is a relation *R* on $O^{(m+n)}$ such that the *Response Logic* of *P* is *consistent* (shows the *Expected Logic*) on inputs $x_1, x_2, ..., x_n \in I$ and $f_1, f_2, ..., f_m$ are applicable on $x_1, x_2, ..., x_n$ imply that $R(P(x_1), P(x_2), ..., P(x_n), P(f_1(x_1, x_2, ..., x_n)), P(f_2(x_1, x_2, ..., x_n)), ..., P(f_m(x_1, x_2, ..., x_n)))$, where $x_1, x_2, ..., x_n$ are the *seed* test cases, and $f_1(x_1, x_2, ..., x_n)$, $f_2(x_1, x_2, ..., x_n)$, ..., $f_m(x_1, x_2, ..., x_n)$ are the mutation functions on $x_1, x_2, ..., x_n$.

The *Response Logic* here is the logic of the similarity of the injection response content (such as S_1 in Fig. 1) with the original response content (such S_0 in Fig.1), including "*True*" or "*False*" corresponding to "*Similar*" or "*Different*". The *Expected Logic* is the *Response Logic* of *P* injected by *seeds* $x_1, x_2, ..., x_n$. The *Response Logic* of mutation test cases $f_1(x_1, x_2, ..., x_n)$, $f_2(x_1, x_2, ..., x_n)$, ..., $f_m(x_1, x_2, ..., x_n)$ should be *consistent* with the *Expected Logic* if they are valid, where *consistent* represents the consistency of the background logical regularity, such as "*True=!False*". If the IMR $R(P(x_1), P(x_2), ..., P(x_n), P(f_1(x_1, x_2, ..., x_n)), ..., P(f_m(x_1, x_2, ..., x_n)))$ is verified to be *consistent*, then $f_1(x_1, x_2, ..., x_n)$, $f_2(x_1, x_2, ..., x_n)$, ..., $f_m(x_1, x_2, ..., x_n)$ can be classified as *New Seed Test Cases* (NSTC). An IMR can be described as follows:

$$C(x_1, x_2, ..., x_n) \Rightarrow R(P(x_1), P(x_2), ..., P(x_n), P(f_1(x_1, x_2, ..., x_n)), P(f_2(x_1, x_2, ..., x_n)), ..., P(f_m(x_1, x_2, ..., x_n)))$$

Consider the logical structure of background SQL query in *P*, such as the one in the example in Fig. 1. A mutation function *f(y)* is a combination of test case mutation operators, which can be defined on the input domain *I* of blind SQLIV penetration test cases "*f(AND 1=1) = 'AND 1=1 -- '*", where *f(y)* adds a comment symbol "--" at the end of the test case "*AND 1=1*". The Response Logic result of *P("AND 1=1")* is the *Expected Logic* of the injection logic of the mutation of the test case "*AND 1=1*". The above mutation function can derive an IMR such as:

$$C(AND 1=1): P(AND\ 1=1)=True \ \wedge \ P(AND\ 1=1--\ )=True$$
$$\Rightarrow IMR:\ P(f(AND\ 1=1)) = P(AND\ 1=1)$$

The applicability condition *C(AND 1=1)* here is *P(AND 1=1)=True*, which is the *Response Logic* of the *seed* test case "*AND 1=1*". After we inject the mutation test case "*AND 1=1-- *", the *Response Logic* of *P(AND 1=1--)* is still "*True*", which can derive the above IMR. Making inference about testing context is a important feature in our approach because the testing context is unknown. The whole process of IMT for penetration tests can be described as follows:

1) For a target injection point, we choose a set of test cases (which involve exposing SQLIV *alerts*) as *seed Test Cases* (STC), i.e., $STC=\{x_1, x_2, ..., x_n\}$, and make *NSTC=STC*, where *NSTC* is the test case set of new *seeds*.

2) We infer the *Inferential Testing Context* (ITC) from *seed* test case set *NSTC* and conditions *C(NSTC)*, and deduce a new mutation function set $F=(f_1, f_2, ..., f_m)$ and a new $IMR= MR(P(x_1), P(x_2), ..., P(x_n), P(f_1(x_1, x_2,..., x_n)), P(f_2(x_1, x_2,..., x_n)), ..., P(f_m(x_1, x_2,..., x_n)))$ from *C(NSTC)* and the *Inferential Testing Context* (ITC).

3) We apply every mutation function in set $F=(f_1, f_2, ..., f_m)$ on each *seed* test case *x* ($x \in NSTC$) and get a *Mutational Test Case* (MTC) set, i.e, $MTC=(f_1(x_1, x_2,..., x_n), f_2(x_1, x_2,..., x_n), ..., f_m(x_1, x_2,..., x_n))$.

4) We inject all test cases in the mutation set MTC to *P*, and record all the output *Response Logic* $P(f_1(x_1, x_2,..., x_n)), P(f_2(x_1, x_2,..., x_n)), ..., P(f_m(x_1, x_2,..., x_n))$. If $P(f_m(x_1, x_2,..., x_n))$ is consistent with the deduced IMR, then we consider this input-output pair as an *evidence*. If so, we add $f_m(x_1, x_2,..., x_n)$ to set NSTC and add the input-output pair $(f_m(x_1, x_2,..., x_n), P(f_m(x_1, x_2,..., x_n)))$ to the condition set *C(NSTC)*.

5) We stop the *IMT* testing process whenever some prescribed *Testing Threshold Expression* (TTE) is satisfied. And if another threshold *Evidence Threshold Expression* (ETE) is not satisfied. Then we consider this as an FP, and the testing procedure jumps to step *2)* for next round of IMT.

From the above process of IMT, we can see that the standard of *FP* in this paper meets the *Threshold Expression* $(!ETE \wedge TTE)$. During an IMT, the *Inferential Testing Context* is based on inferences, so it is impossible to meet every IMRs. We use the threshold expression ETE to judge whether it is a FP. For instance, if we have "$!ETE: (N_E < 4) \vee (N_T < 2) \vee (N_F < 2)$" ($N_E, N_T, N_F$ are variables defined by tester, which respectively represent the number of total *evidences*, "*True*" response logic evidences and "*False*" response logic evidences). And expression "$TTE: N_S > 100$" (N_S represents the number of the generated test cases) means that, if the total number of *evidences* is less than four, or if the number of the *evidences* with the *Response Logic* "*True*" or "*False*" is less than two, then this *alert* is an FP.

B. IMT Operators

1) Mutation Operators

The proposed approach is based on test case mutation [8][12]. We introduce four kinds of *Mutation Operators* (MO) in this paper, including *Keyword Mutation Operators* (KMO), *Condition Changing Operators* (CCO), *Syntax Changing Operators* (SCO) and *Mathematical Mutation Operators* (MMO), as showed in Table I. Without further specification, the syntax of the operators is based on the Mysql database which is the most popular open source relational database.

TABLE I. THE PORPOSED MUTATION OPERATORS

Cat.	Operators	Mutation Operators Description
KMO	*M_{KW}(input, kw)*	Keyword mutation operation, *kw ∈{AND, OR, IF, SLEEP, HAVING, LIKE, IN, UNION, ORDER}*
CCO	*M_T (input)*	Changing the logic of the input (test case) to "*TRUE*" logic predicate (Tautology)
	M_F (input)	Changing the logic of the input (test case) to "*FALSE*" logic predicate (Contradiction)
SCO	*M_{AK}(input, kw, L)*	Adding a clause with certain keyword to the input (test case), *kw ∈{AND, OR, LIKE, IN, ...}*
	M_{RK}(input, kw, L)	Removing a clause with certain keyword to the input (test case), *kw ∈{AND, OR, LIKE, IN, ...}*
	M_{CMT} (input)	Adding a comment operator to the end of the input (test case), such as "*AND 1=2*" to "*AND 1=2 --* "
MMO	*M_{RD} (input)*	Randomizing Number or String Value in the input (test case), such as "*OR 1=1*" to "*OR 756=756*"
	M_C (input)	Adding math calculating operations into the input (test case), such as "*1=2*" to "*(11-10)=(1+1)*"

KMO includes M_{KW}, and $M_{KW}(input, kw)$ represents the operation that changes first keyword to "*kw*". For example, M_{KW}*("AND 1=1","OR")* changes the first keyword "*AND*" of the test case "*AND 1=1*" to "*OR*". M_T or M_F in CCO can change the logic condition to a tautology or a contradiction. SCO can change the syntax structure of the original SQL query, in which M_{AK} or M_{RK} represents the operation of adding or removing a sub-query of specific *Keyword*, respectively. M_{AK}*("AND 1=1", "AND", T)* means adding a sub-query with keyword "*AND*" and the logic is "*True*", and producing the mutation test case "*AND 1=1 AND 1=2*". Operator M_{CMT} will add one comment symbol "--" at the end of a test case, such as "*AND 1=1-- *". MMO includes two mathematical mutation operators, namely M_{RD} and M_C. The operator M_{RD} conducts an operation of randomization to number or character values in a test case, such as "*OR 867=867*". Operator M_C can modify the number values to an equivalent calculation formula, such as "*AND (2-1) = (0+1)*". The mutation function *f* in *Definition 1* is actually the combination of mutation operators, such as function "*f:* $M_{CMT}(M_F(M_{KW}$*("AND 1=1", 1,"OR")))*" is the combination of mutation operators M_{CMT}, M_F *and* M_{KW}, which provides "*OR 1=2-- *". With function *f*, we can get new test cases.

TABLE II. THE PROPOSED LOGIC OPERATORS

Operator	Logic Operators Description
U	The query logic corresponds to Full Set, such as "1=1"
$Ø$	The query logic corresponds to Empty Set, such as "1=2"
S_L	Left query logic divided by the injection point, a subset of U
S_R	Right query logic divided by the injection point, a subset of U
$\cap$	Intersection Operator, keywords: AND, HAVING, LIKE, …
$\cup$	Union Operator, keywords: OR, UNION, …
Cmt	Remove the Suffix-Logic after the injection point
$\triangledown$	The position of injection point

2) *Logic Operators*

To deduce the *Inferential Testing Context* (ITC) and get the *Inferential Metamorphic Relation* (IMR), we introduce a series of *Logic Operators* (LO) presented in Table II. With them, we can infer IMR or ITC with logic calculations. Operator U represents the query logic that can have a full set of data from a table in the background database, such as tautology "*1=1*". $Ø$ is corresponding to the query logic of getting an empty data set from a table, such as contradiction "*1=2*". S_L and S_R respectively represents the left part and the right part of the original SQL query that divided by the injection point, which are subsets of a full set U, and not empty, as showed in Fig. 2. $\cap$ represents intersection logic operator, which conducts the intersection operation in a logic calculation. For instance, the logic formula of test case "*AND 1=1*" is "$\cap U$". Operator $\cup$ conducts union operation, which is related to the keywords "*OR*", "*UNION*", etc.. Such as, keyword "*OR*" conducts a union calculation in an injection clause "*OR 1=2*", which logic formula is "$\cup Ø$". Operator *Cmt* is a comment operator, which is corresponding to a comment symbol that can remove the logic of the right part after the injection point in the original SQL query, such as comment symbol "--". Operator $\triangledown$ shows the position of the injection point in a logical calculation formula, which will be replaced by an injection logic formula.

C. *IMR Generation and ITC Deduction*

The generation of *Inferential Metamorphic Relation* (IMR) for SQLIV penetration test (especially blind SQLIV) is based on the testing context assumption and deduction, called *Inferential Texting Context* (ITC) Inferring. And the deduced IMR can be utilized to conduct IMT and deduce more detailed information of ITC.

Fig. 2 displays the logical structure of a typical SQLIV test case. A common SQLIV payload (test case) is usually consisting of four parts: *Prefix, Keyword, Condition* and *Suffix*. Where, *Prefix* and *Suffix* are used for syntax fixing, *Keyword* is the injection keyword to conduct additional intersection or union operation on the original SQL query, *Condition* is normally logical condition of tautology or contradiction formula. At the injection point (located in the *Where* clause in Fig. 2), a SQL injection payload divides the original SQL query into two parts. We define them as *Left Condition* (LC) and *Right Condition* (RC). Usually, because of the effect of *Prefix* and *Suffix*, the influence of a common SQLIV test case can be limited to the condition expression in the targeted sub clause, such as *Where* clause here. And the targeted sub clause is also separated into *Left Condition* (LC) and *Right Condition* (RC) by the injection test case.

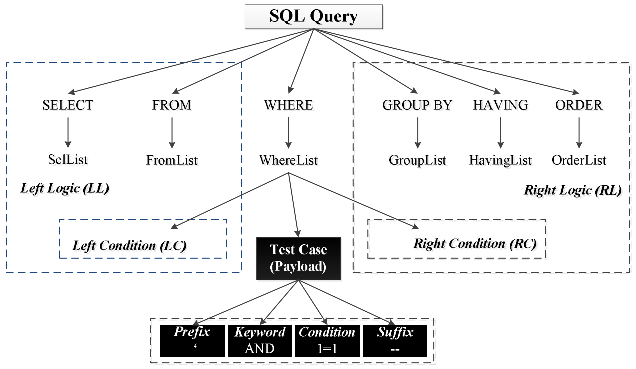

Figure 2. Injection Logic Structure of Background SQL Query

Essentially, SQL injection is to change the original query logic with the payload's logic, and conducts the altered query logic calculation. The original SQL query logics can be described by *Inferential Testing Context* (ITC) "$LC \Diamond RC$ ($LC \in \{U, Ø, S_L\}$, $RC \in \{U, Ø, S_R\}$, $\Diamond \in \{\cap, \cup\}$)", where LC and RC are the *Left* and the *Right* part of a query logic *Condition*, which is divided by a SQL injection payload (test case). *ITC* is actually the formula abstraction form of background test context. The SQLIV test cases can also be described by *Injection Payload Expression (IPE)* "$\Diamond IL\#$ ($IL \in \{U, Ø\}$, $\Diamond \in \{\cap, \cup\}$, $\# \in \{Cmt, null\}$)", where IL is the *Injection Logic* of test case, and # is the placeholder of comment character, which is empty if # equals "*null*". If there exists a SQLIV, we can deduce its *Response Logic* from the logic calculation of the *Inferential Test Context* "$LC \Diamond RC$" and the *Injection Payload Expression* "$\Diamond IL\#$".

Table III shows the examples of basic IMR and ITC deduction criteria, in which we take a pair of test cases "*AND 1=1*" and "*AND 1=2*" as *seeds*, and the related ITC is "$LC \triangledown \cap RC$". From the previous test, the conditions we obtained are *C("AND 1=1", "AND 1=2"): {("AND 1=1", True), ("AND 1=2", False)}*. The *Injection Payload Expression* (IPE) of these test cases are "$\cap U$" and "$\cap Ø$". To infer the possible logic of ITC, we make an assumption that all the combinations of LC, $\Diamond$ and RC are possible for the ITC "$LC \Diamond RC$", which is $|LC| \times |\Diamond| \times |RC|$ kinds of testing context scenarios in our approach. When situations that involving both S_L and S_R, we still need to consider four kinds of set relationships between S_L and S_R, including "$S_L \subset S_R$", "$S_L \supset S_R$", "$S_L \cap S_R = Ø$", and "$S_L \cap S_R \neq Ø, S_L \cap S_R \neq S_L, S_L \cap S_R \neq S_R$". And the number of basic IPEs we use in this paper is $|\Diamond| \times |IL| \times |\#|$. With the proposed ITC and IPE, we can obtain the list of *Inferential Metamorphic Relations* (IMR). Table III shows part of basic testing context scenarios we used, which logic expressions are "$U \triangledown \cap U$" and "$S_L \triangledown \cap S_R$ ($S_L \subset S_R$)". The columns of *Test Case* and *Response Logic (RP =P(TC))* are actually the forms of IMR for IMT, such as *P("AND 1=1")=P("AND 1=1 – ")=True.*

TABLE III. EXAMPLE OF INFERENTIAL METAMORPHIC RELATION AND INFERENTIAL TESTING CONTEXT DEDUCTION

Inferential Testing Context	*LC*	*RC*	*LC & RC* Logic Relation	Injection Payload Expression	Test Case (TC)	Logic Before SQL Injection	Logic After SQL Injection	Response Logic (RL)
$LC \triangledown\cap RC$	S_L	S_R	$LC \subset RC$ $(S_L \subset S_R)$	$\cap U$	**AND 1=1**	$S_L \cap S_R = SL$	$S_L \cap U \cap S_R = S_L$	**TRUE**
				$\cap \emptyset$	**AND 1=2**	$S_L \cap S_R = SL$	$S_L \cap \emptyset \cap S_R = \emptyset$	**FALSE**
				$\cap U$ *Cmt*	**AND 1=1 --**	$S_L \cap S_R = SL$	$S_L \cap U \cancel{\cap S_R} = S_L$	**TRUE**
				$\cap \emptyset$ *Cmt*	**AND 1=2 --**	$S_L \cap S_R = SL$	$S_L \cap \emptyset \cancel{\cap S_R} = \emptyset$	**FALSE**
				$\cup U$	**OR 1=1**	$S_L \cap S_R = SL$	$S_L \cup (U \cap S_R) = S_R$	**FALSE**
				$\cup \emptyset$	**OR 1=2**	$S_L \cap S_R = SL$	$S_L \cup (\emptyset \cap S_R) = S_L$	**TRUE**
				$\cup U$ *Cmt*	**OR 1=1 --**	$S_L \cap S_R = SL$	$S_L \cup U \cancel{\cap S_R} = U$	**FALSE**
				$\cup \emptyset$ *Cmt*	**OR 1=2 --**	$S_L \cap S_R = SL$	$S_L \cup \emptyset \cancel{\cap S_R} = S_L$	**TRUE**
	U	U	$LC = RC$	$\cap U$	**AND 1=1**	$U \cap U = U$	$U \cap U \cap U = U$	**TRUE**
				$\cap \emptyset$	**AND 1=2**	$U \cap U = U$	$U \cap \emptyset \cap U = \emptyset$	**FALSE**
				$\cap U$ *Cmt*	**AND 1=1 --**	$U \cap U = U$	$U \cap U \cancel{\cap U} = U$	**TRUE**
				$\cap \emptyset$ *Cmt*	**AND 1=2 --**	$U \cap U = U$	$U \cap \emptyset \cancel{\cap U} = \emptyset$	**FALSE**
				$\cup U$	**OR 1=1**	$U \cap U = U$	$U \cup (U \cap U) = U$	**TRUE**
				$\cup \emptyset$	**OR 1=2**	$U \cap U = U$	$U \cup (\emptyset \cap U) = U$	**TRUE**
				$\cup U$ *Cmt*	**OR 1=1 --**	$U \cap U = U$	$U \cup U \cancel{\cap U} = U$	**TRUE**
				$\cup \emptyset$ *Cmt*	**OR 1=2 --**	$U \cap U = U$	$U \cup \emptyset \cancel{\cap U} = U$	**TRUE**

To generate basic IMRs, we choose four mutation operators M_{KW}, M_T, M_F and M_{CMT}, which has little affection on the grammar structure of the original query. The mutation criteria are based on test case expression IPE: "$\Diamond IL\#$ *(IL* $\in \{U, \emptyset\}, \Diamond \in \{\cap, \cup\}, \# \in \{Cmt, null\})$". The condition logic should mutate between $\cap$ and $\cup$. After a round of IMT iteration, the IMRs that trigger *evidences* will be added to the conditions *C*, and related test cases will be marked as *seeds*. Meanwhile, more information about ITC can be derived from new IMRs. For instance, if *P("OR 1=1")=!P("OR 1=2")=False* is valid, then "$S_L \triangledown\cap S_R$ $(S_L \subset S_R)$" can be inferred as the most possible ITC, because other logical relation scenarios are not consistent with this IMR.

With new ITC and the extended *seeds*, we can generate more proper IMR and conduct systematic tests until the predefined *Testing Threshold Expression* (TTE) is reached.

IV. EMPIRICAL STUDY

A. Experiment Implementation

To demonstrate the effectiveness of our approach, a prototype tool is developed to implement the proposed IMT approach in the environment of "Visual Studio 2010 + .Net 3.5 + C#", which is utilized to test and discover FP produced by other SQLIV penetration test tools. The input of the tool is the basic URL and test information of discovered *alerts* of a SQLIV penetration test tool, and the output is whether these *alerts* are TP or FP.

Three state-of-the-practice benchmarking tools are chosen to assess our approach, including *WVS Acunetix*[1], *IBM Appscan*[2] and *SQLmap*[3], which are three of the most used and famous penetration test tools. To protect their confidentiality and avoid brand comparison, we refer them as *Tool A*, *Tool B* and *Tool C* (without a particular order).

[1] http://www.acunetix.com/vulnerabilityscanner
[2] http://www-03.ibm.com/software/products/en/appscan
[3] http://Sqlmap.org

For target dataset, we choose 3567 URLs, including 1067 URLs obtained from a web application vulnerability evaluation project *Wavsep* [4] *(The Web Application Vulnerability Scanner Evaluation Project)*, which is an open source project of OWASP [1] and contains 133 SQLIVs, and 2500 real Internet URLs with highly dynamic Web pages that may cause FP in high possibility.

B. Preparation

To prepare our experiment, benchmarking *Tools A*, *B* and *C* are used to testing on 3567 Web pages of the target dataset to collect the data of their *alerts*, FP and TP, as showed in Table IV. Here, we manually check to identify FP and TP from the *alerts* reported by the three test tools. In the table, *URL#*, *TP#* and *FP#* represent the number of URLs, TP and FP respectively. *FP_R* is the result of FP Rate, and we have "*FP_R = FP#/(FP#+TP#)*". *FP_F* represents the FP occurring Frequency on each Web page, and we have "*FP_F = FP#/URL#*". *Rqsts* represents the total HTTP requests sent during testing, which can be used to evaluate testing efficiency of assessed approaches. The total numbers of *URL#*, *TP#* and *FP#* are the sum of three tools, and an *alert* includes the information of the corresponding URL, injection point and test cases.

TABLE IV. FALSE POSITIVE DATA OF THREE TOOLS

	URL#	**TP#**	**FP#**	**FP_R**	**FP_F**	**Rqsts**
Tool A	3567	126	16	11.27%	0.45%	1413722
Tool B	3567	113	27	19.29%	0.76%	451772
Tool C	3567	120	19	13.67%	0.53%	573566
Total	10701	359	62	17.27%	0.58%	2439060

C. Experiment and Results

In the experimental phase, we conduct our prototype tool on all *alerts* that three benchmarking tools generate, and we

[4] https://code.google.com/p/wavsep

have “*T.Alerts# = T.TP# + T.FP# = 421*”, in which *T.* represents the “*Total Number*”. Table V displays the detailed FP testing and reducing results, in which *T.TP#* and *T.FP#* means the total number of TP and FP caused by three benchmarking tools. *TPV* represents the TP that verified by our approach, which can evaluate the test coverage. *FPRm* represents the number of FP that discovered and Removed by our approach. *FPRm_R* represents the Rate of identified and Removed FP in all the FP previously reported, and “*FPRm_R =FPRm/T.FP#*”. Parameter *Rqsts* here is the total number of HTTP requests sent by our approach during test, which can assess the efficiency of our approach.

We use three different *Threshold Expressions* (TE) as the judges of FP respectively, which including *Evidence Threshold Expression* (ETE) and *Testing Threshold Expression* (TTE), and the strict degree increase from *TE1* to *TE3* gradually. The TE we use is “$TE\_n: !ETE \wedge TTE$”, in which “$ETE = ((N_A \geq 2n) \wedge (N_T \geq n) \wedge (N_F \geq n))$”, and “$TTE = (N_S \geq 100)$”. Here, N_E, N_T, N_F and N_S represents the number of the discovered *evidences*, “*True*”, “*False*” *evidences* and used test cases respectively. Here, the standard of FP is that, TTE is satisfied and ETE is not satisfied.

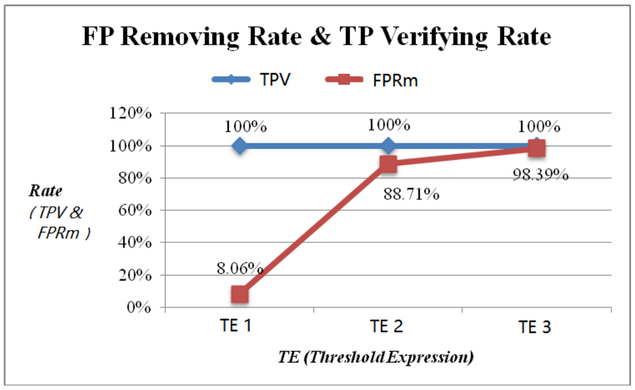

Figure 3. Data of FP Removing Rate & TP Verifying Rate

The results in Fig. 3 shows that the more strict the TE degree applied the more FP discovered. When “*n=3*”, the FP removing rate reaches to 98.39%. And there is no missing TP in our experiment, which means that our approach can maintain good coverage. The HTTP requests sent by our approach are much less than those in an entire testing, which means that our approach will not bring about too much additional resource consumption. From the current experiment, the results we obtained so far are promising.

TABLE V. FP VERIFYING AND REDUCING RESULTS

	T.TP#	T.FP#	TPV	FPRm	Rqsts	FPRm_R
TE1	359	62	359	5	21168	8.06%
TE2	359	62	359	55	21168	88.71%
TE3	359	62	359	61	21168	98.39%

V. CONCLUSIONS

We have presented a mutation-based *Inferential Metamorphic Testing* (IMT) approach to reducing FP in SQLIV penetration tests. First, we defined *Inferential Metamorphic Relation* (IMR) and its mutation operators. Second, we presented logic expressions calculation based on a set of *Logic Operators* (LO). These logic expressions and logic operators can be use to generate IMR and deduce *Inferential Testing Context* (ITC). Finally, we presented an iterative IMT process based on IMR and the inferred ITC heuristically. Our empirical study on three benchmarking tools demonstrates the effectiveness of IMT on reducing FP in SQLIV penetration tests. For future work, we are studying on the application of the proposed approach to other vulnerabilities testing scenarios.

ACKNOWLEDGMENT

This research work was supported by the National Natural Science Foundation of China (Grant No. 61402264 and 61502260) and the Science and Technology Planning Project of Tianjin (Grant No. 17JCZDJC30700).

REFERENCES

[1] OWASP, “Top ten most critical web application security risks”, https://www.owasp.org , reviewed in 2017.

[2] Lawal M. A., Abu B. M. S., Ayanloye O., et al. “Systematic Literature Review on SQL Injection Attack”, International Journal of Soft Computing, 2016, 11(1): 26-35.

[3] Antunes N., Vieira M., “Evaluating and Improving Penetration Testing in Web Services”, 23rd IEEE International Symposium on Software Reliability Engineering (ISSRE 2012), November, 2012: 27-30.

[4] Halfond W. G. J., Viegas J., Orso A., “A Classification of SQL-Injection Attacks and Countermeasures”, IEEE International Symposium on Secure Software Engineering, 2006: 65-81.

[5] Antunes N., Vieira M., “Assessing and Comparing Vulnerability Detection Tools for Web Services: Benchmarking Approach and Examples”, IEEE Trans on Services Computing, 2015, 8(2): 269-283.

[6] Halfond W. G. J., Choudhary S. R., Orso A., “Improving penetration testing through static and dynamic analysis”, Software Testing, Verification and Reliability, 2011, 21(3): 195-214.

[7] Kieyzun A., Guo P. J., Jayaraman K., et al. “Automatic creation of SQL injection and cross-site scripting attacks”, Software Engineering, ICSE 2009, IEEE 31st International Conference, 2009: 199-209.

[8] Shahriar H., Zulkernine M., “MUSIC: Mutation-based SQL injection vulnerability checking”, Quality Software 2008. QSIC'08, The Eighth International Conference on. IEEE, 2008: 77-86.

[9] Appelt D., Nguyen C. D., Briand L. C., et al. “Automated testing for SQL injection vulnerabilities: An input mutation approach”, Proceedings of the 2014 International Symposium on Software Testing and Analysis, ACM, 2014: 259-269.

[10] Shin Y., Williams L., Xie T., “SQLUnitgen: Test Case Generation for SQL Injection Detection”, North Carolina State University, Raleigh Technical Report, NCSU CSC TR 21: 2006.

[11] Ciampa A., Visaggio C. A., Di Penta M., “A heuristic-based approach for detecting SQL-injection vulnerabilities in Web applications”, Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems, ACM, 2010: 43-49.

[12] Zhu H., “Jfuzz: A tool for automated java unit testing based on data mutation and metamorphic testing methods”, Trustworthy Systems and Their Applications (TSA), 2015 Second International Conference on. IEEE, 2015: 8-15.

[13] Chen T. Y., Kuo F. C., Ma W., et al. “Metamorphic Testing for Cybersecurity”, Computer, 2016, 49(6): 48-55.

[14] Huang Y. W., Huang S. K., Lin T. P., et al. “Web Application Security Assessment by Fault Injection and Behavior Monitoring”, Computer Networks, vol 48, Aug, 2005: 739-761.