Search-Based SQL Injection Attacks Testing using Genetic Programming

Benjamin Aziz, Mohamed Bader, and Cerana Hippolyte

School of Computing
University of Portsmouth
Portsmouth PO1 3HE
United Kingdom

Abstract. Software testing is a key phase of many development methodologies as it provides a natural opportunity for integrating security early in the software development lifecycle. However despite the known importance of software testing, this phase is often overlooked as it is quite difficult and labour-intensive to obtain test datasets to effectively test an application. This lack of adequate automatic software testing renders software applications vulnerable to malicious attacks after they are deployed as detected software vulnerabilities start having an impact during the production phase. Among such attacks are SQL injection attacks. Exploitation of SQL injection vulnerabilities by malicious programs could result in severe consequences such as breaches of confidentiality and false authentication. We present in this paper a search-based software testing technique to detect SQL injection vulnerabilities in software applications. This approach uses genetic programming as a means of generating our test datasets, which are then used to test applications for SQL injection-based vulnerabilities.

Keywords: Genetic Programming, Search-Based Testing, SQL Injections

1 Introduction

Over the past few years, many organisations have rapidly adopted Web applications to solve increasingly complex business problems making them widely available on the Internet. However, with this high usability comes the risk of rendering such business applications a natural target to malicious minds. Applications available through Web browsers have become exponentially more vulnerable in recent years [1] due to a wide range of attacks such as buffer overflow attacks, cross-site scripting, SQL Injection attacks and many others.

Among the top ten Web application vulnerabilities published by the Open Web Application Security Project (OWASP) [2] are SQL Injection Attacks (SQLIAs). In 2011, the National Institute of Standards and Technology's National Vulnerability Database [3] reported 289 SQL injection Vulnerabilities (SQLIVs) in websites, including those of IBM, Hewlett-Packard, Cisco, WordPress, and Joomla.

In December 2012, security experts in SANS Institute reported a major SQL injection attack that affected approximately 160,000 websites using Microsoft’s Internet Information Services (MS-IIS), ASP.NET, and SQL Server frameworks.

An SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of the SQL Server for parsing and execution. These types of attacks are particularly harmful as they could give attackers direct access to the database thus enabling them to leak out very sensitive and confidential information with severe consequences for any organisation. More worrying is that SQLIAs have now been documented for well over a decade [4] but still remain an active method of attacks as a result of the lack of effective techniques for detecting and preventing such attacks. Numerous techniques, such as defensive programming and sophisticated input validation, have been proposed and implemented to prevent some types of SQLIAs. However, such techniques have proven incapable of withstanding new forms of SQLIAs as attackers continue to find new exploits that can avoid the checks programmers put in place. Moreover, defensive programming has proven to be very labour-intensive as it requires constant interaction with testers therefore making it an expensive exercise in terms of resources required.

Software testing is a key phase of many development methodologies as it provides a natural opportunity for integrating security early in the software development lifecycle. Inadequate testing of software applications during their development renders these applications vulnerable attacks such as SQLIAs when they are deployed into a live production environment. However despite the known importance of software testing, this stage is often considered lightly as it could be a laborious task. Most software development teams adopt a manual approach in generating test cases thus limiting the number of test cases which could be built and executed within the project’s budget.

Automated software testing techniques, such as Search-Based Software Testing (SBST), are essential for the development of complex systems as they aim to lower the cost of writing tests by enabling users to generate tests automatically. We therefore present a new approach using SBST techniques for the detection of SQLIVs in SQL-based database systems. This new approach will help solve two of the biggest issues with existing SQLIA detection techniques; being fully automated in terms of test case generation, it requires very little interaction with testers, and it provides adequate levels of robustness and flexibility to deal with new emerging patterns of SQLIA attacks.

The rest of the paper is organised as follows. In Section 2, we give an overview of literature covering work related to the problem of test generation for SQLIVs. In Section 3, we introduce the various types of SQLIAs and the technique they each use. In Section 4, we introduce the design of the GP grammar corresponding to SQLIAs. In Section 5, we define our anti-SQLIA system design and discuss its implementation. Finally, in Section 6, we discuss the results obtained from the implementation of the system and in Section 7 we conclude the paper.

2 Related Work

A wide range of approaches have been proposed by researchers to address SQL injection-based threats to Web applications. Appelt et al. [5] proposed a black-box automated testing approach targeting SQLIVs, called U4SQLi. The approach rests on a set of mutation operators that manipulate legitimate inputs to create new test inputs to trigger new SQLIAs. With this approach, it is possible to use a combination of different mutation operators to generate a wide range of attacks and generate inputs that contain new attack patterns, thus increasing the likelihood of detecting vulnerabilities. The approach follows a similar pattern with our proposed approach in that it uses SBST to detect SQLIVs. The difference is with the algorithm used, U4SQLi uses genetic algorithms whereas in our case, we use grammar-based genetic programming.

Su et al. [6] propose a grammar-based approach to detect and stop queries containing SQLIAs by implementing their SQLCheck tool. The proposed approach tracks users' input using a special symbol to mark the beginning and end of each input string. This annotated query is called an *augmented query*. The main idea is to forbid the augmented query from modifying the syntactic structure of the rest of the query. This is achieved by constructing an augmented grammar for this augmented query based on the grammar for a standard SQL statement. At runtime, the augmented query is validated based on this grammar, and is therefore rejected if it does not conform to the grammar.

Shahriar and Zulkernine [7], presented a mutation-based testing approach for SQLIV testing. The authors proposed the use of nine injection operators that inject SQLIVs in application source code. The nine mutation operators were divided into two categories. The first category consists of four operators, which inject faults into WHERE conditions of SQL queries, and the second category consists of five operators, which inject faults into database API method calls. These operators generate mutants, which can be killed with test data containing SQLIAs. This approach is similar to the technique presented by Appelt et al. [5], in that they both use mutants to test the presence of SQLIVs in SQL databases. However, it differs from the approach of [5] in that the mutation is applied to the source code of the application rather than the test input. Hence, it would not adequately solve the problem, which we are trying to address in making testing a less expensive process by automatically generating test cases, as it would require high levels of user involvement to conduct testing using this approach.

Chan et al. [8] presented fault-based testing of SQL database applications. This technique used seven mutation operators to represent faults of entity relationships model of a database driven application. These operators were used to modify the cardinality of queries (e.g., replace "SELECT count (column1)" with "SELECT count (column2)"), replace attributes with similar types (e.g., change one column name with another of a similar data type) and replace participation constraints (e.g., replace EXIST with NOT EXIST) and so on. Like the MUSIC tool proposed by Shahriar and Zulkernine [7], this approach mutates the code and not the test cases, and therefore it would not fulfil our desired goal of having an automated way of generating test data to detect SQLIVs.

Shin et al. [9] proposed an approach for SQL injection vulnerability detection, automated by a prototype tool called SQLUnitGen. The tool combines static analysis, runtime detection and automatic testing to identify input manipulation vulnerabilities. Kosuga et al. [10] presented a technique named Sania, for detecting SQLIVs during the development and debugging phases. Sania investigates HTTP requests and SQL queries to try to discover SQLIAs by constructing parse trees of intended SQL queries written by developers. Terminal leafs of parse trees would represent vulnerable spots, which are filled by possible attack strings. The difference between the initial parse tree and the modified parse tree generated from user-supplied attack strings results in warnings of SQLIAs. Both of [9] and [10] differ to our case as neither injects SQLIAs. There is also a high level of user involvement in both of these approaches and therefore making them inadequate for our objective, which is to automate the generation of test cases.

Ciampa et al. [11] proposed an approach to perform penetration testing of Web applications. This approach differs to ours and many other existing tools in that it does not randomly generate test data but relies on a knowledge base of heuristics to guide the generation of test data. This is achieved by firstly analysing the Web application with the aim of determining its hyperlinks structure and of identifying its input forms. Then it starts seeding a series of standard SQLIAs with the objective of letting the Web application report an error message. Such standard attacks consist of a set of query strings that are not dependent on the Web application. It then matches the output produced by the Web application against an (extensible) library of regular expressions related to error messages that databases can produce. It continues the attack using text mined from the error messages with the objective of identifying likely table of field names, until it is able to retrieve (part of) the database structure. A limitation of this approach is that it depends on known SQL injection patterns to detect errors during testing. This practice have sometimes proven insufficient to test an application as it is incapable of handling unlearned or new attacks. Also there might be a large number of different representations for the same pattern, for example, using different encodings, which may not be captured in the knowledge base.

Tuya et al. [12] proposed a set of mutation operators for SELECT queries and then tested the mutants using a set of queries drawn from the NIST SQL conformance test suite [13]. They then perform further experiments, aimed at reducing the cost of testing using two different approaches: reducing the number of mutants (selective mutation) and reducing the number of test cases (by selecting the order in which mutants are killed). The number of test cases are reduced by ordering mutants from the most difficult to the easiest to be killed.

SQL DOM was proposed by McClure and Kruger [14] as a way of automating the defensive coding testing technique since the manual approach has often proven to be labour-intensive and error-prone. SQL DOM contains a set of classes that enable automated data type validation and escaping. When using this tool, developers provide their own database schema and construct SQL statements us-

ing the SQL DOM API. The tool has proven to be quite useful when developers need to use dynamic queries instead of parameterised queries for greater flexibility. However, this solution could only be used with a new software application under development, as it would require considerable amount of refactoring to get it to work with legacy systems. However, our approach could easily be adapted to any project as it entails black-box testing, which does not require knowledge of the application code.

Thomas et al. [15] proposed an automated vulnerability removal approach, which finds potentially vulnerable (dynamic) SQL statements in programs and replaces them with parameterised SQL statements. Similar to the work of [14], this approach is based on white-box testing requiring some knowledge of the internal structure of the application.

Boyd et al. [16] proposed a tool called SQLrand, which prevents injection attacks that contain keywords. Developers construct queries, which use randomised keywords rather than the normal SQL keywords. This is different from our work in that it only protects against attacks that contain SQL keywords, so although an attacker may not be able to inject code containing keywords without the secret key to randomisation, this approach would not prevent an attacker from injecting other codes, which do not contain SQL keywords.

Finally, Halfond et al. [17] implemented the AMNESIA (Analysis for Monitoring and Neutralising SQL Injection Attack) tool to detect and prevent SQLIAs. The tool first scans the application code to identify hotspot points in the application code that issue SQL queries to the underlying database. For each hotspot, a model is built which represents all the possible SQL queries that may be generated at that hotspot. Calls are then added to the runtime monitor for each hotspot in the application. If a generated query is not consumed by the query model, then it is considered an attack.

3 SQL Injection Attacks (SQLIAs)

SQLIAs occur when an attacker exploits an SQLIV by changing the intended logic underlying an SQL query through inserting new SQL characters or keywords into the query. Here, we describe some common SQLIA examples, although our approach is general enough to cover any other SQLIAs.

3.1 Tautologies

Tautology attacks are attacks where the attacker injects code into conditional statements so that these would always evaluate to a logical True value. The consequences of such attacks generally depend on how the result of a query is used by an application. These types of attacks are generally used to bypass authentication and to return all data in a particular table. For example, an attacker could submit,

```
anything' OR 'x=x
```

instead of a password in an input HTML form. The resulting SQL query becomes:

```
SELECT id FROM users WHERE username='Joe' AND password='anything' OR 'x=x'
```

This transforms the entire WHERE clause into a tautology, which could return every record in the users database.

3.2 Union Query

Union queries provide more flexibility in allowing legitimate queries to retrieve additional information from the database. In such attacks, the attacker injects a statement of the form UNION <injected query> to the original query. For example:

```
' Union Select cardNo FROM creditCards WHERE acctNo = 7909
```

when injected into the username field, modifies the original query to become:

```
SELECT id FROM users WHERE username = '' Union Select cardNo FROM  
creditCards WHERE acctNo = 7909 -- AND password =
```

When this modified SQL query is executed by the application, the original query would return null and the injected query returns the creditCard number for the given account.

3.3 Piggyback Queries

Piggyback queries are similar to Union queries in that they append additional queries to the original query. However unlike Union queries, the intention is not to modify the original query but to just introduce new queries, which piggyback on the original query. For example, in the case of our authentication query, an attacker may submit the following query:

```
SELECT id FROM users WHERE username='Joe' AND password='';  
drop table users;
```

which as expected, may result in the users table being dropped.

3.4 Malformed Queries

Malformed query attacks are used by an attacker to gather information about the database. These attack types are often deployed in conjunction with other SQLIAs such as Union and Piggyback queries, which require the attacker to have some *a priori* knowledge of the database's schema. Malformed queries take advantage of error messages returned by an SQL server, for example the attacker could inject,

```
convert(int,(SELECT top 1 name FROM sysobjects WHERE xtype='u'))
```

into a pin input field for an authentication query. This would result in a new *malformed* SQL query:

```
SELECT id FROM users WHERE username = 'Joe' AND password = 'xxx' AND  
pin = convert(int, (SELECT top 1 name FROM sysobjects WHERE xtype='u'))
```

If we assume that the database is indeed an SQL Server, the error message returned may be of the sort “Microsoft OLE DB Provider for SQL Server (0x80040E07) Conversion failed when converting the nvarchar value 'CreditCards' to data type int”. This error message provides useful information to aid other attacks, since it informs the attacker that in fact an SQL server is running at the backend. Additionally, the second part of the error message reveals that the first user-defined table in the database is actually 'CreditCards'. The attacker could also use a similar approach to find all the columns in the table.

3.5 Inference Queries

The final type of SQLIAs we discuss here is called *Inference queries*, and these are similar to malformed queries in that they allow the attacker to discover information about the SQL database. With this type of attacks, code is injected to allow the application to behave differently based on the results of a particular query. For example, if the attacker inputs the following two queries at two different times,

```
'legalUser' AND 1=0 -- '  
'legalUser' AND 1=1 -- '
```

then an original SQL query would be modified into two different versions,

```
SELECT id FROM users WHERE username= 'legalUser' AND  
1=0 -- ' AND password=" AND pin=0  
SELECT id FROM users WHERE username= 'legalUser' AND  
1=1 -- ' AND password=" AND pin=0
```

Let us suppose the application is an insecure application. When the first query is run, and since $1 = 0$ is always False, the application will return a login error. However, at this stage, it is impossible for the attacker to determine whether the error message was actually the result of the application validating the input correctly and blocking the attack attempt. If the attacker submits the second query, which always evaluates to True, and there is no login error, the attacker can confirm that the username parameter is indeed vulnerable.

4 Design of the GP Grammar

In this section, we propose our automated technique for detecting SQLIVs. The technique applies the concept of Genetic Programming (GP) to evolve legitimate inputs to create new inputs, which could trigger SQLIAs in an application

under test. The proposed technique is driven by an evolutionary computation system called ECJ [18], a framework that supports a variety of evolutionary computation techniques such as genetic programming and genetic algorithms. The ECJ framework was chosen as it is based on a well-engineered structure, which makes heavy use of Java inheritance, abstraction and pattern-oriented design, has great flexibility, with nearly all classes (and all of their settings) dynamically determined at runtime by a user-provided parameter file. This means it is possible to support common functions of GP such as population initialisation, fitness, selection and variation operators without requiring additional user-written code.

We now discuss the preparatory steps for defining our ECJ-based GP.

4.1 Terminal Sets

Terminal (and function) sets, specify the language used to evolve programs used in GP. A terminal set represents the leaves of the parse tree, which corresponds to the program inputs and it typically consists of variables, constants and functions with no arguments. Our implementation included only one terminal node called *X*, which holds an alphanumeric string. *X* is the program external input, a string which will be evolved to generate SQL injection test cases. This node could be used as a child for each of the seven internal nodes defined in the function set.

4.2 Functions Sets

Function sets are the interior nodes of the parse tree of the GP. They are usually all the functions allowed in the program and are driven based on the nature of the problem domain. Our function set contains a total of seven functions. These functions are classified into two broad categories based on the purpose of the functions; i.e. whether they are *behaviour-changing* or *syntax-repairing*. The definition of these functions was inspired by previous mutation testing frameworks by Appelt et al. [5], matching six of the mutation operators defined in [5].

Behaviour-Changing Functions An SQLIA occurs when an attacker changes the intended effect of an SQL query by inserting new SQL keywords or operators into the query. This class of functions from our function set are intended to evolve our legitimate inputs with SQLIAs to take advantage of vulnerabilities in an application. Although there are a wide variety of behaviour-changing attacks, we have restricted this function set to three examples, which represent the most forms of simple SQLIAs. These examples include the AND, OR and SEMI functions. The OR function accepts one input and appends “OR x=x” to the end of the input where x could be a random number or any string enclosed in quotes. For the AND function, it accepts one input and appends “AND x=y” to the end of the input, where x and y could be different random numbers or strings enclosed in quotes. When this resulting output is added to the WHERE clause of an SQL query, it changes the behaviour of the query so that it always evaluate to False therefore no records are returned. Finally, SEMI accepts one

input and appends a semicolon “;” followed by an additional SQL statement. When this resulting string is added to the WHERE clause of the SQL query, it changes the behaviour of the query by including a new and distinct query that *piggybacks* itself to the original one. We show the relationship between SQLIAs and these functions in Table 1.

Table 1. SQL Injection Attacks and Proposed Functions

Attack	Function
Tautologies	OR
Union Queries	SEMI
Piggybacked Queries	SEMI
Inference Attacks	AND
Malformed Queries	OR, SEMI, AND

Syntax-Repairing Functions An SQLIA attempt will only be successful if the resulting query is syntactically correct. Often, malicious inputs may cause SQL syntax errors to appear when these are combined with the original SQL statements, which hinder attempts to undermine the system. Therefore, this next class of functions, PARA, CMT, QUOTE and DoubleQUOTE, evolve inputs with the goal of *repairing* SQL syntax errors when these are encountered. Such functions are not used on their own, but in combination with the behaviour-changing functions. First, we define PARA as a function that accepts a valid input and appends a closing parenthesis to the end of the input. This is often needed as sometimes the input provided is inserted within parenthesis used as a parameter in an SQL function call or within a nested SELECT statement. In such cases, a vulnerability can only be exploited if the opening parenthesis is matched with a closing one. Next, the CMT function adds an SQL comment (i.e. double dashes `--` or a hash character `#`), ensuring that anything which follows the comment is not executed. The QUOTE function accepts one input and adds a single quote (`'`) to the end of the input. This is usually necessary for string inputs, which are enclosed in quotes in the predefined SQL statement. Finally, DoubleQUOTE accepts a single input and adds a double quote (`"`) to the end of the input. This is usually necessary for string inputs, which are usually enclosed in quotes in the SQL statement.

4.3 Fitness Function

Whilst the first two preparatory steps define the primitive set for the GP, and therefore indirectly define the search space that the GP will explore, these two steps are unable to instruct the GP system about which elements or regions of the search space are good. This is the task of the *fitness measure*, the most difficult and most important concept of genetic programming. The fitness function

determines how well a program is able to solve the problem. It varies greatly from one type of program to the next. In our GP implementation, the fitness of an individual is assessed by determining the number of possible injection types that this individual could generate when inserted into the WHERE clause of an SQL query and calculating the sum of all those injection types.

$$Fitness = \sum (1/(1 + S)) \quad (1)$$

Where S is the number of different injection types found in one individual. The more injections generated the smaller the fitness value and the fitter the individual is. If an individual reaches a fitness of 1, it means the generated string is likely to possess a syntactical error and would therefore not result in an SQLIA, whereas a fitness of 0 represents a definite SQLIA.

4.4 Parameters

The fourth preparatory step specifies the control parameters for the run. The most important control parameter is the *population size*. Most of the parameters defined in the *koza.params* seem adequate for our implementation. The important parameters inherited by *koza.params* are as follows.

Initialisation. Our implementation used Koza’s halfBuilder technique to create the initial population. HalfBuilder is essentially a mixture of grow and full technique, with a ramp from 2 to 6 inclusive. The ramp is essentially an initial random number between 2 and 6 inclusive, which is the *maximum* tree size.

Selection. In our implementation, *tournament selection* was used for selecting individuals with a tournament size of 7. Tournament selection is a technique whereby a number of individuals are chosen at random from the population, then they are compared with one other to determine which will be the parent.

Operators. Our implementation used a mixture of mutation, reproduction and crossover to evolve our inputs.

4.5 Termination and Solution Designation

The fifth preparatory step consists of specifying the termination criterion and the method of designating the result of the run. In our implementation, the criterion was the *maximum number of generations* to be run. The single individual was harvested and designated as the result of the run, which was returned as the solution to our problem domain.

Fig. 1. Components of our System showing Data Flows

5 System Design and Implementation

Our testing system consists of three main components: An AntiSQLInjection tool, an ECJ-based test generator and a vulnerable Web service. Figure 1 shows the key components of our testing system. Our AntiSQLInjection tool uses heavily an open source tool called General SQL Parser [19] that functions as a database proxy, which serves to intercept communications between the target system and its database in order to identify if an input is potentially harmful or not. On the other hand, the Web service in this case was the subject application, i.e. SuiteCRM [20]. We next outline how various components were implemented to map our problem to the ECJ framework.

5.1 Representation of Individuals

When setting-up a GP problem in ECJ, we begin by defining the individuals and the species that they belong to. Species available in ECJ vary from integer vectors, to object lists and trees. An evolutionary run in ECJ would allow one population an unlimited number of sub-populations containing an array of the defined individuals, the species and fitness functions. Here, individuals are defined as SQL injected input strings. The implementation used Abstract Syntax Trees (ASTs) to represent our individuals, as this is able to hold an arbitrary number of inputs. Our AST was defined using the built-in ECJ facility for defining and generating ASTs.

The tree structure in ECJ is typically defined by its types, tree constraints, node constraints and a function set, which were all specified in the parameters file. The tree constraints contain data elements shared by the tree. This includes the definition of the tree, its return types and function set. The type essentially defines all the possible node types, which could be found in a tree. ECJ's type objects are of two kinds: atomic types and set types. An atomic type is just a

single object (in fact, it is theoretically just a symbol) [18]. A set type is a set of atomic types. Node constraints define what nodes can be children of other nodes and how many children each node could have. The function set is used to link the node constraints and their implementation in the Java class files.

We identified four atomic types and one set type:

- *strValue*: this type holds a string value, used as a terminal node
- *intValue*: this type holds an integer value, used as a terminal node
- *syntax*: this type is the output of a string where a syntax-repairing function is applied, hence, it is the return value of a syntax-repairing function
- *behaviour*: this type is a string where a behaviour-changing function is applied, hence, it is the return value of a behaviour-changing function
- *before-or-after*: this is a set type with two members, *strValue* and *syntax*.

This would serve as a child to any behaviour changing function

The node constraints for our tree structure when applied to a GPNode would describe three properties of the node: Number of children, the type of children and the type of the parent of the children. Our tree consists of tree nodes representing the function sets (behaviour-changing and syntax-repairing), which were implemented as java class files in the ec.app.sqli package. Their entries in our parameter files are as follows, along with their respective constraints:

```
# We have one function set, of class GPFunctionSet
    gp.fs.size = 1
    gp.fs.0 = ec.gp.GPFunctionSet
# We call the function set "f0"
    gp.fs.0.name = f0
# We have seven functions in the function set. They are:
    gp.fs.0.size = 7
    gp.fs.0.func.0 = ec.app.sqli.X
    gp.fs.0.func.0.nc = nc0
    gp.fs.0.func.1 = ec.app.sqli.Or
    gp.fs.0.func.1.nc = nc1
    gp.fs.0.func.2 = ec.app.sqli.And
    gp.fs.0.func.2.nc = nc1
    gp.fs.0.func.3 = ec.app.sqli.Cmt
    gp.fs.0.func.3.nc = nc3
    gp.fs.0.func.4 = ec.app.sqli.Quote
    gp.fs.0.func.4.nc = nc2
    gp.fs.0.func.5 = ec.app.sqli.Para
    gp.fs.0.func.5.nc = nc2
    gp.fs.0.func.6 = ec.app.sqli.Semi
    gp.fs.0.func.6.nc = nc1
```

5.2 Evaluation and Fitness

After the representation of an individual is defined, the ECJ framework requires the user to define a problem, which evaluates individuals and assigns fitness values to them. As was mentioned earlier, the behaviour of each node type is defined in a separate Java class with a single crucial method overridden:

```

public void eval(final EvolutionState state,
    final int thread, final GPData input, final ADFStack stack,
    final GPIndividual individual, final Problem problem)
    {DoubleData rd = ((DoubleData)(input));
    children[0].eval(state,thread,input,stack,individual,problem);
    rd.x = rd.x + " or 1=1";}

```

The method is called when the GPNode is being executed in the course of executing the tree. The execution proceeds depth-first like the evaluation of a standard parse tree. The *eval()* method has several arguments fairly straightforward to understand. The GPData argument is a simple data object passed around amongst the GPNodes when they execute one another. This is how data is passed from one node to another. The GPData object is defined in the parameter file as *eval.problem.data = ec.app.sqli.StringData*. During evaluation, our GPData object passes through the tree. The object contains a simple string “*x*”. Therefore, when this object arrives at the “OR” node, it becomes the case of simply retrieving *x* and concatenating it with “OR 1=1”, which is then returned, so that the value could then be used by some other syntax-repairing function. The result of the syntax-repairing function is the return value of the tree, which is stored in a text file and serves as the generated test case for our GP run. This process will be done based on the maximum number of runs we have specified in our parameters file.

Like most other grammar-based evolutionary GP systems implemented using the ECJ framework, our implementation uses KozaFitness. This is a fitness function that stores an individual’s fitness. In KozaFitness, standardised fitness and raw fitness are considered the same (there are different methods for these, but they return the same thing). Standardised fitness *f* ranges from 0.0 inclusive (the best) to infinity exclusive (the worst). Adjusted fitness converts standardised fitness, using the formula $adj_f = 1/(1 + f)$, into a scale from 0.0 exclusive (worst) to 1.0 inclusive (best). Our standardised fitness, which is passed to the ECJ system was derived from the definition of 1 earlier in Section 4.3.

6 Results and Analysis

There is no standard benchmark application for testing the existence of SQLIVs. However we evaluated the effectiveness of our approach on one open source system called SuiteCRM [20]. SuiteCRM provides a software suite for the management of popular customer relationships. The application was implemented using PHP, with a MySQL backend database, and provides a SOAP-based Web Service API. The use of this application was motivated by related work [5], in which a version of SugarCRM [21] was used to evaluate the mutation testing approach proposed in [5]. Our evaluation approach is based on the following:

1. First, we identify the distinct list of injectable parameters for the collection of Web services from our subject application. We define an injectable parameter as an input parameter to a Web application whose value is used to build part

- of a query that is sent to the database. We identified a total of 4 injectable parameters that are being used across the 26 Web services.
2. We identify legitimate inputs for each of the injectable parameters so that these are passed to our GP tool to generate SQLIAs, which will be used to test for SQLIVs.
 3. We create a repository of SQL statements for each injectable parameter to mimic a similar structure to what exists in the subject application.
 4. We generate test cases from our GP tool from the legitimate inputs identified for each parameter in 2.
 5. And finally, we use test cases obtained from our GP tool and run these through our AntiSQLInjection tool for each injectable parameter to conduct our tests.

Though most of the run parameters were already predefined by inheriting the Koza parameter file, it is worth mentioning here the values used for each parameter during a GP run, as shown in Table 2.

Table 2. GP Experiment Parameters

Parameter	Value
Population size	1024
Generation size	50
Crossover rate	90%
Mutation rate	10%
Selection method	Tournament selection
Tournament size	7
Number of runs	20

During the experiment evaluation, we investigated the first research question, the effectiveness of our technique in detecting and preventing SQLIAs. The evaluation did not include all parameters of the subject application as it would have taken a substantial amount of time to execute every parameter. The focus therefore was only on the vulnerable parameters which were already identified in a related work [5]. Starting with two legitimate inputs for each parameter, our GP system generated 2048 malicious inputs for each such parameter. These malicious inputs were then applied using the AntiSQLInjection tool. For each parameter, we tracked the number of inputs, which were able to exploit the vulnerability in our subject application. The test results are shown in Table 3.

The table shows for each parameter of our chosen subject the number of unsuccessful attacks (Unsuccessful), the number of successful attacks (Successful), and the number of our generated inputs, which resulted in a syntax error and would therefore not be successful in launching an attack. As the table shows, our tool was able, for all vulnerable parameters of our subject application, to successfully create test cases, which would enable us to exploit vulnerabilities of the application.

Table 3. Results of our SQLIA Tests

Operation	Parameter	Successful (Flagged)	Unsuccessful (Unflagged)	Syntax Error
get_entry_list	query	136	261	1651
get_entry_list	order_by	202	690	1257
search_by_module	assigned_user_id	56	1233	778
get_relationships	related_module_query	478	115	1455
get_entries_count	query	145	343	1560
set_relationships	value	1203	0	845

Whilst our tool was able to launch successful attacks for all parameters, we did note an exceptionally high number of inputs resulting in syntax errors. This could perhaps be corrected in the future by having stricter type constraints on our GP implementation. We also noted a significant number of syntactically correct, yet un-flagged inputs. With further investigation we discovered that this was an issue related to the AntiSQLInjection tool and not our GP system as when these same inputs were applied directly to our subject application, they were able to cause successful attacks. This implies a weakness in the implementation of the AntiSQLInjection tool, where it is sometimes unable to identify attack patterns.

7 Conclusion and Future Work

We presented a GP-based approach for automatically generating test cases for the detection of SQLIAs. Our approach is general in the sense that it is capable of generating any malicious SQL query possible to construct based on the syntax of SQL commands. We demonstrated how our solution can be implemented using open source toolkits and we discussed the results against a subject application, namely SuiteCRM [20]. For future work, we would like fine-tune our implementation to obtain better rates of syntactic errors in the generated test cases, which would improve the performance of our tool. We would also like to widen the set of functions used in generating SQLIAs, including functions related to other forms of SQL-based attacks such as cross-site scripting.

References

1. Fossi, M., Turner, D., Mack, E.J.T., Adams, T., Blackbird, J., Entwisle, S., Graveland, B., McKinney, D., Mulcahy, J., Wueest, C.: Symantec Global Internet Security Threat Report: Trends for 2009. Technical Report XV, Symantec (2010)
2. OWASP: OWASP Top 10-2010-The Ten Most Critical Web Application Security Risks. The Open Web Application Security Project (2010)
3. NIST: National Vulnerability Database: Automating Vulnerability Management, Security Measurement and Compliance Checking. <https://nvd.nist.gov> (Accessed: 2015-10-14)

4. Jeff Forristal: NT Web Technology Vulnerabilities. *Phrack Magazine* **8**(54) (1998)
5. Appelt, D., Alshahwan, N., Nguyen, C.D., Briand, L.C.: Black-box SQL Injection Testing. Technical report, University of Luxembourg and University College London (2014)
6. Su, Z., Wassermann, G.: The Essence of Command Injection Attacks in Web Applications. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '06, New York, NY, USA, ACM (2006) 372–382
7. Shahriar, H., Zulkernine, M.: MUSIC: Mutation-based SQL Injection Vulnerability Checking. In: Quality Software, 2008. QSIC '08. The Eighth International Conference on. (2008) 77–86
8. Chan, W., Cheung, S., Tse, T.: Fault-based testing of database application programs with conceptual data model. In: Quality Software, 2005. (QSIC 2005). Fifth International Conference on. (2005) 187–196
9. Shin, Y., Williams, L., Xie, T.: Sqlunitgen: Test Case Generation for SQL Injection Detection. Technical Report NCSU CSC TR, 21:2006, North Carolina State University (2016)
10. Kosuga, Y., Kernel, K., Hanaoka, M., Hishiyama, M., Takahama, Y.: Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection. In: Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual. (2007) 107–117
11. Ciampa, A., Visaggio, C.A., Di Penta, M.: A Heuristic-based Approach for Detecting SQL-injection Vulnerabilities in Web Applications. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems. SESS '10, New York, NY, USA, ACM (2010) 43–49
12. Tuya, J., Suárez-Cabal, M.J., la Riva, C.d.: Mutating Database Queries. *Information and Software Technology* **49**(4) (2007) 398–417
13. NIST: Structured Query Language (SQL) Test Suite. http://www.itl.nist.gov/div897/ctg/sql/_form.htm (Accessed: 2015-10-14)
14. McClure, R., Kruger, I.: SQL DOM: compile time checking of dynamic SQL statements. In: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on. (2005) 88–96
15. Thomas, S., Williams, L., Xie, T.: On Automated Prepared Statement Generation to Remove SQL Injection Vulnerabilities. *Information and Software Technology* **51**(3) (2009) 589–598
16. Boyd, S.W., Keromytis, A.D.: SQLrand: Preventing SQL Injection Attacks. In: Jakobsson, M., Yung, M., Zhou, J., eds.: ACNS. Volume 3089 of Lecture Notes in Computer Science., Springer (2004) 292–302
17. Halfond, W.G.J., Orso, A.: AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. ASE '05, New York, NY, USA, ACM (2005) 174–183
18. Luke, S., Panait, L., Balan, G., Paus, S., Skolicki, Z., Bassett, J., Hubley, R., Chircop, A.: ECJ: A Java-based Evolutionary Computation Research System. <https://cs.gmu.edu/~eclab/projects/ecj/> (Accessed: 2015-10-14)
19. Gudu Software: GSP: General SQL Parser. <http://www.sqlparser.com> (Accessed: 2015-10-14)
20. SuiteCRM: SuiteCRM CRM for the world. <https://suitecrm.com> (Accessed: 2015-10-14)
21. SugarCRM: SugarCRM. <https://www.sugarcrm.com> (Accessed: 2015-10-14)