SAFELI – SQL Injection Scanner Using Symbolic Execution

Xiang Fu
School of Computing and Mathematics
Georgia Southwestern State University
Americus, GA 31709
xfu@canes.gsw.edu

Kai Qian
School of Computing and Software Engineering
Southern Polytechnic State University
Marietta, GA 30060
kqian@spsu.edu

ABSTRACT

This paper presents the current progress, main algorithm, and the open problems of a tool set called "SAFELI," for detecting SQL Injection vulnerabilities resident in Web applications. SAFELI instruments the bytecode of Java Web applications and utilizes symbolic execution to statically inspect security vulnerabilities. At each location that submits SQL query, an equation is constructed to find out the initial values of Web controls that lead to the breach of database security. The equation is solved by a hybrid string solver where the solution obtained is used to construct test cases. SQL injection attacks are replayed by SAFELI to designers, step by step. We also raise open problems on more powerful string solver techniques that work at the semantics level.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic Execution*; H.2.7 [**Database Management**]: Database Administration— *Security, Integrity, and Protection*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based Services*

General Terms

Security, Verification

Keywords

SQL Injection Attack, Symbolic Execution, Constraint Solver, Automated Testing

1. INTRODUCTION

SQL Injection Attack (SIA) [2, 3] has been one of the major threats to the security of Web applications. SIA results from the fact that many Web applications construct SQL queries on the fly but do not apply a thorough user input validation. Attackers can trick server into executing malicious SQL code which is able to manipulate back-end database for the attackers' interests. Recently, many researchers have proposed solutions to capture and defend SIA, e.g., tainted data tracking [17], intrusion detection based on static analysis [7], black-box testing [12, 8], and SQL randomization [13, 4].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
TAV-WEB – Workshop on Testing, Analysis and Verification of Web Software, July 21, 2008
Copyright 2008 ACM 978-1-60558-053-1/08/07 ...$5.00.

```
String massage(String strInput)
{
  //1. SQL keyword search
  if(strInput.IndexOf("- -")!=-1
        || strInput.IndexOf("OR")!=-1
        || strInput.IndexOf("drop")!=-1)
     throw new Exception(
        "Possible SQL Injection Attack: " + strInput
     );

  //2. massage the data for single quote
  String sOut = strInput.Replace("'","''");
  sOut = sOut.Substring(0,16);
  return sOut;
}
```

Figure 1: Vulnerable User Input Validation

This paper presents a different approach for handling SQL injection attack. We propose the use of symbolic execution to tackle the problem. The bytecode of Java Web applications is first instrumented for symbolic execution engine (SE). Then the program is run by SE. Whenever a hotspot which submits SQL query is encountered, a hybrid string equation is constructed to explore the initial values of Web controls which might be used to apply SQL injection attack. Once the equation is successfully solved, the solution of the equation is used to construct a test case which is replayed by an automated GUI testing tool. The attack scenario can be demoed to designers step by step.

The aforementioned techniques are partly implemented in a tool set called SAFELI. The design blue-print of SAFELI was presented in [10]. This paper concentrates on the presentation of the most recent progress of the tool and the open problems we faced.

The motivation of SAFELI is based on the observation that delicate user input validation bugs can lead to very tricky SQL Injection. In practice, one typical approach against SIA is to filter out special characters such as single quote and "- -". However, it does not always work [2]. To make the story complete, we demonstrate the point using a non-trivial SIA vulnerability that can not be easily detected by black-box testing tools. The example is originally presented in [10]. It is motivated by the vulnerability example of string size restriction in [2].

Fig. 1 presents a `massage()` function that does the safety screening of user input strings. The designer decides that every user input string (e.g., in textbox controls) should go through the `massage()` function before being embedded in SQL query.

The `massage()` function has two parts. The first part does the check on suspicious SQL keywords. It searches for the suspicious

SQL keywords such as "- -", "OR", and "drop". If any of them is found, the function throws exception. The second part deals with the notorious single quote characters. Notice that in SQL language, a single quote character can be treated as a data character by escaping. The escaping form of single quote is "″". massage() substitutes each single quote character with "″", i.e., the escaping character. Then the single quote will not be treated as control characters any more. Finally, massage() tries to provide further protection by restricting the output within the length of sixteen characters. Notice that here "sixteen" is simply a constant magic number, its motivation is to limit the possibility that users can play tricks. A sample use of the massage() function is displayed as below:

```
"SELECT uname, pass FROM users WHERE \n uname='"
+massage(sUname)+ "' AND pass='" +massage(sPwd)+ "'"
```

At the first sight, the above code does a good job at defending SIA. Single quote characters will be replaced by "″" and hence causing no harm. The massage() function in Fig. 1, however, has a very delicate bug. Consider the following input for user name and password, respectively:

```
123456789012345'
 OR     uname<>'
```

Notice both strings are sixteen characters long. After going through the massage operations, the following SQL statement is generated.

```
SELECT uname,pass FROM users WHERE
 uname='123456789012345'' AND pass=' OR     uname<>''
```

Notice that its WHERE clause is always a tautology. It consists of two conditions: (1) whether uname is equal to constant string "123456789012345″ AND pass= " (note the escaping character "″" inside), or (2) uname is not an empty string. Obviously "uname<>″" always evaluates to true. The trick is that the length of malicious strings are both sixteen. Although the Replace function generates escaping characters "″", half of the "″" is then cut off by the Substring method at the end of massage(). Such delicate bugs cannot be easily discovered by black-box testing tools.

This paper is organized as follows. Section 2 presents the general architecture of SAFELI. Section 3 discusses the application of bytecode instrumentation and symbolic execution. Section 4 introduces the current hybrid string solver technique employed in SAFELI. Section 5 outlines the test case generation and re-player module. Section 6 proposes the open problems we faced when enhancing the SAFELI tool set. Section 7 concludes the paper.

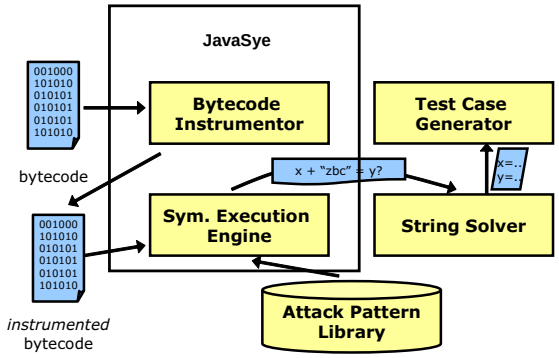

Figure 2: SAFELI Framework

2. SAFELI FRAMEWORK

We now present the architecture of SAFELI in Fig. 2. SAFELI stands for Static Analysis Framework for discovEring sqL Injection vulnerabilities. SAFELI is a loosely coupled tool set which consists of the following components. Many of the components are independent projects.

- *Java Symbolic Execution Engine (JavaSye):* JavaSye consists of two modules: a Java bytecode instrumentor and a symbolic execution engine. The bytecode instrumentor injects additional logic into the target Java bytecode so that it can be executed by the symbolic execution engine. The symbolic execution engine is built upon the Java reflection package. It is able to iteratively examine Java servlets. When hotspots are reached (e.g., the location that submits SQL query or throws exceptions), a library of pre-set attack patterns is consulted, based on which a hybrid string constraint is constructed and sent to constraint solver for generating vulnerability evidence.

- *Library of Attack Patterns:* The module stores a collection of pre-set attack patterns. Each attack pattern consists of two components: the syntax trees of SQL query before and after attack. Patterns can be instantiated/parameterized by real table/column names to form an equation expressed using regular expression.

- *Hybrid String Solver:* Given a constraint, the solver tests its satisfiability and generates valuation of variables that satisfy the constraint. Different than other popular platforms of symbolic execution, the Constraint Solver of SAFELI can solve string constraints.

- *Test Case Replayer (APOGEE):* When initial valuations are generated, they are passed to the Test Case Generator. The module then injects the values into HTML fields and posts the web page back to server. It then uses a heuristic algorithm to analyze the response from server. When vulnerability is verified, a step by step error trace is generated. Currently, the module (APOGEE - Automated Project Grading and Instant Feedback System for Web Engineering Education) is also used for automated grading in teaching.

3. JAVA SYMBOLIC EXECUTION ENGINE

This section presents the JavaSye project, which is a part of the SAFELI framework. JavaSye has two major components: the symbolic execution engine and the bytecode instrumentor.

3.1 Symbolic Execution

The history of symbolic execution can be dated back to 1970's [15]. The idea of symbolic execution is to symbolically interprets and verifies correctness of sequential programs. At the beginning of symbolic execution, initial values of input variables are represented using symbolic literals. The execution is traced by a global variable called "path condition." When an exception is encountered or some system safety property is violated, the path condition is sent to a constraint solver for generating the corresponding initial values of input variables. Symbolic execution has been widely applied. Typical examples include automated test case generation [19], exploration of Operating System vulnerabilities [21], and analyzing heap configurations and data structures [14].

```
1.  public class intro{
2.
3.   public int add(int a, int b) throws Exception{
4.         int c = a + b;
5.         if(c<0){
6.           throw new Exception("The sum is negative!");
7.         }
8.         return c;
9.   }
10. }
```

Figure 3: Java Program Sample

3.2 Instrumentation

The instrumentor module of JavaSye generally follows the idea of the work by S. Khurshid *et al.* [14]. It relies on a Java bytecode engineering library called Javassist [6]. Javassist is able to manipulate Java bytecode and provides basic functions such as replacing types, references, variables, and methods. Based on Javassist, JavaSye is able to inspect the bytecode of Java Servlets, replacing the basic data types such as integer and strings with internal wrapper classes in the symbolic execution framework. Each interesting operation, such as integer addition and string operations are intercepted, and replaced with calls to corresponding wrapper functions in the symbolic execution framework. Similar efforts of symbolic execution include S. Anand's symbolic execution extension [1] using Java Path Finder (JPF) [5]. We decide to directly instrument Java programs at the machine code level due to the greater flexibility provided by the bytecode engineering library.

```
public int add(int, int)

0 iload_1
1 iload_2
2 iadd
3 istore_3
4 iload_3
5 ifge 18
8 new Exception
11 dup
12 ldc "result is negative!"
14 invokespecial Exception.<init>
        (Ljava/lang/String;)V(String):void
17 athrow
18 iload_3
19 ireturn
```

Figure 4: Bytecode Before Instrumentation

We illustrate the idea of JavaSye bytecode instrumentor via an example in Fig. 3. The code snippet computes and returns the sum of two integer variables. If the sum is negative, the function throws an exception. Its bytecode is displayed in Fig. 4. The disassembled bytecode is generated by another bytecode engineering library called BCEL [9]. As shown in Fig. 4, the bytecode of the `add()` function involves several JVM integer instructions. The control flow is straight forward.

Using JavaSye, we can instrument the bytecode directly. The instrumented code for Fig. 4 is presented in Fig. 5. There are several interesting observations. First, all integer variables are converted to instances of `IntExpr`, an internal data type of the symbolic execution engine. For example, pay attention to the first two instructions (bytecode offset 0 and 1) in Fig. 5. They are converted to `aload` from `iload`. The first letter `a` in the instruction opcode indicates that the parameters of the `add` function are treated as object references instead of integer. Second,

```
public int add(int, int)

0 aload_1
1 aload_2
2 invokestatic javasye.constraint.IntExpr.add
    (Ljavasye/constraint/IntExpr;
     Ljavasye/constraint/IntExpr;)
    Ljavasye/constraint/IntExpr;
5 astore_3
6 aload_3
7 new javasye.constraint.IntConst
10 dup
11 iconst_0
12 invokespecial javasye.constraint.IntConst.<init>
    (I)V(int):void
15 sipush 9001
18 invokestatic
   javasye.SERunTime.makeRandomChoiceOnIntComparison
      ...
      (javasye.constraint.IntExpr,
      javasye.constraint.IntExpr, int):boolean
21 ifeq 49
24 new Exception
27 dup
28 ldc "result is negative!"
30 invokespecial Exception.<init> ...
33 aload_0
34 ldc "intro"
36 ldc "add"
38 ldc "(II)I"
40 bipush 33
42 sipush 3002
45 invokestatic
   javasye.SERunTime.SaveCurrentPathCondition_Exception
   (...):Exception
48 athrow
49 aload_0
50 ldc "intro"
52 ldc "add"
54 ldc "(II)I"
56 bipush 50
58 sipush 3001
61 invokestatic javasye.SERunTime.SaveCurrentPathCondition
   (...):void
64 aload_3
65 areturn
```

Figure 5: Bytecode After Instrumentation

the condition in the `if` branch statement is replaced by a random choice function which generates a random boolean value (at offset 18 where `makeRandomChoiceOnIntComparison` is called). At the beginning of each branch, the current path condition is updated correspondingly (see bytecode offset 45 and 61 in Fig. 5, where `SERunTime.SaveCurrentPathCondition` are called). By running the instrumented function twice, the symbolic execution engine will be able to discover the condition that leads to exception.

4. HYBRID STRING SOLVER

This section introduces the hybrid string solver. Once JavaSye finds a hotspot which submits SQL query (by intercepting JDBC calls), JavaSye constructs an equation on the initial values of Web controls (i.e., non-initialized variables in Servlets). Then the equation is passed on to the hybrid string solver. A collection of heuristic algorithms are used in the solver. Detailed description of the heuristic algorithm can be found in [10]. In the following we give one example to illustrate the design idea. The example is taken from [10]. We noticed that D. Shannon has recently developed a similar string analysis technique [18].

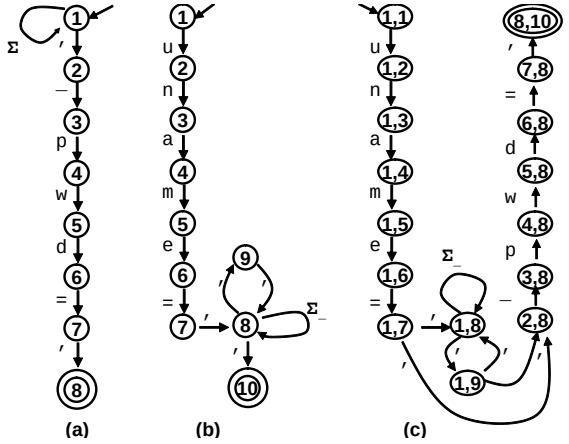

Figure 6: Solving String Equation

EXAMPLE 4.1. Let Σ represent the ASCII alphabet. $\Sigma_- = \Sigma - \{'\}$ and $\Sigma_+ = \Sigma_- \cup \{''\}$. Clearly, Σ_+ represents the alphabet with single quote replaced by its escaping form. Consider the following equation, where $\equiv$ is used to separate the left and right hands of the equation. The "+" represents string concatenation.

$$\texttt{uname='} + s + \texttt{' pwd='} \equiv \texttt{uname='}\Sigma_+{}^*\texttt{'}$$

The above equation is actually an attack pattern. The equation asks: if we concatenate the two constant strings (which are intended to test two data columns `uname` and `pwd`) with s, is it possible to generate one single condition that tests on column `uname` only (and bypasses the password check)? Note that, for the simplicity of the discussion, the attack pattern above is a simplified case of the example given in Section 1. The above equation can be converted to an equivalent set of two equations as below:

$$s_1 + \texttt{' pwd='} \equiv \texttt{uname='}\Sigma_+{}^*\texttt{'} \quad (1)$$

$$\texttt{uname='} + s \equiv s_1 \quad (2)$$

As the solution process is similar for the two equations, we only solve the first one. Equation (1) belongs to the category of concatenation equations. Its left side is a concatenation of a variable and a constant string, and the right side is a regular expression.

To solve concatenation equations, we will first compute an over approximation of the left side. The over approximation is represented by a regular string Σ^*`' pwd='` (let it be s_a). Clearly, the Σ^* represents an over approximation of the variable s_1. We then construct the finite state machine for s_a, as shown in Fig. 6(a). Then we construct the finite automaton for the right side of the equation, as shown in Fig. 6(b). The intersection of both left and right sides is presented in Fig. 6(c).

The next step of the heuristic algorithm is to do a search in the intersection automaton. We search for those states which can reach the final states via the constant string `' pwd='`. Clearly, any path from the initial state to these states represent possible solutions for s_1. We drop the old final states and mark the states found by the search process as final states. Then the resulting automaton represents the solution to Equation 1. The resulting automaton renders the following regular expression as solution.

$$\texttt{uname=} \mid \texttt{uname='(''}\mid\Sigma_-)^*\texttt{'}$$

Similarly we can solve Equation (2), and the regular expression solution of s is displayed below:

$$(\texttt{''}\mid\Sigma_-)^*\texttt{'}$$

Readers can verify that these are indeed attack strings that can bypass password check. ∎

5. TEST GENERATION AND REPLAY

One interesting problem of SAFELI is how to generate equations. We now describe the idea of attack pattern library and test case replayer (which is currently under development).

When symbolic execution reaches each hotspot, attack patterns have to be retrieved from library to construct equations. Note that as JavaSye does not know about the database schema of the Web application, this information has to be extracted on the fly. One simple way is to randomly generate some commonly used strings (not malicious) and instantiates the dynamically constructed SQL statement. Then from the instantiated string, an abstract SQL syntax tree can be constructed. From the syntax tree, all table and column names involved are now known to JavaSye. Then depending on the syntax tree, JavaSye pulls from the attack pattern library a set of applicable attack patterns. Each attack pattern consists of two syntax trees. The first syntax tree represents the "normal" syntax tree of a query before attack, and the second represents the syntax tree for the "malicious" query after SQL injection attack. The table and column names used in the syntax trees can be parameterized (i.e., replaced by the actual table and column names at run time). JavaSye matches the structure of the first syntax tree of the attack pattern against the abstract syntax tree just extracted from the instantiated "normal" SQL query string. If there is a match, an equation is established to relate the two syntax trees in the attack pattern, with the real table and column names embedded into both syntax trees.

Once an equation is solved by the hybrid string solver, the solution of the equation constitutes a test case (attack scenario). The attack scenario can be replayed by SAFELI. The technique is implemented in an independent project called APOGEE (Automated Project Grading and Instant Feedback System) [11]. APOGEE is based on an automated GUI testing library called WatiN [20]. WatiN treats web browser as a DOM object, and can simulate human tester actions on web browser via submitting commands to DOM object. For example, using WatiN, testers can write scripts to click buttons on a Web page, enter text into a textbox, select an item of a drop-down select, etc. Using the technique, given the test case, APOGEE is able to display an attack scenario step by step to designers.

Fig. 7 displays a sample report page of a test case. The system provides two approaches for demonstrating the running result. In the first approach, APOGEE has already automatically recorded the snapshots and generated the text description of the test case. In the second approach, APOGEE is able to generate a Ruby [16] script on the fly. If user download and run the Ruby script, the attack scenario can be replayed step by step to user.

APOGEE is currently used for automated Web application grading in authors' Web engineering classes. We are currently working on the interface of APOGEE to interact with JavaSye for replaying test cases generated by JavaSye.

6. OPEN PROBLEMS

At the present, the weakest part of the SAFELI framework is its attack pattern library. This is the only part that needs the input from human testers and security specialist. In the future we plan

Figure 7: APOGEE Test Case Replay Sample

to remove the attack pattern library from the SAFELI tool set and make SAFELI a completely automated security inspection tool.

To remove the use of attack patterns raises several interesting open problems as below. We assume that G is the context free grammar of SQL and S is a statement that produces SQL query on the fly. S consists of string operations such as concatenation, substring, etc. as well as constant strings and string variables.

- **Problem 1:** Given SQL grammar G and statement S, is it possible to find two different valuations of variables of S such that after parsing using G, two structurally different syntax trees are generated?
- **Problem 2:** Is there a (heuristic) algorithm for generating two sets of variable valuations for Problem 1?
- **Problem 3:** Given G and statement S, is it possible to find valuation of variables in S such that the string produced by S is a tautology (based on G and the associated semantics rules)?

We do not know the solution to Problem 3 so far. For Problems 1 and 2, we plan to solve them as below.

It is not hard to see that most string operations (e.g., concatenation, substitution, substring, etc.) can be represented as finite state machines (FSA) such as Moore machines that produce string outputs. Clearly the set of string outputs can be also recognized by a standard FSA. Hence we have the following proposition:

PROPOSITION 6.1. *If a Java statement S involves only constant strings, string variables, and string operations {*`concatenation`*,* `substring`*,* `substitution`*}, then S can be represented by a Moore machine and all possible outputs of S can be represented by a regular expression.*

According to Proposition 6.1, let $L(S)$ represent the set of strings that can be produced by S, and let $L(G)$ represent the SQL language. It is well known that the intersection of a regular language and a context-free language is context free. Clearly, $L(S) \cap L(G)$ is also context free. This fact leads to one simple heuristic algorithm for solving Problems 1 and 2: (1) derive the context free grammar for $L(S) \cap L(G)$, (2) examine each non-terminal in the interested set[1]. For each non-terminal, there should be only one production rule and the production rule should have no logical OR relation. (3) Problem 2 can be resolved by using different production rules to produce two different syntax trees.

7. CONCLUSION

This paper presents the most recent progress of the SAFELI[10] tool set. Based on the symbolic execution technique, SAFELI can statically inspect the bytecode of a Web application and automatically generate SQL injection attack scenarios. The novelty of the tool lies in its satisfiability decision/approximation procedure for string constraints. We raised open problems in this paper to solve string constraints at a higher semantics level.

8. REFERENCES

[1] Saswat Anand, Corina S. Pasareanu, and Willem Visser. Jpf-se: A symbolic execution extension to java pathfinder. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2007)*, pages 134–138, Braga, Portugal, March 2007.

[1]The interested set includes those non-terminals in syntax tree that need to be compared when doing structure comparison. For example, you might not want to include a non-terminal such as “variable" that is recognized by a right linear production rule.

[2] C. Anley. Advanced SQL Injection In SQL Server Applications. Next Generation Security Software LTD. White Paper, 2002.

[3] C. Anley. More Advanced SQL Injection. Next Generation Security Software LTD. White Paper, 2002.

[4] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, volume 3089 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 2004.

[5] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder - a second generation of a Java model checker. In *Proceedings of the Workshop on Advances in Verification*, 2000.

[6] Shigeru Chiba. Java assist tutorial. `http://www.csg.is.titech.ac.jp/~chiba/javassist/`.

[7] A. Christensen, A. Mller, and M. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the International Static Analysis Symposium (SAS'03)*, 2003.

[8] SPI Dynamics. Webinspect: Security throughout the application lifecycle. SPI Dynamics. Datasheet. `http://www.spidynamics.com/assets/documents/WebInspect_DataSheets.pdf`.

[9] Apache Software Foundation. Bytecode engineering library (bcel). `http://jakarta.apache.org/bcel/manual.html`.

[10] X. Fu, X. Lu, K. Qian, B. Peltsverger, and S. Chen. A Static Analysis Framework for Detecting SQL Injection Vulnerabilities. In *Proceedings of 31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, pages 87 – 96, 2007.

[11] X. Fu, K. Qian, B. Peltsverger, L. Tao, and J. Liu. APOGEE - Automated Project Grading and Instant Feedback System for Web Based Computing . In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2008)*, pages 77 – 81, Portland, OR, March 2008.

[12] Y.W. Huang, S.K. Huang, T.P. Lin, and C.H. Tsai. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 11th International World Wide Web Conference (WWW 2003)*, 2003.

[13] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction set randomization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'03)*, 2003.

[14] S. Khurshid, C. S. Pasăreănu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *LNCS*, 2003.

[15] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[16] Yukihiro Matsumoto. Ruby programming language. `http://www.ruby-lang.org/en/`.

[17] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the 20th IFIP International Information Security Conference*, 2005.

[18] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. (TAICPART-MUTATION 2007)*, pages 13–22, September 2007.

[19] N. Tillmann and W. Schulte. Parameterized unit tests with unit meist. In *Proceedings of the 10th European Software Engineering Conference Joint with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2005.

[20] Jeroen van Menen. Web application testing in .net. `http://watin.sourceforge.net/`.

[21] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P 2006)*, 2006.