CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks

PRITHVI BISHT
University of Illinois, Chicago
P. MADHUSUDAN
University of Illinois, Urbana-Champaign
V.N. VENKATAKRISHNAN
University of Illinois, Chicago

SQL injection attacks are one of the topmost threats for applications written for the Web. These attacks are launched through specially crafted user inputs, on web applications that use low level string operations to construct SQL queries. In this work, we exhibit a novel and powerful scheme for automatically transforming web applications to render them safe against all SQL injection attacks.

A characteristic diagnostic feature of SQL injection attacks is that they change the intended structure of queries issued. Our technique for detecting SQL injection is to dynamically mine the programmer-intended query structure on any input, and detect attacks by comparing it against the structure of the actual query issued. We propose a simple and novel mechanism, called CANDID, for mining programmer intended queries by dynamically evaluating runs over benign candidate inputs. This mechanism is theoretically well founded and is based on inferring intended queries by considering the symbolic query computed on a program run. Our approach has been implemented in a tool called CANDID that retrofits Web applications written in Java to defend them against SQL injection attacks. We have also implemented CANDID by modifying a Java Virtual Machine, which safeguards applications without requiring retrofitting. We report extensive experimental results that show that our approach performs remarkably well in practice.

Categories and Subject Descriptors: K.6 [**Security and Protection**]: Unauthorized access; H.3 [**Online Information Services**]: Web-based services

General Terms: Security, Languages, Experimentation

Additional Key Words and Phrases: SQL injection attacks, retrofitting code, symbolic evaluation, dynamic monitoring.

1. INTRODUCTION

The widespread deployment of firewalls and other perimeter defenses for protecting information in enterprise information systems in the last few years has raised the bar

Author's address: Prithvi Bisht, Department of Computer Science, University of Illinois at Chicago, Chicago, 60607, Email: pbisht@cs.uic.edu.

This research is supported in part by NSF grants (CNS-0716584), (CNS-0551660) and (CCF-0747041).

for remote attacks on networked enterprise applications. However, such protection measures have been penetrated and defeated quite easily with simple script injection attacks, of which the SQL Command Injection Attack (SQLCIA) is a particularly virulent kind. An online application that uses a back end SQL database server, accepts user input, and dynamically forms queries using the input, is an attractive target for an SQLCIA. In such a vulnerable application, an SQLCIA uses malformed user input that alters the SQL query issued in order to gain unauthorized access to the database, and extract or modify sensitive information.

SQL injection attacks are extremely prevalent, and ranked as the second most common form of attack on web applications for 2006 in CVE (Common Vulnerabilities and Exposures) list at `http://cve.mitre.org`. The percentage of these attacks among the overall number of reported attacks rose from 5.5% in 2004 to 14% in 2006. The 2006 SQLCIA on CardSystems Solutions [1] that exposed several hundreds of thousands of credit card numbers is an example of how such attack can victimize an organization and members of the general public. By using Google code search, analysts have found several application programs whose sources exhibit these vulnerabilities.[2] Several reports suggest that a large number of applications on the web are indeed vulnerable to SQL injection attacks [Sutton 2006], that the number of attacks are on the increase, and is on the list of most prevalent forms of attack [2].

Research on SQL injection attacks can be broadly classified into two basic categories: *vulnerability identification* approaches and *attack prevention* approaches. The former category consists of techniques that identify vulnerable locations in a web application that may lead to SQL injection attacks. In order to avoid SQL injection attacks, a programmer often subjects all inputs to input validation and filtering routines that either detect attempts to inject SQL commands or render the input benign [Anley 2002; O. Maor and A. Shulman 2002]. The techniques presented in [Xie and Aiken 2006; Livshits and Lam 2005] represent the prominent static analysis techniques for vulnerability identification, where code is analyzed to ensure that every piece of input is subject to an input validation check before being incorporated into a query (blocks of code that validate input are manually annotated by the user). While these static analysis approaches scale well and detect vulnerabilities, their use in addressing the SQL injection problem is *limited* to merely identifying potential unvalidated inputs. The tools do not provide any way to check the *correctness* of the input validation routines, and programs using incomplete input validation routines may indeed pass these checks and still be vulnerable to injection attacks. (See [Wassermann and Su 2007] for progress on using static analysis to verify the correctness of validation functions.)

A much more satisfactory treatment of the problem is provided by the class of attack prevention techniques that *retrofit* programs to shield them against SQL injection attacks [Halfond et al. 2005; Valeur et al. 2005; Nguyen-Tuong et al. 2005; Pietraszek and Berghe 2006; Xu et al. 2006; Halfond et al. 2006; Buehrer et al. 2005; Su and Wassermann 2006]. These techniques often require little manual annotation, and instead of detecting vulnerabilities in programs, offer preventive

[1] Secureworks Press release, July 2006, http://www.secureworks.com/press/20060718-sql.html
[2] Using Google Code search to find security bugs.
http://www.oreillynet.com/onlamp/blog/2006/10/using_google_code_search_to_fi.html

mechanisms that solve the problem of defending the web application against SQL injection attacks.

Relying on input validation routines as the sole mechanism for SQL injection defense is problematic. Although they can serve as a first level of defense, it is widely agreed [Halfond et al. 2006] that they cannot defend against sophisticated attack techniques (for instance, those that use alternate encodings and database commands to dynamically construct strings) that inject malicious inputs into SQL queries.

A more fundamental technique to solve the problem of preventing SQL injection comes from the commercial database world, in the form of `PREPARE` statements. These statements, originally created for the purpose of making SQL queries more efficient, have an important security benefit. They allow a programmer to declare (and finalize) the structure of every SQL query in the application. Once issued, these statements do not allow malformed inputs to influence the SQL query structure, thereby avoiding SQL injection vulnerabilities altogether. The following statement:

```
SELECT * FROM phonebook WHERE username = ?  AND password = ?
```

is an example of a `PREPARE` statement. The question marks in the statement are used as "place-holders" for user inputs during query parsing, and therefore ensure that these possibly malicious inputs are prevented from influencing the structure of the SQL statement. Thus, `PREPARE` statements allow a programmer to easily isolate and confine the "data" portions of the SQL query from its "code".

Thus, `PREPARE` statements are in fact a *robust* and *effective* mechanism to defend against SQL injection attacks. However, retrofitting an application to make use of `PREPARE` statements requires manual effort in specifying the intended query at every query point, and the effort required is proportional to the complexity of the web application.

The above discussion raises a natural question: Could we *automatically infer* the structure of the programmer-intended queries at every query issue point in the application? A positive answer to this question will address the retrofitting problem, thereby providing a robust defense for SQL injection attacks.

In this paper we offer a technique, *dynamic candidate evaluation*, that automatically (and dynamically) mines programmer-intended query structures at each SQL query location, thus providing a robust solution to the retrofitting problem. Central to our approach are two simple but powerful ideas: (1) the notion that the string operations computed on any particular program path capture the symbolic structure of the corresponding programmer-intended query, and (2) a simple dynamic technique to mine these programmer-intended query structures using candidate evaluations.

Based on these ideas, we implemented a tool called CANDID(CANdidate evaluation for Discovering Intent Dynamically). CANDID[3] retrofits web applications written in Java through a program transformation. CANDID's natural and simple approach turns out to be very powerful for detection of SQL injection attacks. We support this claim by evaluating the efficacy of our approach in preventing a large

[3]Webster: free of malice

set of attacks on several real-world examples.
This paper makes the following contributions:

—The dynamic candidate evaluation approach for mining the structures of programmer intended queries.
—A formal basis for this dynamic approach using the notion of symbolic queries.
—Two mechanisms, one based on automated bytecode transformation and another based on a modified virtual machine, that implement this approach for Java programs to defend against SQL injection attacks.
—A discussion of practical issues and resilience of our approach to various artifacts of Web applications.
—Two distinct implementation approaches that are evaluated based on effectiveness of attack detection and performance overheads.

The problem of SQL injection is one of information flow integrity [Biba 1977; Sabelfeld and Myers 2003]. The semantic notion of data integrity requires that untrusted input sources (i.e., user inputs) must not affect trusted outputs (i.e., structure of SQL queries constructed). Notions of *explicit information flows* that track a relaxed version of the above data integrity problem are suitable for this problem. Such dynamic solutions have been implemented by mechanisms such as tainting [Pietraszek and Berghe 2006; Nguyen-Tuong et al. 2005; Xu et al. 2006] and bracketing [Su and Wassermann 2006]. Our solution is structurally very different from these approaches and we present detailed comparisons in the section on related work.

The paper is organized as follows. In Section 2 we informally present our approach through an example. We present the formal basis for the idea of deriving intended query structures in Section 3. Section 4 presents the program transformation techniques used to compute programmer-intended query structures. Section 5 discusses various ways to implement CANDID and our prototype implementations. Section 6 presents the functional and performance evaluation of our approach through experiments on our prototype implementations. Related work on other approaches is discussed in Section 7. In Section 8, we discuss some limitations of our approach and compare it in detail with dynamic tainting. Section 9 concludes the paper.

2. OVERVIEW OF CANDID

2.1 An example

To illustrate SQL injection, let us consider the web application given in Figure 1. The application is a simple online phone book manager that allows users to view or modify their phone book entries. Phone book entries are private, and are protected by passwords. To view an entry, the user fills in her user name, and chooses the `Display` button. To modify an entry, she chooses the `Modify` button, and enters a different phone number that will be updated in her record. If this entry is left blank, and the modify option is chosen, her entry is deleted. The program that processes the input supplied by the form is also shown in Figure 1. Depending on the display / modify check-box and depending on whether the phone number is supplied or not, the application issues a `SELECT`, `UPDATE`, or `DELETE` query.


```
void process-form(string uname, string pwd, bool modify,
                                                string phonenum) {
 if (modify == false){ /* Path 1. only display */
  query = "SELECT * from phonebook WHERE username = '" +
                uname + "' AND password = '" + pwd + "'";
 }
 else{  /* modify telephone number */
   if (phonenum == "") /* Path 2. delete entry */
     query = "DELETE from phonebook WHERE username='" +
                uname + "' AND password = '" + pwd + "'";
   else               /* Path 3. update entry */
     query = "UPDATE phonebook SET phonenumber =" + phonenum +
       "WHERE username = '"  + uname +
       "' AND password = '" + pwd + "'";
 }
 sql.execute(query);
}
```

Fig. 1. Web Application Example

The inputs from the HTML form are directly supplied to procedure `process-form`, and hence the application is vulnerable to SQL injection attacks. In particular, a malicious user can supply the string "`John' OR 1=1 --`" for the username, and "`not-needed`" for the password as inputs, and check the display option, which will make the program issue the SQL query (along Path 1):

```
SELECT * from phonebook WHERE username='John' OR 1=1 --' AND
                password='not-needed'
```

This contains the tautology 1=1, and given the injected OR operator, the `SELECT` condition always evaluates to true. The sub-string "`--`" gets interpreted as the comment operator in SQL, and hence the portion of the query that checks the password gets commented out. The structure of the original query that contained the "AND" operator is changed to a query that contains (1) an "OR" operator and

(2) uses a tautology. The net result of executing this query is that the malicious user can now view all the phone book entries of all users. Using similar attack queries, the attacker can construct attacks that delete phone number entries or modify existing entries with spurious values. A program vulnerable to an SQL injection attack is often exploitable further, as once an attacker takes control of the database, he can often exploit it (for example using command-shell scripts in stored procedures in the SQL server) to gain additional access.

In order for an attack to be successful, the attacker must provide input that will ultimately affect the construction of a SQL query statement in the program. In the above example, the user name "`John' OR 1=1 --' `" is an attack input, whereas the input name "`John`" is not. An important observation made in prior work [Buehrer et al. 2005; Su and Wassermann 2006] is that a successful attack always changes the structure of the SQL query intended by the programmer of the application. For the example given above, the attack input "`John' OR 1=1 --'`" results in a query structure whose condition consists of an "OR" clause, whereas the corresponding query generated using non-attack input "`John`" has a corresponding "AND" clause. Detecting change in query structure that departs from the one intended by the programmer is therefore a robust and uniform mechanism to detect injection attacks. `PREPARE` statements also capture the programmer's intention at every query-issue location. By enabling the programmer to fix (and finalize) the parse structure of the SQL query, `PREPARE` statements disallow any injection attacks.

From the above discussion, it follows that the main problem at hand is to learn the structure of programmer-intended `PREPARE` query structures for various query issuing locations in the program. If this can be accomplished, then during program execution, the syntactic structures of the programmer-intended query and the actual query can be compared in order to detect attacks.

Several options are available to learn programmer intended queries. One approach is to construct valid query structures from safe test inputs [Valeur et al. 2005]. The problem with a purely testing-based strategy is that it may miss some control paths in the program and may not be exhaustive, leading to rejection of valid inputs when the application is deployed. Another possibility is to use *static analysis* techniques [Halfond et al. 2005] to construct the programmer-intended queries for each program point that issues a query. The effectiveness of static analysis is dependent on the precision of the string analysis routines. As we show in the related work section (Section 7), precise string analysis using static analysis is hard, especially for applications that use complex constructs to manipulate strings or interact with external functions to compute strings that are used in queries.

2.2 Our approach

To deduce (at run-time) the query structure intended by a programmer, our high-level idea is to dynamically construct the structure of the programmer-intended query whenever the execution reaches a program location that issues a SQL-query. Our approach is to compute the intended query by running the application on *candidate* inputs, that are self-evidently non-attacking. For the above example, the candidate input for variable `name` set to "`John`" and the variable `modify` set to `false`, elicit the intended query along the branch that enters the first `if-then` block (`Path 1`) in Fig.1.

In order for a candidate input to be useful, it must satisfy the following two conditions:

(1) *Inputs must be benign.* The candidate input must be evidently non-attacking, as envisioned by the programmer while coding the application. The queries constructed from these inputs, therefore, will not be attack queries.

(2) *Inputs must dictate the same path in a program.* An actual input to the program will dictate a control path to a point where a query is issued. To deduce the programmer-intended query structure for this particular path (i.e., the *control path*), the candidate inputs must also exercise the same control path in the program.

Given such candidate inputs, we can detect attacks by comparing the query structures of the programmer-intended query (computed using the candidate input) and the possible attack query.

The above discussion suggests the need for an *oracle* that, given a control path in a program, returns a set of benign candidate inputs that exercise the same control path. This oracle, if constructed, may actually offer us a clean solution to the problem of deducing the query structure intended by the programmer. Unfortunately, such an oracle is hard to construct, and is, in the general case, impossible (i.e., the problem of finding such candidate inputs is *undecidable*). (See [Emmi et al. 2007] and references therein for prior work on *testing* database applications using incomplete solutions based on constraint-solving and random testing.)

The crux of our approach is to *avoid* the above problem of finding candidate inputs that exercise a control path, and instead derive the intended query structure directly from the same control path. We suggest that we can simply *execute* the statements along the control path on *any* benign candidate input, *ignoring* the conditionals that lie on the path. In the above example, Path 1 is taken for the attack input `John' OR 1=1 --'`. We can execute the statements along that path, in this case the lone `SELECT` statement, using the benign input "`John`" and dynamically discover the programmer intended query structure for the same path.

The idea of executing the statements on a control path, but not the conditionals along it, is, to the best of our knowledge, a new idea. It is in fact a very intuitive and theoretically sound approach, as shown by our formal description in the next section. Intuitively, when a program is run on an actual input, it exercises a control path, and the query constructed on that path can be viewed as a *symbolic* expression parameterized on the input variables. A natural approach to compute the intended query is then to substitute benign candidate inputs in the *symbolic expression for the query*. This substitution is (semantically) precisely the same as evaluating the non-conditional statements on the control path on the candidate input, as shown in the next section.

3. FORMAL ANALYSIS USING SYMBOLIC QUERIES

In this section, we formalize SQL injection attacks and, through a series of gradual refinements and approximations, we derive the detection scheme used by CANDID. In order to simplify and concentrate on the main ideas in this analytic section, we will work with a simple programming language.

P	::=	defn ; stmt; `sql.execute`(s_0)	(program)
defn	::=	`int` n | `str` s | `input int In`	
	::=	| `input str Is` | defn ; defn	(variable declaration)
stmt	::=	stmt ; stmt | $n := ae$ | $s := se$ |	
		| `skip` | `while` (be) {stmt}	(statement)
		| `if` (be) `then` {stmt} `else` {stmt}	
ae	::=	c | n | $f_i(t_1, \ldots, t_k)$	(arithmetic expressions)
se	::=	$cstr$ | s | $g_i(t_1, \ldots, t_k)$	(string expressions)
be	::=	`true` | `false` | $h_i(t_1, \ldots, t_k)$	(boolean expressions)

where $n \in \mathcal{I}$, $s \in \mathcal{S}$, $c \in \mathbb{Z}$ is any integer constant, $cstr$ is any string constant, each t_i is either ae, be or se, depending on the parameters for f_i, g_i, h_i, respectively.

Fig. 2. A simple while language

We first define a simple while-programming language (see Fig. 2) that has only two variable domains: integers and strings. We fix a set of integer variables $\mathcal{I}$ and a set of string variables $\mathcal{S}$, and use n, n_i, etc., to denote integer variables and s, s_j, etc., to denote string variables. A subset of these are declared to be *input* variables using the keyword `input`, and is used to model user-inputs to a web application.

Let us also fix a set of functions f_i each of which takes a tuple of values, each parameter being a string / integer / boolean, and returns an integer. Likewise, let's fix a set of functions g_i (respectively h_i) that take a tuple of string / integer / boolean values and return a string (respectively boolean). A special string s_0 is the query string, and a special command `sql.execute`(s_0) formulates an SQL-query to a database; we will assume that the query occurs exactly once and is the last instruction in the program. The syntax of programs is given in Fig. 2. The semantics is the natural one (and we assume each non-input integer variable is initialized to 0 and each non-input string variable is initialized to the null string ϵ).

Note that the functions f_i, g_i, and h_i are native functions offered by the language, and include arithmetic functions such as addition and multiplication, string functions such as concatenation and sub-string, and string-to-integer functions such as finding the length of a string.

For example, if `concat` is a function that takes two strings and concatenates them, then consider a string expression of the form:

`concat`("`SELECT * FROM employdb WHERE name='`", s)

This example denotes the concatenation of the constant string with the variable string s. For readability, however, we will represent concatenation using '+', for example:

"`SELECT * FROM employdb WHERE name='`" + s

The formal development of our framework will be independent of the functions the language supports. For the technical exposition in this section, we will assume that all functions are *complete* over their respective domains.

Figure 3 illustrates a program that will serve as the running example throughout this section. The program takes an integer n and a string s as input, and depending

```
Program P
input int n; input str s; str s0;
if (n == 0) then                    ;; Path 1
  {s0:="SELECT * FROM employdb WHERE name='"+s+"' ";}
else                                ;; Path 2
  {s0:="SELECT * FROM employdb WHERE name='"+s+"' AND status='cur'";}
sql.execute(s0);
```

Fig. 3. An example program

on whether n (which could be a check-box in a form) is 0 or not, forms a dynamically generated query s_0. Note that the query structures generated in the two branches of the program are different. The input determines the control path taken by the program, which in turn determines the structure of the query generated by the program.

3.1 SQL injection defined

Let us assume a standard syntax for SQL queries, and define two queries q and q' to be equivalent (denoted $q \approx q'$) if the parse tree structures of the two queries are the same. In other words, two queries are equivalent if the parse trees of q and q' are isomorphic.

Let P be a program with inputs $I = \langle In_1, \ldots, In_p, Is_1, \ldots, Is_q \rangle$. An input valuation is a function v that maps each In_j to some integer and maps each Is_j to some string. Let IV denote the set of all input valuations. For any input valuation v, the program P takes a unique control path Run_v (which can be finite, or infinite if P doesn't halt). We will consider only halting runs, and hence Run_v will end with the instruction `sql.execute(`s_0`)`. Note that the path Run_v, and hence the structure of the query s_0, could depend on the input valuation v (e.g., due to conditionals on input variables as in Fig. 3).

Intuitively, the input valuations are partitioned into two parts: the set of *valid* inputs V which are benign, and the complement $\overline{V}$ of *invalid* inputs, which include all SQL injection attacks. A definition of SQL injection essentially defines this partition.

Our definition of SQL injection is based on following two primary principles:

—(**P1**) the structure of the query on any valid valuation v is determined solely by the control path the program takes on input v.

—(**P2**) an input valuation is invalid iff it generates a query structure different from the structure determined by its control path.

The principle (P1) holds for most practical programs that we have come across as well as any natural program we tried to write. An application, such as the one in Figure 3, typically will generate different query structures depending on the input (the input value of the variable n, in this case). However, these query structures are generated differentially using conditional clauses that check certain values in the input (typically check-boxes in Web application input), as in Figure 1. As shown

in our comparison studies in Section 7, (P1) is actually a more general principle than the underlying ideas suggested in earlier works [Halfond et al. 2005; Su and Wassermann 2006].

Let v be an input and $\pi = Run_v$ be the path the program exercises on it. By (P1), we know that there is a unique query structure associated with π. (P2) says that every invalid input disagrees with this associated query structure. (P2) is a well-agreed principle in other works on detecting SQL injection [Buehrer et al. 2005; Su and Wassermann 2006; Hansen and Patterson 2005]. Given the above principles, we now can define SQL injection.

Let us first assume that we have a *valid representation* function $VR : IV \rightarrow V$, which for any input valuation v, associates a *valid* valuation v' that exercises the *same* control path as v does, i.e., if $VR(v) = v'$, then $Run_v = Run_{v'}$. Intuitively, the range of VR is a set of *candidate inputs* that are benign and exercise all feasible control paths in the program. This function may not even exist and is hard to statically or dynamically determine on real programs; we will circumvent the construction of this function in the final scheme.

Now we can easily define when an input v is invalid: v is invalid iff the structure computed by the program on v is different from the one computed on $VR(v)$.

Definition 1. *Let P be a program and $VR : IV \rightarrow V$ be the associated valid-representation function. An input valuation v for P is an* SQL-injection attack *if the structure of the query q that P computes on v is different from that of the query q' that P computes on $VR(v)$ (i.e., $q \not\approx q'$).*

Turning back to our example in Fig. 3, the input $v : \langle n \leftarrow 0; s \leftarrow$"Jim' OR 1=1--"$\rangle$ is an SQL-injection attack since it generates a query whose structure is:

```
SELECT ? FROM ? WHERE ?=?  OR ?=?
```

while its corresponding candidate input $VR(v) = v_1 : \langle n \leftarrow 0; s \leftarrow$"John"$\rangle$ exercises the same control path and generates a different query structure:

```
SELECT ? FROM ? WHERE ?=?
```

Alternate definition using symbolic expressions

Let us now reformulate the above definition of SQL injection in terms that explicitly capture the *symbolic expression* for the query at the end of the run Run_v. Intuitively, on an input valuation v, the program exercises a path that consists of a set of assignments to variables. The symbolic expression for a variable summarizes the effect of all these assignments using a single expression and is solely over the input variables $\langle In_1, \ldots, In_p, Is_1, \ldots, Is_q \rangle$.

For example, consider the input $v : \langle n \leftarrow 0$ and $s \leftarrow$ "Jim' OR 1=1--"$\rangle$ for the program in Fig 3. This input exercises Path 1, and the `SELECT` statement is the only statement along this path. The symbolic expression for the query string s_0 on this input at the point of query is $Sym_\pi(s_0)$:

"SELECT * FROM employdb WHERE name='"+s+"' "

Definition 2 Symbolic expressions. *Let U be a set of integer and string variables, and let π be a sequence of assignments involving only variables in U.*

Then the symbolic expression after executing π, for any integer variable $n \in U$ is an arithmetic expression, denoted $Sym_\pi(n)$, and the symbolic expression for a string variable $s \in U$ is a string expression, denoted $Sym_\pi(s)$. These expressions are defined inductively over the length of π as follows:

—*If $\pi = \epsilon$ (i.e., for the empty sequence),*

$$\begin{array}{lcll} Sym_\epsilon(n) & = & n & \text{if } n \in U \\ & = & 0 & \text{otherwise} \\ Sym_\epsilon(s) & = & s & \text{if } s \in U \\ & = & \epsilon & \text{otherwise} \end{array}$$

—*If $\pi = \pi'.(n' := ae(t_1, \ldots, t_k))$, then*

$$\begin{array}{lcl} Sym_\pi(n) & = & Sym_{\pi'}(n), \text{ for every } n \in U,\ n \neq n' \\ Sym_\pi(n') & = & ae(Sym_{\pi'}(t_1), \ldots, Sym_{\pi'}(t_k)) \\ Sym_\pi(s) & = & Sym_{\pi'}(s), \text{ for every } s \in U. \end{array}$$

—*If $\pi = \pi'.(s' := se(t_1, \ldots, t_k))$, then*

$$\begin{array}{lcl} Sym_\pi(n) & = & Sym_{\pi'}(n), \text{ for every } n \in U \\ Sym_\pi(s') & = & se(Sym_{\pi'}(t_1), \ldots, Sym_{\pi'}(t_k)) \\ Sym_\pi(s) & = & Sym_{\pi'}(s), \text{ for every } s \in U,\ s \neq s'. \end{array}$$

For an input valuation v, let π_v denote the set of *assignments* that occur along the control path Run_v that v induces, i.e., π_v is the set of statements of the form $n := ae$ or $s := se$ executed by P on input valuation v. Then the symbolic expression for the query s_0 on v is defined to be $Sym_{\pi_v}(s_0)$.

Observe that for any program P and input valuation v, the value of any variable x computed on v is $Sym_{\pi_v}(x)$. That is, the value of any variable can be obtained by substituting the values of the input variables in the symbolic expression for that variable.

Note that if v and v' induce the same run, (i.e., $Run_v = Run_{v'}$), then $\pi_v = \pi_{v'}$, and hence the symbolic expression for the query computed for v is *precisely* the same as that computed for v'.

We can hence reformulate SQL injection as in Definition 1 precisely as:

DEFINITION 3. *Let P be a program and $VR : IV \rightarrow V$ be the associated valid-representation function. An input valuation v for P is an* SQL-injection attack *if the symbolic expression for the query, $exp = Sym_{\pi_v}(s_0)$ when evaluated on v has a different query structure than when evaluated on $VR(v)$ (i.e., $exp(v) \not\approx exp(VR(v))$).*

Consider the benign candidate input: $v_1 : \langle n \leftarrow 0$ and $s \leftarrow$ "John"$\rangle$ corresponding to the input $v : \langle n \leftarrow 0$ and $s \leftarrow$ "Jim' OR 1=1- -" $\rangle$ for the program in Fig 3 as they exercise the same path (Path 1). The symbolic expression for s_0 on this valid input at the point of query is $Sym_\pi(s_0)$:

"SELECT * FROM employdb WHERE name='"+s+"' "

Note that the *conditionals* that are checked along the control path are *ignored* in this symbolic expression. Substituting any input string for s tells us exactly the query computed by the program along this control path. Consequently, we infer that the input v is an SQL-injection attack since it follows the same path as

the valid input above, but the structure of the query obtained by substituting $s\leftarrow$ "John" in the symbolic expression is *different* from that obtained by substituting $s\leftarrow$ "Jim' OR 1=1- -".

Observe that the solution presented by the above definition is hard to implement. Given an input valuation v, we can execute the program P on it, extract the path followed by it, and compute the symbolic expression along v. Now if we *knew* another valuation v' that exercised the same control path as v does, then we can evaluate the symbolic expression on v and v', and check whether the query structures are the same. However, it is very hard to find a valid input valuation that exercises the same path as v does.

Approximating the SQL injection problem

The problem of finding for every input valuation v, a corresponding valid valuation that exercises the same path as v does (i.e., finding the function VR) is a hard problem. We now argue that a simple approximation of the above provides an effective solution that works remarkably well in practice.

We propose to simply drop the requirement that v' exercises the same control path as v. Instead, we define $VR(v)$ to be the valuation v_c that maps every integer variable to 1 and every string variable s to $a^{v(s)}$, where a^i denotes a string of a's of length i. That is, v_c maps s to a string of a's precisely as long as the string mapped by v.

We note that v_c is manifestly benign and non-attacking for *any* program. Hence substituting this valuation in the symbolic query must yield the *intended query structure* for the control-path executed on v. Consequently, if this intended query structure does not match the query the program computes on v, then we can raise an alarm and prevent the query from executing.

The fact that the candidate valuation v_c may not follow the same control path as v is not important as in any case we will not follow the control path dictated by v_c, but rather simply substitute v_c in the symbolic expression obtained on the control path exercised on v. Intuitively, we are *forcing* the program P to take the same path on v_c as it does on v to determine the intended query structure that the path generates. We will justify this claim using several practical examples below.

Consider our running example again (Fig. 3). The program on input $v : \langle n\leftarrow 0, s\leftarrow$ "Jim' OR 1=1--" $\rangle$ executes the `if`-block, and hence generates the symbolic query exp:

"`SELECT * FROM employdb WHERE name='`" + s+"`' `"

Substituting the input values in this expression yields

```
Q1:  SELECT * FROM employdb WHERE name='Jim' OR 1=1--'
```

Consider the valuation $v_c : \langle n\leftarrow 1, s\leftarrow$ "$aaaaaaaaaaaaa$" $\rangle$. The program on this path follows a *different* control path (going through the `else`-block), and generates a query whose structure is quite unlike the query structure obtained by pursuing the `if`-block. However, substituting v_c in the symbolic expression exp yields

```
Q2:  SELECT * FROM employdb WHERE name='aaaaaaaaaa'
```

which is indeed the correct query structure on pursuing the `if`-block. Since the query structures of `Q1` and `Q2` differ, we detect that the query input is an SQL-injection attack. Note that an input assigning $\langle n \leftarrow 0, s \leftarrow$ "*Jane*"$\rangle$ *will match* the structure of the candidate query.

The above argument leads us to an approximate notion of SQL injection:

DEFINITION 4. *Let P be a program, and v be an input valuation, and v_c the benign candidate input valuation corresponding to v. An input valuation v for P is a* SQL-injection attack *if*

—*the symbolic expression exp for the query string s_0 on the path exercised by v results in different query structures when evaluated on v and v_c (i.e., $Sym_{\pi_v}(v) \not\approx Sym_{\pi_v}(v_c)$).*

The above scheme is clearly implementable as we can build the symbolic expression for the query on the input to the program, and check the structure of the computed query with the structure of the query obtained by substituting candidate non-attacking values in the symbolic query. However, we choose a simpler way to implement this solution: we transform the original program into one that at every point computes values of variables both for the real input as well as the candidate input, and hence evaluates the symbolic query on the candidate input in tandem with the original program.

4. THE CANDID TRANSFORMATION

In this section, we discuss the transformation procedure for the dynamic candidate evaluation technique described in the earlier section. We accomplish this using a simple source transformation of the web application.

For every string variable v in the program, we add a variable v_c that denotes its candidate. When v is initialized from the user-input, v_c is initialized with a benign candidate value that is of the same length as v. If v is initialized by the program (e.g., by a constant string like an SQL query fragment), v_c is also initialized with the same value. For every program instruction on v, our transformed program performs the same operation on the candidate variable v_c. For example, if x and y are two variables in the program, the operation:

$$\texttt{partialQuery} = \text{“}\texttt{name=}\text{”} \ + \ x \ + \ y$$

results in the construction of a partial query string. The transformer performs a similar operation immediately succeeding this operation on the candidate variables:

$$\texttt{partialQuery\_c} = \text{“}\texttt{name=}\text{”} \ + \ x_c \ + \ y_c$$

The operations on the candidate variables thus mirror the operations on their counterparts. The departure to this comes in handling conditionals, where the candidate computation needs to be forced along the path dictated by the real inputs. Therefore, our transformer does not modify the condition expression on the `if-then-else` statement. At run-time, the conditional expression is then only

	Grammar	Production		Definition of the function $\Gamma()$	
defn	$\rightarrow$	`int` n	{	`int` n	} (1a)
	|	`str` s	{	`str` s_c; `str` s	} (1b)
	|	defn_1 ; defn_2	{	$\Gamma(\text{defn}_1)$; $\Gamma(\text{defn}_2)$	} (1c)
	|	`input-int` n	{	`input-int` n	} (1d)
	|	`input-str` s	{	`input-str` s ;	
				`str` s_c := `str-candidate-val(s)`	} (1e)
stmt	$\rightarrow$	`skip`	{	`skip`	} (2a)
	|	$s := se$	{	$s_c := \Gamma(se)$; $s := se$	} (2b)
	|	$n := ae$	{	$n := ae$	} (2c)
	|	stmt_1 ; stmt_2	{	$\Gamma(\text{stmt}_1)$; $\Gamma(\text{stmt}_2)$	} (2d)
	|	`if` be stmt_1	{	let t-stmt_1 = $\Gamma(\text{stmt}_1)$ and	
		`else` stmt_2		t-stmt_2 = $\Gamma(\text{stmt}_2)$ in	
				`if` be t-stmt_1 `else` t-stmt_2	} (2e)
	|	`while` be stmt_1	{	let t-stmt_1 = $\Gamma(\text{stmt}_1)$ in	
				`while` be t-stmt_1	} (2f)
ae	$\rightarrow$	c	{	c	} (3a)
	|	n	{	n	} (3b)
	|	$f_i(\text{t}_1,\ldots,\text{t}_k)$	{	$f_i(\text{t}_1,\ldots,\text{t}_k)$	} (3c)
se	$\rightarrow$	*cstr*	{	*cstr*	} (3d)
	|	s	{	s_c	} (3e)
	|	$g_i(\text{t}_1,\ldots,\text{t}_k)$	{	$g_i(\Gamma(\text{t}_1),\ldots,\Gamma(\text{t}_k))$	} (3f)
be	$\rightarrow$	`false`	{	`false`	} (3g)
	|	`true`	{	`true`	} (3h)
	|	$h_i(\text{t}_1,\ldots,\text{t}_k)$	{	$h_i(\text{t}_1,\ldots,\text{t}_k)$	} (3i)
`sql`	$\rightarrow$	`sql.execute(`se`)`	{	let t-se = $\Gamma(se)$ in	
				compare-parse-trees(se,t-se);	
				`sql.execute(`se`)`	} (4)

Fig. 4. Transformation to compute candidate queries

evaluated on the original program variables, and therefore dictates candidate computation along the actual control path taken by the program. The transformation for the `while` statements are similar.

Function calls are transformed by adding additional arguments for candidate variables. Due to the type safety guarantees of our target language (Java), we only maintain candidates for string variables. We also do not transform expressions that do not involve string variables. For those expressions that involve use of non-string variables in string expressions, we directly use the original variable's values for the candidate.

The transformation for the SQL query statement `sql.execute` calls a procedure `compare-parse-trees` that compares the real and candidate parse trees. This procedure throws an exception if the parse trees are not isomorphic. Otherwise, the original query is issued to the database.

Figure 5 gives the transformed code for the program illustrated in Figure 1. Can-

```
void process-form(string uname, string pwd,
      bool modify, string phonenum) {

 string uname_c = stringofa(strlength(uname));
 string pwd_c = stringofa(strlength(pwd));
 string phonenum_c = stringofa(strlength(phonenum));

 if (modify == false){                    /* Path 1. only display */

  query = "SELECT * from phonebook WHERE username = '" +
               uname + "' AND password = '" + pwd + "'";
  query_c = "SELECT * from phonebook WHERE username = '" +
               uname_c + "' AND password = '" + pwd_c + "'";
 }
 else{                               /* modify telephone number */
   if (phonenum == ""){                    /* Path 2. delete entry */

     query = "DELETE from phonebook WHERE username='" + uname + "' AND
                password = '" + pwd + "' ";
     query_c = "DELETE from phonebook WHERE username='" + uname_c + "' AND
                password = '" + pwd_c + "' ";
    }
   else{                             /* Path 3. update entry */

     query = "UPDATE phonebook SET phonenumber =" + phonenum +
                " WHERE username = '"  + uname +
                "' AND password = '" + pwd + "'";
     query_c = "UPDATE phonebook SET phonenumber =" + phonenum_c +
                 " WHERE username = '" + uname_c +
                 "' AND password = '" + pwd_c + "'";
   }
 }
 compare-parse-trees(query,query_c); /* throw exception if no match */
 sql.execute(query);
}
```

Fig. 5. Transformed source for the example in Figure 1

didate variables `uname_c, pwd_c, phonenum_c` are initialized on entering the method. Method `stringofa` takes an integer as length argument, and returns a string of a's of that length. The actual transformation rules for the while language discussed in the previous section, are presented in Figure 4.

Our tool, CANDID, is implemented to defend applications written in Java, and works for any web application implemented through Java Server Pages or as Java servlets. Figure 6 gives an overview of our implementation. Candid consists of two components: an offline Java program transformer that is used to instrument the application, and an (online) SQL parse tree checker. The program transformer is implemented using the SOOT [Vallée-Rai et al. 1999] Java transformation tool. The SQL parse-tree checker is implemented using the JavaCC parser-generator.

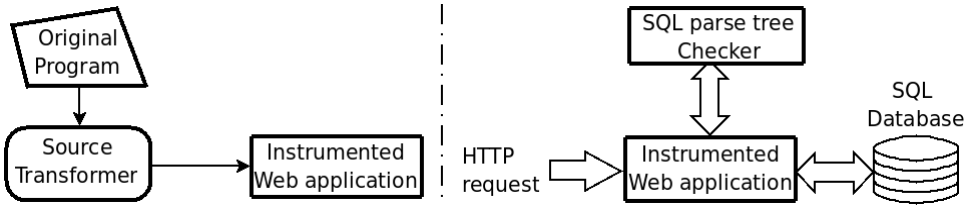

Fig. 6. Overview of CANDID (a) offline view (b) run-time view

A note on parse tree comparison. As mentioned earlier, before a query is sent to the database, we compare the parse trees of the real and candidate queries for attack detection. It is worthwhile to mention here that all lexical entities (including *comments*) need to be included in the parse tree. This idea of including comment tokens in parse trees is slightly different from the traditional understanding of the parse trees which are constructed after the lexical analysis phase where comments are stripped out. The importance of including comment tokens is discussed elaborately in [Buehrer et al. 2005], and is crucial for the robustness of this technique. For instance, consider the following queries `Q1` (benign) and `Q2` (attack):

```
Q1:  SELECT * FROM employdb WHERE name='Jim' OR name='John'

Q2:  SELECT * FROM employdb WHERE name='Jim' OR 'John'='John' --'OR name=''
```

On a first glance, it may seem that the two queries have the same parse structure. In fact, they do not, as there is a mismatch in the token types in the condition after the OR connective. Even if it was the case that there is a match in token types in that condition, the second query has an additional comment token, which when included in the parse tree will make the parse trees differ (Q1's parse tree will not have a comment token). Principle (P2) on Section 3 was in fact based on such a comparison of parse trees. Our CANDID implementation, in fact demands the following (stronger) match criteria: (a) that the tree structures are isomorphic (b) the tokens in the leaves match (inclusive of comment and operator tokens) and (c) the comment strings in both queries (if present) have an exact match, and therefore ensure that comments in SQL queries are not derived from malicious user inputs.

4.1 Resilience of CANDID

The transformation of programs to dynamically detect intentions of the programmer using candidate inputs as presented above is remarkably resilient in a variety of scenarios. We outline some interesting input manipulations Web applications perform, and illustrate how CANDID handles them. Several approaches in the prior literature for preventing SQL injection attacks fail in these simple scenarios (see Section 7).

Conditional queries. Conditional queries are differential queries constructed by programs depending on predicates on the input. For example, a program may form different query structures depending on a boolean input (such as in Fig 1), or perhaps even on particular values of strings. The candidate input may not match

the real queries on these predicates, and hence may take a different path than the real input. However, in the CANDID approach, conditionals are always evaluated on the real inputs only, and hence the candidate query is formed using the same control path the real input exercises. An illustrative example: consider a program that issues an INSERT-query if the input string `mode` is "ADD" and issues a DELETE-query if `mode` is "MODIFY". For a real query with `mode`="ADD", the candidate query will assign `mode`="`aaa`" which, being an invalid mode, may actually cause the program to throw an exception. However, the test for `mode` is done on the real string and hence the candidate query will be an INSERT-query with appropriate values of the candidate inputs substituted for the real inputs in the query.

Input-splitting. Programs may not atomically embed inputs into queries. For example, a program may take an input string `name`, which contains two words, such as "Alan Turing", and may issue a SELECT query with the clauses `FIRSTNAME='Alan' and LASTNAME='Turing'`. In this case, the candidate input is a string of a's of length 11, and though it does not have any white-space, the conditional on where to split the input acts on the real query, and the candidate query will have the clauses `FIRSTNAME='aaaa' and LASTNAME='aaaaaa'`, which elicits the intended query structure.

Preservation of lengths of strings. The method of forcing evaluation of candidate inputs along the control path taken by the real inputs may at first seem delicate and prone to errors. An issue is that the operations performed on the candidate variables may raise exceptions. The most common way this can happen is through string indexing: the program may try to index into the i^{th} character of a string, and this may cause an exception if the corresponding string on the candidate evaluation is shorter than i. This is the reason why we choose the candidate inputs of precisely the same length as the real inputs. Moreover, for all relevant string operations we can show that the lengths of the real and candidate strings are preserved to be equal. More precisely, consider a function g that takes strings and integers as input and computes a string. The function g is said to be *length preserving*, if the length of the string returned by g as well as whether g throws an exception depends only on the *lengths* of the parameter strings and the values of the integer variables. All string methods in the Java String class (such as concatenation and substring methods) are in fact length-preserving. We can show that the strings for candidate variables are precisely as long as their real variable counterparts across any sequence of commands and calls to length-preserving functions. Therefore, they will not throw any exception on the candidate evaluation. In all the experiments we have conducted, the candidate path never raises an exception.

External functions and stored queries. CANDID also handles scenarios where external functions and stored queries are employed in a program. When an external function *ext* (for which we do not have the source) is called, as long as the function is free of side-effects, CANDID safely calls *ext* twice, once on the real variables and once on the candidate variables. Methods such as tainting are unsuitable here as tracking taints cannot be maintained on the external method; however, CANDID can still keep track of the real and intended structures using this mechanism.

Stored queries are snippets of queries stored in the database or in a file, and programs use these snippets to form queries dynamically. Stored queries are commonly used to maintain changes to the database structure over time and to reflect changes in configurations. Changes to stored queries pose problems for static methods as the code requires a fresh analysis, but poses no problems to CANDID as it evaluates queries dynamically on each run.

5. IMPLEMENTATION

The CANDID transformation is fairly simple to implement for any web-application platform; our prototype implementations aim to demonstrate the CANDID approach for the Java platform. Through the Java Server Pages (JSP) framework, Java has become a popular platform for building web applications. Java is a type-safe language, and therefore guarantees that the values of the candidate variables cannot be manipulated by unsafe operations in the language. (An implementation of CANDID for applications written in the C language, for instance, will need to address this issue.)

The implementation of CANDID for Java can be realized at the following layers:

—*Source-code transformation.* An implementation that directly applies the transformation discussed in the previous section on the application's source code.

—*Byte-code transformation.* Here, the transformation will act on JVM bytecodes, an intermediate code representation, as input. As many Java applications are distributed as bytecodes without source, bytecode transformation has the advantage that it can be applicable even when the application's original source code is not available.

—*JVM-level implementation.* This approach is different from the above two approaches in the sense that the original program or bytecode is never transformed. Instead, a Java Virtual Machine is augmented to keep track of candidate values.

Indeed the simplicity of the CANDID approach, readily yields implementation at all these layers. In the rest of this section, we discuss issues related to implementing CANDID for each of these layers.

Source Code Transformation. A source-level transformation is the most natural way to implement CANDID. In fact, this was our first choice for implementation, and we used the SPOON [Pawlak et al. 2006] Java source transformation tool.

An object-oriented and elegant approach to implement CANDID as a Java source transformation, would be to subclass the Java `java.lang.String` class, (say `CandidString`) and introduce the candidate string as a field member of the `CandidString` class. The member methods of the `java.lang.String` class (such as `strcat`) can be overridden in `CandidString`, to additionally perform the corresponding operations on candidate values. Therefore, the transformation of the original program is limited to changing the type of its original `String` variables into `CandidString` variables. During runtime, Java's dynamic method binding will automatically result in calling `CandidString` methods, thus enabling the corresponding operations on candidate strings.

Unfortunately, the `String` class in Java is declared `final`, effectively preventing creation of such a subclass. One can indeed remove the `final` flag from the source

in the Java String libraries, but this will require a special `String` library to run any CANDID transformed application. We note that the String type is implemented differently across Java library implementations and requiring this customized `String` class, will pin down the CANDID transformation to one such Java library implementation. This would defeat our goal of transparently extending the original web application, without requiring customized libraries of basic types such as `String`.

Another approach to achieve the same effect is to define `CandidString` as a new class (i.e., without subclassing `java.lang.String`), but having two `String` variables that represent the real and candidate strings as field members. As before, the transformation of the original program is limited to changing its original `String` variables into `CandidString`, with `CandidString` essentially acting as a "wrapper" class for the `String` class. However, this approach does not work in the presence of the string concatenation operator "+" in Java. The reason is that while Java does not support operator overloading, the "+" operator is overloaded in an *ad-hoc* fashion in the language (the language syntax restricts the use of the "+" operator to only certain basic types defined in the language). The language offers no support for programmer defined operator overloading (such as in the C++ language), and therefore standard compiler tools will not be able to directly compile the transformed application into bytecode.

It is possible to overcome the restrictions due to overloading of the concatenation operator +, by preprocessing an expression such as `a+b` into `new StringBuffer("a").append("b")`. However, in general, source transformation encounters difficulty in directly handling third party methods in a nested expression that involve strings. For example, processing a nested string expression such as `a+f(b)+c`, where `f` is a third party method call (i.e., source code unavailable), requires us to split the expression into three temporary expressions in order to obtain the candidate value corresponding to the call to `f`. While these challenges are addressable at the source level, a low level intermediate representation such as JVM bytecode, expands these nested expressions into simpler instructions, and eases the transformation.

Bytecode Transformation. A second approach to implement CANDID is to transform the bytecode of the Java classes. Each Java source statement corresponds to one or more bytecode statements in the JVM bytecode and these can be transformed using our transformation rules.

In such an implementation, for every variable of string type, an additional variable can be added to track the corresponding candidate value. All the statements that manipulate string variables, can be replicated to manipulate corresponding candidate variables. In JSP / Java based web applications, user inputs are generally retrieved by invoking a fixed set of methods such as `getParameter` and `getAttribute`, that return these possibly malicious inputs. For each such method, a candidate string is initialized.

The key benefit of implementing CANDID at the bytecode level is that the problems posed for source code transformation that were described earlier are avoided. At the bytecode level, all nested expressions are transformed to multiple simpler statements and the "`+`" operator is represented in terms of `append` operations on objects of the `StringBuffer` class.

We implemented above mentioned transformation as an extension to the SOOT optimization framework [Vallée-Rai et al. 1999] and found the *Jimple* bytecode representation of SOOT to be apt for CANDID transformation. Each statement in the Jimple representation is a three-address instruction - two addresses for representing the operands, and one address to store the result, very close to the syntax of the simple *while* language that was used in the previous section. (In contrast, the JVM bytecode is based on a zero-address instruction format in which the operands and results are stored onto stack .) For instance, the *Jimple* representation for a nested expression such as `d = a+f(b)+c` is shown below (simplified for illustration) :

```
d = d.append(a);
tmp = f(b);
d = d.append(tmp);
d = d.append(c);
```

In such a representation, the transformation rules are straightforward. For every three address statement that acts on String type variables, a new three address instruction is added to the code e.g., `d = d.append(a);` causes addition of statement d_c `=` d_c`.append(`a_c`);` where d_c, a_c are candidate variables. Candidate initialization statements are added after every statement that retrieves the user inputs, and the parse tree comparison method is called before the SQL queries are executed.

The CANDID implementation analyzes each Jimple statement of the web application class. Our implementation handles all fifteen types of Jimple statements e.g., `InvokeStmt, AssignStmt`, etc. An example of the transformed statements is the Jimple *identity statement*, which is used to pass parameters to methods. The user defined methods must pass candidate parameters for every real String parameter. To achieve candidate parameter passing, our transformation adds a candidate parameter in the method signature and an identity statement in the method body.

Given the class file for original application, bytecode transformer adds necessary statements for candidate tracking, and creates a CANDID protected class file - ready to be deployed.

A JVM modification approach to track candidate values. A third approach to implement CANDID is to have the Java Virtual Machine keep track of candidate values. An implementation will mainly require modifications to the `java.lang.String` class, in which an additional field will keep track of the candidate value for the corresponding `String` object. All the string operations will also need to be modified to operate on their corresponding candidate values (this approach is similar to the `CandidString` approach at the source level, except that it is implemented internal to the `String` class).

From a software engineering viewpoint, a modification to the JVM is more likely prone to the maintenance problems. Every change made to the JVM needs to be carefully checked, to ensure that the CANDID implementation keeps up with the modification. This is in contrast to source or bytecode transformations, which operate on standardized sources or bytecodes, and therefore are less likely to result in maintenance issues.

However, a JVM implementation presents a very attractive option from a different viewpoint: a web application that runs on a modified JVM *does not* need to

be modified. Protection from SQL injection attacks is available transparently to any application that runs on such a modified VM. This is a very distinct advantage, and may even be preferable in certain practical situations where source or bytecode modifications are not the preferred approaches. Therefore, we have also implemented the CANDID approach in a Java Virtual Machine.

Our first choice of the JVM was the HotSpot VM from SUN MicroSystems. However, we ran into several problems in obtaining and modifying the source of the HotSpot VM, some of whose components were not available under a free / open license. Also, adding a candidate field to the String class triggered a failure result for some hard-coded integrity checks in the HotSpot JVM, as the implementation expects these classes to be unmodified. The lack of customizability in the SUN JVM led us to chose Harmony [4], a free / open source Java VM produced by the Apache foundation, as our implementation platform. Harmony is a clean-room implementation of a JVM that has a simple design that is highly customizable. (On the downside, Harmony lacks some of the advanced memory and speed optimizations of the HotSpot VM.) We were able to implement the initial modifications to the Harmony VM to implement candidate evaluation in three weeks, and further tested the VM's ability to prevent attacks. We report these results in the next section.

6. EVALUATION

We evaluated our approach on three dimensions :

—*Comprehensiveness* in defending many different types of SQL code injection attacks and

—*Applicability* to real world attacks, and

—*Acceptable* performance overheads.

Comprehensive defense evaluation was performed using an attack test suite, obtained from an independent research group [Halfond et al. 2005]. This attack test suite contained more than 50,000 query strings - both benign queries and attack queries based on different vectors of SQL code injection. *Applicability* of CANDID to real world attacks, was evaluated on some commercial Java applications that were vulnerable to SQL code injection attacks reported in the CVE database [MITRE].

6.1 Test-bed Evaluation

Application examples. The test suite obtained from an independent research group (that authored [Halfond et al. 2005]) contained seven applications, five of which are commercial applications: Employee Directory, Bookstore, Events, Classifieds and Portal. These are available at `http://www.gotocode.com`. The other two applications - Checkers and Officetalk, were developed by the same research group. These applications were medium to large in size (4.5KLOC - 17KLOC).

Table I summarizes the statistics for each application. The number of servlets in the second column gives the number exercised in our experiments, with the total

[4]http://harmony.apache.org/

Table I. Applications from the AMNESIA test suite

Application	LOC	Servlets	SCL
Employee Dir	5,658	7 (10)	23
Bookstore	16,959	3 (28)	71
Events	7,242	7 (13)	31
Classifieds	10,949	6 (14)	34
Portal	16453	3 (28)	67
Checkers	5421	18 (61)	5
Officetalk	4543	7 (64)	40

number of servlets in brackets. Our goal was to perform large-scale automated tests (as described below), and some servlets could only be accessed through a complex series of interactions with the application that involved a human user, and therefore were not exercised in our tests. The column SCL reports the number of SQL Command Locations, which issue either a `sql.executeQuery` (mainly `SELECT` statements) and `sql.executeUpdate` (consisting of `INSERT`, `UPDATE` or `DELETE` statements) to the database. Immediately preceding this command location, the CANDID instrumentation calls the parse tree comparison checker.

Attack Suite. Our attack tests contained attacks from the AMNESIA testbed which contained both attack and non-attack inputs for each application. Attack inputs were based on different vectors of SQL code injections. Overall, the attack suite contained 30 different attack string patterns (such as tautology-based attacks, UNION SELECT based attacks [Halfond et al. 2006]), that were constructed based on real attacks obtained from sources US/CERT and CERT/CC advisories. Based on these attack strings, the attack test suite exercised each servlet's injectable web inputs.

The non-attack (benign) inputs of the attack test suite, tested the resilience of the application on legitimate inputs that "look like" attack inputs. These inputs contained data that may possibly break the application in the presence of SQL input validation techniques. Our objective was to deploy two versions of each application: (1) an original uninstrumented version and (2) a CANDID protected version. Also, to simulate a live-test scenario, we wanted to conduct attacks simultaneously on each of these two versions and observe the results. We wanted the original and instrumented versions to be isolated from each other, so that they do not affect the correctness of tests. For this reason, we decided to run them on two separate machines.

In order that the state of the two machines be the same at the beginning of the experiments, we adopted the following strategy: on a host RedHat GNU / Linux system, we created a virtual machine running on VMware also running RedHat EL 4.0 guest operating system. We then installed all the necessary software in this virtual machine: the Apache webserver and Tomcat JSP server, MySQL database server, and the source and bytecode of all Java web applications (original and instrumented versions) in our test suite. Through an automated script, we also populated the databases with tables required by these applications. After configuring the applications to deployment state, we *cloned* this virtual machine by copying all the virtual disk files to another host machine with similar configurations. This resulted in two machines that were identical except for the fact that they ran the original and instrumented versions of the web applications.

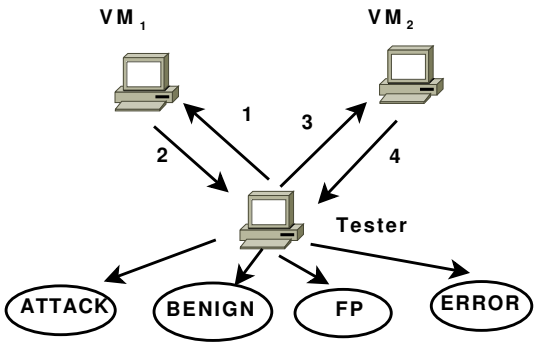

Fig. 7. Testbed setup

Figure 7 illustrates the testbed setup. The original application was deployed on virtual machine VM_1 and the instrumented application was deployed on virtual machine VM_2. A third machine, *Tester*, was used to launch the attacks over HTTP on the original and instrumented web applications, and to immediately analyze the results. For this purpose, a suite of Perl scripts utilizing the `wget` command were developed and used. The master script that executed the attack scripts, ran the following sequence as shown in the Figure 7: (1) first, it launched the attack on the original application and (2) recorded the responses. It then (3) launched the attack on the instrumented application and (4) recorded the responses. After step (4), another post-processing script compared the output from the two VMs and classified the result into one of the following cases (a) the attack was successfully detected by the instrumented application (b) the instrumented application reported a benign string as an attack (c) the instrumented application reported a benign string as benign (d) errors were reported by the original or instrumented application.

Attack evaluation. We ran the instrumented applications with the attack suite, and the results are summarized in Table II. The second column lists the number of input attempts, and the third lists the number of successful attacks on the original application. The number of attacks detected by the instrumented application is shown in the same column. The fourth column shows the number of non-attack benign inputs and any false positives for the instrumented applications. CANDID instrumented applications were able to defend all the attacks in the suite, and there were zero false positives.

The test suite we received had a large number of attack strings that resulted in invalid SQL queries and are reported in column 5. We used a standard SQL parser based on SQL ANSI 92 standard, augmented with MySQL specific language extensions. To ensure the correctness of our parser implementation, we verified that these queries were in fact malformed by using an online SQL Query formatter. [5]

[5]http://www.wangz.net/gsqlparser/sqlpp/sqlformat.htm

Table II. Attack evaluation results

Application	Input Attempts	Succ. Attacks	FPs/ Non-attacks (Benign)	Parse Errors
EmployeeDir	7,038	1529/1529	0/2442	3067
Bookstore	6,492	1438/1438	0/2930	2124
Events	7,109	1414/1414	0/3320	2375
Classifieds	6,679	1475/1475	0/2076	3128
Portal	7,483	2995/2995	0/3415	1073
Checkers	8,254	262/262	0/7435	557
Officetalk	6,599	390/390	0/2149	4060

6.2 Evaluation of applicability to real world attacks

Application examples. For this evaluation, we selected following three JSP / Java based real world applications from the CVE repository, with reported SQL code injection vulnerabilities -

—*Adobe Dreamweaver* - a web application development suite. [6]

—*Daffodil CRM* - a large scale open source software for client relationship management. [7]

—*FileLister* - a filesystem indexing tool with a Web based frontend. [8]

For each of these applications, we obtained the vulnerable versions of the software. The process of setting up these applications involved creating and populating application-specific MySQL databases, creating users with appropriate privileges for applications to connect to MySQL databases through JDBC, installing appropriate versions of the MySQL JDBC Connector drivers, defining appropriate application contexts in the Tomcat server's XML configuration file, and compiling and deploying the application's source code. As shown in Figure 8, these applications were small (814 LOC, Adobe DreamWeaver generated vulnerable application) to large (15093 LOC, Daffodil CRM) in size.

Attack Suite. Figure 8 shows short descriptions and CVE identifiers of vulnerabilities present in these applications. We used CVE description of vulnerabilities to reproduce the reported exploits. Reproducing these exploits required a significant effort both in terms of setting up the applications and constructing the exploit strings.

These vulnerabilities are briefly described for each case: *Adobe Dreamweaver 8.0 and MX 2004 (CVE-2006-2042)* - Dreamweaver 8.0 and MX 2004 generated code has several SQL code injection vulnerabilities - we mention one particularly interesting one below [9] -

[6] http://www.adobe.com/products/dreamweaver
[7] http://crm.daffodilsw.com
[8] http://freshmeat.net/projects/filelister/
[9] based on: http://kb.adobe.com/selfservice/viewContent.do?externalId=585ac720&sliceId=1

CVE	Program	Version and LOC	Attack Details	Candid Prevention
CVE-2006-2042	Adobe Dreamweaver	8 and MX 2004, 814	incorrect use of PREPARE stmt	Success
CVE-2006-0510	Daffodil CRM	1.5, 15093	via login/password to userlogin.jsp	Success
CVE-2005-4040	FileLister	0.51, 9651	via searchwhat to definesearch.jsp	Success

Fig. 8. The real world SQL code injection attacks used in applicability evaluation

```
String userId = (String) request.getParameter("ID");
PreparedStatement ps =
       conn.prepareStatement("SELECT * FROM uid WHERE ID = " + userId + "");
ResultSet rs = ps.executeQuery();
```

The above code snippet is automatically generated by the vulnerable versions of the Adobe Dreamweaver. This snippet uses `PreparedStatement` construct in an insecure fashion. Instead of passing the user input through the `PreparedStatement` exposed "setter" methods, above code directly embeds it in the structure of the `PreparedStatement`. As a result, constructed statement is vulnerable to SQL code injection. In earlier discussion, we emphasized that all new applications can guard effectively against SQL code injection attacks by using `PreparedStatement`. However, an incorrect use of `PreparedStatement` will still leave the applications vulnerable, as exemplified by this instance.

Adobe has provided update and a detailed set of instructions for manually retrofitting the code to fix the vulnerabilities. [10] However, presence of deployed applications (that were generated by older versions of Dreamweaver) vulnerable to this attack cannot be ruled out.

Daffodil CRM (CVE-2006-0510) - The login screen of this application prompts users to supply userid and password. Directly using these inputs in SQL query construction, leaves the application vulnerable to SQL code injection.

FileLister (CVE-2005-4040) - The FileLister application generates a database by indexing the files present in a user specified directory. This database can be searched by a web interface for user specified keywords. The constructed query embeds the user specified search value, and this can be exploited by a suitably constructed attack string.

Experiment setup. For evaluating efficacy in preventing real world attacks, we used JVM based implementation of the CANDID. We created two virtual machines - VM_1 with original Harmony JVM, and VM_2 with modified Harmony JVM. The Tomcat JSP web application servers on both machines, were configured to use the appropriate versions of Harmony JVM.

As discussed earlier, the JVM based CANDID implementation would not require any changes to the web applications to be defended. All the tasks needed to track candidate values are transparently performed in the appropriate JVM classes e.g.,

[10] http://kb.adobe.com/selfservice/viewContent.do?externalId=585ac720&sliceId=1

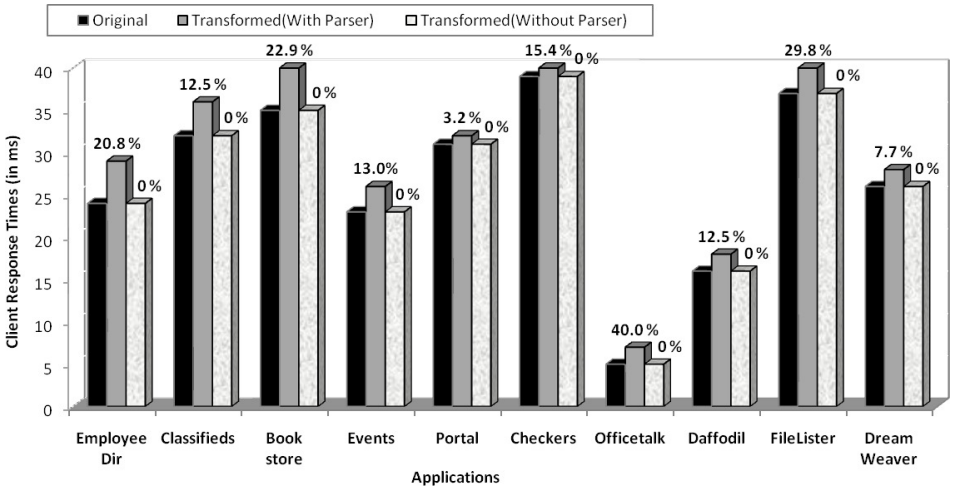

Fig. 9. Performance overheads of the CANDID bytecode implementation

`getParameter()` method of class `HttpServletRequest` initializes manifestly benign candidate values for user inputs, `executeQuery()` method of class implementing `Statement` interface performs SQL parse tree comparisons before executing the queries.

Attack evaluation. We ran the instrumented applications with the reproduced exploits. All three exploits were successfully defended in applications with modified Harmony JVM. Also, verifying expected behavior with benign inputs, indicated zero false positives for the instrumented applications.

6.3 Performance evaluation

Our performance evaluation consists of two different evaluations (1) the CANDID implementation through bytecode rewriting and (2) the CANDID JVM implementation. For both these experiments, we measured the performance impact using JMeter [11] tool from Apache Foundation, which is an industry standard tool for measuring performance impact on web applications.

For both the implementations, we computed the overhead imposed by the approach on one servlet that was chosen from each application, and prepared a detailed test suite for each application. As typical for web applications, the performance was measured in terms of differences in response times as seen by the client. The server was on a Red Hat Enterprise GNU / Linux machine with a 2GHz Pentium processor and 2GB of RAM, that ran in the same Ethernet network as the client. Note that this scenario does not have any network latencies that are typical for many web applications, and is therefore an indicator of the worst case overheads in terms of response times.

[11] http://jakarta.apache.org/jmeter/

6.3.1 *Performance evaluation of the* CANDID *bytecode implementation.* For each test, we took 1000 sample runs and measured the average response time for each run, with caching disabled on the JSP / Web/ DB servers. The results are shown in Figure 9. The figure depicts the time taken by the original application, the transformed code, and also the transformed code without the parser component.

Figure 9 indicates that instrumented applications without SQL parser calls had negligible overhead over the original applications (also optimized for performance using SOOT), when compared to uninstrumented applications.

Figure 9 also indicates that instrumented applications with SQL parser code had varying overheads and ranged from 3.2% (for Portal application) to 40.0% (for OfficeTalk application). These varying overheads are mainly attributed to varying numbers of SQL parser calls in the tested control path e.g., Bookstore application invoked SQL parser code 7 times for the selected page, whereas Portal application only invoked it once. OfficeTalk application's high percentage overhead is attributed to the fact that client response time for the uninstrumented application is very small (5ms) when compared to other applications (23ms - 39ms). This application's actual execution time is dwarfed by factors like class load time and resulted in the higher overhead for the instrumented application. In other applications actual execution time is considerable, and thus the overheads are significantly less.

Overall, above results clearly indicate that bytecode implementation imposes overheads that are quite acceptable. The vanilla SQL parser that we built using the JavaCC parser generator is bulky for online use and contributes to most of the overhead. Notably, the two class files of SQL parser we used are large– 54KB and 21KB - and are frequently loaded. The performance can be improved with a lighter, hand coded SQL parser that is significantly smaller in size. Also, by performing data-flow analysis, we can avoid candidate tracking for string operations that do not contribute to the queries.

6.3.2 *Performance evaluation of the* CANDID *JVM implementation.* While the Harmony VM is robust and has a simple and modular design / implementation that enabled us to perform candidate tracking operations in a short span of time, it does lack various memory and speed optimizations of the HotSpot VM. For instance, our web applications over the TomCat server run about three times slower in an unmodified Harmony VM in comparison to the HotSpot VM. However, despite our rapid prototyping effort we were still able to focus on various performance optimizations in the JVM internal implementation that we describe below.

Figure 10 shows the performance overheads for all ten applications for CANDID JVM implementation. The figure depicts the time taken by the applications running with original Harmony JVM, and with CANDID Harmony JVM implementation. The results demonstrate modest overheads for our JVM approach.

A comparison of Figure 10 (JVM approach) with Figure 9 (bytecode approach) indicates that in terms of absolute response times, the JVM implementation shows much higher numbers. However, the CANDID JVM implementation resulted in less overheads (compared to unmodified JVM executions) than the corresponding overheads in our bytecode implementation. We attribute this improved performance to some optimizations that we performed in the CANDID JVM implementation based on the lessons learnt from our original bytecode implementation.

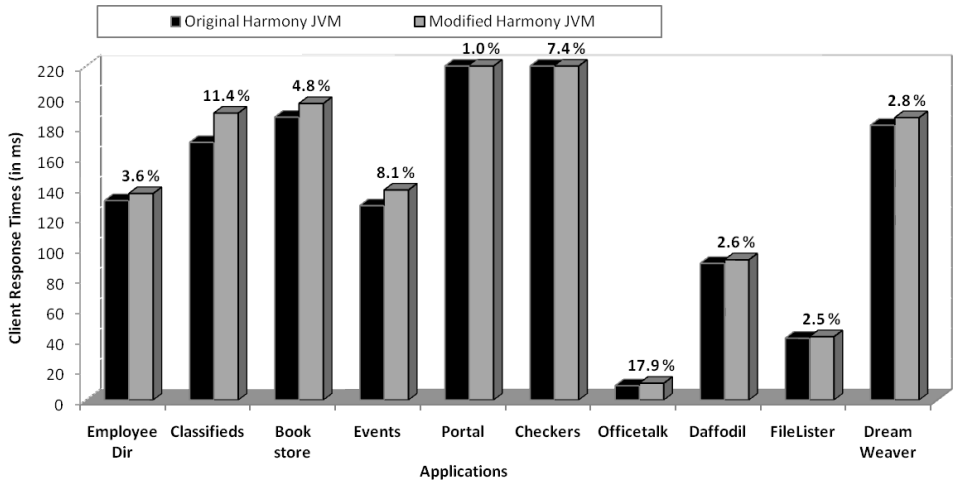

Fig. 10. Performance overhead for CANDID JVM Implementation

First, we restricted the candidate string tracking to only variables that received untrusted data, effectively limiting candidate tracking to partial queries that were derived from untrusted data. This is implemented internally in the JVM by having all real and candidate variable strings initially share the same base character array that stores the string contents until any such string receives user input. Secondly, from the inference that parse tree comparisons were one of the main contributors to the overheads, we performed a few optimizations to the parse tree comparison code. The easiest and most effective change was to limit the parse tree comparisons to cases where real and candidate queries did not match verbatim. This effectively leads to relatively inexpensive operations when the web application issues constant string queries i.e., those not derived from user inputs. A more detailed treatment of these optimizations and implementation is available in a technical report [Chauhan 2008].

Our main objective with the implementation of Harmony was to demonstrate that implementing candidate evaluation in a JVM is an interesting alternative to code transformation implementations. We demonstrated that this is feasible, and our evaluation presents reasonable overheads in an experimental JVM implementation. Our implementation with the Harmony VM thus demonstrates that we have successfully realized this objective. More research on optimizations and implementation techniques is required to scale the JVM implementation approach of candidate evaluation to deployment standards.

7. RELATED WORK

This article is a revised version of [Bandhakavi et al. 2007] in which we introduced the candidate evaluation approach. In this journal version, we also added a more refined and detailed treatment of our approach and provide detailed comparisons with other techniques. These additional discussions appear in this section and in section 8. In addition to these revisions, this paper describes a new JVM

implementation and evaluation that appeared in the previous section. In addition, an evaluation with several real-world examples from CVE was also added to this journal version. We also recently explored the use of candidate evaluation to successfully defend cross-site scripting (XSS) attacks [Bisht and Venkatakrishnan 2008]. However, discussing cross-site scripting attacks is beyond the scope of this paper.

Overview of related work. There has been intense research in detection and prevention mechanisms against SQL injection attacks in past. We can classify these approaches broadly under three headings: (a) coding practices that incorporate defensive mechanisms that can be used by programmers, (b) vulnerability detection using static analysis techniques that warn the programmer of possible attacks, and (c) *defense* techniques that detect vulnerabilities and simultaneously prevent them.

Defensive coding practices include extensive input validation and the usage of `PREPARE` statements in SQL. Input validation is an arduous task because the programmer must decide the set of valid inputs, escape special characters if they are allowed (for example, a name-field may need to allow quotes, because of names like `O'Niel`), must search for alternate encodings of characters that encode SQL commands, look for presence of back-end commands, etc. `PREPARE` statements semantically separate the role of keywords and data literals. Using `PREPARE` statements is very effective against attacks and is likely to become the standard prevention mechanism for freshly written code; augmenting legacy programs to prepare statements is difficult to automate. Two similar approaches, SQL DOM [McClure and Krüger 2005] and Safe Query Objects [Cook and Rai 2005], provide executable mechanisms that enable the user to construct queries that isolate user input.

Vulnerability detection using static analysis

There are several approaches that rely solely on static analysis techniques [Livshits and Lam 2005; Xie and Aiken 2006] to detect programs vulnerable to SQLCIA. These techniques are limited to identifying sources (points of input) and sinks (query issuing locations), and checking whether every flow from a source to the sink is subject to input validation ([Livshits and Lam 2005] is flow-insensitive while [Xie and Aiken 2006] is flow-sensitive). Typical precision issues with static analysis, especially when dealing with dynamically constructed strings, mean that they may identify several such illegal flows in a web application, even if these paths are infeasible. In addition, the user must *manually* evaluate and declare the sanitizing blocks of code for each application, and hence the approach is not fully automatable. More importantly, the tools do not in any way help the user determine whether the sanitization routines prevent all SQL injection attacks. Given that there are various flawed sanitization techniques for preventing SQL injection attacks (several myths abound on Internet developer forums), we believe there are numerous programs that use purported sanitization routines that are not correct, and declaring them as valid sanitizers will result in vulnerable programs that pass these static checks.

All the previous static approaches do not track vulnerabilities across web application modules, and typically lose precision. [Balzarotti et al. 2007] refer to these vulnerabilities as multi-module vulnerabilities and develop an approach that models an application's extended state (for instance an application's session variables)

to identify vulnerabilities that span across modules. Extended state based attacks pose no problem for our approach. Data carried through session variables have their candidate counterparts which denote corresponding benign inputs, and can be used to detect attacks.

Defensive techniques that prevent SQLCIA

Defensive techniques that prevent SQL injection attacks are significantly different from vulnerability analysis as they achieve the more complex (and more desirable) job of transforming programs so that they are guarded against SQL injection attacks. These techniques do not demand the programmer to perform input validation to stave off injection attacks, and hence offer effective solutions for securing applications, including legacy code. We discuss three approaches in detail below; for a more detailed account of various other techniques and tools, including proxy filtering of input, and testing, we refer the reader to a survey of SQL injection and prevention techniques [Halfond et al. 2006].

Using Randomization. One can randomize SQL keywords in parts of the query generated by an application and look for correctly randomized keywords in SQL statements issued to the database to detect attacks. This is the approach taken by SQLrand [Boyd and Keromytis 2004]. Randomization is an interesting, orthogonal approach for detecting code injection attacks, that relies on the secrecy of the randomized keywords. It is also resilient to some of the difficult and subtle scenarios that we describe later (such as the blind-copy scenario we describe in Section 8). SQLrand follows an implementation approach where a developer manually retrofits the web application that dynamically constructs SQL queries, which limits its scalability.

A more general problem with randomization is that it could result in a change of semantics of the program even on benign inputs. For instance, consider the example of an application that checks and limits the length of one of its SQL queries, to say a constant number, and denies any query which is longer. In this case, the randomized keywords will increase the length of the query, and therefore may cause the application to deny the query from being issued. What we are describing through the example here, is an instance of a general problem with randomization, which is that it *changes* the original application data, which therefore may change the intended semantics of the application. Changing a program's original data variables can have unintended consequences even on benign inputs, as illustrated by the example above. CANDID does not modify the values of original program variables (only the additional candidate variables are acted on), and does not therefore change the semantics of the original application on benign inputs.

Learning programmer intentions statically. One approach in the literature has been to learn the set of all intended query structures a program can generate and check at run-time whether the queries belong to this set. The learning phase can be done statically (as in the AMNESIA tool [Halfond et al. 2005]) or dynamically on test inputs in a preliminary learning phase [Valeur et al. 2005]. The latter has immediate drawbacks: incomplete learning of inputs result in inaccuracies that can stop execution of the program on benign inputs.

A critique of AMNESIA: Consider a program that takes in two input strings `nam1` and `nam2`, and issues a select query that retrieves all data where the `name`-field matches either `nam1` or `nam2`. If `nam2` is empty, then the select query issues a search only for `nam1`. Further, assume the program ensures that neither `nam1` nor `nam2` are the string "`admin`" (preventing users from looking at the administrators entries). There are two intended query structures in this program:

```
"SELECT * FROM employdb WHERE name='" + nam1 +  "'"          (1)
"SELECT * FROM employdb WHERE name='" + nam1 +  "'" +        (2)
                        " OR name='" + nam2 +  "'"
```

with the requirement that neither `nam1` nor `nam2` is "admin".

We tested the Java String Analyser (the string analyzer used in AMNESIA [Halfond et al. 2005] to learn query structures statically from Java programs) on the above example. First, JSA detected the above two structures, but could not detect the requirement that `nam1` and `nam2` cannot be "admin". Consider now an attack on the program where `nam1 = "John' OR name='admin"` and `nam2` is empty. The program will generate the query:

```
SELECT * FROM employdb WHERE name='John' OR name='admin'
```

and hence retrieve the administrator's data from the database. Note that though the above is a true SQL injection attack, a tool such as AMNESIA would allow this as its structure is a possible query structure of the program on benign inputs. The problem here is of course flow-sensitivity: the query structure computed by the program must be compared with the query structure the programmer intended along *that particular path* in the program. Web application programs use conditional branching heavily to dynamically construct SQL queries and hence require a flow sensitive analysis. The CANDID approach learns intentions dynamically and hence achieves more accuracy and is flow-sensitive.

Dynamic tainting approaches. Dynamic approaches based on tainting input strings, tracking the taints along a run of the program, and checking if any keywords in a query are tainted before executing the query, are a powerful formalism for defending against SQL injection attacks. It is important to note that these tainting mechanisms are based on *explicit tracking* that track data but ignore implicit flows. [12] Tracking implicit flows is not appropriate for SQL injection detection as it can raise false alarms in very simple common scenarios where programs allow benign implicit flows from user input to SQL keywords. For example, in the phone-book example from Fig. 1, whether the SQL query issued is a DELETE or an UPDATE query (Path 2 and Path 3) depends on the phone-number being empty or not. This intended implicit flow should not be construed as an attack.

Four recent taint platforms [Nguyen-Tuong et al. 2005; Pietraszek and Berghe 2006; Xu et al. 2006; Halfond et al. 2006] offer compelling evidence that the method

[12] [Xu et al. 2006] provide support for implicit flow in their approach, but they do not give any clear automatic criteria as to when this facility will be enabled.

is quite versatile across most real-world programs, both in preventing genuine attacks and in maintaining low false positives. The taint-based approach fares well on all experiments we have studied and several common scenarios we outlined in Section 4.1.

Our formalism is complementary to the tainting approach. There are situations where the candidate approach results in better accuracy compared to the taint approach. Typical taint strategies require the source code of the entire application to track taint information. When application programs call procedures from external libraries or calls to other interpreters, the taint based approach requires these external libraries or interpreters to also keep track of tainting or make the assumption that the return values from these calls are entirely tainted. The second choice may negatively impact tainting accuracy. In our approach, we can call the functions twice, one for the real input and one for the candidate input, which works provided the external function is length preserving and does not have side-effects. We offer a detailed formal comparison with tainting in Section 8.

Dynamic bracketing approaches. [Buehrer et al. 2005] provide an interesting approach where the application program is *manually* transformed at program points where input is read, and the programmer explicitly brackets these user inputs (using random strings) and checks right before issuing a query whether any SQL keyword is spanned by a bracketed input. While this is indeed a very effective mechanism, it relies on the programmer to correctly handle the strings at various stages; for example if the input is checked by a conditional, the brackets must be stripped away before evaluating the conditional.

In [Su and Wassermann 2006], the authors propose a first formalization and an automatic transformation based on the above solution. The formalism is the first formal definition of SQL injection in the literature, and formalizes attacks as changes to query structure, where intentions are captured using bracketing of input. The automatic transformation adds random meta-characters that bracket the input, and adds functions that detect whether any bracketed expression spans an SQL-keyword. However, the formalism and solution set forth in [Su and Wassermann 2006] has some drawbacks:

• The implemented solution of meta-bracketing may not preserve the semantics of the original program even on benign inputs. For example, a program that checks whether the input is well-formed (like whether a credit card number has 16 digits) may raise an error on correct input because of the meta-characters added on either side of the input string. Adding meta-characters only *after* such checks are done in the program is feasible in manual transformation [Buehrer et al. 2005] (though it would involve tedious effort), but is very hard to automate and sometimes impossible (for example if properties of the input are used later in the program, say when the input gets output in a tabular form where the width of tables depends on the length of the inputs). These problems can be overcome by hiding the bracketing in a lower layer (like tainting); however even this will not make the technique work in the case of input-splitting programs (see Section 4.1) (since the input word would span across keywords).

• The formal framework of SQL injection defined in [Su and Wassermann 2006] is based on viewing the program, on an input valuation, to essentially compute a

query as a concatenation of functions of its input (these functions change for each input valuation). SQL injection is then defined by demanding that all keywords emanate from constants in the expression for the query. However, there is no formal mapping from real programs to these query expressions, and it is not clear how program flows such as those caused by conditionals, loops, etc. affect these expressions. Our formalism is much more robust in this regard and the definitions of SQL injection in Definition 1 and Definition 3 are elegant and accurate that work on realistic programs.

In summary, we believe that the dynamic taint-based approach and the CANDID approach presented in this paper are the only techniques that promise a real scalable automatic solution to dynamically detect and prevent SQL injection attacks.

8. DISCUSSION

Limitations of CANDID

Though CANDID is resilient and effective in transforming a large number of natural programs and scenarios (see Section 4.1 on resilience), there are many situations where CANDID fails.

First, there are simple situations where CANDID fails. For instance, if an external function f (that is not CANDID transformed) returns an exception if the input string contains an 'a', then any piece of code that uses this function can easily throw exceptions on even benign input. There are a number of such contrived cases (which the reader can easily construct) that will fail with CANDID.

Second, there are some natural situations where CANDID can fail when implemented at the wrong level. Consider, for example, a function `first` and `second`, that return the first word and second word in their input string, identified using white space, and with the convention that if the input has no white space, then the functions throw exceptions.

Consider the following program snippet (where x is a string input by the user):

```
y = first(x);
z = second(x);
q = "SELECT * FROM db WHERE NAME=" + x " OR " + " NAME=" + y;
```

On a valid input, say "Alan Turing", the above program will succeed in forming a valid non-attacking query. However, it may seem that the CANDID-transformed program will evaluate the candidate input x_c as a string of a's that does not have a white space, and calling `first` on it would result in an exception being thrown.

However, if the *code* for the individual functions was given and transformed as well using CANDID, then the transformation will most likely succeed (it will fail only if the functions were written unnaturally). For example, a natural implementation of `first` may find the first position *pos* where a white space occurs in x and set y to be the prefix of x till position $(pos - 1)$ and z to be the suffix of x from position $pos + 1$. Recall that since we do not keep candidate values for integers, the value of *pos* (real) will be used to index the candidate string to split it. The instrumented code for these functions will hence create y_c and z_c with two strings of a's that match the lengths of the real values of y and z. For the "Alan Turing" example, for instance, the length's of y_c and z_c will be 4 and 6 respectively, will therefore not

result in an exception being thrown and produce an equivalent query q_c. Moreover, the transformation will indeed stave all SQL injection attacks.

In summary, the key to CANDID being successful depends on handling external functions correctly. Natural external functions that are *length-preserving* (see Section 4.1) do not need to undergo any transformation, and that is the reason our implementation and evaluation has been successful on a large set of examples. If these external functions are not length-preserving, the code for these functions need to be transformed as well.

Finally, there is a third class of scenarios where CANDID fails truly because of limitations of the approach. Consider the following program that simply copies a character x to y using the following code:

```
if x='a' then y='a'
else if x='b' then y='b'
else if ...
```

We can extend the above in the same vein to copy a string x to a string y, iterating through the positions of x in an outer loop. Let us call this a *conditional-copy* function.

When instrumented with candidate computation, notice that for a particular value of x (say x="OR"), the set of assignments along the control flow assign the string "OR" to y. In fact the symbolic expression for y on this path will be "O"+"R", and the starting candidate input x_c will be completely ignored, and y_c will evaluate always to "OR". If this keyword ends up in an SQL query, CANDID will wrongly declare that the keyword was intended by the programmer, while in actuality it was not. The *conditional-copy* function is in fact an example that breaks not just CANDID, but *every dynamic SQL detection technique we are aware of* including dynamic tainting.

We note that the above example is contrived, and is very unlikely to appear in a typical web application. (We note that in a scenario in which one deals with malicious programs, the above example has interesting significance.) In the context of web applications, it is only the user input that is untrusted, while the web application code is generally trusted, and therefore this scenario is unlikely.

A formal difference between CANDID and dynamic tainting

In programs that do not call external functions (i.e., when external function code is always available and can be transformed), CANDID and tainting resemble each other in many ways. For instance, both ignore conditionals (in tainting, when a conditional is evaluated on a string, the taint is "below the surface" and ignored), and hence, in a way, both force the program and the shadowed computation (either the taint or the candidate evaluation) along the same path. (By tainting we only refer to approaches that track explicit flows only, not implicit flows.)

One stark example where CANDID fails is the conditional-copy example defined above where a program copies a string x into y, character by character (or bit by bit) using conditional statements and constant assignments. In this contrived but illustrative example, dynamic tainting also fails to detect a flow from x to y.

Over a large class of natural examples as well as simple contrived examples, we noticed that dynamic tainting and CANDID either both work or both fail. This leads

us to the natural question: given that exceptions do not occur and there are no external function calls that are untransformed, are tainting and CANDID essentially the same?

We show that this is not the case and in fact that there is a formalizable difference between tainting and CANDID even in the most idealistic settings.

Let $\texttt{to}-\texttt{char}$ be a native function call that takes in a number i as input and returns a string containing the i^{th} ASCII character. Let *len* be the function that returns the length of a string. Consider the following program.

```
y := to-char(len(x)-len(x)+20)
```

If x is tainted, then a tainting approach would propagate the taint to y as well (assuming that integers keep track of taint as well). However, in the CANDID approach (whether we keep track of candidate integer variables or not), y_c will receive a value *independent* of x_c. In the context of SQL injection, if y was later concatenated with other strings to form a query, and the portion of y actually formed a keyword parsed in SQL, then CANDID would correctly deduce that y *did not flow from* x. However, tainting would detect a flow from x and abort the program.

Contrived as it may be, the above example actually illustrates a formal difference between tainting and CANDID. Assume a program P (say a while-program introduced in Section 3) where all functions return without exceptions and there are no external functions. Given an input valuation v, let π_v be the sequence of assignments in the program that are executed by P on input v, and let Sym_{π_v} denote the symbolic expression computed on this path.

Notice that on a candidate input v_c, CANDID will compute, for any variable y, the exact value $Sym_{\pi_v}(v_c)$ on this path. More precisely, CANDID computes values of the variables according to the *semantic* definition of the assignments along this path. In other words, if we replaced the sequence of assignments in this path by a semantically equivalent sequence of assignments, then CANDID would compute the same value on both paths. Tainting, on the other hand, *may not compute the same taint on semantically equivalent sequences of statements.*

In the above example, for instance, CANDID computes the same candidate value for y whether we execute the assignment

```
y := to-char(len(x)-len(x)+20)
```

or the semantically equivalent assignment

```
y := to-char(20)
```

. Tainting, however, produces a different taint on these two assignments (tainting y in the former and leaving y untainted in the latter)

The above examples destroy all hope of reconciling tainting and CANDID in a common paradigm. We conclude that it is indeed interesting that tainting and CANDID work with similar success on a large classes of natural examples, despite being fundamentally very different techniques.

9. CONCLUSIONS

We have presented a novel technique to dynamically deduce the programmer intended structure of SQL queries and used it to effectively transform applications so that they guard themselves against SQL injection attacks. We have also shown strong evidence that our technique will scale to most web applications.

At a more abstract level, the idea of computing the symbolic query on sample inputs in order to deduce the intentions of the programmer seems a powerful idea that probably has more applications in systems security. There are many approaches in the literature on mining intentions of programmers from code as such intentions can be used as *specifications* for code, and detection of departure from intentions can be used to infer software vulnerabilities and errors [Ammons et al. 2002; Alur et al. 2005; Weimer and Necula 2005]. The idea of using candidate inputs to mine programmer intentions is intriguing and holds much promise.

ACKNOWLEDGMENTS

We heartily thank Sruthi Bandhakavi, who was involved in this research and much of its ideas and implementation, and co-authored the conference version of this paper. We would also like to thank Megha Chauhan for her support with the JVM implementation. We thank William Halfond and Alessandro Orso for providing us their test suite of applications and attack strings. We thank Zhendong Su and Gary Wassermann for insightful email discussions. Thanks are due to Tejas Khatiwala, Mike Ter Louw, Saad Sheikh and Michelle Zhou for their suggestions on improving the draft. Finally, we thank the anonymous referees of TISSEC 2008 and CCS 2007 for their feedback.

REFERENCES

Alur, R., Černý, P., Madhusudan, P., and Nam, W. 2005. Synthesis of Interface Specifications for Java Classes. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 98–109.

Ammons, G., Bodík, R., and Larus, J. R. 2002. Mining Specifications. *SIGPLAN Not. 37,* 1, 4–16.

Anley, C. 2002. Advanced SQL Injection in SQL Server Applications, White paper, Next Generation Security Software Ltd. Tech. rep.

Balzarotti, D., Cova, M., Felmetsger, V. V., and Vigna, G. 2007. Multi-Module Vulnerability Analysis of Web-based Applications. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*. 25–35.

Bandhakavi, S., Bisht, P., Madhusudan, P., and Venkatakrishnan, V. N. 2007. CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations. In *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM, New York, NY, USA, 12–24.

Biba, K. J. 1977. Integrity Considerations for Secure Computer Systems. Tech. Rep. ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA. Apr.

Bisht, P. and Venkatakrishnan, V. N. 2008. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In *Detection of Intrusions, Malware and Vulnerability Analysis*. Lecture Notes in Computer Science, vol. 5137. Springer, 23–43.

Boyd, S. W. and Keromytis, A. D. 2004. SQLrand: Preventing SQL Injection Attacks. In *Applied Cryptography and Network Security*. Vol. 3089/2004. Springer Berlin / Heidelberg, 292–302.

Buehrer, G., Weide, B. W., and Sivilotti, P. A. G. 2005. Using Parse Tree Validation to Prevent SQL Injection Attacks. In *SEM '05: Proceedings of the 5th International Workshop on Software Engineering and Middleware*. ACM, New York, NY, USA, 106–113.

Chauhan, M. 2008. An Efficient Implementation of Candidate Evaluation in a Java Environment. https://alcazar.sisl.rites.uic.edu/wiki/pub/Main/CANDIDJavaImplementation/Project_Report_Megha.pdf.

Cook, W. R. and Rai, S. 2005. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*. ACM, New York, NY, USA, 97–106.

Emmi, M., Majumdar, R., and Sen, K. 2007. Dynamic Test Input Generation for Database Applications. In *ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 151–162.

Halfond, W. G., Viegas, J., and Orso, A. 2006. A Classification of SQL-Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*. Arlington, VA, USA.

Halfond, W. G. J., Orso, A., and Manolios, P. 2006. Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, USA, 175–185.

Halfond, W. G. J., Orso, A., and Orso, A. 2005. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 174–183.

Hansen, R. and Patterson, M. 2005. Stopping Injection Attacks with Computational Theory. In *Black Hat Briefings*.

Livshits, V. B. and Lam, M. S. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *SSYM'05: Proceedings of the 14th Conference on USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 18–18.

McClure, R. A. and Krüger, I. H. 2005. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*. ACM, New York, NY, USA, 88–96.

MITRE. Common Vulnerabilities and Exposures List. http://cve.mitre.org/.

Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D. 2005. Automatically Hardening Web Applications using Precise Tainting. In *SEC*. 295–308.

O. Maor and A. Shulman. 2002. SQL Injection Signatures Evasion. White paper, Imperva. Tech. rep.

Pawlak, R., Noguera, C., and Petitprez, N. May 2006. SPOON: Program Analysis and Transformation in Java. Tech. Rep. 5901, INRIA Research Report.

Pietraszek, T. and Berghe, C. V. 2006. Defending Against Injection Attacks through Context-Sensitive String Evaluation. In *Recent Advances in Intrusion Detection*. Vol. Volume 3858/2006. Springer Berlin / Heidelberg, 124–145.

Sabelfeld, A. and Myers, A. C. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications 21,* 1 (January).

Su, Z. and Wassermann, G. 2006. The Essence of Command Injection Attacks in Web Applications. In *POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 372–382.

Sutton 2006. Dark Reading Security Analysis. Internet. http://www.darkreading.com/document.asp?doc_id=103774&WT.svl=news1_3.

Valeur, F., Mutz, D., and Vigna, G. 2005. A Learning-Based Approach to the Detection of SQL Attacks. In *Intrusion and Malware Detection and Vulnerability Assessment*. Vol. 3548/2005. Springer Berlin / Heidelberg, 123–140.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. 1999. SOOT - A Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. 125–135.

Wassermann, G. and Su, Z. 2007. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*. ACM Press New York, NY, USA, San Diego, CA.

Weimer, W. and Necula, G. C. 2005. Mining Temporal Specifications for Error Detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 461–476.

Xie, Y. and Aiken, A. 2006. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX-SS'06: Proceedings of the 15th Conference on USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA.

Xu, W., Bhatkar, S., and Sekar, R. 2006. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *USENIX-SS'06: Proceedings of the 15th Conference on USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA.

Received Month Year; revised Month Year; accepted Month Year