CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations

Sruthi Bandhakavi
University of Illinois
Urbana-Champaign, USA
sbandha2@uiuc.edu

Prithvi Bisht
University of Illinois
Chicago, USA
pbisht@cs.uic.edu

P. Madhusudan
University of Illinois
Urbana-Champaign, USA
madhu@cs.uiuc.edu

V.N. Venkatakrishnan
University of Illinois
Chicago, USA
venkat@cs.uic.edu

ABSTRACT

SQL injection attacks are one of the topmost threats for applications written for the Web. These attacks are launched through specially crafted user input 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 to detect attacks by comparing them against the intended query structure. 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 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

Keywords

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 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 in 2006 in CVE (Common Vulnerabilities and Exposures list [17]). The percentage of these attacks among the overall number of attacks reported rose from 5.5% in 2004 to 14% in 2006. The recent SQLCIA on CardSystems Solutions [14] 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]. Recent reports suggest that a large number of applications on the web are indeed vulnerable to SQL injection attacks [24], that the number of attacks are on the increase, and is on the list of most prevalent forms of attack [14, 26].

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 [5, 19]. The techniques presented in [28, 15] 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.

A much more satisfactory treatment of the problem is provided by the class of attack prevention techniques that *retrofit* programs to shield themselves against SQL injection attacks [11, 25, 18, 20, 29, 12, 9, 23]. These techniques often require little manual annotation, and instead of detecting vulnerabilities in programs, offer preventive mechanisms that solve for the programmer the problem of defending the code against 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 [13] that they cannot defend against sophisticated attack techniques (for instance, those that use alternate encodings, and database commands to dynamically construct strings) to inject malicious inputs to SQL queries.

A more fundamental technique to the problem of defending SQL injection comes from the commercial database world, in the form of `PREPARE` statements. These statements 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 further influence the SQL query structure, thereby avoiding SQL vulnerabilities altogether. This is in fact a *robust* mechanism to prevent 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 query structures 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 solution, *dynamic candidate evaluation*, a technique 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 symbolic query computed on a program path captures the intention of the programmer, and (2) a simple dynamic technique to mine these programmer-intended query structures using candidate evaluations.

Based on these ideas, we build a tool called Candid (CANdidate evaluation for Discovering Intent Dynamically). Candid 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 the tool in preventing a large set of attacks on a class of 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.

- A fully automated, program transformation mechanism for Java programs that employs this technique, with a discussion of practical issues and resilience to various artifacts of Web applications.

- A comprehensive evaluation of the effectiveness of attack detection and performance overheads.

The problem of SQL injection is one of information flow integrity [7, 21]. 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 solutions have been implemented by mechanisms such as tainting [20, 18, 29] and bracketing [23]. 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 presents the functional and performance evaluation of our approach through experiments on our tool Candid. Related work on other approaches is discussed in Section 6, and Section 7 concludes with a brief discussion.

2. OVERVIEW OF CANDID

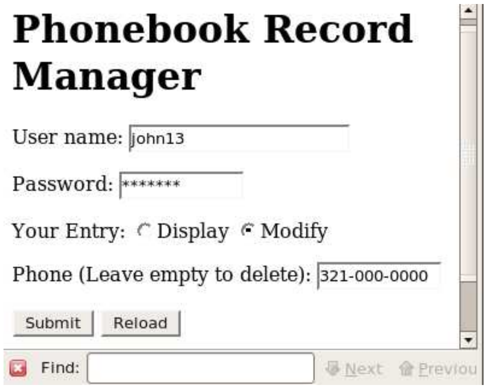
2.1 An example

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.

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 --'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 now



Phonebook Record Manager

User name: john13

Password: *******

Your Entry: ○ Display ● Modify

Phone (Leave empty to delete): 321-000-0000

```
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);
}
```

Figure 1: Approach overview

changed to a query that contained an “OR” operator that 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 that is used in SQL `PREPARE` statements, and also in recent work [9, 23] 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.

The problem then 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 [25]. 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 [11] 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 the related work section (Section 6), 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`, elicits 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*).

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.

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 take a tuple of values, each parameter being a string/integer/boolean, and return 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 a string expression of the form:

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

denotes the concatenation of the constant string with the variable string s. For readability, however, we will represent concatenation using '+' (eg. "`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.

```
Program P
input int n; input str s; str s_0;
if (n = 0) then                              ;; Path 1
  {s_0:="SELECT * FROM employdb WHERE name="+s}
else                                         ;; Path 2
  {s_0:="SELECT * FROM employdb WHERE name="+s+
        " AND status='cur' "};
sql.execute(s_0)
```

Figure 3: An example program

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 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 structures of the two queries is 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 includes all SQL injection attacks. A definition of SQL injection essentially defines this partition.

The primary principles on which our definition of SQL injection is based on are the following:

- (**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.

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

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.

Figure 2: A simple while language

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 in Figure 1. As shown in our comparison studies in Section 6, (P1) is actually a more general principle than the underlying ideas suggested in earlier works [11, 23].

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. As mentioned in the earlier section, `PREPARE` statements are based on (P2). Moreover, (P2) is a well-agreed principle in other works on detecting SQL injection [9, 23]. Given the above principles, we 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 π for any integer variable $n \in V$ is an arithmetic expression, denoted $Sym_\pi(n)$, and the symbolic expression for a string variable $s \in V$ 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),*
$$\begin{array}{rcll} Sym_\epsilon(n) & = & n & \text{if } n \in I \\ & = & 0 & \text{otherwise} \\ Sym_\epsilon(s) & = & s & \text{if } s \in I \\ & = & \epsilon & \text{otherwise} \end{array}$$

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

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

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 Def 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 $v_c : \langle n \leftarrow 1, s \leftarrow$ “*aaaaaaaaaa*” $\rangle$. 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., $exp(v) \not\approx exp(v_c)$).*

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 program 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

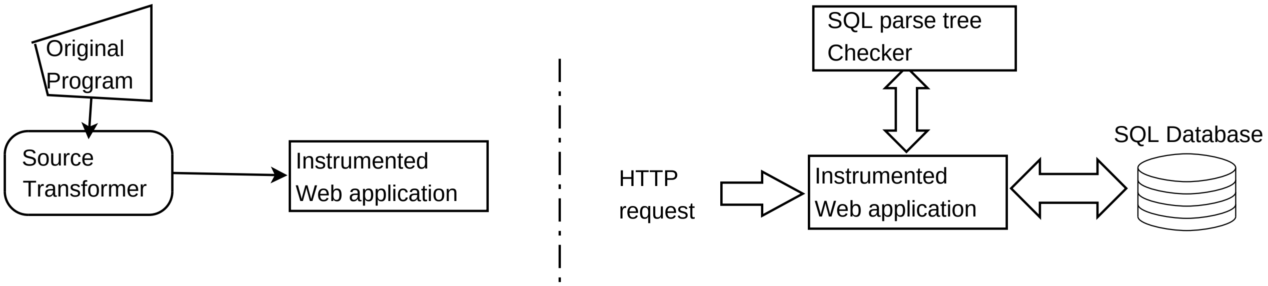


Figure 4: Overview of Candid (a) offline view (b) run-time view

performs the same operation on the candidate variable v_c. For example, if x and y are two variables in the program, the operation:

$$\text{“}\texttt{SELECT * FROM employdb WHERE name=}\text{”} + x + y$$

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

$$\text{“}\texttt{SELECT * FROM employdb WHERE name=}\text{”} + x_c + y_c$$

The operation 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 translator does not modify the condition expression on the `if-then-else` statement. At runtime, the conditional expression is then only 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 samples. 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 4 gives the transformed code for the program illustrated in Figure 1. The actual transformation rules for the while language presented in the previous section is presented in Figure 6.

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 4 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 [22] Java transformation tool. The SQL parse-tree checker is implemented using the JavaCC parser-generator.

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 recent literature for preventing SQL injection attacks fail in these simple scenarios (see Section 6).

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 input substituted for the real input 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 is done 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 input may at first seem delicate and prone

```
void process-form(string uname, string uname_c, string pwd, string pwd_c, bool modify,
          string phonenum, string phonenum_c) {
 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);
}
```

Figure 5: Transformed source for the example in Figure 1

Grammar Production				Definition of the function Γ()		
defn	→	`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	→	`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 `else` stmt_2	{	let $\text{t-stmt}_1 = \Gamma(\text{stmt}_1)$ in let $\text{t-stmt}_2 = \Gamma(\text{stmt}_2)$ in `if` be t-stmt_1 `else` t-stmt_2	}	(2e)
	|	`while` be stmt_1	{	let $\text{t-stmt}_1 = \Gamma(\text{stmt}_1)$ in `while` be t-stmt_1	}	(2f)
ae	→	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	→	$cstr$	{	$cstr$	}	(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	→	`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.execute(`se`)`	→	`sql.execute(`se`)`	{	let $t\text{-}se = \Gamma(se)$ in compare-parse-trees(se,$t\text{-}se$); `sql.execute(`se`)`	}	(4)

Figure 6: Transformation to compute candidate queries

to errors. An issue is that the operations performed on the candidate path 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 to be 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 functions in the Java String class (such as concatenation and substring function) 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 infeasible in this scenario 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 that are dynamically changed over time 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 attacks dynamically on each run.

5. IMPLEMENTATION AND EVALUATION

Transformation. The automated transformation was implemented for Java byte-code using an extension to the SOOT optimization framework [22]. SOOT provides a three-address intermediate byte-code representation, Jimple, suitable for code analysis and optimization. Class files of the uninstrumented applications were processed using the SOOT framework with CANDID to generate instrumented class files for deployment.

The transformer handles all fifteen types of Jimple statements e.g., `InvokeStmt, AssignStmt`, etc. If a statement is found to be acting on, producing or leading to `String` type objects, the transformer adds appropriate statements to perform candidate evaluation; for example, identity statements are used to pass parameters to methods. For user defined methods, corresponding to each String parameter, a candidate parameter is added to the method signature and an identity statement is inserted in the method body for parameter passing.

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

Table 1: Applications from the test suite

As mentioned earlier, we compare the parse trees of the real and candidate queries for attack detection. It is worthwhile to mention here that even the slightest mismatch of the parse trees is detected as an attack.

Application examples. We evaluated our technique using a suite of applications that was obtained from an independent research group [11]. This test suite 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 two other applications, Checkers and Officetalk, were developed by the same research group. These applications were medium to large in size (4.5KLOC - 17KLOC).

Table 1 summarizes the statistics for each application. The number of servlets in the second column gives the number exercised in our experiment, with the total 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. The attack test suite was also obtained from the authors of [11]. It consists, for each application, both attack and non-attack inputs that test several kinds of SQL injection vulnerabilities. Overall, the attack suite contains 30 different attack string patterns (such as tautology-based attacks, UNION SELECT based attacks [13]), 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 makes use of each servlet's injectible web inputs.

The test suite also contained non-attack (benign) inputs that tested the resilience of the application on legitimate inputs that "looked like" attack inputs. These inputs contain data that may possibly break the application in the face of SQL input validation techniques.

Experiment setup. 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 deploy attacks simultaneously on each of these two versions and observe the

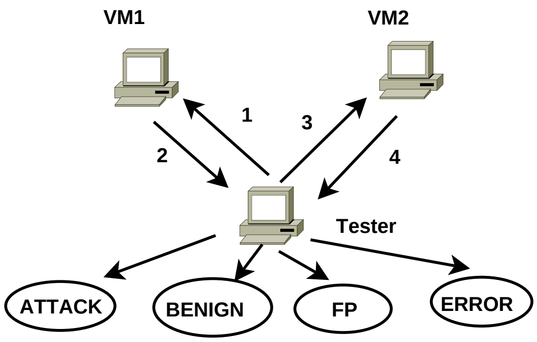


Figure 7: Testbed Setup

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 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 database 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.

Figure 7 shows the test scenario. The original application was deployed on virtual machine VM1 and the instrumented application was deployed on virtual machine VM2. A third machine ("Tester") was used to launch the attacks over HTTP on the original and instrumented web applications, and also was used to immediately analyze the result. For this purpose, a suite of Perl scripts utilizing the `wget` command were developed and used. The master script that ran the attack scripts ran in the following sequence, as shown in the figure (1) it launched the attack first 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 application with the attack suite, and the results are summarized in Table 2. 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 application. CANDID instrumented applications were able to defend all the attacks in the suite, and there were zero false positives.

Application	Input attemp.	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

Table 2: Attack Evaluation results

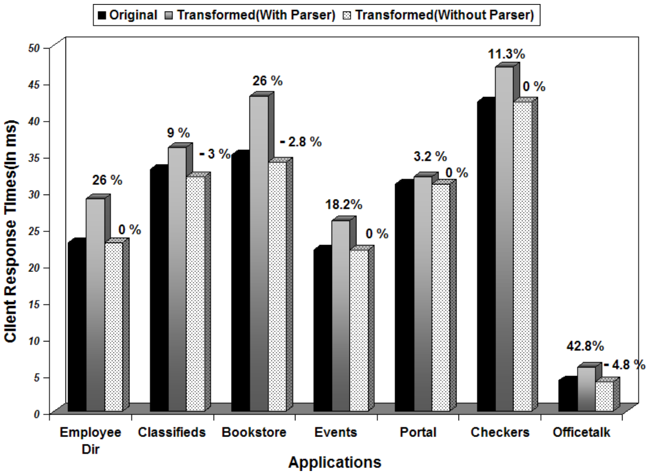


Figure 8: Performance Overhead

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 using an online SQL Query formatter [1].

Performance evaluation. For testing the performance impact we used the web application benchmarking tool JMeter [6] from Apache Foundation, an industry standard tool for measuring performance impact on Java web applications.

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 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.

For each test, we took 1000 sample runs and measured the average numbers for each run, with caching disabled on the JSP / Web/ DB servers. The results are shown in Figure 8. The figure depicts the time taken by the original application, the transformed code, and also the transformed code without the parser component.

Figure 8 indicates that instrumented applications without SQL parser calls had negligible overhead when compared to uninstrumented applications. Also, in some cases e.g., Classifieds application, instrumented applications performed better than uninstrumented applications. We attribute this to the fact that the instrumented class files created by SOOT are optimized, and in some cases are smaller (5% - 9%) in size than the corresponding uninstrumented class files.

Figure 8 also indicates that instrumented applications with SQL parser code had varying overheads and ranged from 3.2% (for Portal application) to 42.8% (for OfficeTalk application). These varying overheads are mainly attributed to varying numbers of SQL parser calls in each page 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 overhead is attributed to the fact that client response time for uninstrumented application is very small (8ms) when compared to other applications (22ms - 42ms). This application's actual execution time is dwarfed by factors like class load time and resulted in high overhead for instrumented application. In other applications actual execution time is considerable, and thus the overheads are significantly less.

The above results clearly show that CANDID's overheads are quite acceptable. We briefly discuss some of the possible optimizations here that could further improve the performance. The SQL parser contributes to most of the overhead and can be greatly improved. Notably, the two class files of SQL parser we used are large– 54KB and 21KB, and are frequently loaded. A hand coded SQL parser can be significantly smaller in size. Also, by performing flows-to/reachability analysis, we can avoid transformations of string operations that do not contribute to the query.

6. COMPARISON WITH RELATED WORK

There has been intense research in detection and prevention mechanisms against SQL injection attacks recently. 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 hard to automate and not viable. Two similar approaches, SQL DOM [16] and Safe Query Objects [10], 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 [15, 28] 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 ([15] is flow-insensitive while [28] 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 the 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.

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 paradigms such as instruction set randomization [8], proxy filtering of input, and testing, we refer the reader to a survey of SQL injection and prevention techniques [13].

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 [11]) or dynamically on test inputs in a preliminary learning phase [25]. 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 + “’”
“SELECT * FROM employdb WHERE name=’” + nam1 + “’” +
                    “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 [11] 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 of 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.

Four recent taint platforms [18, 20, 29, 12] offer compelling evidence that the method is quite versatile across most real-world programs, both in preventing genuine attacks and in maintaining low false positives. The taint-based approach fairs well on all experiments we have studied and several common scenarios we outlined in Section 4.1.

Our formalism is complimentary 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 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 does not have side-effects.

Dynamic Bracketing approaches. Buehrer et al. [9] 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 [23], the authors propose both a formalization and an automatic transformation based on the above solution. The formalism is the only other formal definition of SQL injection in the literature, and formalizes changes of query structure using randomized 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 [23] has several drawbacks:

• The 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. There are several other scenarios outlined in Section 4.1 where the scheme fails: conditional querying (where say a string input determines the query structure, but would fail with meta-brackets), input splitting (since the input word would span across keywords), etc. Adding meta-characters only *after* such checks are done in the program is feasible in manual transformation [9] (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).

• The above problems are in fact deep-rooted in the formalism developed in [23], which considers an overly simple notion of an application program that essentially takes in the input, applies a single filter function on it, and concatenates them to form a query. Program constructs such as conditionals and loops are ignored and is the source of the above problem (formally, a function applied on a bracketed input can behave very differently than when applied on the real input). Our formalism is much more robust in this regard and the definition of SQL injection in Definition 1 and Definition 3 are elegant and accurate definitions 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.

7. 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 [4, 3, 27]. The idea of using candidate inputs to mine programmer intentions is intriguing and holds much promise.

Acknowledgements This research is supported in part by NSF grants (CNS-0716584), (CNS-0551660), (IIS-0331707), (CNS-0325951), and CNS-0524695. We thank William Halfond and Alessandro Orso for providing us their test suite of applications and attack strings. 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 for their feedback.

8. REFERENCES

[1] Online SQL syntax checker. http://www.wangz.net/gsqlparser/sqlpp/sqlformat.htm.

[2] Using Google code search to find security bugs. Internet Bulletin, Oct 2006.

[3] Alur, R., Cerný, P., Madhusudan, P., and Nam, W. Synthesis of interface specifications for java classes. In *POPL* (2005), pp. 98–109.

[4] Ammons, G., Bodík, R., and Larus, J. R. Mining specifications. In *POPL* (2002), pp. 4–16.

[5] Anley, C. Advanced SQL injection in SQL server applications, White paper, Next Generation Security Software Ltd. Tech. rep., 2002.

[6] Apache. The JMeter project. http://jakarta.apache.org/jmeter/.

[7] Biba, K. J. Integrity considerations for secure computer systems. Tech. Rep. ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, Apr. 1977.

[8] Boyd, S. W., and Keromytis, A. D. Sqlrand: Preventing sql injection attacks. In *ACNS* (2004), pp. 292–302.

[9] Buehrer, G., Weide, B. W., and Sivilotti, P. A. G. Using parse tree validation to prevent sql injection attacks. In *SEM* (2005).

[10] Cook, W. R., and Rai, S. Safe query objects: statically typed objects as remotely executable queries. In *ICSE* (2005), pp. 97–106.

[11] Halfond, W., and Orso, A. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *ASE* (2005), pp. 174–183.

[12] Halfond, W., Orso, A., and Manolios, P. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. In *FSE* (2006), pp. 175–185.

[13] Halfond, W. G., Viegas, J., and Orso, A. A Classification of SQL-Injection Attacks and Countermeasures. In *SSSE* (2006).

[14] Secureworks press release. Internet news report, July 2006. http://www.secureworks.com/press/20060718-sql.html.

[15] Livshits, V. B., and Lam, M. S. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium* (2005).

[16] McClure, R. A., and Krüger, I. H. Sql dom: compile time checking of dynamic sql statements. In *ICSE* (2005), pp. 88–96.

[17] MITRE. Common vulnerabilities and exposures list. http://cve.mitre.org/.

[18] Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D. Automatically hardening web applications using precise tainting. In *SEC* (2005), pp. 295–308.

[19] O. Maor and A. Shulman. SQL injection signatures evasion. White paper, Imperva. Tech. rep., 2002.

[20] Pietraszek, T., and Berghe, C. V. Defending against injection attacks through context-sensitive string evaluation. In *RAID* (2005).

[21] Sabelfeld, A., and Myers, A. C. Language-based information-flow security. *IEEE JSA*, (2003).

[22] Soot: a java optimization framework. http://www.sable.mcgill.ca/soot/.

[23] Su, Z., and Wassermann, G. The essence of command injection attacks in web applications. In *POPL* (2006), pp. 372–382.

[24] Dark reading security analysis. Internet, September 2006. http://www.darkreading.com/document.asp?doc_id=103774&WT.svl=news1_3.

[25] Valeur, F., Mutz, D., and Vigna, G. A learning-based approach to the detection of sql attacks. In *DIMVA* (2005), pp. 123–140.

[26] Top five vulnerabilities. IT management security report. http://www.computerweekly.com/Articles/2004/04/16/201840/Top+five+threats.htm.

[27] Weimer, W., and Necula, G. C. Mining temporal specifications for error detection. In *TACAS* (2005), pp. 461–476.

[28] Xie, Y., and Aiken, A. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium* (2006).

[29] Xu, W., Bhatkar, S., and Sekar, R. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium* (2006).