Automatically Preparing Safe SQL Queries

Prithvi Bisht, A. Prasad Sistla, and V.N. Venkatakrishnan

Department of Computer Science
University of Illinois, Chicago, USA
{pbisht,sistla,venkat}@cs.uic.edu

Abstract. In this paper, we present the first sound method for automatically transforming the code of a legacy web application to generate queries that employ PREPARE statements in place of unsafe SQL queries. Our approach has been implemented as a source code transformation tool for PHP based web applications. We demonstrate the effectiveness of our approach over a number of web applications that are vulnerable to SQL injection attacks, and were able to successfully retrofit them to guarantee security by construction.

Key words: Static program transformation, Security by construction, Symbolic evaluation, SQL injection

1 Introduction

In the last decade, the Web has rapidly transitioned to an attractive platform, and web applications have significantly contributed to this growth. Unfortunately, this transition has resulted in serious security problems that target web applications. A recent survey by the security firm Symantec suggests that malicious content is increasingly being delivered by Web based attacks [2], of which SQL injection attacks (SQLIA) have been of widespread prevalence. For instance, the SQLIA based Heartland data breach[1] allegedly resulted in information theft of 130 millions credit/debit cards.

SQL injection attacks are a prime example of malicious input that change the behavior of a program by sly introduction of query structure into the input strings. An application that does not perform input validation (or employs error-prone validation) is vulnerable to SQL injection attacks. Although useful as a first layer of defense, input validation often is hard to get right [9, 27, 15]. The absence of proper input validation has been cited as the number one cause of vulnerabilities in web applications [23].

There is an emerging consensus in the software industry that using PREPARE statements, a facility provided by many database platforms, to construct SQL queries constitutes a robust defense against SQL injections. PREPARE statements are objects that contain precompiled SQL query structures (without data). This allows a programmer to easily *isolate* and *confine* the "data" portions of the SQL query from its "code", *avoiding* the need for (error-prone) sanitization of user inputs. In addition, they are efficient because they do not require any runtime tracking, and provide opportunities for the DBMS server for query optimization [1, 11].

The existing practice to transform an *existing application* to make use of PREPARE statements requires detailed manual effort, which can be tedious and prohibitively expensive for large applications. This paper presents the first automated *sound* program trans-

[1] http://www.wired.com/threatlevel/2009/08/tjx-hacker-charged-with-heartland

formation approach that transforms an existing web application to make use of PREPARE statements. The main challenge in doing this transformation is to ensure that the semantics of the transformed program on non-attack inputs is the same as the original program.

Our approach is implemented in a tool called TAPS (Tool for Automatically Preparing SQL queries). To retrofit an existing application, TAPS uses a novel approach to obtain an understanding of the string operations of the program using symbolic evaluation, and effectively rewrites the program with this understanding.

Our approach and tool has been successfully applied to several real world applications, including one with over 22,000 lines of code. In addition, some of these applications were vulnerable to widely publicized SQL injection attacks present in the CVE database, and our transformation renders them safe *by construction*.

As a concluding remark to the introduction, we note that there is a rich body of literature on SQL injection detection and prevention (see the next section). Our objective is to not propose "one more defense" to this problem. Instead, our contribution is quite the opposite: to develop an automatic technique that will assist developers and system administrators to automatically retrofit their programs with the "textbook defense" for SQL injection.

This paper is organized as follows: Section 3 presents the problem description along with a running example. Section 4 describes our approach in detail. Section 5 presents evaluation of TAPS over several open source PHP applications. We conclude in Section 6.

2 Related Work

There has been extensive work on detecting SQL injection vulnerabilities as well as approaches for defending attacks. Due to space limitations, we briefly summarize them here (see [27] for a detailed discussion).

Defenses based on static analysis There has been extensive research on static analysis to detect whether an application is vulnerable [21, 32, 9, 10, 34, 31, 12, 29]. The most common theme of detection approaches is to reason about sources (user inputs) and their influence on query strings issued at sinks (sensitive operations) or intermediate points (sanitization routines). Our approach provides means for fixing such vulnerabilities through PREPARE statements.

Defenses based on dynamic analysis Dynamic prevention of SQLIA is fairly well researched area and has a large body of well understood prevention techniques: taint based [8, 33, 13, 17], learning based [28, 27, 24, 19, 25, 4, 5, 3, 30], proxy based [26, 20].

At a high level, all these techniques track use of untrusted inputs through a reference monitor to prevent exploits. Unlike the above approaches, the high-level goal of TAPS is not to monitor the program – the goal here is to modify the program to eliminate the root causes of vulnerabilities – isolation of program generated queries from user data while avoiding any monitoring costs.

Automated PREPARE statement generation [7] investigates the problem of automatically converting programs to generate PREPARE statements. This approach assumes that the entire symbolic query string is directly available at the sinks. This assumption does not hold in many typical applications that construct queries dynamically.

3 Background and Problem Statement

We use the following running example: a program that computes a `SELECT` query with a user input `$u`:

```
1. $u = input();
2. $q1 = "select * from X where uid LIKE '%";
3. $q2 = f($u); // f - filter function
4. $q3 = "%' order by Y";
5. $q = $q1.$q2.$q3;
6. sql.execute($q);
```

The above code applies a (filter) function (`f`) on the input (`$u`) and then combines it with constant strings to generate a query (`$q`). This query is then executed by a *SQL sink* (query execution statement) at `line 6`.

The running example is vulnerable to SQL injection if input `$u` can be injected with malicious content and the filter function `f` fails to eliminate it. For example, the user input `' OR 1=1 --` provided as `$u` in the above example can break out of the expected string literal context and add an additional `OR` clause to the query. Typically, user inputs such as `$u` are expected to contribute as literals in the parse structure of any query : more specifically, in one of the two literal *data contexts*: (a) a string literal context which is enclosed by program supplied string delimiters (single quotes) (b) in a numeric literal context. SQL injection attacks violate this expectation by introducing input strings that do not remain confined to these literal data contexts and directly influence the structure of the generated queries [5, 27].

`PREPARE` statement confines all query arguments to the expected data contexts. These statements allow a programmer to declare (and finalize) the structure of every SQL query in the application. Once constructed, the parse structure of a `PREPARE` statement is frozen and cannot be altered by malformed inputs. The following is an equivalent `PREPARE` statement based program for the running example.

```
1. $q = "select * from X where uid LIKE ? order by Y";
2. $stmt = prepare($q).bindParam(0, "s", "%".f($u)."%");
3. $stmt.execute();
```

The question mark in the query string `$q` is a "place-holder" for the query argument `%f($u)%`. In the above example, providing the malicious input `u = ' or 1=1 --` to the prepared query will not result in a successful attack. This is because the actual query is parsed with these placeholders (`prepare` instruction generates `PREPARE` statement), and the actual binding to placeholders happens *after* the query structure is finalized (`bindParam` instruction). Therefore, the malicious content from `$u` cannot influence the structure of query.

The Transformation Problem: In this paper, we aim to replace all queries generated by a web application with equivalent `PREPARE` statements. A web application can be viewed as a SQL query generator, that combines constant strings supplied by the program with computations over user inputs.

Given a large web application, making a change to `PREPARE` statements, is challenging and tedious to achieve through manual transformation. To make the change, a developer must consider each SQL query execution location (sink) of the program and queries that it may execute. Depending on the control path a program may generate and execute different

SQL queries at a sink. Looping behavior may be used to introduce a variety of repeated operations, such as construction of conditional clauses that involve user inputs. Sinks that can execute multiple queries need to be transformed such that each control path gets its corresponding PREPARE statement. This requires a developer to consider all control flows together. Also, each such control flow may span multiple procedures and modules and thus requires an analysis spanning several procedures across the source code.

A second issue in making this change is : for each control flow, a developer must extract query arguments from the original program statements. This requires reasoning about the data contexts. In the running example, the query argument `%f($u)%` is generated at line 5, and three statements provide its value: `f($u)` from line 3, and enclosing character `(%)` from line 2 and 4, respectively. The above mentioned issues make the problem of isolating user input data from the original program query quite challenging.

4 Our approach

We will use the running example from the previous section. This application takes a user input `$u` and constructs a query in the partial query string variable `$q`. A *partial query string variable* is a variable that holds a query fragment consisting of some string constants supplied by the program code together with user inputs. Our approach makes the following assumption about partial query strings.

Main Assumption:: We require the web application to be transformed, to not perform content processing or inspection of partial query string variables.

To guarantee the correctness of our approach, we require this assumption to hold. To explain this assumption for the running example, we require that once the query string `$q` is formed in line 5 of the application by concatenating filtered user input `f($u)` with program generated constant strings in variables `$q1` and `$q3`, it does not undergo deep string processing (i.e., splitting, character level access, etc.,) further en route to the sink. To ensure that this assumption holds, our approach and implementation checks that the program code only performs the following operations on partial query string variables: (a) append with other program generated constant strings or program variables (b) perform output operations (such as writing to a log file) that are independent of query construction and (c) equality comparison with string constant `null`. Checking the above three conditions is sufficient to guarantee that our main assumption holds.

The above conditions are in fact conservative and can be relaxed by the developer, but we believe that the above assumption is not very limiting based on our experimental evaluation of many real world open source applications. In fact, the above assumption has been implicitly held by many prior approaches for SQL injection defense. Defenses such as SQLRand [4], SQLCheck [27] are indeed applicable to real world programs because this assumption holds for their target applications. We note that all of these approaches change the original program's data values. SQLRand randomizes the program generated keywords, SQLCheck encloses the original program inputs with marker tags. These approaches then require that programs do not manipulate their partial query strings in arbitrary ways. For instance, if a program splits and acts on a partial query string after its SQL keywords has been randomized, it introduces the possibility of losing the effect of randomization. A small minority of query generation statements in some programs may

not conform to our main criteria; in this case, our tool reports a warning and requires programmer involvement as discussed in section 4.5.

4.1 Intuitions behind our Approach

As mentioned earlier, user inputs are expected to contribute to SQL queries in string and numeric data literal contexts. Our approach aims to isolate these (possibly unsafe) inputs from the query by replacing existing query locations in the source code with `PREPARE` statements, and replacing the unsafe inputs in them with safe placeholder strings. These placeholders will be bound to the unsafe inputs during program execution (at runtime).

In order to do this, we first observe that the original program's instructions already contain the programmatic logic (in terms of string operations) to build the structure of its SQL queries. This leads to the crucial idea behind our approach: *if we can precisely identify the program data variable that contributes a specific argument to a query, then replacing this variable with a safe placeholder string (?) will enable the program to programmatically compute the `PREPARE` statement at runtime.* The above approach will work correctly if our main assumption is satisfied. We indeed can ensure that the resulting string with placeholders at the original SQL sink will have (at runtime) the body of a corresponding `PREPARE` statement.

The problem therefore reduces to precisely identifying query arguments that are computed through program instructions. In our approach, we solve this problem through symbolic execution [18], a well-known technique in program verification. Intuitively, during any run, the SQL query generated by a program can be represented as a symbolic expression over a set of program inputs (and functions over those inputs) and program-generated string constants. For instance, by symbolically executing our running example program, we obtain the following symbolic query expression :

```
SELECT ... WHERE uid LIKE '%f($u)%' ORDER by Y
```

Notice that the query is expressed completely by constant strings generated by the program, and (functions over) user inputs. (We will define these symbolic expressions formally later.)

Once we obtain the symbolic expression, we analyze its parse structure to identify data arguments for the `PREPARE` statement. In our running example, the only argument obtained from user input is the string `%f($u)%`.

Our final step is to traverse the program backwards to the program statements that generate these arguments, and modify them to generate placeholder (?) instead. Now, we have changed a data variable of a program, such that the program can compute the body of the `PREPARE` statement at runtime.

In our running example, after replacing contributions of program statements that generated the query data argument `%f($u)%` with a placeholder (?), `$q` at line 5 contains the following `PREPARE` statement body at runtime:

```
SELECT ... WHERE uid LIKE ? ORDER by Y, %$q2%
```

The corresponding query argument is the value `%$q2%`. Note that the query argument includes contributions from program constants (such as `%`) as well as user input (through `$q2`) .

Approach overview Figure 1 gives an overview of our approach for the running example. For each path in the web application that leads to a query, we generate a derivation tree that

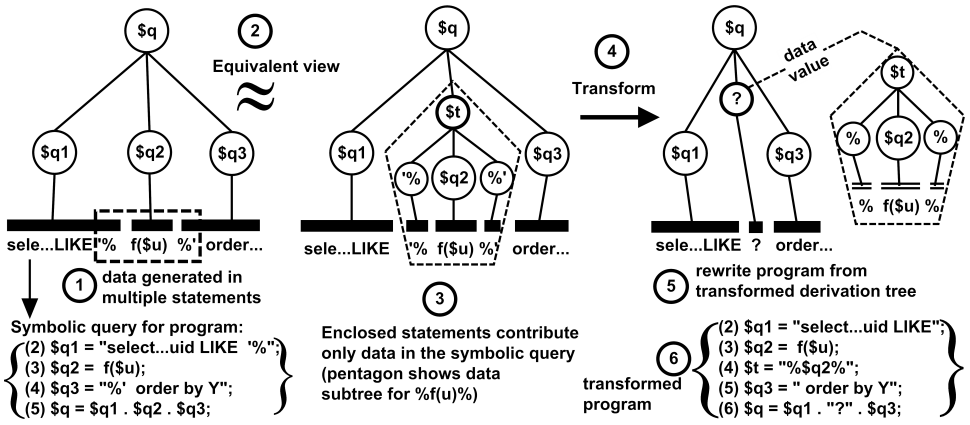

Fig. 1. TAPS: step (1) generates symbolic queries, steps (2-3) separate data reaching the queries, step (4) removes data from symbolic queries, and steps (5-6) generate the transformed program.

represents the structure of the symbolic expression for that query. For our example, `$q` is the variable that holds the query, and step 1 of this figure shows the derivation tree rooted at `$q` that captures the query structure. The structure of this tree is analyzed to identify the contributions of user inputs and program constants to data arguments of the query, as shown in steps 2 and 3. In particular, we want to identify the subtree of this derivation tree that confines the string and numeric literals, which we call the *data subtree*. In step 4, we transform this derivation tree to introduce the placeholder value, and isolate the data arguments. This change corresponds to a change in the original program instructions and data values. In the final step 5, the rewritten program is regenerated. The transformed program programmatically computes the body of the `PREPARE` statement in variable `$q` and the associated argument in variable `$t`.

4.2 Formal description for straight line programs

We give a more precise description using a simple well defined programming language. We assume that all the variables in the language are string variables. Let $\cdot$ denote string concatenation operator. The allowed statements in the language are of the following forms: $x = f(),\ x = y,\ x = y_1 \cdot y_2$ where x is a variable and y is a variable or a constant, y_1, y_2 are variables or constants with the constraint that at most one of them is a constant, and $f()$ is any function including the input function that accepts inputs from the user. Here we describe our approach for straight line programs. Processing of more complex programs, that include conditional statements and certain type of simple loops, is presented later in this section. The approach for such complex programs uses the procedure for straight line programs as a building block.

Derivation Trees. Now consider a straight line program P involving the above type of statements. Assume that P has l number of statements. We let S_i denote the i^{th} statement in P. With each i, $1 \leq i \leq l$, we define a labeled binary tree T_i as follows. Let $x = e$ be the statement S_i. Intuitively, T_i shows the derivation tree for the symbolic value of x immediately after execution of S_i. The root node r_i of T_i is labeled with the pair $\langle i, x \rangle$ and

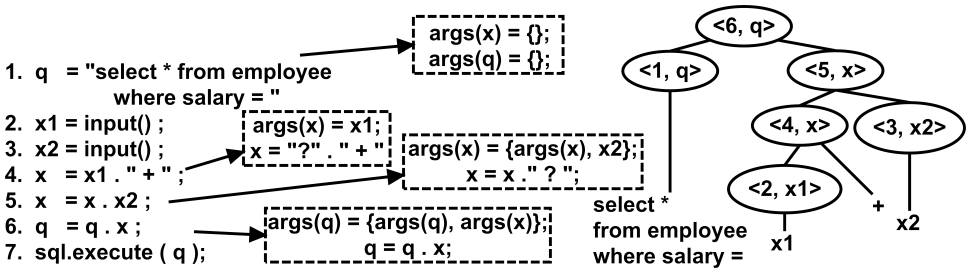

Fig. 2. Labeled derivation tree for symbolic values of q after execution of statement 6.

its left and right children (T_l, T_r) are defined as follows.

$$(\mathbf{T_l}\ ,\ \mathbf{T_r}) = \left\{ \begin{array}{ll} (\ (label = x)\ ,\ \_\ ) & if\ e\ =\ f() \\ (\ (label = c)\ ,\ \_\ ) & if\ e\ =\ c \\ (\ T_j\ ,\ \_\ ) & if\ e\ =\ y \\ (\ T_j\ ,\ T_k\ ) & if\ e\ =\ y\ \cdot\ z \end{array} \right\} \begin{array}{l} \text{Here } T_j \text{ and } T_k \text{ are the} \\ \text{derivation trees of last} \\ \text{statements } j \text{ and } k \text{ before } i \\ \text{that update } y \text{ and } z, \text{ respectively.} \end{array}$$

The derivation tree T_i has two sub-trees only when e is $y \cdot z$. Note that if y (or z) is a constant then the left (or right) sub-tree is a leaf node labeled with the constant, otherwise it is a copy of of some T as defined above. Figure 2 gives a program and the tree T_6 for this program.

Symbolic strings. For the program P, we construct the trees T_i, for $1 \leq i \leq l$. For each tree T_i, we define a symbolic string, called the string generated by T_i, as the string obtained by concatenating the labels of leaves of T_i from left to right. If S_i is of the form $x = e$, then we define the symbolic value of x after S_i to be the symbolic string generated by T_i. For the program given in figure 2, the symbolic value of q after statement 6 is the string `select * from employee where salary =` $x_1 + x_2$

Data sub-strings. Assume that the last statement of P is $sql.execute(q)$ and that this is the only sql statement in P. Also assume that statement i is the last statement that updated q. We obtain the symbolic value s of q after statement i from the tree T_i and parse it using the sql parser. If it is not successfully parsed then we reject the program. Otherwise, we do as follows. From the parse tree for s, we identify the sub-strings of s that correspond to data portions. We call these sub-strings as data sub-strings. For each data sub-string u, we identify the *smallest* sub-tree τ_u, called data sub-tree, of T_i that generated u. Note that τ_u is a copy of T_j for some $j \leq i$. Clearly, u is a sub-string of the string generated by τ_u. Now, we consider the case when the following property (*) is satisfied. (If (*) is not satisfied we transform P into an equivalent program P' that satisfies (*) and we invoke the following procedure on P'; this transformation is described later).

Property (*): *For each data sub-string u, u is equal to the string generated by τ_u.*

Program Transformation: We modify the program so that data sub-strings in symbolic strings are replaced by "?" ($Rule_1$) and all such data sub-strings are gathered into argument lists ($Rule_1$ and $Rule_2$). We achieve this as follows. For each relevant variable x, we introduce a new variable $args(x)$ that contains its list of arguments and initialize it to the

empty lists in the beginning. We use [ ] to represent a list and # to represent a list append operator.

Let the root node of T_i be r_i and the root node of sub-tree τ_u in T_i be r_u. We traverse the tree T_i from node r_u to its root and let $t_1, \ldots, t_k$ be the nodes on this path in that order. Note that $t_1 = r_u$ and $t_k = r_i$. For each j, $1 \leq j \leq k$, let the label of node t_j be given by $\langle j, var_j \rangle$ where var_j represents the variable being updated at the node t_j (note that t_j cannot be a leaf node).

$\mathbf{Rule_1}$*: Eliminating data subtrees* Let j' be the smallest integer such that $1 < j' \leq k$ and $t_{j'}$ has two children. Clearly, the statement $S_{j'}$ is of the form $var'_j = y' \cdot z'$. If $var_{j'-1} = y'$ i.e., τ_u appears in the left subtree of $t_{j'}$. We replace $S_{j'} : var_{j'} = y'.z'$ by the following two statements.

$$args(var_{j'}) = \begin{Bmatrix} [\, y' \,] & \text{if } z' \text{ is a constant} \\ [\, y' \,\#\, args(z') \,] & \text{if } z' \text{ is a variable} \end{Bmatrix}$$
$$var_{j'} = \text{“?”} \cdot z'$$

Note that the second statement above introduces “?” in the query and the first one adds corresponding data sub-string to the argument list. Here $[y']$ represents a list consisting of the single variable y' and $[y' \,\#\, args(z')]$ represents a list obtained by adding y' to the front of the list $args(z')$. If $t_{j'-1}$ is a right child of $t_{j'}$ then $Rule_1$ is applied in a symmetric fashion i.e., $var_{j'} = y' \cdot \text{“?”}$, variable z' is used in place of y', $args(y')$ is used in place of $args(z')$, and z' is added at the end of the list $args(y')$. This rule is applied to transform the lines 4 and 5 of the Figure 2.

$\mathbf{Rule_2}$*: Propagating arguments* For each j'', $j' < j'' \leq k$, the following rule adds an additional statement immediately before the $S_{j''}$ to propagate the argument introduced by $Rule_1$.

$$args(var_{j''}) = \begin{Bmatrix} args(z'') & \text{if } S_j'' : \ var_{j''} = z'' \\ [\, args({y_1}'') \,\#\, args({y_2}'') \,] & \text{if } S_j'' : \ var_{j''} = {y_1}'' \cdot {y_2}'' \end{Bmatrix}$$

The argument lists for $var_{j''}$ is obtained by concatenating the lists $args({y_1}'')$ and $args({y_2}'')$ in that order. If either one of ${y_1}''$ or ${y_2}''$ is a constant string, the above rule sets the argument list to be the argument list of the non-constant variable. Note that z'' can not be a constant string. This rule is used to transform the line 6 in the Figure 2.

Ensuring property (*): Now we consider the case when property (*) is not satisfied. In this case, we transform the program P into another equivalent program for which the property (*) is satisfied. Let Δ be the set of all data sub-strings u of the query string s such that property (*) is violated for them, i.e., u is a strict sub-string of the string generated by τ_u.

Now, observe that r_u has two children, otherwise τ_u will not be the smallest sub-tree that generated u. Let the label of r_u be $\langle m, y \rangle$. Clearly S_m is of the form $y = z_1 \cdot z_2$. Observe that each leaf node of T_i is labeled with a constant string or the name of a variable. For each $u \in \Delta$, we transform P as follows. Fix any such u. Chose a new variable x_u and add a new statement at the beginning of P initializing x_u to the empty string.

The transformation outlined below removes part of u that was computed in z_1 and stores it in x_u. Let v be a leaf node of τ_u such that the *left most* element of u falls in the label of v. The label of v can be written as $s' \cdot s''$ such that s'' is the part that falls in u. Let $t_1, \ldots, t_k$ be the sequence of nodes in τ_u from the parent of v to r_u where r_u is the root node of τ_u. For $1 \leq j < k$, replace S_j by $New(S_j)$ as defined below.

$$New(S_j) = \begin{Bmatrix} \{x_u = s'' \ \cdot \ x_u \ ; \ var_j \ = \ s'\} \text{ if } j = 1 \ \& \ S_1 : var_j = s' \cdot s'' \\ \{x_u = x_u \cdot z; var_j = var_{j-1}\} \text{ if } 1 < j < k \ \& \ S_j : var_j = var_{j-1} \ \cdot \ z \end{Bmatrix}$$

After this, we identify the leaf node w of τ_u such that the *right most* element of u falls in the label of w. P is modified in a symmetric fashion updating variable x_u. Finally, we replace S_m (root of the τ_u) by the following two statements — $x_u \ = z_1 \cdot x_u$; $y \ = x_u \cdot z_2$.

The above transformation is done for each $u \in \Delta$. We say that changes corresponding to two different strings in Δ are conflicting if both of them require changes to the same statement of P. Our handling of the cases of conflicting changes is explained in the next section. Here we assume that changes required by different strings in Δ are non-conflicting; Let P' be the resulting program after changes corresponding to data strings in Δ have been carried out. It can be easily shown that P' is equivalent to P, i.e., the query string generated in the variable q by P' is same as the one generated by P. Further more, P' can be shown to satisfy the property (*).

4.3 Handling of Conditionals and Procedures

In this section, we discuss our approach and implementation for programs that include branching, functions and loops.

Let us first consider branching statements. For programs that include these constructs, TAPS performs inter-procedural slicing of system dependency graphs (SDGs) [14]. Intuitively, for all queries that a SQL sink may receive, the corresponding SDG captures all program statements that construct these queries (data dependencies) and control flows among these statements. TAPS then computes backward slices for SQL sinks such that each slice represents a unique control path to the sink. Each of these control paths is indeed a straight line program, and is transformed according to our approach described in the previous section. A key issue here is the possibility of conflicts: when path P_1 and P_2 of a program share an instruction (statement) $\mathcal{I}$ that contributes to the data argument, then instruction $\mathcal{I}$ may not undergo the same transformation along both paths, and TAPS detects such conflicts. Conflict detection and resolution is described in more detail in Section 4.5. Also note that the inter-procedural slicing segregates unique sequences of procedures invoked to construct SQL queries. Such sequences may have multiple intra-procedural flows e.g., conditionals. These SDGs are then split further for each procedure in above construction such that each slice contains a unique control flow within a procedure.

The above discussion captures loop-free programs. Handling loops is challenging as loops in an application can result in an arbitrary number of control paths and therefore we cannot use the above approach of enumerating paths.

4.4 Loop Handling

First of all, let us consider programs that construct an entire query inside a single iteration of the loop. Let us call the query so constructed *loop independent* query. In this case, the body of the loop is a loop-free program that can be handled according to the techniques described earlier. To ensure whether a query location is loop independent, our approach checks for the following sufficient conditions (1) the query location is in the loop body and (2) every variable used in the loop whose value flows into the query location does not depend on any other variable from a previous iteration. Once these conditions are satisfied, our approach handles loop independent queries as described in the earlier section.

However, there may be other instances where loop bodies do not generate entire queries. The most common example are query clauses that are generated by loop iterations. Consider the following example:

```
1. $u1 = input(); $u2 = input();
2. $q1 = "select * from X where Y =".$u1;
3. while ( --$u2 > 0){
4. $u1 = input();
5. $q2 = $q2." OR Y=".$u1;
6. }
7. $q = $q1.$q2;
8. sql.execute($q);
```

In this case, our approach aims to summarize the contributions of the loop using the symbolic regular expressions. In the above case, at the end of the loop, our objective is to summarize the contribution of `$q2` as `( OR Y=$u1)`*, so that the symbolic query expression can now be expressed as `select * from X where Y = $u1( OR Y=$u1)`*.

The goal of summarization is essentially to check whether we can introduce placeholders in loop bodies. Once we obtain a summary of the loop, if it is indeed the case that the loop contribution is present in a "repeatable" clause in the SQL grammar, we can introduce placeholders inside the loop. In the above example, since each iteration of the loop produces an `OR` clause in SQL, we could introduce the placeholder in statement `6`, and generate the corresponding `PREPARE` statement at runtime.

Previous work [34] has shown that the body of a loop can be viewed as a grammar that represents a language contributing to certain parts of the SQL query, and a grammar can be automatically extracted from the loop body as explained there. We will need to check whether the language generated by this grammar is contained in the language spawned by the repeatable (pumped) strings generated by the SQL grammar. Note that this containment problem is not the same as the undecidable general language containment problem for CFGs, as the SQL grammar is a fixed grammar. However, a decision procedure specific to the SQL grammar needs to be built.

We instead take an alternative approach for this problem by ensuring that the loop operations produce regular structures. To infer this we check whether each statement in the body of the loop conforms to the following conditions: (1) the statement is of the form $q \rightarrow x$ where x is a constant or an input OR (2) it is left recursive of the form $q \rightarrow qx$, where x itself is not recursive, i.e., resolves to a variable or a constant in each loop iteration. It can be shown that satisfaction of these conditions yields a regular language. The symbolic parser is now augmented to see if the regular structure only generates repeatable strings in the SQL language. If this condition holds, we introduce placeholders as described earlier. We find our strategy for loops quite acceptable in practice, as shown in the next section.

4.5 Implementation

We implemented TAPS to assess our approach on PHP applications by leveraging earlier work Pixy [22, 16] and extending it with algorithms to convert programs to Static Single Assignment(SSA) format [6], and then implementation of the transformation described earlier. We briefly discuss some key points below.

We used an off-the-shelf SQL parser and augmented it to recognize symbolic expressions in query strings. The only minor change we had to make was to recognize query strings with associative array references. An associate array access such as `$x['member']` contains single quotes and may conflict with parsing of string contexts. To avoid premature termination of the data parsing context, TAPS ensures that unescaped string delimiters do not appear in any symbolic expression.

Limitations and Developer Intervention TAPS requires developer intervention if either one of the following conditions hold (i) the main assumption is violated (Section 4) or (ii) a well-formed SQL query cannot be constructed statically (e.g., use of reflection, library callbacks) (iii) the SQL query is malformed because of infeasible paths that cannot be determined statically (iv) conflicts are detected along various paths (v) query is constructed in a loop that cannot be summarized.

TAPS implements static checks for all of the above and generates reports for all untransformed control flows along with program statements that caused the failure. A developer needs to qualify a failure as (a) generated by an infeasible path and ignore or (b) re-write of violating statements possible. The number of instances of type (a) can be reduced by more sophisticated automated analysis using decision procedures. In case of (b), TAPS can be used after making appropriate changes to the program. In certain cases, the violating statements can be re-written to assist TAPS e.g., a violating loop can be re-written to adhere to a regular structure as described earlier. The remaining cases can either be addressed manually or be selectively handled through other means e.g., dynamic prevention techniques.

In case of failures, TAPS can also be deployed to selectively transform the program such that control paths that are transformed will generate prepared queries, and those untransformed paths will continue to generate the original program's (unsafe) SQL queries. The sufficient condition to do this in a sound manner is that the variables in untransformed part be not dependent (either directly or transitively) on the variables of the transformed paths. In this case, the transformation can be done selectively on some paths. All sinks will be transformed to `PREPARE` statements, and any untransformed paths will make use of the `PREPARE` statements (albeit with unsafe strings) to issue SQL queries with an empty argument list.

5 Evaluation

Our evaluation aimed to assess TAPS on two dimensions (a) *effectiveness* of the approach in transforming real world applications, and (b) *performance* impact of transformation induced changes.

5.1 Effectiveness

Test suite Table 1 column 1 lists SQLIA vulnerable applications from another research project on static analysis [31] and applications with known SQLIA exploits from Common Vulnerabilities and Exposures (CVE 2009). This table lists their codebase sizes in lines of code and any known CVE vulnerability identifiers (column 2 and 3), number of analyzed SQL sinks and control flows that execute queries at SQL sinks (column 4 and 5), transformed SQL sinks and control flows (column 6 and 7) and number of control flows that required developer intervention (column 8). In this test suite, the larger applications

Application	Size (LOC)	CVE Vulnerability	Analyzed SQL Sinks	Analyzed Control Flows	Transformed SQL Sinks	Transformed Control Flows	Human Intervention Flows
WarpCMS	22773	-	14	200	14	186	14
Utopia NewsPro	7323	-	2	336	2	333	3
AlmondSoft	6633	CVE-2009-3226	22	33	17	27	6
PortalXP TE	5121	CVE-2009-3148	122	122	122	122	0
Gravity Board	2422	CVE-2009-1277	62	62	62	62	0
MyNews	1792	CVE-2009-0739	1	34	1	34	0
Auth	284	CVE-2009-0738	1	5	1	5	0
BlueBird	288	CVE-2009-0740	1	5	1	5	0
Yap Blog	264	CVE-2009-1038	2	6	2	6	0

Table 1. Effectiveness suite applications, transformed SQL sinks and control flows: TAPS transformed over 93% and 99% of the analyzed control flows for the two largest applications.

invoked a small number of functions to execute SQL queries. This caused the number of analyzed sinks and control flows to vary across applications.

Transformed control flows For the three largest applications, TAPS transformed 93%, 99% and 81% of the analyzed control flows. Although smaller in LOC size, the Utopia news pro application had a greater fraction of code involving complex database operations and required analyzing more control flows than any other application. For the remaining applications, TAPS achieved a transformation rate of 100%. This table suggests that TAPS was effective in handling the many diverse ways that were employed by these applications to construct queries.

TAPS did not find any partial query string variables used in operations other than append, null checks and output generation / logging (supports main assumption from Section 4). Further, TAPS did not encounter conflicts while combining changes to program statements required for transformed control flows.

Untransformed control flows The last column of the Table 1 indicates that TAPS requires human intervention to transform some control flows.

As TAPS depends on symbolic evaluation, it did not transform flows that obtained queries at run time e.g., the Warp CMS application used SQL queries from a file to restore the application's database. In two other instances, it executed query specified in a user interface. In both these cases, no meaningful `PREPARE` statement is possible as external input contributes to the query structure. If the source that supplies the query is trusted, then these flows can be allowed by the developer. The limitations of the SQL parser implementation were responsible for two of the three failures in the Utopia news pro application and the rest are discussed below.

Queries computed in loops A total of 18 control flows used loops that violated restrictions imposed by TAPS and were not transformed (11 - Warp CMS, 1 - Utopia news pro, 6 - AlmondSoft). These control flows generated queries in loop bodies that used conditional statements or nested loops. We also found 23 instances of queries computed in loops, including a summarization of `implode` function, that were successfully transformed. In all

Application	Statements changed (%)	Args extracted Avg (max)	Functions traversed Avg (max)	SSA conversion time (%)	Flow enumeration time (%)	Static checks time (%)	Transf-ormation time (%)
WarpCMS	438 (1.9%)	6.6 (27)	2.2 (3)	98.6	0.4	0.4	0.6
Utopia News Pro	333 (4.5%)	1.1 (8)	2.9 (6)	86.9	5.3	6.7	1.1
AlmondSoft	46 (0.7%)	1.3 (4)	1.3 (2)	61.3	12.2	0.1	26.4
PortalXP TE	332 (6.5%)	1.5 (9)	1.0 (1)	96.7	1.0	2.2	0.1
Gravity Board	172 (7.1%)	1.5 (15)	1.0 (1)	94.8	1.3	3.3	0.6
MyNews	56 (3.1%)	2.4 (5)	2.5 (3)	80.7	10.8	2.2	6.3
Auth	17 (6.1%)	3.0 (4)	2.0 (2)	23.4	37.3	8.9	30.4
BlueBird	17 (6.0%)	3.0 (4)	2.0 (2)	23.5	34.6	12.4	29.5
Yap Blog	8 (3.0%)	4.0 (7)	2.0 (2)	53.5	14.2	16.8	15.5

Table 2. Transformation changed less than 5% lines for large applications.

such cases queries were either completely constructed and executed in each iteration of the loop or loop contributed a repeatable partial query.

For untransformed flows TAPS precisely identified statements to be analyzed e.g., the Warp CMS application required 195 LOC to be manually analyzed instead of complete codebase of 22K LOC. This is approximately two orders of magnitude reduction in LOC to be analyzed.

Changes to applications As shown in the second column of Table 2 a small fraction of original LOC was modified during transformation. The columns 3 and 4 of this table show average (maximum) number of data arguments extracted from symbolic queries and functions traversed to compute them, respectively. 2% of changes in LOC were recorded for Warp CMS - the largest application, whereas approximately 5% of lines changed for database intensive Utopia news pro application. We noticed that a significant portion of code changes only managed propagation of the data arguments to `PREPARE` statements. Some of these changes can be eliminated by statically optimizing propagation of arguments list e.g., for all straight line flows that construct a single query, `PREPARE` statement can be directly assigned the argument list instead of propagating it through the partial queries. Overall, this small percentage of changes points to TAPS's effectiveness in locating and extracting data from partial queries.

Further, as columns 3 and 4 suggest, TAPS extracted a large number of data arguments from symbolic queries constructed in several non-trivial inter-procedural flows. For a manual transformation both of these vectors may lead to increased effort and human mistakes and may require substantial application domain expertise. For successfully transformed symbolic queries the deepest construction spanned 6 functions in the Utopia news pro application and a maximum of 27 arguments (in a single query) were extracted for the Warp CMS application, demonstrating robust identification of arguments.

5.2 Performance Experiment

Performance of transformed applications TAPS was assessed for performance overhead on a microbench that consisted of an application to issue an `insert` query. This application did not contain tasks that typically interleave query executions e.g., HTML

generation, formatting. Further, the test setup was over a LAN and lacked typical Internet latencies. Overall, the microbench provided a worst case scenario for performance measurement.

We measured end-to-end response times for 10 iterations each with TAPS transformed and original application and varied sizes of data arguments to `insert` queries from 256B to 2KB. In some instances TAPS transformed application outperformed the original application. However, we did not find any noteworthy trend in such differences and both applications showed same response times in most cases. It is important to note here that dynamic approaches typically increase this overhead by 10-40%. Whereas, TAPS transformed application's performance did not show any differences in response times. Overall, this experiment suggested that TAPS transformed applications do not have any overheads.

Performance of the tool We profiled TAPS to measure the time spent in the following phases of transformation: conversion of program to SSA format, enumeration of control flows, static checks for violations described earlier, derivation tree generation and changing the program. The time taken by each phase is summarized in the last four columns of Table 2. The largest application took around 2 hours to transform whereas the rest took less than an hour. The smallest three applications were transformed in less than 5 seconds. For large applications TAPS spent a majority of time in the SSA conversion. The only exception to this case occurred for AlmondSoft application which had smaller functions in comparison to other applications and hence SSA conversion took lesser time. We wish to note here that TAPS is currently not optimized. A faster SSA conversion implementation may improve performance of the tool and by summarizing basic blocks some redundant computations can be removed. For a static transformation these numbers are acceptable.

6 Conclusion

In this paper, we presented TAPS, a static program transformation tool that modifies web applications to make use of `PREPARE` statements. We presented experimental results with several open-source applications to assess the effectiveness of TAPS. Our approach provides evidence that it is possible to successfully design retrofitting techniques that guarantee security (by construction) in legacy applications, and eliminate well known attacks.

References

1. JDBC: Using prepared statements. http://java.sun.com/docs/books/tutorial/jdbc/basics/prepared.html.
2. Symantec Internet Security Threat Report. Technical report, March 2007.
3. Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V.N. Venkatakrishnan. CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations. In *CCS*, 2007.
4. Stephen W. Boyd and Angelos D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *ACNS*, 2004.
5. Gregory Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. In *SEM '05*, 2005.
6. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *PLAS*, 1991.
7. Fred Dysart and Mark Sherriff. Automated fix generator for SQL injection attacks. *ISSRE*, 2008.
8. Anh Nguyen-Tuong et al. Automatically Hardening Web Applications using Precise Tainting. In *20th International Information Security Conference*, 2005.

9. D. Balzarotti et al. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *IEEE Security and Privacy*, 2008.
10. W. Halfond et al. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *ASE '05*.
11. H. Flak. MYSQL prepared statements. http://dev.mysql.com/tech-resources/articles/4.1/prepared-statements.html.
12. Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. A static analysis framework for detecting SQL injection vulnerabilities. In *COMPSAC '07*, 2007.
13. William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks. In *FSE*, 2006.
14. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI*, 1988.
15. Michael Howard and David Leblanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, USA, 2001. Foreword By-Valentine, Brian.
16. Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *PLAS*, 2006.
17. Adam Kieżun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE*, 2009.
18. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), 1976.
19. Yuji Kosuga, Kenji Kono, Miyuki Hanaoka, Miho Hishiyama, and Yu Takahama. Sania: Syntactic and semantic analysis for automated testing against SQL injection. In *ACSAC*, 2007.
20. Anyi Liu, Yi Yuan, Duminda Wijesekera, and Angelos Stavrou. SQLProb: a proxy-based architecture towards preventing SQL injection attacks. In *SAC*, 2009.
21. V. Benjamin Livshits and Monica S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium*, 2005.
22. E. Kirda N. Jovanovic, C. Kruegel. Pixy: a static analysis tool for detecting web app vulnerabilities. In *IEEE Security and Privacy*, 2006.
23. OWASP. The ten most critical web application security vulnerabilities. http://www.owasp.org.
24. Tadeusz Pietraszek and Chris Vanden Berghe. Defending Against Injection Attacks through Context-Sensitive String Evaluation. In *RAID*, 2006.
25. Frank S. Rietta. Application layer intrusion detection for SQL injection. In *ACM-SE 44*, 2006.
26. R. Sekar. An efficient black-box technique for defeating web application attacks, ndss'09.
27. Zhendong Su and Gary Wassermann. The Essence of Command Injection Attacks in Web Applications. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2006.
28. Stephen Thomas, Laurie Williams, and Tao Xie. On automated prepared statement generation to remove SQL injection vulnerabilities. *IST*, 2009.
29. Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *SIGPLAN Not.*, 44(6):87–97, 2009.
30. Fredrik Valeur, Darren Mutz, and Giovanni Vigna. A Learning-Based Approach to the Detection of SQL Attacks. In *DIMVA*, 2005.
31. Gary Wassermann and Zhendong Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *PLDI*, 2007.
32. Yichen Xie and Alex Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security Symposium*, 2006.
33. Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *USENIX Security Symposium*, 2006.
34. Y. Minamide. Static approximation of dynamically generated Web pages. In *WWW '05*.