ABSTRACT

MARRI, MADHURI REDDY. PaRaM: Path-Sensitive Monitoring of Web Applications against SQL Injection Attacks. (Under the direction of Dr. Tao Xie.)

Web applications are ubiquitous and are accessed by a large number of users, making these applications susceptible to various types of attacks. Among these attacks, SQL Injection attacks belong to one of the most popular attack types, since a web application's vulnerability to SQL injection attacks poses serious threat to information security. The primary reason for these attacks is dynamic construction of SQL queries using string concatenation. In this thesis, we present a novel runtime-monitoring approach, called PaRaM, that guards a web application against SQL injection attacks. The key insight of our approach is that two executions that dictate the same program path result in SQL queries with the same structure, i.e., queries with the same sequence of SQL keywords. Therefore, the structure of the queries can be predetermined and mapped to the execution paths of the application. In general, runtime monitoring requires instrumentation of the web application and can cause high runtime overhead on the performance of the application. To address this challenge, PaRaM includes a minimization algorithm that reduces the number of program points to be monitored without sacrificing the effectiveness of monitoring against attacks. To evaluate the effectiveness of PaRaM, we apply PaRaM on five web applications. Our results show that our execution-path-sensitive approach is more effective than a related path-insensitive approach, and also causes low performance overhead.

PaRaM: Path-Sensitive Monitoring of Web Applications against SQL Injection Attacks

by
Madhuri Reddy Marri

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2010

APPROVED BY:

Dr. Laurie Williams	Dr. Ting Yu

Dr. Tao Xie
Chair of Advisory Committee

DEDICATION

To my parents, Papi Reddy Marri and Laxmi Marri.

BIOGRAPHY

Madhuri Marri was born in Hyderabad (known as the "City of Pearls"), Andhra Pradesh, India. She completed her bachelor's degree in Information Systems from Birla Institute of Technology & Science, Pilani, India in the year 2005. After graduating, she went on to work with Mercedes-Benz Research & Development India (formerly DaimlerChrysler Research and Development India) from June, 2005 to November, 2007. She then headed to North Carolina State University (NCSU) in January 2008 for Master of Science in Computer Science. She has been working with Dr. Tao Xie (thesis advisor) from August, 2008 broadly in the area of software testing and security. During her graduate school she spent one summer (2008) working with Dr. Giuliano Antoniol (Ecole Polytechnique de Montreal) on automatically detecting defects in C programs. In the summer 2009, she interned at ABB Corporate Research, under the supervision of Dr. Aldo Dagnino, working on software effort estimation. She will receive her Master of Science degree from NCSU in August, 2010.

ACKNOWLEDGEMENTS

First of all, I would like to thank my parents and sister for their encouragement and unwavering trust in my ability. I owe my deepest gratitude to my boy friend who has supported me in a number of ways during my graduate school. Nothing would have been possible without the love and support of these loved ones. I would like to thank my advisor Dr. Tao Xie for providing me the technical and monetary support to complete this work. During this stint of working with him, I learned many important things from him which helped me develop my technical skills. I would also like to thank Dr. Laurie Williams and Dr. Ting Yu for agreeing to serve on my committee and for their suggestions on my work. I would like to express my sincere gratitude to my colleagues at Automated Software Engineering research group without whose help, reviews, and insightful discussions this work would not have been possible. I would also like to thank my graduate school friends who believed in my strengths and helped me in many ways. Special thanks go to my wonderful friends and cousins for being there for me always.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Web applications are ubiquitous and are accessed by a large number of users. These applications often deal with sensitive data and any unauthorized access to the data in these applications can cause irrevocable damage to the data. Existing security technologies such as anti-virus applications and network firewalls offer comparatively secure protection at the host and network levels, but not at the application level [8]. Such technologies can be easily compromised with attacks in the form of malicious inputs sent by the attackers. These attacks may be hard to be detected at host or network levels and have to be handled at the application level. On the other hand, such attacks are more prominent as these malicious inputs can come from any online user. A popular type of such attacks are the *SQL Injection Attacks*, which are rated as one of the top-priority attacks that the Internet is facing [16].

SQL injection attacks (SQLIAs) can occur in web applications when parts of user inputs are unexpectedly treated as SQL code. Most web applications take user inputs (e.g., through web application forms) that are used for retrieving or storing information from or to the database, respectively. Therefore, when an application is vulnerable to SQLIAs, the vulnerability poses a serious threat to the information security. An attacker can submit SQL commands directly to the database (e.g., SQL commands to delete database records) or change the semantics of a SQL query to access unauthorized information (e.g., retrieve information without authentication).

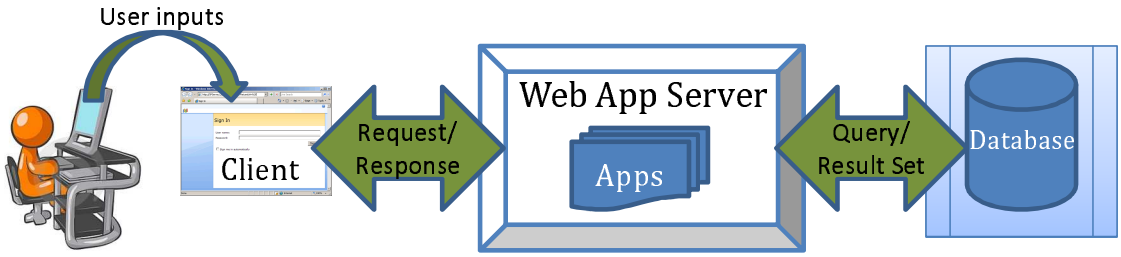

Figure 1.1: Typical flow of inputs in a web application.

Two recent discoveries of systems hacked via SQLIAs that caused 32.6 million passwords [21] and 130 million credit card numbers [20] to be exposed, highlight the threat caused by this type of attacks.

1.1 Research Challenges

The primary reason for SQLIAs is the common coding practice to dynamically construct SQL queries using user inputs. One way of ensuring security of a web application against SQLIAs is to adopt defensive coding mechanisms and carry out sufficient user input validation or *sanitization* [3]. However, in practice, programmers either do not adopt these coding mechanisms or do not produce defect-free code, resulting in vulnerable web applications that can be easily exploited by attackers. On the other hand, secure methodologies suggest the use of parameterized queries, which bind the variables to parameters and avoid the modification of the semantics of the queries by the user inputs. Nevertheless, despite a state-of-the-practice that can ensure security of the applications against SQLIAs, these practices impose major modifications on the legacy web applications. Therefore, SQLIA vulnerabilities are still found to be a common prevalence in many web applications [15]. Figure 1.2 shows the risk assessment of injection attacks, which include SQLIAs, and which top the list of the major application security issues.

To deal with the preceding issues, runtime monitoring of a web application's interactions with a database can be employed to protect the application at runtime, i.e., after the application is deployed. Nevertheless, there are two major challenges with employing a runtime monitoring

Figure 1.2: OWASP's risk assessment of injection attacks [15].

approach: (1) building an effective oracle such that the application can be monitored at runtime to filter out potential attacks and execute only genuine SQL queries, and (2) minimizing the effect of monitoring overhead on the performance of the application.

1.2 Contributions

To address the challenges presented in the preceding section, in this thesis, we present a novel *execution-path-sensitive runtime-monitoring* approach, called PaRaM. The key insight of our approach is that two executions that dictate the same program path result in SQL queries with the same structure, i.e., queries with the same sequence of SQL keywords. Therefore the structure of the query can be determined based on the execution path of the application. Based on this insight, we first dynamically build a model, instrument the application based on the built model, and then deploy the instrumented application to allow path-sensitive monitoring of the queries issued at runtime. The objective of the monitoring is to classify a query that does not belong to the model as a potential SQLIA and avoid the execution of such a query.

PaRaM uniquely blends a state-of-the-art input-generation technique, dynamic path tracing, and runtime monitoring of a web application to secure the application against SQLIAs. PaRaM monitors a PaRaM-instrumented application at runtime, based on a model built before the application is deployed. Furthermore, to minimize the runtime overhead due to monitoring, we design a minimization algorithm that reduces the amount of instrumentation by reducing the number of program points to be monitored and ensures low performance overhead without sacrificing the effectiveness of monitoring against SQLIAs.

In essence, PaRaM achieves runtime monitoring in three major steps: (1) applying an

automatic input-generator on an application to build a model for serving as an oracle to monitor the application at runtime, (2) instrumenting the application to monitor execution paths at runtime, (3) instrumenting the application such that at runtime every SQL query is checked against the model before being issued to the database.

This thesis makes the following major contributions:

- A novel approach called PaRaM that aims to achieve effective path-sensitive monitoring of web applications against SQLIAs. In addition to the path-sensitive approach, we also present the characteristics of query-issuing points when a path-sensitive approach will be more effective in preventing SQLIAs than a path-insensitive approach.

- A minimization algorithm that reduces the number of program points required to be instrumented to trace the execution path at runtime.

- An evaluation that measures (1) the effectiveness of PaRaM in guarding an application against SQLIAs when compared to a path-insensitive approach [6], and (2) the runtime overhead caused by PaRaM on the performance of the application. Our results show that PaRaM effectively prevents SQL injection attacks with low performance overhead, while the path-insensitive approach generates a total of 501 false negatives for the same inputs.

1.3 Thesis Structure

The rest of the thesis is organized as follows. Chapter 2 presents an example of a SQLIA and illustrates how an existing path-insensitive runtime-monitoring approach and the proposed path-sensitive approach deal with such an attack. Chapter 3 presents a formal definition of the problem of preventing SQLIAs. Chapter 4 presents PaRaM in detail and illustrates the minimization algorithm with an example. Chapter 5 presents the details of the prototype implementation of PaRaM. Chapter 6 presents the evaluation setup and discusses results of our evaluation. Chapter 7 discusses related work. Chapter 8 presents the analysis of PaRaM and discusses limitations of our approach. Finally, Chapter 9 concludes.

Chapter 2

Motivating Example

We next present an example SQLIA and illustrate how our path-sensitive runtime-monitoring approach can prevent such attacks. Figure 2.1 shows an example C# program that interacts with MySql database and the program includes a SQL issuing point (SQL-IP) in Line 16. Executing the program leads to two structurally different SQL queries to be issued at SQL-IP, i.e., two queries containing different SQL keywords and column names can reach the SQL-IP based on the value of `isLoggedIn` (Line 07). Basically, when the user's credentials are previously provided and the `isLoggedIn` flag is set to `true`, then the information is retrieved without using the `passcode`; otherwise, the information is retrieved using both the `username` and `passcode`.

An SQLIA is possible when the `isLoggedIn` value is `False`, the `username` value is `me' OR 'x'='x';--`, and `passcode` is `fdfs` (or any other value). The resulting SQL query (value of `query` when execution reaches Line 16) is `"SELECT * from EmployeeDB WHERE username = 'me' OR 'x'='x';--' AND passcode = 'fdfs'"`. In the preceding query, the `username` value creates a tautology (`'x'='x'`) and bypasses the authentication, since the MySql parser interprets "`--`" as comments and ignores the following text till the end of the line. Consequently, the execution of this query retrieves entire information from the `EmployeeDB` table without the required authentication, causing data leakage. The example shows a simple SQLIA; however, an attacker can launch a range of malicious attacks and achieve unintended data exposure, privilege escalation,

```
01:   bool isLoggedIn;
02:   void RetrieveInfo(string uname, string pcode)
03:   {
04:     if (!isValid(uname)) /**Branch B1**/
05:        ShowError("Invalid username");
06:     string query;
07:     if (!isLoggedIn) /**Branch B2**/
08:     {
09:        query = "SELECT * FROM EmployeeDB WHERE username = '" +
                        uname + "' AND passcode = '" + pcode + "'";
10:     }
11:     else
12:     {
13:        query = "SELECT * FROM EmployeeDB WHERE username = '" +
                                                      uname + "'";
14:     }
15:   ...
16:     sql.execute(query);
17:   ...
18:   }
```

Figure 2.1: Example program vulnerable to SQLIAs.

and even remote command execution [19]. We next illustrate how a state-of-the-art existing approach, Amnesia [6], prevents such attacks.

2.1 Existing Path-Insensitive Runtime Monitoring

To avoid such SQLIAs, Amnesia first statically extracts a model of the queries that reach every SQL-IP. Amnesia next instruments the application with a query checker at each SQL-IP to monitor whether a SQL query to be issued is an expected SQL query, i.e., if it belongs to the model. Figure 2.2 shows the transformed program. The instrumented condition check (that is path insensitive) shown in Line 16 in the transformed program checks whether the `query` structure is either similar to `s1` or `s2` and only if so, allows the query to be executed. However, such monitoring of the application does not ensure that the application is guarded against SQLIAs when the execution of different paths results in queries with different structures at the same SQL-IP.

Consider the program execution when the value of `isLoggedIn` is `False`, the value of `username`

```
01:   bool isLoggedIn;
02:   void RetrieveInfo(string uname, string pcode)
03:   {
04:     if (!isValid(uname)) /**Branch B1**/
05:        ShowError("Invalid username");
06:     string query;
07:     if (!isLoggedIn) /**Branch B2**/
08:     {//s1=SELECT * FROM EmployeeDB WHERE username = AND passcode =
09:        query = "SELECT * FROM EmployeeDB WHERE username = '" +
                       uname + "' AND passcode = '" + pcode + "'";
10:     }
11:     else
12:     {//s2=SELECT * FROM EmployeeDB WHERE username =
13:        query = "SELECT * FROM EmployeeDB WHERE username = '" +
                                                      uname + "'";
14:     }
15:     ... //model = {s1, s2}
16:     if (model.contains(getStructure(query))){
17:         sql.execute(query);
18:     }
19:     else {
20:        // throw Exception
21:     }
22:   ...
23:   }
```

Figure 2.2: The example transformed to guard against SQLIAs.

is `me'--`, and `passcode` is `fdfs`. The execution flows through the `True` branch of Line 07 and the resulting SQL query (the value of `query` in Line 16) is `"SELECT * from EmployeeDB WHERE username = 'me'-- AND pcode = 'fdfs'`, causing to bypass the use of `passcode` for retrieving the information. This SQLIA is allowed by the Amnesia-instrumented query checker at Line 16, since the resulting query matches the structure `s2`; however, it was supposed to match the structure `s1` based on the path executed by the input values. This example illustrates the ineffectiveness caused by using a path-insensitive model.

2.2 Proposed Path-Sensitive Runtime-Monitoring

To address the preceding issues, PaRaM applies *path-sensitive* runtime-monitoring of the application. In Figure 2.1, when the *control decisions*[1] for branch `B1` and for branch `B2` are `False`,

[1]We refer to a decision of a branching condition as referred to as *control decision*.

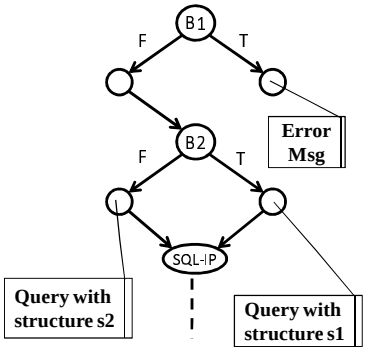

Figure 2.3: Control-flow graph of program in Figure 2.1.

B1_False ∧ B2_True ⇒ "SELECT * FROM EMPLOYEEDB USERNAME = AND PASSCODE ="
B1_False ∧ B2_False ⇒ "SELECT * FROM EMPLOYEEDB USERNAME ="

Figure 2.4: Example model built by PaRaM before the application is deployed.

the query that reaches Line 16 should have the structure `s1`. On the other hand, when the control decision for Branch `B1` is `False` and for Branch `B2` is `True`, then the query should have the structure `s2` ((illustrated in Figure 2.3, which shows a control-flow graph of the example shown in Figure 2.1)). Therefore, PaRaM builds a model that stores all execution paths (in terms of sequence of control decisions) and their corresponding queries that reach a SQL-IP. Figure 2.4 shows the constructed model. We then transform the program in a way such that, at runtime, the application's execution path is monitored (i.e., the executed control decisions are tracked) and the generated query is checked against the model before the query is issued to the database.

Although PaRaM's path-sensitive approach is more effective than path-insensitive approaches in guarding against SQLIAs, PaRaM can cause more runtime overhead due to the need to monitor control decisions for each execution. To deal with this issue, we develop a novel minimization algorithm that aims to reduce the number of control decisions to be monitored to a very essential number of control decisions, and consequently assist in reducing the performance overhead of our approach. Thus, our proposed approach achieves high effectiveness in preventing SQLIAs at runtime while inducing low performance overhead.

Chapter 3

Problem Definition

We next provide a formal description[1] that leads to our key idea in preventing SQLIAs using a *path-sensitive* approach.

We denote a path as a sequence of executed control decisions. That is, given that D is the set of all control decisions in an application, then a path p is given as a sequence of a subset of control decisions[2] as:

$$p = d_1 + d_2 + ... + d_m, \ where \ d_j \in D \ and \ \ 1 \leq j \leq m$$

Consider that a SQL query $q \in Q$ is a sequence of (1) SQL keywords (denoted as K) including the operators used in the predicates, (2) names of database tables and columns (denoted as T), and (3) literals (denoted as L), such as string or numeric values. Consider that a web application W accepts a set of inputs $I \in \Sigma^*$ (the set of all strings that can be formed from Σ, which is a set of symbols) and interacts with a database through a set of SQL queries Q at different *critical points*.

Definition 3.1 Critical Point. A critical point c is a program point (in a web application W), where W issues a SQL query q.

[1] Definitions 3.4 and 3.5 are inspired by the formalization provided by Su and Wasserman [18].

[2] '+' is used to denote that the elements are in a sequence.

Therefore, an application that interacts with a database contains a set of critical points C. For the example in Figure 2.1, the SQL-IP in Line 16 is a critical point.

Thus, an execution of the application W issues a query $q \in Q$ to the database, when a set of inputs $i \in I$ dictate a path $p \in P$ such that the execution reaches a critical point $c \in C$. Each query q can be characterized by its *SQL signature* and the issued query is successfully executed on the database only when the query is *valid*.

Definition 3.2 SQL Signature. The SQL signature of a query q is the syntactic structure of the query, given as:

$$\hat{q} = (a_1, a_2, ..., a_m),\ where\ a_j \in K \cup T\ and\ 1 \leq j \leq m$$

For example, the SQL signature for a query `SELECT age FROM t.emp_dir WHERE name = 'ABC'` is given as `(SELECT, age, FROM, t.emp_dir, WHERE, name, =)`.

Definition 3.3 Valid Query. A valid query is a query with a valid syntax according to the SQL grammar[3], i.e., the application's database can parse a valid query without any errors. A query's SQL signature is valid only when the query is valid.

Definition 3.4 Syntactically Same Queries. Two queries are syntactically same (denoted as $\equiv$) when both queries have the same SQL signature, i.e., $q \equiv q^{'}$ iff $\hat{q} == \hat{q^{'}}$.

Definition 3.5 SQL Injection Attack (SQLIA). Given K, a SQLIA is successfully launched on an application W, when one or more inputs $i \in I$ transform to $l \in L$ in a $q \in Q$, such that the introduced l can cause one or more of $k \in K$ or $t \in T$ in q to change to k_α or t_α, respectively, where $k_\alpha == \phi \ ||\ k_\alpha \in K$ and $t_\alpha == \phi \ ||\ t_\alpha \in T$, resulting in a query q_α that satisfies the following two conditions:

1. q_α is a *valid* query.

2. q_α is syntactically different from q, i.e., $q_\alpha \not\equiv q$.

[3] `http://docs.openlinksw.com/virtuoso/GRAMMAR.html`

The inputs $i \in I$ that can cause such change of q to q_α are called SQLIA-causing inputs. When an execution of the application W with a set of inputs I follows a program path p, and results in a query q at a critical point c, then let the execution be denoted as $\langle I, p, q\rangle \rightarrow c \xrightarrow{q} database$, indicating that the query is successfully issued to the database. Using this notation, we next formally define our key idea of preventing SQLIAs.

Preventing SQLIAs. If the application's execution with a set of benign inputs I_1 (that do not cause an attack) is intended to issue a query q_1, i.e., if $\langle I_1, p_1, q_1\rangle \rightarrow c_1 \xrightarrow{q_1} database$ is intended, and if at runtime any execution of the application with inputs I_2 follows the same path p_1, and reaches the same critical point c_1, but the execution results in a different query q_α that satisfies the preceding two conditions, then inputs I_2 should be identified as SQLIA-causing inputs and the query q_α should not be issued to the database, i.e., $\forall q_\alpha \not\equiv q_1$, $\langle I_2, p_1, q_\alpha\rangle \rightarrow c_1 \not\xrightarrow{q_\alpha} database$, indicating the query is not issued to the database.

Chapter 4

Approach

To effectively guard web applications against SQLIAs, we propose PaRaM, a path-sensitive runtime-monitoring approach. The objective of PaRaM is to guard a web application to reject those user inputs that modify the intended SQL queries issued by the application. Figure 4 gives an overview of PaRaM, which includes three distinct phases: (1) the learning phase learns a guarding-model set that describes the expected behavior of the web application in terms of the SQL queries that can be issued by the web application, (2) the minimization phase minimizes the guarding-model set, and (3) the instrumentation phase instruments the application based on the minimized guarding-model set to guard the web application. The key insight for our approach is that a guarding-model set (with all the query signatures expected at each critical point) can be extracted before deploying the application and that the extracted model set can be used to protect the application at runtime. Based on this insight, we adopt dynamic analysis to generate a set of inputs and execute the application to learn a model set that includes all the possible unique sets of critical points, execution paths, and SQL signatures. We then use the model set to instrument the control decisions given by the collected paths in the model set and instrument all critical points to check queries against the model set. Nevertheless, monitoring every collected control decision could be expensive; therefore, as an intermediate step, before instrumenting the application, we apply a minimization algorithm that reduces the

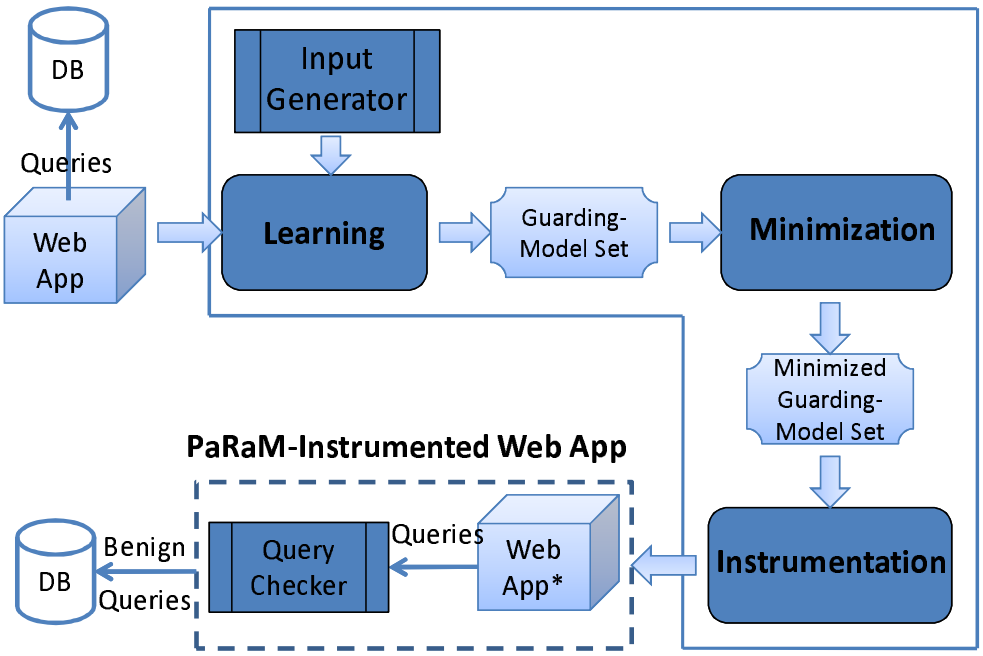

Figure 4.1: Overview of PaRaM.

control decisions to a small number, such that instrumenting these small number of control decisions is sufficient for path-sensitive monitoring of the application at runtime. Thus, this less-instrumented web application can be deployed being guarded against SQLIAs. We next present the characteristics of critical points where a path-sensitive approach is needed and then provide details of the different phases in our approach.

4.1 Characteristics of Critical Points for Path-Sensitive Monitoring

In general, a path-sensitive monitoring approach is more effective in preventing SQLIAs when compared to a path-insensitive approach when a web application issues multiple SQL queries from a critical point such that the structure of the query being issued is path-sensitive. Based on this insight, we present two characteristics of a critical point, where PaRaM should be used instead of a path-insensitive approach in preventing SQLIAs:

1. At the critical point, at least two SQL queries with different signatures can be issued to the database.

2. At the critical point, at least one of the SQL queries can be transformed by the user inputs into a query with a different SQL signature, such that both the SQL signatures are expected at the critical point (via different paths).

For user inputs to successfully transform a SQL query, say q_1, of one signature to SQL query, say q_2, with a different signature, both the query signatures should have common prefix up to the first location of a *literal* (refer to Chapter 3) in one of the queries, i.e., at the least the SQL queries should be performing the same SQL action, e.g., `SELECT` action. Figure 2.1 shows a critical point (Line 16) with these two characteristics. On the other hand, if the `SELECT` queries being issued at the critical point select different columns (such as q_1: `SELECT *..` and q_2: `SELECT count(*)..`), then q_1 cannot be transformed to q_2 by any user input, thereby not satisfying the second characteristic and therefore, a path-insensitive approach would be equally effective for the later case.

We next explain the three phases of PaRaM in detail.

4.2 Phase 1. Learning a Guarding-Model Set

The learning phase generates inputs and executes the web application using the generated inputs to learn the possible SQL queries that the application executes. Furthermore, to achieve a path-sensitive monitoring, it is also necessary to learn the path information of the executions, along with the SQL query(s) resulting from each execution.

Input Generation. To generate inputs for a given web application, we adopt Dynamic Symbolic Execution (DSE) [10, 4, 17] that generates inputs to achieve high structural coverage of a given application. In particular, given an application, DSE explores the application with random or default values and collects constraints along the execution path. DSE next systematically negates parts of the collected constraints and solves the resulting constraints to

generate concrete values that guide application execution through alternate paths. Thus, DSE generates concrete values to execute distinct paths in the application. The rationale behind using DSE for our input generation is that DSE aims to generate inputs that can cover all the feasible paths (that can reach all the critical points) in a given application. Consequently, we execute the inputs generated by DSE to learn the SQL queries that the web application can issue to a database.

Model Construction. While executing the web application with the generated test inputs, we collect the execution information required to build an information pool for assisting in path-sensitive monitoring of the application.

PATH-SENSITIVE MONITORING. When the execution of a particular path p (a sequence of control decisions) issues a query q at a critical point c, then path-sensitive monitoring expects that every other execution of the application that follows the same sequence of control decisions as in p should issue a syntactically same query as q at the same critical point c.

Therefore, to construct the information pool, we collect the following information for every critical point[1] $c \in C$, (1) *Path Signature* representing the execution path information starting from the application's entry point[2] to c, (2) SQL Signature[3] $\hat{q}$, representing the SQL query issued at c during the execution.

Definition 4.1 Path Signature. Path signature $\hat{p}$ for a path p reaching a critical point c is the sequence of control decisions before reaching c and is given by $\hat{p} = (d_1, d_2, ..., d_m)$ where $\forall 1 \leq j \leq m$, d_j represents a control decision that is executed when following p to reach c. Therefore, d_m represents the last control decision before the execution reaches c and d_1 is the first control decision after the execution starts.

For a given input application W, we accumulate the preceding information for all executions and build a *CP-Model* for each critical point.

[1]There could be more than one critical point in an execution, and the subsequent set of information is collected for each critical point during an execution.

[2]We denote a program point (in the application) that is triggered by a user action as an entry point.

[3]Recall that different executions reaching the same critical point can generate structurally different SQL queries (as shown in our example in Figure 2.1).

Definition 4.2 CP-Model. A CP-model $m_c \in M$ for a critical point $c \in C$ is the mapping of SQL signatures of the queries issued at c to the corresponding path signatures of the paths that reach c. Assuming that $\hat{Q}_c$ is the set of the SQL signatures of queries issued at c, and $\hat{P}_c$ is the set of n path signatures of the paths that reach c, then the corresponding CP-model m_c is given by,

$$m_c = \left\{ \langle \hat{q}, \{\hat{p}_1, \hat{p}_2, ..., \hat{p}_l\} \rangle \mid \begin{array}{l} \hat{q} \in \hat{Q}_c, \\ \hat{p}_j \in \hat{P}_c, 1 \leq j \leq l \text{ and } l \leq n \end{array} \right\}$$

The CP-models for all the critical points C for the given W are denoted together as *Guarding-Model Set* G, given by

$$G = \{\langle c, m_c \rangle \mid c \in C \text{ and } m_c \in M\}$$

Thus, for each execution of the application with the generated inputs, PaRaM's learning phase collects path and query signatures to build the model set.

Figure 4.5 shows an example application with its control-flow graph (Figure 4.2) and its guarding-model set G resulting from our learning phase (Figure 4.3). The edges $d_1, d_2, ...d_{14}$ in the control-flow graph are the control decisions of the application. The three dark nodes in the graph represent 3 critical points; $m1$, $m2$, and $m3$ are the CP-models for critical points $c1$, $c2$, and $c3$, respectively.

4.3 Phase 2. Model Set Minimization

Our minimization phase first reduces the collected control decisions (in path signatures of the model set) to a small number of *essential control decisions*, such that instrumenting the smaller number of control decisions would be sufficient for our path-sensitive monitoring, without sacrificing the effectiveness of preventing SQLIAs. First, we minimize each path signature in the guarding-model set such that the path signature includes only a set of essential control decisions. Consider D, the set of all control decisions of an application W, assume a CP-model $m_c = \{\langle \hat{q}_1, \{\hat{p}_1\} \rangle, \{\langle \hat{q}_2, \{\hat{p}_2\} \rangle\}$. The crux of our algorithm is that if removing a control decision

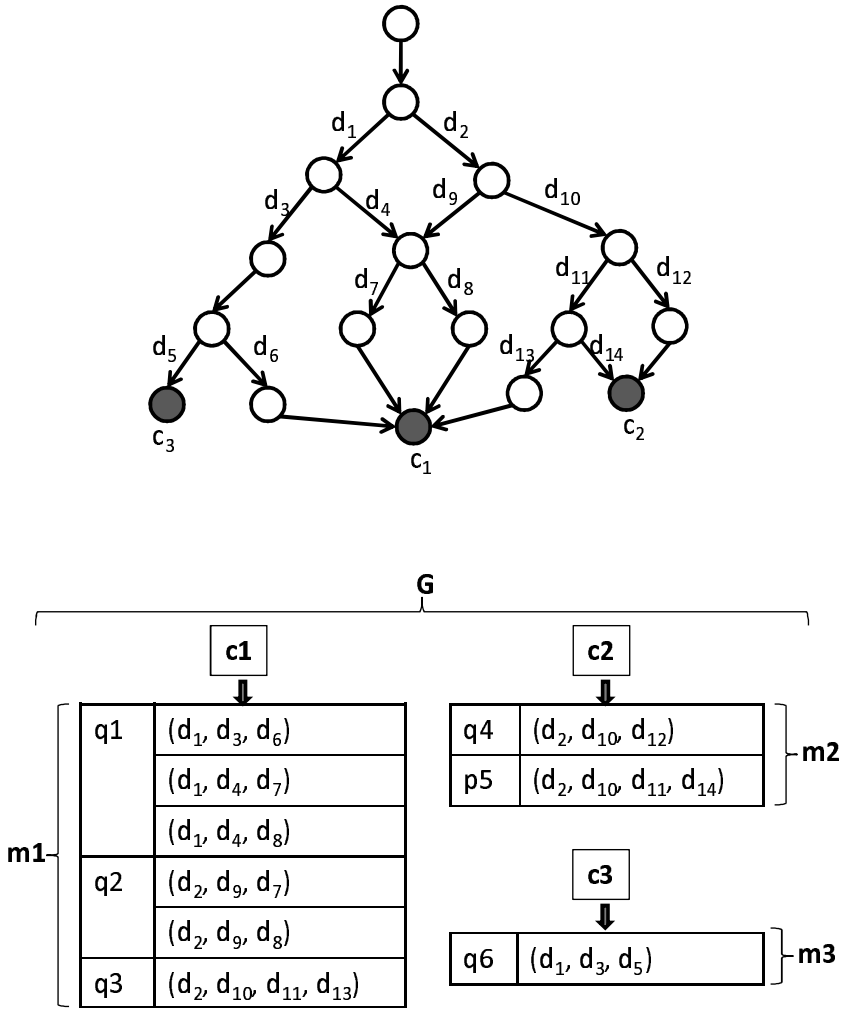

Figure 4.3: Guarding-model set G resulting from the learning phase.

$d \in D$ from $\hat{p}_1$ or $\hat{p}_2$ results in a CP-model such that $\hat{p}_1 = \hat{p}_2$, i.e., the CP-model does not uniquely bind the query signature to the path signature and therefore loses its capability to prevent SQLIAs. Such a d is an essential control decision and needs to be monitored.

Definition 4.3 Essential Control Decision. An essential control decision, δ, is a control decision (in the application W) that is essential to be monitored to preserve the model set's capability during path-sensitive monitoring of W against SQLIAs.

We denote the set of all essential control decisions of a given application W as $\mathcal{D}$. Our minimization algorithm uses three techniques to identify $\mathcal{D}$ for a given model set G, and then transforms each path signature in G to a sequence of $\delta \in \mathcal{D}$.

MINIMIZATION PROBLEM: Given a guarding-model set $G = \{\langle c, m_c \rangle\}$, minimization is to extract the set of essential control decisions $\mathcal{D}$ and minimizes G to $G' = \{\langle c, m'_c \rangle\}$ where m'_c can be represented as

$$m'_c = \{\langle \hat{q}, \{\hat{\rho}_1, \hat{\rho}_2, ..., \hat{\rho}_k\} \rangle\}$$

where $\forall 1 \leq j \leq k$, $\hat{\rho} = (\delta_1, \delta_2, ..., \delta_t)$ and $\forall 1 \leq z \leq t,\ \delta_z \in \mathcal{D}$

we next present details of three major steps in our minimization algorithm (shown in Algorithm 1). The first two steps of the algorithm focus on reducing the input model set G by eliminating the control decisions that need not be monitored, and the last step extracts $\mathcal{D}$ from the reduced G and reconstructs the path signatures of the model set G in terms of $\mathcal{D}$.

4.3.1 Within-SQL Minimization

In the first step, *within-SQL minimization* (Lines 5-10 in Algorithm 1), we deal with each set of path signatures corresponding to a single SQL signature in each CP-model individually. We reduce these path signatures based on the following rationale: *If a set of paths reach a particular critical point resulting in queries with the same SQL signature, then the common control decisions (if exist) in these paths are the essential control decisions that are instrumental in the SQL query structure.* Therefore, our objective of applying within-SQL minimization for each SQL signature is to reduce the set of path signatures (which are sequences of control decisions) to a set of the sequence of *common control decisions.* We denote a control decision as a common control decision if the control decision appears in all path signature in the given set of path signatures. Therefore, the objective of our within-SQL minimization step is to transform each CP-model m_c and can be represented as follows:

$$\langle \hat{q}, \{p_1, p_2, ..., p_l\} \rangle \Rightarrow \langle \hat{q}, \{p'_1, p'_2, ..., p'_k\} \rangle$$

Algorithm 1: Minimization Algorithm.

input : Guarding-model set $G = \{< c, m_c >\}$
output: Minimized guarding-model set G'

1 **foreach** *critical point* $c \in C$ **do**
2 $m \leftarrow G[c]$;
3 $\hat{Q}_c \leftarrow m.QuerySignatures$;
4 **if** $count(\hat{Q}_c) > 1$ **then**
  `/* Within-SQL Minimization */`
5 **foreach** $\hat{q} \in \hat{Q}_c$ **do**
6 $\hat{P}_c \leftarrow m.paths[\hat{q}]$;
7 **if** $count(\hat{P}_c) > 1$ **then**
  `/* CD - Common Decisions */`
8 $\hat{P'}_c \leftarrow ReduceToCommonCDs(\hat{P}_c)$;
9 $m'.add(\hat{q}, \hat{P'_c})$;
10 **else** $m'.add(\hat{q}, \hat{P})$;
  `/* CP-Model Minimization */`
11 $toMinimize = true$;
12 $m'' \leftarrow RemoveCommonCDs(m'.paths())$;
  `/*` $toMinimize$ `is set` $false$ `if CP-model min. is not applicable */`
13 **if** $!toMinimize$ **then** $G[c] = m'$;
14 **else** $m' = m''$;
15 **else** $m'.Add(\hat{q} \in \hat{Q}_c, \phi)$;
16 $G'[c] = m'$;
  `/* Global Reconstruction */`
17 Collect critical CDs $\mathcal{D}$;
18 Build final path signatures $\mathcal{P}$ using $\hat{p} \cap \mathcal{D}$;
19 $G' = G.replace(P, \mathcal{P})$

where $\forall\ 1 \leq j \leq k, 1 \leq i \leq l, p'_j = \bigcap\{p_1, p_2, .., p_i\}$ and $k \leq l$. In essence, for each SQL signature, we carry out the within-SQL minimization by recursively finding the common control decisions[4] for the corresponding path signatures. Let c be a critical point whose corresponding CP-model m_c is undergoing the within-SQL minimization. We then replace the path signatures with the sequence of common control decisions by eliminating uncommon control decisions recursively. Line 8 in Algorithm 1 carries out this step of minimization for each SQL signature at each

[4]The problem of finding common control decisions is similar to finding common subsequence in a given set of strings.

critical point. As a result, each CP-model can be reduced in two dimensions, the length of a path signatures, and the size of the set of path signatures.

In the example shown in Figure 4.5, when within-SQL minimization is applied on the CP-model $m1$ for SQL signature $q1$, the first invocation of *ReduceToCommonCDs* results in minimized path signatures (d_1, d_4) and (d_1, d_3, d_6). Further invocation of *ReduceToCommonCDs* results in the path signature (d_1) indicating that d_1 is a potential δ. Figure 4.5(a) shows the reduced guarding-model set after within-SQL minimization.

4.3.2 CP-Model Minimization

The second step (operating on the model set resulting from the within-SQL minimization) of our minimization (Lines 11-14 in Algorithm 1), called *CP-Model minimization*, reduces all the path signatures at each critical point, i.e., this step operates on each CP-model in the model set at a time. The objective of this step is to reduce the path signatures at each critical point based on the following rationale: *For the paths that reach the same critical point, but do not result in queries with the same SQL signature, the common control decisions across these paths are not required to be monitored as they are not instrumental in the differences in the query structure.* The objective of our CP-model minimization step can be represented as follows:

$$\{\langle \hat{q}, \{\hat{p}'_1, \hat{p}'_2, ..., \hat{p}'_l\}\rangle\} \Rightarrow \{\langle \hat{q}, \{\hat{p}''_1, \hat{p}''_2, ..., \hat{p}''_l\}\rangle\}$$

where, $\forall 1 \leq j \leq l$, $\hat{p}''_j \subseteq \hat{p}'_j$, and $\forall\ 1 \leq r \leq l$, $j \neq r$, $\hat{p}''_j \cap \hat{p}''_r = \phi$.

In essence, this step finds and removes any common control decisions between every path corresponding to a SQL query and every path corresponding to other SQL queries in the CP-model. Unlike the within-SQL minimization, the CP-model minimization should reduce only the length of the path signatures at a critical point but not the size of the set of the path signatures. The CP-model minimization step is not always completely applicable and can result in an unsound[5] guarding-model set. To address this issue, our algorithm includes two

[5]The algorithm results in an unsound guarding-model set only when the minimization eliminates a control

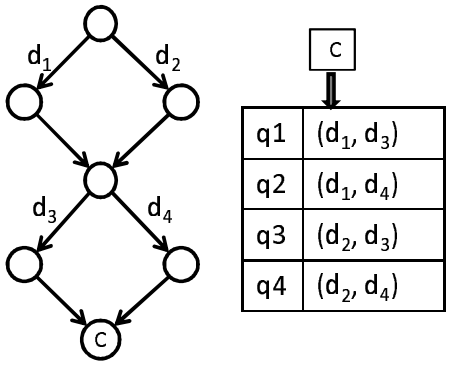

Figure 4.4: Example where CP-model minimization is not applicable.

particular validations in the CP-model minimization step:

- In a situation when two path signatures (p'_1 and p'_2) are minimized, and one path signature is a complete subsequence of another path signature ($p'_1 \subset p'_2$), the result of eliminating the common control decisions could lead to an empty path signature ($|p''_1| = 0$), and thus can result in an unsound model set. Therefore, for each iteration of the CP-model minimization (for each CP-model), we check for such a case, and minimize only the path signature that is super-sequence among the two, i.e., only minimize p'_2 in the example[6].

- When the CP-model minimization step affects the size of the set of path signatures, this step is not applicable for minimizing the CP-model and the minimization for that particular CP-model is terminated with the within-SQL minimization; Line 13 in Algorithm 1 checks for this condition where $toMinimize$ is expected to be set to $True$ when CP-model minimization is applicable. An example of the CP-model where CP-model minimization cannot be applied is shown in Figure 4.3.2 where the critical point is associated to four different path signatures that result in four different SQL signatures, respectively.

decision that is essential for path-sensitive-monitoring of the application.

[6]Note that a case of $p'_1 == p'_2$ will never arise.

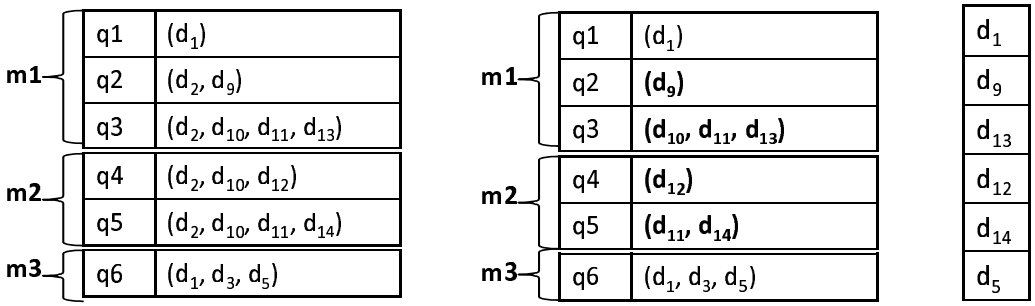

Figure 4.5: (a) The example application's G after within-SQL minimization. (b) G after CP-model minimization. (c) $\mathcal{D}$, the set of essential control decisions.

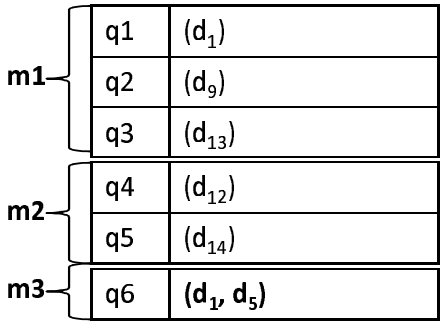

Figure 4.6: G after global reconstruction, $G^{'}$.

4.3.3 Global Reconstruction

In the step of our global reconstruction (Lines 17-19 that operate on the model set resulting from the CP-model minimization step), we form the set of essential control decisions $\mathcal{D}$ and reconstruct all the path signatures to include control decisions $\delta \in \mathcal{D}$.

The objective of our global reconstruction is to transform the path signatures in G to include only essential control decisions. The transformation of G can be represented as follows:

$$\{c, \{\langle \hat{q}, \{\hat{p}_1, \hat{p}_2, ..., \hat{p}_l\}\rangle\}\} \stackrel{\mathcal{D}}{\Rightarrow} \{c, \{\langle \hat{q}, \{\hat{\rho}_1, \hat{\rho}_2, ..., \hat{\rho}_l\}\rangle\}\}$$

where, $\forall\ 1 \leq j \leq l$, $|\hat{\rho}_j| = \phi \ ||\ |\hat{\rho}_j| << |\hat{p}_j|$.

First, we form $\mathcal{D}$, the set of the essential control decisions constituting of (1) the last

control decision of each path signature from the CP-models for which CP-model minimization was successful, and (2) all the control decisions of all path signatures from the CP-models for which CP-model minimization was unsuccessful. From the minimized path signatures, these selected control decisions are sufficient to preserve the soundness of the model set, and therefore form the set of essential control decisions. Figure 4.5(c) shows the set $\mathcal{D}$ for our example.

We then construct the final path signatures $\mathcal{P}$, where each $\hat{\rho}$ includes only one or more of $\delta \in \mathcal{D}$. We construct $\hat{\rho}$ for a path signature $\hat{p}$, by including only the control decisions in $\hat{p}$ that exist in $\mathcal{D}$, i.e., we construct a final path signature $\hat{\rho} = \hat{p} \cap \mathcal{D}$. Consequently, through this reconstruction of the path signatures, we build a minimized model set, where the path signatures consist of a smaller number of control decisions compared to the original guarding-model set. Figure 4.6 shows the model set G' after our global reconstruction. It can be observed that the total number of control decisions in the model set decreased from 14 to 6, showing the significance of our minimization algorithm.

Thus, through our minimization algorithm, we find a small number of control decisions that are essential for our path-sensitive monitoring and minimize path signatures to include only these essential control decisions. Although reducing the number of control decisions to be monitored to as less as possible is the goal of our minimization algorithm, this reduction is not always guaranteed. For example, when the minimization terminates in the CP-model minimization step for a particular CP-model, all the control decisions in that CP-model are included in our essential control decisions set and are required to be monitored. We further discuss the performance gain in terms of the number of the reduced control decisions by our algorithm in Chapter 8.

4.4 Phase 3. Application Instrumentation

The instrumentation phase transforms the application under analysis to a PaRaM-instrumented application such that every query that is issued by the application to the database is checked to be *benign* before the query is issued to the database.

Definition 4.4 Benign query. A SQL query generated at runtime is benign, when the query's signature matches with the expected signature as given by the guarding-model set $G^{'}$.

If $\hat{p}$ is the path signature of the path p that reaches a critical point c, and $\hat{q}$ is the signature of the generated query, then $\hat{q}$ is benign if and only if $\hat{p} \in m'_c[\hat{q}]$, i.e., $\hat{p}$ is one of the path signatures associated with $\hat{q}$ at c (recall that m'_c is a CP-model in G').

To check whether every query is benign before the query is issued to the database, in this phase, we (1) instrument the control decisions given by the guarding-model set G' so that an execution's path signature[7] is monitored at runtime, and (2) instrument the critical points in the application to pass the query to a *query checker* that checks whether the generated query is benign. The query checker checks whether the generated query at each critical point is a benign query based on the execution's path signature obtained using the instrumented control decisions. The instrumentation of essential control decisions and the critical points enables us to carry out path-sensitive monitoring of the application against SQLIAs at runtime. For our example shown in Figure 4.5, we instrument the essential control decisions $d_1, d_9, d_{13}, d_{12}, d_{14}, d_5$ to be monitored, and instrument the three critical points with invocations to the query checker.

As a result of this phase, only benign queries that cannot launch any SQLIAs are issued to the database at runtime. Therefore, the PaRaM-instrumented application is effectively guarded against SQLIAs.

[7]Recall that G' consists of path signatures that include only the essential control decisions.

Chapter 5

Implementation

We implemented a prototype of PaRaM for the purpose of evaluation. In our prototype, we use an existing off-the-shelf DSE-based [10, 4, 17] test-generation tool called Microsoft Pex [23] to generate inputs for input-generation part of the learning phase. For the purpose of learning a model set using the generated inputs, we implemented a tracing module using ExtendedReflection (ER) [14], a dynamic analysis framework for .NET languages that instruments the intermediate-language code and provides callbacks for various program events, such as a method entry and exit. We built our tracing module using the ER framework to trace the executions and record signatures of paths that reach an application's critical points, starting from a given set of entry points (refer to the definition of *path signature* in Chapter 4). To extract the SQL signature from each SQL, we extended an open-source SQL parser[1] for our prototype. Since parsing of SQL queries is time-consuming with the `XmlDocument` class used by the existing implementation, we use our own data structures to optimize the parsing. Furthermore, the existing parser can only parse simple `SELECT` queries and we extended the existing parser to be able to parse complex queries. We also implemented a minimization module based on the minimization algorithm presented in Section 4.3.

[1] `http://www.codeproject.com/KB/dotnet/SQL_parser.aspx`

Chapter 6

Evaluation

PaRaM aims to achieve effective monitoring of a given web application against SQLIAs with a small overhead on the performance of the web application. To show the effectiveness of PaRaM, we applied PaRaM on five open-source web applications. In our evaluation, we address the following research questions:

- RQ1: How effective is PaRaM in dealing with SQLIAs? In particular, how effective is PaRaM's path-sensitive monitoring when compared to a path-insensitive approach [6]?
- RQ2: How many control decisions (to be monitored) are reduced by our minimization algorithm and how much runtime overhead is caused by PaRaM on the web application over a path-insensitive approach? The first part of this research questions helps to show the performance gain achieved by using our minimization algorithm, since a low number of control decisions results in low overhead caused by the instrumentation. The second part of the this research questions helps to evaluate the amount of overhead caused by PaRaM's additional monitoring of control decisions.

Metrics. To evaluate the effectiveness of our approach, we measure the number of attacks that the five subject applications were successfully guarded against, when the PaRaM-instrumented applications were executed with a set of *evaluation inputs*. This set of evaluation inputs used

in our study are an augmented version of the set of inputs obtained from the authors of [5, 6] and include both attack-causing and benign inputs. We augmented the original set of inputs to increase the number of SQL issuing points that these inputs can reach. To compare the effectiveness of our approach with existing path-insensitive approaches such as Amnesia [6], we execute the subject applications with the same set of inputs and use the same guarding-model sets excluding the mapping to the execution paths, i.e., the model set used for path-insensitive approach contains only SQL signatures associated with the critical points and an attack is reported at the critical point only when the generated query's signature does not match any query signature in the CP-model. We compare the number of *false positives* (the number of benign inputs that are classified by the approaches as attack-causing inputs) and the number of *false negatives* (the number of actual attack-causing inputs that are not classified as attack-causing inputs) caused by the two approaches.

To measure the performance overhead caused by our monitoring, we execute the five subject applications with a set of benign inputs with and without PaRaM-instrumentation and compare the *execution time* values. We also measure the *number of control decisions* that are included in the guarding-model set before and after applying our minimization algorithm.

6.1 Subject Applications

In our evaluation, we used five subjects that were obtained from GotoCode[1]. These five web applications are available in different languages and have been previously used as subjects in related work [2, 5, 6, 11, 19]. We used the C# versions of the applications in our evaluation, since we use Pex [23], a .NET test generation tool for our learning phase. The primary objective of our evaluation is to show the effectiveness of our approach when execution paths dictate the structure of a SQL query being issued, i.e., the subject applications have the characteristics discussed in Section 4.1. Such characteristics are often found in real applications such

[1] `http://www.gotocode.com/`

as Bugzilla[2], and Basic Analysis & Security Engine[3] (PHP applications). For example, the `Bug.Search` function in Bugzilla (versions 3.3.2 to 3.4.1) has a critical point with the characteristics given in Chapter 4. The function issues `SELECT` queries on the same table, but with different `WHERE` clauses, depending on user inputs that dictate different execution paths[4]. In our evaluation, since we use the C# versions of the benchmark applications, we refactored these five applications and created SQL issuing points, such that more than one similar SQL queries are issued at the same critical point. In particular, we refactored the applications such that queries that perform the same action (e.g., `SELECT`) on the same table are issued from the same critical point. Our refactoring preserves the semantics of these applications, while changing the applications such that the structure of the SQLs issued from the same issuing point is path-sensitive.

Table 6.1 gives the details of the subject applications. Column "LOC" gives the lines of code of each application measures using CLOC[5], an open-source tool that can be used to measure code metrics for many programming languages. Column "Critical Points" gives the number of critical points in the application *before* and *after* our refactoring (given by the two subcolumns). Column "# Handlers" gives the number of *handlers* in the applications. Handlers are methods that are directly invoked as a result of a user action. For example, in the Employee Directory application, when a user clicks on the Login button, the method `Login_login_click` is invoked. Similarly, when a user enters a URL to a page, the corresponding `Page_Load` method is invoked.

6.2 Evaluation Setup

The inputs used to evaluate PaRaM, referred to as evaluation inputs, are a modified version of the inputs obtained from the authors of [6, 5]. From the obtained inputs, we removed any invalid inputs, i.e., those inputs that either do not conform to the expected syntax of the application

[2]http://www.bugzilla.org/

[3]http://base.secureideas.net/

[4]A SQLIA was actually reported in September, 2009 in the `Bug.Search` function. Refer to `http://www.bugzilla.org/security/3.0.8/`.

[5]http://cloc.sourceforge.net/

Table 6.1: Details of the Subject Applications.

Subject	Description	LOC		Critical Points		# Handlers
		C#	Other	Before Refac.	After Refac.	
Classfieds	Online management system for classifieds	4572	3139	43	40	10
BookStore	Online bookstore	7182	5047	75	72	10
Employee Directory	Online employee directory	2585	2263	23	21	8
Events	Event tracking system	3123	2005	31	30	7
Portal	Portal for SVEC club	6927	7260	67	65	4

and are rejected by the application, or inputs that are as expected by the application, but generate a syntactically wrong query (i.e., SQL parser throws a parse exception). The rationale behind excluding these inputs is that these do not contribute to measuring the effectiveness of the approach, since the inputs either do not reach the critical points or those that reach will not be executed by the database since they are syntactically-invalid queries. We further augment the remaining inputs to include more critical points and more attack patterns. For example, none of the existing inputs use a combination of in-line comments `/* */` (with respect to MySql) that can be carefully crafted to be split between two input values. In summary, the evaluation inputs are of two types: *benign* and *attack* inputs. Benign inputs are those inputs that conform to the expected syntax of the application, and that lead to a syntactically correct and expected query, i.e., a query that does not cause any attack. Attack inputs are those inputs that conform to the expected input syntax of the application and lead to a syntactically correct query, but is not intended to be issued by the application. We manually analyzed these inputs to ensure their category and used external parsers[6] to verify the syntactic-correctness of the queries. Table 6.2 summarizes the evaluation inputs used in our evaluation. Column "#Total"

[6]http://www.dpriver.com/pp/sqlformat.htm

Table 6.2: Details of the Evaluation Inputs.

Subject	Evaluation Inputs		
	# Total	# Benign	# Attacks
Classifieds	5926	2140	3786
Bookstore	5535	2398	3137
EmployeeDirectory	6966	2944	4022
Events	9611	3295	6316
Portal	3945	966	2979
Total	**31983**	**11743**	**20240**

gives the total number of evaluation inputs used for the corresponding subject, and Columns "#Benign" and "#Attack" gives the numbers of benign and attack inputs among the total inputs, respectively.

In our evaluation, we first applied Pex on different handlers of the subject applications to generate inputs. We then executed the applications with the generated inputs (execute the different handlers for each page) and learned guarding-model sets with the help of the tracing module. We then applied our minimization algorithm on the model sets. We next transformed the subjects to PaRaM-instrumented applications with the essential control decisions instrumented and the critical points guarded with query checkers. We also separately transformed the subjects to be guarded with the path-insensitive approach, i.e., instrumented only the critical points with query checkers such that at each critical a generated query is checked if it matches any of the queries in the corresponding CP-model. We then executed both the path-insensitive-guarded applications and PaRaM-instrumented applications with the evaluation inputs and measured the number of false positives and false negatives.

Table 6.3: Results of executing applications with the evaluation inputs.

Subject	Path InSensitive			PaRaM		
	False Positives	False Negatives	Attacks Prevented	False Positives	False Negatives	Attacks Prevented
Classifieds	0	220	3566	0	0	3786
Bookstore	0	12	3125	0	0	3137
Employee Directory	0	259	3763	0	0	4022
Events	0	0	6316	0	0	6316
Portal	0	10	2969	0	0	2979
Total	**0**	**501**	**19739**	**0**	**0**	**20240**

6.3 Results

6.3.1 Effectiveness

In this section, we address the research question RQ1 of how effectively PaRaM guards the applications against SQLIAs, when compared to the path-insensitive approach. To address this question, we measure the number of false positives and false negatives produced by both the approaches, PaRaM and the path-insensitive approaches. We say that an input leads to a false negative (i.e., a successful attack that the approach did not detect) if executing the application with the input results in executing at least one SQL query that includes the attack input. Note that invocation of a handler in an application, e.g., on clicking the login button, can lead to issuing multiple SQL queries that include the inputs provided by a user.

Table 6.3 shows the results of RQ1 for the five subject applications. Column "Path Insensitive" gives the results of executing the application with the evaluation inputs when the applications are guarded with a path-insensitive approach; Subcolumn "False Positives" gives the false positives produced by the approach, i.e., the number of benign inputs identified as attack inputs, and Subcolumn "False Negatives" give the number of false negatives produced by the approach, Subcolumn "Attacks Prevented" gives the number of attack inputs that the applications were completely guarded against. Similarly, Column "PaRaM" shows the results

Table 6.4: CP-Model for a critical point in Employee Directory.

Id	SQL Signature	Path Sig.
S1	:SELECT:count:(:*:):FROM:emps:WHERE:emp_login:=:and:emp_password:=:	:CD2:
S2	:SELECT:emp_id:FROM:emps:WHERE:emp_login:=:and:emp_password:=:	:CD2:CD2:
S3	:SELECT:emp_level:FROM:emps:WHERE:emp_login:=:and:emp_password:=:	:CD2:CD2:CD2:
S4	:SELECT:count:(:*:):FROM:emps:WHERE:emp_login:=:	:CD2:CD64:
S5	:SELECT:name:FROM:deps:WHERE:dep_id:=:	:CD43:
		:CD2:CD43:

of executing the subject applications with the evaluation inputs.

Results show that PaRaM was effective in completely guarding the applications against SQLIAs and also produced no false positives. In contrast, though the path-insensitive approach did not generate any false positive, unlike PaRaM, the path-insensitive approach was not effective in guarding the applications against attack inputs and produced a maximum of 259 (6.44%) false negatives for the EmployeeDir application and produced a total of 501 false negatives for all the applications. We manually analyzed and confirmed that these false negatives were caused due to the path-sensitive characteristics of the critical points. Though the path-insensitive approach produced a small % of false negatives (2.59% average), allowing even such a small number of attacks poses a serious threat to information security since only a single attack is sufficient to compromise the information security.

We next explain an example attack that was not successful with PaRaM, but that was successful with path-insensitive approach. In the EmployeeDir application, the login handler issues three SQL queries for completing the login transaction of user: one for checking the authentication of the provided information, and the other two for loading the employee id and user rights after successful authentication. The CP-model for the critical point where all of these `SELECT` queries are issued is given in Table 6.4 (Column "Id" is used for illustration purpose only). For authentication, the issued query is expected to be of the structure $S1$. $S2$ and $S3$ are the structures of the queries used to load the employee id and employee level (i.e., user rights) after login. For an input URL `http://localhost/GotoCode/EmployeeDirectory/Login.aspx?Password=bypass*/`

`&Login=mrmarri'/*`, the values of parameters `Password` and `Login` are used by the login handler to issue the queries of structure $S1$, $S2$, and $S3$. While constructing the first query, there is a defect in the application code that does not replace the single quotes in the login value with double quotes; such character replacement is a common practice to prevent SQLIAs. Due to this defect, the query to authenticate the login using these inputs is `SELECT count(*) FROM emps WHERE emp_login='mrmarri'/*' and emp_password='bypass*/'` where the `/* */` part is treated as in-line comments by MySql. That is, the structure of the query is the same as that of $S4$ ($S4$ is the structure of the query used to check if a user with the same login already exists when a new user is being added) and not that of $S1$. Path-insensitive approach ignores the difference in path signatures for these two queries and finds that the generated query structure is benign at that critical point (with respect to $S4$) and allows the execution of the query, whereas for the subsequent query (expected to be of structure $S2$) generated using the inputs, the approach detects that the inputs are attack inputs. On the other hand, PaRaM identifies that for the executed path `:CD2:`, the generated query is not as expected, i.e., the structure does not match $S1$ and therefore does not allow the execution of the generated query.

In summary, our results support that PaRaM's path-sensitive approach is more effective in preventing SQLIAs than a path-insensitive approach, when the web applications issue SQL queries whose structure is path sensitive.

6.3.2 Performance Gain & Runtime Overhead

In this section, we address RQ2 of how many control decisions are reduced by our minimization algorithm, referred to as *performance gain* and how much runtime overhead is caused by PaRaM's additional instrumentation compared to the path-insensitive approach. Our minimization algorithm reduced the number of control decisions (CDs) to be instrumented from $45 - 101$ (min-max range of CDs of subject applications) to $3 - 7$.

To compare the additional overhead caused by PaRaM over a path-insensitive approach, we measure the execution times for both the approaches when the applications (guarded with

Table 6.5: Results showing the runtime time overhead caused by PaRaM.

Subject	#Inputs	Running time for all inputs (ms)		#CDs	Avg. Path Sig Length	Avg. Overhead/ input (ms)
		Path-Insensitive	PaRaM			
Classifieds	2140	43280.48	43601.50	7	5	0.15
Bookstore	2398	20303.16	20671.18	7	8	0.15
EmployeeDir	2944	33497.92	33890.94	3	1	0.13
Events	3295	14990.86	15438.88	3	4	0.14
Portal	966	12885.74	12973.74	5	9	0.09
Average	2349	24991.63	25315.25	5	5	0.13

PaRaM and the path-insensitive approach) are executed with a benign set of evaluation inputs. Table 6.5 shows the results of execution time values. Column "Running Time for all inputs (ms)" gives the actual execution time values (in milliseconds or ms) for all the inputs for each application. Column "#CDs" gives the number of control decisions monitored for each application. Column "Avg. Path Sig. Length" gives the average length of the path signatures generated when executing the PaRaM-instrumented application with all the inputs. Finally, Column "Avg. Overhead/input (ms)" gives the average overhead (in milliseconds or ms) caused by the additional monitoring in PaRaM when compared to the path-insensitive approach. Our results show that the overhead caused by PaRaM is as low as 0.09 to 0.15 ms on an average for each input when compared to the path-insensitive approach. Furthermore, execution of each PaRaM-instrumented application with the inputs resulted in an overall overhead cost of 5 to 13 milliseconds on an average when compared to the execution of the original applications. This overhead cost should be taken as worst-case values since our prototype implementation is not sophisticated enough to ensure high performance of the SQL parser as well as the construction of signatures. Nevertheless, the overhead cost is low when weighed against the benefit of effectively monitoring the applications against SQLIAs.

In summary, our evaluation results show that the minimization algorithm significantly reduced the number of control decisions to be monitored and PaRaM effectively prevents against SQLIAs with a little overhead cost when compared to a path-insensitive approach.

Chapter 7

Related Work

In this section, we discuss the most related work that deal with SQLIAs in two categories.

7.1 Runtime Monitoring

Halfond and Orso [6] proposed AMNESIA that uses static analysis to build a query model and instruments the application with the model to monitor the application at runtime such that each generated SQL query is checked against the query model. Halfond et al. [5] proposed an approach that uses positive tainting in tracking the flow of untrusted inputs at execution based on policies provided by programmers. Martin et al. [13] proposed a similar approach that takes programmer-specified policies in PQL. Huang et al. [8] proposed an approach that also statically infers annotations to be added to sensitive sections of the application, to achieve runtime inspection of queries before being issued to the database. Sun and Beznosov [19] also adopt static discovery of intended queries and use the discovered intention grammar in combination with monitoring taint inputs to prevent SQLIAs. All the preceding approaches are path-insensitive and build a model or grammar based on static analysis, which could lead to false negatives (as shown in our evaluation). In contrast to these approaches, our approach is path-sensitive and uses dynamic analysis of the application to build a model, and therefore more effective. Bandhakavi et al. [2] proposed to use shadow execution to monitor SQLIAs at

runtime. To achieve guarding of an application against SQLIAs, for every actual execution at runtime, a set of benign inputs are generated to dictate the same path as the actual inputs. Their approach requires a high amount of program transformation since the benign set of inputs should dictate the same path as the actual inputs, while our approach only instruments the essential control decisions in addition to the critical points that are also instrumented in their approach.

7.2 Testing, Vulnerability Detection, and Code Cleaning

Existing testing approaches use input generation, taint propagation, and mutation techniques to generate attack-causing inputs and identify vulnerabilities in web applications. Wassermann et al. [25] proposed a vulnerability-detection approach that carried out dynamic-symbolic execution and used string analysis to create attack-causing inputs using backward slicing of the application under test to alleviate the problem of path explosion. Kieyzun et al. [9] proposed the use of dynamic taint analysis in testing a web application for vulnerabilities by identifying all sets of inputs that reach the application's attack-prone location and mutating the identified set of taint inputs to match a given set of attack patterns. When the mutated inputs reach the attack-prone program points, the application is detected to be vulnerable with respect to those locations. Yu et al. [26] proposed an approach to assist test input generation to exploit vulnerabilities in web applications using forward and backward symbolic execution. In general, researchers have proposed several techniques to assist automatic test generation and vulnerability detection for web applications [1, 7, 12, 24]. While all of these approaches focus on detecting and assisting in hardening web applications against SQLIAs, our approach can directly retrofit an existing application to prevent SQLIAs without having to manually harden the code. Another important category of techniques to deal with SQLIAs are using defensive coding mechanisms. Defensive coding mechanisms include appropriate user input validations and the use of robust mechanisms, such as the use of prepared statements for interacting with database [3]. Thomas et al. [22] proposed a prepared statement replacement algorithm to au-

tomatically replace vulnerable SQL statements with prepared statements for Java applications. These algorithms may not be generic and are both programming language as well as database dependent. For example, PHP only supports the use of server-side prepared statements which are specific to the database being connected to unlike Java or C# that support client-side prepared statements. Our runtime monitoring is not limited by either the language or the database and therefore is easy to adopt for existing applications. Nevertheless, our approach is no replacement to any of the secure coding practices, but can be an effective technique to retrofit an existing web application to prevent SQLIAs.

Chapter 8

Discussion

In our evaluation, PaRaM generated no false positives, and effectively guarded the subject applications with low overhead cost. Nevertheless, effectiveness of PaRaM depends on the inputs used to build the guarding model set. In particular, a model set could be incomplete when inputs used in the learning phase do not exercise either critical points or certain paths that lead to critical points, resulting in false positives. On the other hand, applying minimization algorithm on an incomplete model set could lead to false negatives. We alleviated this issue to a certain extent by using a state-of-the-art DSE-based technique [23] that generates inputs that can achieve high structural coverage (83.6% block coverage[1]). To address this issue further, we plan to use static analysis to enhance our model set to characterize those non-covered paths or critical points.

In addition to the approach's effectiveness, evaluation also showed that our minimization algorithm can bring in high performance gain without comprising the effectiveness of the model set, provided the model set is complete. Furthermore, in practice, our algorithm can bring in a high amount of reduction in overhead cost for programs that recursively include control decisions that are not instrumental in deciding the query structure. On the other hand, like most heuristic algorithms, there could be situations when a model set cannot be minimized and

[1]More details of block coverage are available in `http://research.microsoft.com/en-us/projects/pex/pextutorial.pdf`

therefore the performance gain achieved by our minimization algorithm could be marginal. An example of such a case is shown in Figure 4.3.2: if this control flow graph represents the entire application, i.e., there is only one critical point and each program path results in a query with a unique SQL signature, then our minimization algorithm cannot minimize the guarding model, and thus the performance gain achieved by our model can be 0 in such cases. In future work, we plan to study the existence of such cases and investigate if path signatures involved in such cases could still be minimized.

Chapter 9

Conclusion

Among the various types of attacks that web applications face, SQL Injection attacks belong to one of the most popular attack types, since a web application's vulnerability to SQL injection attacks poses serious threat to the information security. To effectively deal with this type of attacks and to retrofit existing web applications, we presented PaRaM, a path-sensitive runtime monitoring approach. Our approach addresses two major challenges of runtime monitoring by learning a guarding model before the application is deployed and by applying a minimization algorithm to reduce the monitoring overhead on the performance of the application. We evaluated PaRaM by applying it on five web applications. Our results showed that PaRaM effectively guarded web applications when compared to a path-insensitive approach that produced a non-trivial number of false negatives, and also caused low performance overhead.

REFERENCES

[1] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding Bugs in Dynamic Web Applications. In *Proc. ISSTA*, pages 261–272, 2008.

[2] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: Preventing SQL Injection Attacks Using Dynamic Candidate Evaluations. In *Proc. CCS*, pages 12–24, 2007.

[3] William R. Cook and Siddhartha Rai. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. In *Proc. ICSE*, pages 97–106, 2005.

[4] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proc. PLDI*, pages 213–223, 2005.

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

[6] William G. J. Halfond and Alessandro Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks. In *Proc. ASE*, pages 174–183, 2005.

[7] William G.J. Halfond, Saswat Anand, and Alessandro Orso. Precise Interface Identification to Improve Testing and Analysis of Web Applications. In *Proc. ISSTA*, pages 285–296, 2009.

[8] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proc. WWW*, pages 40–52, 2004.

[9] Adam Kieżun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proc. ICSE*, pages 199–209, 2009.

[10] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.

[11] Yuji Kosuga, Kenji Kono, Miyuki Hanaoka, Miho Hishiyama, and Yu Takahama. Sania: Syntactic and semantic analysis for automated testing against sql injection. In *Proc. ACSAC*, pages 107–117, 2007.

[12] Michael Martin and Monica S. Lam. Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking. In *Proc. USENIX Security*, pages 31–43, 2008.

[13] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding Application Errors and Security Flaws using PQL: a Program Query Language. In *Proc. OOPSLA*, pages 365–383, 2005.

[14] Microsoft Research. ExtendedReflection - Dynamic Analysis Framework for .NET, 2010 (accessed). http://research.microsoft.com/en-us/projects/extendedreflection/.

[15] Open Web Application Security Project. Owasp top ten project, 2010. http://www.owasp.org/index.php/Top_10_2010-A1-Injection.

[16] SANS Institute. The Top Cyber Security Risks, 2010. http://www.sans.org/top-cyber-security-risks/.

[17] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proc. ESEC/FSE*, pages 263–272, 2005.

[18] Zhendong Su and Gary Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proc. POPL*, pages 372–382, 2006.

[19] San-Tsai Sun and Konstantin Beznosov. Retrofitting existing web applications with effective dynamic protection against sql injection attacks. *International Journal of Secure Software Engineering*, 2010.

[20] Symantec. Internet Security Threat Report: Volume XV: April 2010, 2010. http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xv_04-2010.en-us.pdf.

[21] Technology Blog. 32.6m passwords may have been compromised in rockyou hack, 2010. http://www.guardian.co.uk/technology/blog/2009/dec/15/rockyou-hacked-passwords.

[22] Stephen Thomas, Laurie Williams, and Tao Xie. On Automated Prepared Statement Generation to Remove SQL Injection Vulnerabilities. *Inf. Softw. Technol.*, 51(3):589–598, 2009.

[23] Nikolai Tillmann and Jonathan de Halleux. Pex-White Box Test Generation for .NET. In *Proc. TAP*, pages 134–153, 2008.

[24] Gary Wassermann and Zhendong Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proc. PLDI*, pages 32–41, 2007.

[25] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic Test Input Generation for Web Applications. In *Proc. ISSTA*, pages 249–260, 2008.

[26] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Generating Vulnerability Signatures for String Manipulating Programs Using Automata-based Forward and Backward Symbolic Analyses. In *Proc. ASE*, pages 605–609, 2009.