View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Concordia University Research Repository

EMPLOYING OPPORTUNISTIC DIVERSITY FOR DETECTING INJECTION ATTACKS IN WEB APPLICATIONS

WEI HUO

A THESIS

IN

CONCORDIA INSTITUTE FOR INFORMATION SYSTEMS ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE IN INFORMATION SYSTEMS SECURITY

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

SEPTEMBER 2014

© WEI HUO, 2014

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Wei Huo**

Entitled: **Employing Opportunistic Diversity for Detecting Injection Attacks in Web Applications**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science in Information Systems Security

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________ Dr. J. Bentahar ______________ Chair

______________ Dr. S. Abdi ______________ External Examiner

______________ Dr. A. Youssef ______________ Examiner

______________ Dr. L. Wang ______________ Supervisor

Approved ______________ Dr. J. Bentahar ______________

Chair of Department or Graduate Program Director

________ 20 ____ ________________________

Dr. Nabil Esmail, Dean

Faculty of Engineering and Computer Science

ABSTRACT

Employing Opportunistic Diversity for Detecting Injection Attacks in Web Applications

Wei Huo

Web-based applications are becoming increasingly popular due to less demand of client-side resources and easier maintenance than desktop counterparts. On the other hand, larger attack surfaces and developers' lack of security proficiency or awareness leave Web applications particularly vulnerable to security attacks. One existing approach to preventing security attacks is to compose a redundant system using functionally similar but internally different variants, which will likely respond to the same attack in different ways. However, most diversity-by-design approaches are rarely used in practice due to the implied cost in development and maintenance, significant false alarm rate is also another limitation. In this work, we employ opportunistic diversity inherent to Web applications and their database backends to prevent injection attacks. We first conduct a case study of common vulnerabilities to confirm the effectiveness of opportunistic diversity for preventing potential attacks. We then devise a multi-stage approach to examine database queries, their effect on the database, query results, and user-end results. Next, we combine the results obtained from different stages using a learning-based approach to further improve the detection accuracy. Finally, we evaluate our approach using a real world Web application.

Acknowledgments

I would really like to express my gratitude to my supervisor, Dr. Lingyu Wang, for his guidance and support to my research and study in Concordia. In addition, I would also like to thank my labmates for their help and suggestions. At last, this thesis cannot be finished without the support of my parents.

Contents

List of Figures viii

List of Tables x

1 Introduction 1

2 Background 5

2.1 Attacks on Web Applications 5

2.2 Diversity in Security 8

2.3 Common Vulnerabilities and Exposures (CVE) Database 10

2.4 Existing Technique of Attack Detection 11

2.4.1 Query String Comparison and Evaluation 11

2.4.2 Database Schema Comparison 13

2.4.3 Behavioral Distance 14

2.4.4 Decision Tree and Learning-Based Approach of Attack Detection 16

3 Related work 20

3.1 Defense of Injection Attacks 20
3.2 Diversity in Security 23
3.2.1 Effectiveness of Improving Security through Diversity 23
3.2.2 Challenges of Building Diversity Systems 24
3.2.3 Diversity System Applications 25
3.3 Existing Techniques of Query and Database Comparison 31
3.3.1 Query String Comparison 31
3.3.2 Database Schema Comparison 32

4 The Case Study **34**
4.1 A Motivating Example 34
4.2 Exhaustive Search in CVE Database 37
4.2.1 Preparation of Exhaustive Search 38
4.2.2 Searching Applications with Variants 38
4.2.3 Matching Common Vulnerabilities 41
4.2.4 Summary 44

5 The Methodology **46**
5.1 Overview 46
5.2 The Controller 47
5.3 The Monitor 48
5.3.1 Stage 1: SQL Query 51
5.3.2 Stage 2: Changes to Database 56

5.3.3 Stage 3: Database Result 59

5.3.4 Stage 4: Application Result 62

5.4 Result Correlation 64

5.5 Summary 68

6 Implementation and Experiments **69**

6.1 Model Implementation 69

6.1.1 Multi-Stage Comparison 69

6.1.2 Anomaly Profile 78

6.2 Experiments 80

6.2.1 The Web Application and Vulnerabilities 80

6.2.2 Training Data 82

6.2.3 Decision Tree Learning and Result Evaluation 87

6.3 Summary 96

7 Conclusion **97**

A CVE Entries for Applications with Common Vulnerability **99**

Bibliography **103**

List of Figures

1 Diversity System Paradigms [1] [2] 9

2 Parse Trees of Normal and Malicious Query [3] 12

3 Tree Representation of XML Schema [4] 13

4 Decision Tree Example [5] 17

5 Overview of Anomaly-based Detection System [6] 18

6 The Model 48

7 The Working Process of the Controller 48

8 Standard AST of Example 4 55

9 The Generated ASTs from Both Variants of Example 4 55

10 Final Decision Making 66

11 Mysql Log 70

12 SQL Server Log 70

13 An Error Web Page Example 77

14 Midicart Interface 81

15 The Learned Decision Tree 89

16 FPR and TNR for Detecting Attack on Common Vulnerabilities 91

17 Detection Accuracy Comparison between Anomaly Approach and Decision Tree . 93

18 FPR Comparison of Detection with Single Feature and Decision Tree 94

19 FNR Comparison of Detection with Single Feature and Decision Tree 94

20 Detection Accuracy Comparison with Single Feature and Decision Tree 94

List of Tables

1 Different Types of SQLIA 7

2 Comparison of SQLIA Defence Techniques according to Attack Types [7] 22

3 Grouping of Applications 40

4 Web Application with Variants and SQL Injection Vulnerability 42

5 Common Vulnerabilities of Midicart 44

6 Attack Detection Features of each Stage 64

7 Anomaly Profile Example 78

8 Midicart Vulnerability Entries in CVE 81

9 Training Data Summary 86

10 Detection Performance of Decision Tree 90

11 Detection Performance of Attack 1 ,2 and 3 91

12 Features and Thresholds for Single-Stage Detection 95

Chapter 1

Introduction

Web-based applications are becoming increasingly popular in an age of cloud computing and mobile devices. In contrast to their desktop counterparts, Web applications demand less client-side resources and are easier to deliver and maintain by employing the Web browser as a thin client. On the other hand, web applications are especially attractive to security attacks due to their larger attack surfaces and the lack of security proficiency or awareness of their developers. Protecting a mission critical web application, such as those used by governments, financial institutions, and health care sectors, means more than just patching known vulnerabilities and deploying firewalls or IDSs. The recent widespread panic about the Heart Bleed vulnerability [8] has clearly demonstrated the importance of improving applications' robustness against novel zero day attacks exploiting undiscovered vulnerabilities. On the other hand, this is clearly a challenging task since signature-based detection mostly works with known attacks, whereas anomaly detection is well known to suffer from a high false alarm rate.

In a slightly different context, software diversity has been regarded as a promising mechanism

for improving the robustness of a software system against unknown attacks [9] (a more detailed review of related work will be given in Chapter 3). By comparing outputs [1] or behaviors [10] of multiple software replicas with diverse implementation details, security attacks may be detected and tolerated as Byzantine faults [11]. Although the earlier diversity-by-design approaches are usually regarded as impractical due to the implied development and deployment cost, recent work show more promising directions of either employing opportunistic diversity already existing in operating systems [12], or automatically generating diversity through randomization of address space [13, 14], instruction set [15], or data space [16]. A dilemma faced by such existing work is that, diversity is either too costly (as in the case of diversity-by-design), or not reliable enough to be used as a stand-alone means for attack detection (as in the case of opportunistic diversity).

In this thesis, we take a novel approach of employing opportunistic diversity already existing in Web applications and their database backends to assist, instead of replacing, traditional anomaly detection methods in reducing false alarms and improving detection accuracy. Specifically,

- First, we conduct a case study on real world vulnerabilities to confirm the effectiveness of the proposed approach. Specifically, we perform an exhaustive search among almost 6,000 Common Vulnerabilities and Exposures (CVE) Web injection vulnerabilities [17] to find all Web applications that have multiple variants written in different languages, and the common vulnerabilities between those variants. Our results indicate a very low occurrence of common vulnerabilities between variants, which shows opportunistic diversity can indeed assist in detecting attacks.

- Second, we propose a multi-stage approach to employ opportunistic diversity for assisting

anomaly detection of injection attacks. Specifically, we design an architecture for monitoring the behavior of multiple variants of an application at four stages, in terms of queries sent by the application to its database backend, the effect of such queries on the database, query results, and user-end results. We propose methods for extracting features at each of those stages, and for comparing such features, or partial anomaly detection results obtained from such features, between different variants. We also devise a learning-based method for combining the partial results obtained at different stages into a final decision.

- Finally, we implement the proposed approach based on a real world Web application and conduct experiments to evaluate the effectiveness of our approach. The experiment results indicate that, by employing diversity between different variants, our approach leads to less false positives than traditional anomaly detection; at the same time, by combining partial results obtained from multiple stages, our approach yields higher detection accuracy than existing work based on a single stage.

The main contribution of this thesis is twofold. First, to the best of our knowledge, this is the first effort that combines the advantages of both opportunistic diversity and anomaly detection; this approach can avoid both the prohibitive cost implied by diversity-by-design and the high false alarm rate inherent to traditional anomaly detection. Second, our approach of combining partial results obtained at different stages of the interaction between users, applications, and databases yields a higher detection accuracy than existing work, and thus leads to a promising direction towards practical security solutions; although we have focused on injection attacks in this thesis, the methodology can be easily extended to prevent other attacks.

The rest of the thesis is organized as follows, we introduce background knowledge of SQL injection attack and diversity in security in Chapter 2, review related work in Chapter 3. We then build intuitions through a motivating example in Chapter 4.1, perform an exhaustive search in CVE vulnerability database in Chapter 4.2, propose the attack detection model in Chapter 5, present the case study in Chapter 6 and conclude the thesis in Chapter 7.

Chapter 2

Background

In this section, we briefly introduce some background knowledge that will be used for further discussion of this paper, including several types of attack on web applications, the use of diversity in security, the common vulnerabilities and exposures (CVE) database and some existing techniques about attack detection.

2.1 Attacks on Web Applications

Most of the attacks on web applications belong to *Cross Site Scripting(XSS)* and *SQL injection attack(SQLIA)* [18]. In this section we will review the knowledge regarding these two attacks, but we focus on SQLIA since we use it later to evaluate our approach.

SQL injection attack(SQLIA) "SQL" stands for Structured Query Language, a language used to access and communicate with databases. SQLIA [19] is a type of security exploit in which

the attacker can add SQL statements through a web application's input field or hidden parameter. As a result, the attacker can pass arbitrary SQL queries into databases. By constructing different Injection string, the attacker can retrieve, modify or even delete data in database. It becomes one of the most serious security issues on the internet.

Example 1. *A typical SQLIA example used for testing if injection vulnerability exist:*

Attacker inject the code to the application.

```
' and '1'='2
```

Application passes this input value to the query statement:

```
SELECT * FROM users WHERE name = 'username';
```

If the input is not sanitized, the query becomes:

```
SELECT * FROM users WHERE name = '' AND '1'='2';
```

Instead of returning the result of the SELECT query, it always returns a false value, thus attacker can use the error page to judge if the application can be exploited.

⊡

By injecting different strings, the attacker can achieve various goals. According to this difference, SQLIAs are classified into different categories, we listed them in Table 1, along with the purpose of different types of SQLIA and corresponding attack examples.

Cross Site Scripting(XSS) XSS [20, 21] is another common type of code injection attack on web applications. Unlike SQLIA in which the code is injected into databases, XSS allows attacker to inject client-side script to the web pages, which could be viewed by normal users. By

Type of Attack	Attack Example	Attack Purpose
Tautology	SELECT accounts FROM users WHERE login=‘ *‘ or 1=1 - -* ’AND pass=‘’	Injection string makes query always true, often used for bypassing password.
Logically incorrect query	SELECT accounts FROM users WHERE login=‘’ AND pass=‘’ AND pin= *convert (int, (select top 1 name from sysobjects where xtype=‘u’))*	Injection string makes query always false, often used for gathering information about target system by the returned error information.
Piggy-backed query	SELECT accounts FROM users WHERE login=‘doe’ AND pass=‘’ *; drop table users* - - ’	Injection string contains a separate query sentence, often used for executing a distinct database command.
Union query	SELECT accounts FROM users WHERE login= ‘*’UNION SELECT cardNo from CreditCards where acctNo=10032 - -* AND pass=‘’	Injection string contains a malicious query after the normal query concatenated by “UNION", often used for gathering table information.
Stored Procedure	SELECT accounts FROM users WHERE login=‘doe’ AND pass=‘*’; exec master..xp_servicecontrol ‘start’,‘FTP Publishing’* - - ’	Injection string contains stored procedures to execute certain commands, attack usage depends on the stored procedure type.
Inference	SELECT accounts FROM users WHERE login=‘*legalUser’and 1=0 - -* ’ AND pass=‘’	Attackers retrieve information by a series of false/true query results. In blind injection, results are represented by the returned pages. In timing attacks, results are represented by server response time.
Alternate encoding	SELECT accounts FROM users WHERE login=‘ *%2527 or 1=1 - -* ’AND pass=‘’	Attackers encode part of the Injection string to avoid filters of server. In this case, symbol “’" is substituted by encoding “%2527".

Table 1: Different Types of SQLIA

injecting malicious script, attacker can get access to sensitive data, e.g. session cookies, that the browser maintains on user's behalf. Attacker can use these data to hijack the victim's session and impersonate the victim.

Common types of XSS attacks are reflected XSS and stored XSS. In the reflected attack, malicious payload are crafted into URL. When the victim visits the link, crafted script would exploit the XSS vulnerability in web server and reflect the code to victim's browser. In stored XSS attacks, a malicious script is submitted by attackers and stored on the web server. The script would be executed when the victim accesses the corresponding web pages.

2.2 Diversity in Security

Using diversity to improve security is not a new idea, it has been proposed in many previous approaches from different aspects. The common principle is to diversify the vulnerabilities of the same application. Identical copies of an application give attackers the opportunity to compromise all released copies by identifying one single vulnerability. This software monoculture leads to an unfair battle between attackers and developers: it is not practical for developers to eliminate all vulnerabilities in their applications, but it is easy for attacker to find one [22, 23]. To decrease attacker's advantage, a number of functionally equivalent but internally different variants of application can be produced, thus exploiting a single vulnerability which would succeed on one variant is not likely to compromise all others. Introducing diversity also means attackers will find it more difficult to generate attack vectors by reverse engineering, since the variants they get are mostly different from the variants existing in their target.

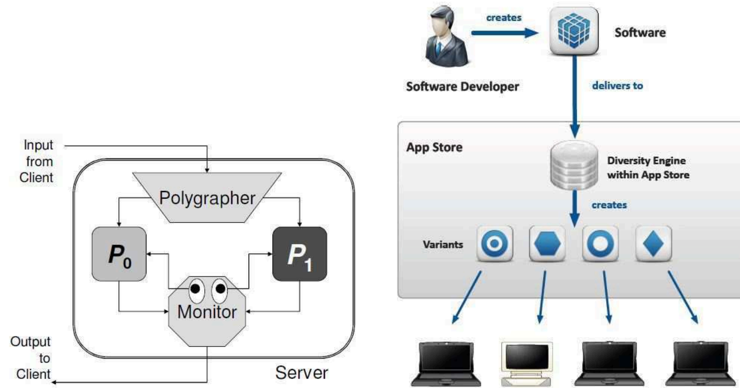


Figure 1: Diversity System Paradigms [1] [2]

Based on the source of variants, diversification can be categorized as specially designed variants and existing product used as variants. Specially designed diversification [13, 16, 24] often includes various code randomization, modification technique and diversified compiling. These methods can be achieved automatically, which makes it efficient to produce massive amount of diversified products, yet they are still susceptible to mimicry attacks. In the other case [25, 26] , functionally similar but internally different existing products are used to compose a diversity system. Compare to the specially designed variants, existing products are more cost efficient and can be used to defend against more types of attacks, since they generally have more unpredictable diversification space. Drawback is there is more synchronization problem, the number of available existing products is also limited. Our approach belongs to the latter category.

Diversity systems have two common paradigms as shown in Figure 1. The first runs multiple variants simultaneously, compares the result in each synchronization point defined by the system, attackers are forced to find a matching vulnerability that can compromise all running variants to achieve a successful attack, otherwise it would be detected. Typical example is *Orchestra* [27], which detects intrusion through parallel execution and monitoring.

In the second paradigm, different variants are distributed to individual users to achieve attack prevention. By using a single vulnerability, attacker can only compromise a portion of variants which has it as common vulnerability. Thus attacker is forced to find different vulnerabilities in order to perform a massive attack. Michael Franz proposed *E unibus pluram* [28], in which this kind of diversity framework is described.

2.3 Common Vulnerabilities and Exposures (CVE) Database

Common Vulnerabilities and Exposures (CVE) Database [17] collects information of publicly known security vulnerabilities and exposures and provides a reference method. It is maintained by MITRE Corporation and used by the *Security Content Automation Protocol (SCAP)* [29]. CVE IDs are listed in the *National Vulnerability Database(NVD)* [30]. It is one of the most widely used vulnerability databases.

In the CVE database, vulnerability information is listed in separate vulnerability entries and each vulnerability entry is assigned with a CVE identifier. According to MITRE, CVE identifier is a unique, common identifiers for publicly known information security vulnerability [31]. Apart from CVE ID, each vulnerability entry also contains a standardized text description of the vulnerability, references of the vulnerability and the date when this entry is added.

We show an example of CVE vulnerability entry below.

CVE-2013-4685	Buffer overflow in flowd in Juniper Junos 10.4 before 10.4S14, 11.4 before 11.4R7, 12.1 before 12.1R6, and 12.1X44 before 12.1X44-D15 on SRX devices, when Captive Portal is enabled with the UAC enforcer role, allows remote attackers to execute arbitrary code via crafted HTTP requests, aka PR 849100.

We use CVE database to thoroughly search existing SQL injection vulnerabilities, and further search for common vulnerabilities in different versions of the same application. If the number of common vulnerabilities is low, we can show that employing diversity can effectively enhance security of web applications. The details of this exhaustive search are given in Chapter 4.2.

2.4 Existing Technique of Attack Detection

In this section, we review some of the existing work that will be used for attack detection in our approach, including query string comparison techniques, database schema comparison techniques, behavioral distance metric and decision tree.

2.4.1 Query String Comparison and Evaluation

Query comparison is often used for defending SQLIAs. By comparing the runtime query and related safe query, potential SQLIA can be detected. The comparison is not simple string comparison. SQL queries are often transformed into parse trees or other expressions that can explicitly reveal the semantic actions of the query. Then the evaluation would be based on whether the semantic actions of two queries are equivalent.

We now review some work in this field that transform the queries into parse trees [3, 32], examples are given below.

Example 2. *The normal query is:*

```
SELECT * FROM 'User_Table' WHERE user_name = 'admin' AND password =
 'hacker_pwd'
```

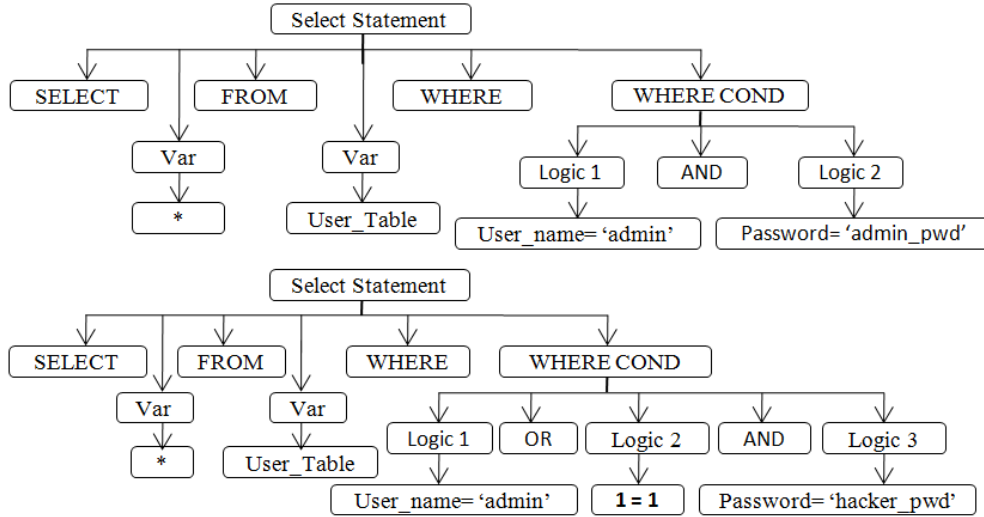


Figure 2: Parse Trees of Normal and Malicious Query [3]

The malicious query with SQL injection code is:

```
SELECT * FROM 'User_Table' WHERE user_name = 'admin' OR '1' = '1'
 AND password = 'hacker_pwd'
```

The parse trees generate from both queries are shown in Figure 2. The semantic actions of two parse trees are apparently different. In [3], the semantic equivalence evaluation is based on two criteria, two queries need to match both to be equivalent:

- *Two queries have the same number of stacked queries.*
- *The semantic actions in each cannonicalized query are equivalent.*

⊡

Existing approach of using query string comparison generally need to setup a safe standard, whether it is standard query string or standard grammar, so that the runtime string can be compared with it. We employ query string comparison in our work to score the difference between variants in

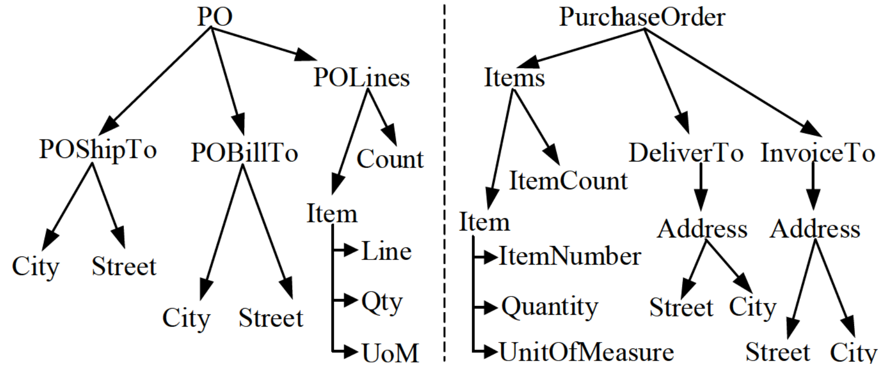


Figure 3: Tree Representation of XML Schema [4]

the first stage. Unlike existing approaches in which a runtime query is only compared to initialized safe profile then an anomaly score is generated. In our case, the anomaly score would be further compared between variants in order to get a more accurate detection result. The details would be given in Section 5.3.1.

2.4.2 Database Schema Comparison

Schema and ontology matching of database is an important problem in schema integration, semantic web, e-commerce, etc. It takes two schemas or ontologies as input, as shown in Figure 3 and output the equivalence relationship of the entities between two inputs.

The equivalence relationship is generally determined based on *linguistic* and *structural* analysis, represented by confidence measure.

Specifically, linguistic analysis often includes string-based techniques such as *prefix*, *suffix* checking, *edit distance comparison* [33,34], language based techniques such as *tokenization*, *elimination*, *expansion* and *lemmatization* [35].

Taking the “POShipTo" element of Figure 3 as an example, the tokenization process would

parse the element into tokens “PO",“Ship",“To", according to the punctuation and letter case. Expansion would expand the element into tokens “Purchase",“Order",“Ship",“To". With these manipulations, it becomes feasible to compare the linguistic similarity between elements.

Structural analysis often includes *graph matching* [36], taxonomy-based techniques such as *bounded path matching* [37]. Taking Figure 3 as an example again, element “POShipTo" has high structural similarity with element “DeliverTo", because their two children in the tree are linguistically similar.

The final similarity score is often a combination of linguistic similarity and structural similarity. An example is the schema matching tool *Cupid* [4], which calculates the final similarity score as the weighted sum of linguistic and structural similarity score.

In our work, we also monitor the changes made in databases, including change of schema and row(s) of data. However, we use a more simplified feature than the similarity score to evaluate the changes. The details would be given in Section 5.3.2.

2.4.3 Behavioral Distance

Behavioral distance is a metric designed for evaluating behavioral difference between replicas, proposed by D. Gao et al. in [10, 38]. Specifically, the problem behavioral distance tries to solve is to detect semantic similarity or difference when replicas in a diversity system process the same input.

In Gao’s work, they measure differences of system call sequences to calculate the behavioral distance between two replicas. System call sequences are extended to the same length by inserting

idle system calls, so that similar system calls can be aligned to each other. Typically seen system call sequences would be called *system call phrases*.

We show an example of identical system call phrases below.

Example 3. *$s1$ and $s2$ are two system call phrases observed in two replicas.*

$s_1 = (open_1; read_1; write_1; close_1)$

$s_2 = (open_2; read_2; idle_2; write_2; close_2)$

In order to make system call $write1$ align to $write2$, an idle system call σ would be insert to $s1$. The modified system call phrases become:

$s'_1 = (open_1; read_1; \sigma; write_1; close_1)$

$s'_2 = (open_2; read_2; idle_2; write_2; close_2)$

$\boxdot$

A distance table of system call phrases is initialized according to the distance of each aligned system call pair between phrases. Afterwards, with the intuition that similar system call phrases should appear in similar frequency in the training system call sequence data, the distance table is adjusted by frequency information. This adjustment is done in iterations. In each iteration, the system call sequence distance is calculated by the updated distance values from the previous iteration and their occurrence information in this iteration, the result would be used to update the distance table. The iteration stops when the distance table converges, which means it will only have very little change after an iteration. The values in the final distance table would be used to measure the distance of runtime system call sequences.

Behavioral distance is a good example showing how to combine multiple differences between

variants to produce the final judgment. In our work, we need to solve similar problems: to combine difference between application variants in multiple stages and produce final detection result. The details of our approach are given in Section 5.4.

2.4.4 Decision Tree and Learning-Based Approach of Attack Detection

Decision Tree

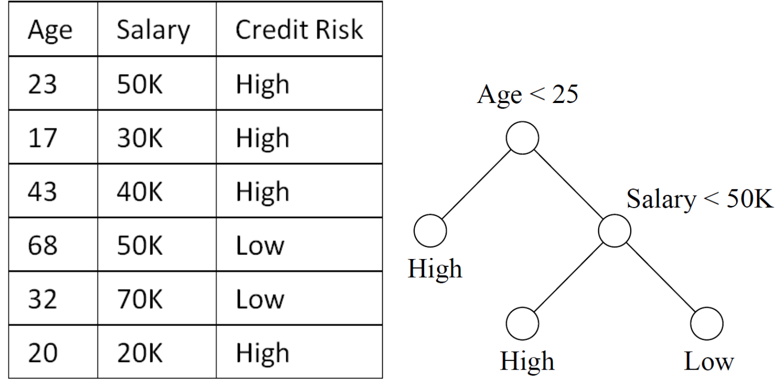
Decision tree [39–41] is a decision analysis tool commonly used for finding the best strategy for a certain goal. Its structure is similar to flow chart, in which the internal nodes represent attributes that classify the result and leaf nodes represent the decisions (class label).

Figure 4 shows an example of decision tree. It uses two attributes: age and salary, to partition the input data set into two classes: high risk and low risk. The tree structure shows split point “age" is higher than “salary", which means input would be classified according to age first.

The priority of split point is determined during the induction of decision tree, attribute that have the closest relevance to data set partitioning has the priority. Formally, we use *information gain* [42], which is based on *information entropy*, to determine the relevance. An attribute with higher information gain will be the upper split point of decision tree since this attribute can “better" partition the data.

Definition 1. Information Entropy *[42]*

Let $H(X)$ be the entropy of discrete random variable X with n possible value $\{x_1, x_2, ..., x_n\}$, $P(x_i)$ is the possibility of x_i and $I(x_i)$ is the information content of x_i.



Age	Salary	Credit Risk
23	50K	High
17	30K	High
43	40K	High
68	50K	Low
32	70K	Low
20	20K	High

Figure 4: Decision Tree Example [5]

$$H(X) = -\sum_{i=1}^{x} P(x_i)I(x_i) = -\sum_{i=1}^{x} \frac{n_i}{N} \log \frac{n_i}{N}$$

Definition 2. Information Gain *[42]*

Let $IG(T, a)$ be the information gain of attribute a, T is a training set in the form $(x, y) = (x_1, x_2, ..., x_k, y)$, where $x_a \in vals(a)$ is value of ath attribute of example X and y is the corresponding class label.

$$IG(T, a) = H(T) - \sum_{v \in vals(a)} \frac{|\{x \in T | x_a = v\}|}{|T|} H(\{x \in T | x_a = v\})$$

Learning-Based Approach of SQLIA Detection

F. Valeur et al. presented a learning-based approach to SQL attack detection in [6]. In this approach, an anomaly-based system learns the profiles of normal database activities characterized by a number of models. This allows the system to detect unknown attacks that have different behaviors compare to normal activities.

The overview of the detection system in this approach is presented in Fig. 5. Event provider

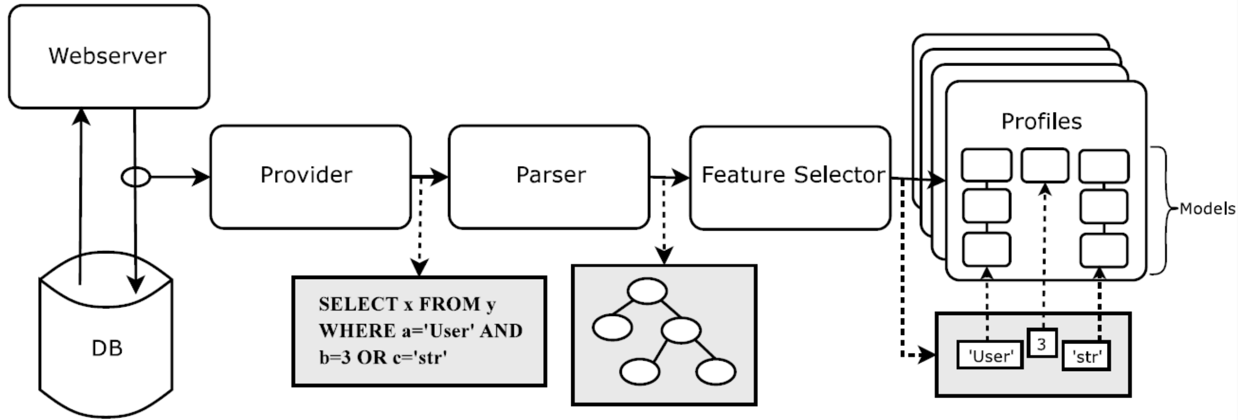


Figure 5: Overview of Anomaly-based Detection System [6]

is in charge of providing the detection system of every SQL query generated by the application. Parser will then process the queries into a high-level view. Feature selector alters the form of the queries so that they can be processed by the models. The actual work of feature selector depends on the status of the system. Namely, there are three status available: training, threshold learning and detection.

In the training phase, feature selector would match the processed queries with existing *profiles*, a collection of statistical models and mapping between features and their corresponding models. If no matching is found, a new profile would be created.

In threshold learning phase, the models would no longer be updated. Instead, an anomaly score is generated to measure how the feature vector matches its model. The highest aggregate anomaly score for each profile would be recorded.

In actual detection phase, anomaly score would be generated in the same way as in threshold learning phase, but it would be compared to the recorded anomaly score. If the runtime score exceeds records, alarm would be raised.

This approach provides two practical anomaly detection models: the string model and data

type-independent model. The string model uses string length and character distribution to detect anomaly. Data type-independent model detects anomaly by checking previously unseen value in runtime queries.

In our work, we also use decision tree as the tool to produce final detection result. Similarity score between variants of each stage are used as attributes of the decision tree. We collect these scores with different actions on the application and use them as training data. A decision tree would be learned from training data and used for attack detection. The details would be given in Section 5.4.

Chapter 3

Related work

In this section, we review existing approaches to SQL injection attack detection, query string comparison, database schema matching and the use of diversity for security.

3.1 Defense of Injection Attacks

Despite many years of research, various forms of injection attacks still remain a major threat to Web applications today (e.g., injection attack is ranked number one in the 2013 top ten critical Web application security risks by the Open Web Application Security Project (OWASP) [43]). Defensive coding is one of the common approaches, with mechanisms like [44] proposed to check user inputs by types, patterns, and detect the malicious input according to signatures. Although it is the most fundamental way to detect SQLIAs, this practice often generates a significant amount of false positives and it also cannot completely cover all the input fields. Xiang Fu et al. also proposed *SAFELI* [45], a mechanism that uses static analysis to detect SQLIAs of web applications during

compilation. Similarly, *JDBC Checker* [46] is a practical tool implementing static analysis. As a static code checker, it verifies the correctness of dynamically-generated SQL queries. However, for SQLIA queries that contains correct type and syntax, this tool can only generate false negatives and is almost useless. The inaccuracy of detection leads to the inefficiency of static analysis.

Further approaches involve the combination of static and dynamic analysis. Typical examples are AMNESIA,CANDID, SQLGuard and SQLCheck. *AMNESIA* [47] is a hybrid solution that combines static and dynamic analysis. It uses static analysis to build a model for web applications and intercept the run-time queries to verify if they match the model. *CANDID* [48] is another dynamic analysis approach; it dynamically runs web applications with candidate inputs. SQLIAs can be detected by comparing them to the structure of candidate queries. *SQLGuard* [49] and *SQLCheck* [50] check SQLIAs based on parse tree models generated by static code analysis. The runtime queries' structures are compared to the model and only the matched ones would be sent to database. Both approaches require code modification of web applications. Although dynamic analysis is introduced in these approaches, their accuracy is still largely based on static analysis.

Another promising SQLIA defense approach is *SQLrand* [51], it uses a randomized instruction set of queries to prevent SQLIAs. The SQL keywords under SQLrand are different from normal so that the injection code with normal SQL keyword cannot have command effect to database. The modification of instruction set is based on a secret key, a proxy filter is introduced to de-randomize the code. This technique is generally very effective. However, if the secret key is compromised, attackers would be able to know the randomized instruction. Also, the interpretation of proxy filter can significantly increase computation cost.

Technique	Tautology	Illegal/ Incorrect	Piggy-back	Union	Stored Procedure	Inference	Alter Encodings
Detective techniques							
AMNESIA [47]	●	●	●	●	×	●	●
CSSE [52]	●	●	●	●	×	●	×
SQLCheck [32]	●	●	●	●	×	●	●
SQLGuard [49]	●	●	●	●	×	●	●
SQLrand [51]	●	×	●	●	×	●	×
Preventive Technique							
JDBC-Checker [46]	–	–	–	–	–	–	–
Java Static Tainting [53]	●	●	●	●	●	●	●
Safe Query Objects [54]	●	●	●	●	×	●	●
Security Gateway [55]	–	–	–	–	–	–	–
SecuriFly [56]	–	–	–	–	–	–	–
SQL DOM [57]	●	●	●	●	×	●	●
WAVES [58]	○	○	○	○	○	–	○
WebSSARI [59]	●	●	●	●	●	●	●

Table 2: Comparison of SQLIA Defence Techniques according to Attack Types [7]

As Table 2 shows, previous SQLIA detective/preventive techniques can effectively counter against some types of SQLIAs, but none of these techniques can detect all kinds of SQLIAs. By employing diversity, our approach should be able to detect all types of SQLIAs in Table 2 except logically incorrect queries. This makes our approach competitive against other SQLIA detection methods.

3.2 Diversity in Security

Researchers have proposed many work on improving security through diversity from different aspects. We review the literature in this field through in categories: effectiveness of improving security through diversity, challenge of building diversity system and diversity system applications.

3.2.1 Effectiveness of Improving Security through Diversity

Although the idea of improving security through diversity is straightforward, quantified evaluation is still in need to prove its effectiveness. M.Garcia et al. [12] evaluated OS diversity system's effectiveness for intrusion tolerance. They obtained vulnerability data of various OSes from NIST National Vulnerability Database (NVD), classified them and searched for common vulnerabilities across different OSes. The use of NVD data brings in some limitations, such as the lack of exploitability information and vulnerability existence confirmation. But the result is still promising, as only for a very small number of non-applications, remotely exploitable common vulnerabilities are found, which proves diversity system can significantly improve security of OS. They also evaluated various OS pairs and announced the pairs with best performance on intrusion tolerance.

3.2.2 Challenges of Building Diversity Systems

Randomization A common approach to diversity is using randomization in the low level code of web servers/web applications. In [24] the authors proposed an *instruction set randomization (ISR)* to achieve diversity and improve security. It combines a set of *Components-Off-The-Shelf (COTS)* web servers to create a redundant system. The use of COTS instead of specifically developed variants reduced the cost of implementation, making it possible to deploy enough number of COTS servers to fulfill intrusion tolerance requirement. An IDS as controller monitors different COTS severs, compares the difference between them, such as response, instruction sequence. Diversified web applications are ran on these COTS servers; they are diversified by instruction set randomization. Lexical analyzer is modified to recognize new instructions, and randomization is applied to code strings. Note that this is a fully automated randomization, which means such techniques cannot detect attacks of logic errors, such as directory traversal or SQL injections.

Another randomization approach is *address space randomization (ASR)*. It obfuscates the location of data and code instead of randomizing the program code itself. *PAX* [14] is a project of Linux which contains such technique to randomize memory regions of program code, called *Address Space Layout Randomization (ASLR)*. Also, in [13], an ASR technique is proposed to mitigate various code injection attacks on executable files, such as stack smashing and format string attacks. Specific ASR techniques involved in this work include memory regions base address randomization, variables/routines order permutation and random gaps between objects. But the randomization in this work still limits to heap and stack instead of all regions of the program file.

Apart from ISR and ASR, a new randomization technique: *Data Space Randomization (DSR)*

is proposed in [16]. It randomizes the representation of different data objects. Namely, each data object in memory is encrypted with a random mask, and they are unmasked before used. By using different masks on different objects, attackers can no longer determine if the intended value is overwritten into the code. Compare to ISR, DSR can defend against various code injection attacks. Compare to ASR, DSR provides randomization on a more thorough region including non-pointer data and pointer-valued data, and also provides a higher range of randomization. The drawback of DSR is a higher overhead than ISR and ASR.

Synchronization With a number of randomization techniques, system running multiple variants in parallel can be built. However, the problem of synchronization emerges in this kind of systems: when a signal is sent, the controller/monitor need to synchronize the delivery to all variants, otherwise divergence occurs between variants and false alarm would be raised. In [60], a synchronization technique is proposed to solve this problem. It chooses the granularity of system calls to synchronize and it is implemented through register and control flow manipulations. Majority voting is involved in the delivery time of system calls. With the control of monitor, each equivalent system call is executed in parallel, thus the false positive created by unsynchronized system calls is minimized. The synchronization of this work results extra delay of execution and it might be exploited by denial of service attack.

3.2.3 Diversity System Applications

Intrusion Tolerance Using design diversity for fault tolerance has been investigated for a long time. Intrusion tolerant system is defined by maintaining security properties while some of its

components are compromised [61], which prevents attackers from exploiting disclosed vulnerabilities. Most approaches in this field implement *Byzantine Fault Tolerant(BFT)* replication as fault tolerance solution. But in order to build a practical system, many additional problems are involved. In [62], most of these problems are listed, including: BFT replication performance, recovery, effectiveness of diversity, confidentiality etc. In [11], some of the challenges are further discussed in single machine environment, such as achieving both effectiveness and efficiency while combining OS and binary randomization techniques. The single machine environment brings in both new problems and opportunities, virtualization techniques need to be involved to achieve isolation between replicas. However, synchronization and transparency become less problematic than they are in multi machine environment.

In [63], an intrusion-avoidance architecture is built based on cross-platform JAVA technologies and *Iaas(Infrastructure-as-a-service)* cloud service providers. Diversity of this system is achieved in four levels: operating system, web server, application server and database management system. A configuration controller retrieves emerging vulnerabilities information from public vulnerability databases and also gathers security patches information. It will then estimate security risk and activate the system combined by diversity components with the least security risk. Platform-independent Java technologies plays an important role in this approach, as JAVA applications can be easily deployed and dynamically reconfigured in different environments.

One of the most crucial factors in the effectiveness of an intrusion tolerance system is that the vulnerability occurs independently. To reduce common vulnerabilities between replicas, security patches and recovery need to be preformed. *DIVErse Rejuvenation SYStem(DIVERSYS)* [64]

provides automatic management of diverse configurations and recoveries between replicas in fault tolerance systems. Diversity components under DIVERSYS is proactively or reactively patched in each period of time to provide recycling. A risk level metric based on common vulnerabilities and time period is proposed to measure the risk level of the system. If the risk level exceeds a threshold, DIVERSYS would start recovery process and then put the system in new life cycle. With this automated patching, independent occurrences of vulnerabilities can be guaranteed to a certain level and security is greatly improved.

However, while diversity intensifies, the problem of incompatibility between replicas also emerges. *DiveInto* [65] is a JAVA tool that improves compliance of diverse server replicas, which is focused on correcting syntax and semantic violations of the replicas instead of malicious violations. It specifies protocol of replicas to an incompletely defined finite state machine, then trace the network traffic and use reverse engineering to determine if the replicas' protocol matches the reference protocol, if not, a possible violation is detected. DiveInto is bound on reverse engineering technique, thus it can only correct violations of some common protocols.

Diversity System Diversity systems are security frameworks that utilize diversity to defend various attacks. As mentioned in Section 2.2, existing approaches of diversity system generally fall into two paradigms:

- Run multiple variants simultaneously; compare the result at each synchronization point.
- Distribute different variants to individual users to achieve attack prevention

N-Version Programming and N-Variant Systems [1, 66, 67] are the typical examples of first category. The N-version programming approach generates $N \geq 2$ functionally equivalent programs and compares their results to determine a faulty version, with metrics for measuring the diversity of software and fault [68, 69]. The main limitation of using diversity for fault tolerance lies in the high complexity of creating different versions, which may not justify the benefit [70]. The use of design diversity as a security mechanism has also attracted much attention [71].

Inspired by N-Version Programming, N-Variant systems is a security framework constructed by a polygrapher, monitoring multiple application variants. It implements two diversification methods: address space partitioning and instruction set tagging. It also gives a formal definition of properties: normal equivalence and detection, and their instances under two diversification methods.

Babak Salamat et al. proposed *Orchestra* [27] with similar idea, it is designed as a fully functioning Multi-Variant Execution Environment (MVEE). The framework of Orchestra is similar to N-Variant systems, with a diversification engine, a monitor and multiple variants. It introduced a notion: *synchronization point*, which is instantiated by invocation of a system call in this work, to evaluate if the variants are in conforming states. Formal definition of the conditions each synchronization point should hold are also given. This paper evaluated the effectiveness of Orchestra with stack-based buffer overflow exploits. The variants are diversified by *reverse stack execution*. Result shows Orchestra can effectively defend this kind of vulnerabilities with acceptable performance overhead. In another work by Babak Salamat et al. [72], the problem of false positives/negatives is also discussed. Race condition may occur when third party trying to manipulate

a file under synchronization, cause divergence between variants and false positives. While the diversifications are based on stack-based techniques, other attacks like heap-based buffer overflows cannot be detected and cause false negatives. The inaccuracy in detection is one of the major problems in MVEE approaches.

Anh Nguyen-Tuong et al. [73] also proposed a diversity system approach using the N-Variant system framework. They formally re-defined some of the key attributes of the previous work, including normal *equivalence* and *detection*. Concrete examples of *reexpression function* are also given regarding common diversification techniques. The effectiveness of the model is evaluated with UID corruption attack, in which an attacker corrupts a data value that causes the original program to execute maliciously [74], evaluation result is promising. It also shows for specific types of attacks, high assurance security can be obtained by low entropy data diversity such as UID data variation.

For the second paradigm, *E unibus pluram* [28] is a typical example. It utilizes a diversification engine, a “multicompiler", to generate unique but functionally equivalent applications for the users, so specific attacks only succeed on a portion of the targets. One of the major advantages is it can effectively prevent attacker from generating attack vectors by reverse engineering. Compared to similar work of this paradigm, this work emphasizes on practical massive-scale availability by introducing online software delivery, reliable compilers and cloud computing. It also discussed the application update problem and the need of trusted delivery system.

Another work falling in this paradigm is [75], which mitigates worm attacks on sensor networks by assigning different versions of applications to different nodes in the network. Instead

of randomly assigning the diversified applications, it treats the assignment as a graph coloring problem, which ensures no two adjacent nodes in the graph share the same color. By solving this problem, it can achieve better isolation between adjacent variants while using only a limited number of diversified applications. Typically, four diversified applications are enough to fulfill the demand.

Moving Target Defence [2] thoroughly describes the idea of using diversity systems to improve software security. It first gives the definition of *attack surface*, and shows how diversity can add uncertainty to the attack surface and make attacker's attempt more difficult. Typical diversity techniques on software are introduced, including randomization and reodering of instruction set, heap layout, stack base, system call number, register, library entry point and program base address etc. By introducing different binary compiling methods, variants of the same application can be automatically generated. It also proposed the concept of synchronization point in order to measure the equivalence of different variants and provide different granularity. The synchronization point includes system calls, arguments of the calls and time. Diversification methods on web applications are also introduced. Instead of binary code diversification, the diversification is on application, web server, and operating system layer. Though there is not many variants on each layer, each application randomly chooses a variant on each layer, as result, randomization of the whole system is sufficient. E unibus pluram [28] is proposed in later chapter of the book, as the implementation of the moving target defence concept.

To evaluate the effectiveness of these diversity systems, researchers have proposed different formalized approaches. A probabilistic based approach is given in [9]. It discussed the roles of

redundancy and diversity for security and proposed a diversity system effectiveness measurement based on effectiveness of IDS on individual variant and the covariance between variants. However, this is a rough and imprecise approach since the parameters of the measurement equation are practically unknowable. In fact, there is no accurate measurement on the effectiveness of security in diversity systems to date. In our work, we try to prove the effectiveness by completing an exhaustive search in CVE vulnerability database, which is shown in Chapter 4.2.

3.3 Existing Techniques of Query and Database Comparison

In this section, we will review some of the existing work on SQL query string and database schema comparison, which is introduced in Section 2.4.1 and Section 2.4.2.

3.3.1 Query String Comparison

A SQLIA detection technique is proposed in [3]; a standard safe query statement for the web application is first generated with known safe user input string, such as “AAA”. Detection is achieved by parsing standard SQL statements, runtime statement and comparing their syntax tree structure. Two SQL statements are considered semantic equivalent if their syntax tree structures are equivalent. If runtime statement fails to be semantic equivalent to its related standard statement, it is considered to be a possible SQLIA.

Context-sensitive string evaluation(CSSE) [52] has more fine grained analysis. By instrumentation of the platform, query strings are separated into user-provided data and developer-provided

data before being sent to database server. Only user-provided data would be examined as it is considered to be untrusted. By using the context of untrusted output string fragment and intercepted API calls, syntactic content inside is either escaped or the execution is prevented.

The work in [76] has similar ideas; query string is divided as hard coded string, string implicitly created by programming language and string originated from external source. Similar to CSSE, only string originated from external source is considered to be untrusted. Syntax evaluation is applied to this part and the query will only execute if the pattern matching has positive result.

The work in [32] tends to achieve prevention of more general injection attack through analyzing the parse tree of query strings. A standard grammar of specific application is pre-defined. User input section of query string is specially marked with random symbol and processed with augmented grammar, the query will only be executed if it complies the syntactic constraints.

3.3.2 Database Schema Comparison

Similarity Flooding [77] utilizes similarity propagation to build a hybrid matching algorithm. It uses directed labeled graphs to represent database schemas. A string-based comparison of vertices labels is performed to obtain initial alignment. With the idea of similarity spreading from similar nodes to adjacent nodes through propagation, in the following iterations, similarity of nodes is computed till they reach a fix-point. The refined alignment would be filtered and becomes the matching result.

Cupid [4] also uses graphs to represent schema. Schema matching of Cupid takes three phase to complete. The first phase is linguistic matching, it computes linguistic similarity coefficients

between element labels through morphological normalization, categorization and various string-based techniques. The second phase is structural matching; structural similarity coefficient is computed based on the context of schema elements and their structural relationship. The final phase computes the weighted similarity coefficient and determines the final matching result according to the threshold reached.

S-Match [35] is a similar hybrid schema matching approach. Particularly, S-Match codifies schema element labels written in natural language into propositional formulas, which turns the matching problem into propositional unsatisfiability problem. With the use of propositional satisfiability deciders, the problem can be resolved. S-Match is designed as a highly modular system which can suitably integrate different element level and structural level schema matchers.

Chapter 4

The Case Study

In this chapter, we first give a motivating example to show how diversity can improve security of a specific web application. We then try to prove diversity can prevent attackers from exploiting most existing vulnerabilities through an exhaustive search on vulnerability database.

4.1 A Motivating Example

Midicart [78] is an online shopping cart application with ASP and PHP versions. We demonstrate how diversity may help detecting injection attacks through an example based on one of the SQL injection vulnerabilities on Midicart PHP version and show this vulnerability does not exist in the ASP version.

The ASP Application Line 20 of "search_list.asp" of the Midicart ASP version reads:

```
search=request.Querystring("searchstring")+request.fo
```

```
rm("searchstring")
search = Replace(search, "'", "''")
search = Replace(search, "/", "//")
...
set rs=conn.execute("SELECT * FROM products where ``& chose & " LIKE '%" &
search & "%' ORDER BY maingroup, secondgroup, code_no")
```

As the code shows, the “searchstring” parameter in file “search_list.asp" is filtered by two “Replace" functions, with “’" and “/" escaped, such that an attacker will not be able to launch an injection attack by enclosing the apostrophe.

The PHP Application However, in the equivalent code section of the Midicart PHP version, the “searchstring" parameter in “search_list.php" is not filtered at all, as shown below.

```
$searchstring=$_REQUEST["searchstring"];
$chose=$_REQUEST["chose"];
...
$result = mysql_query("select * from products WHERE
$chose LIKE '%$searchstring%' ORDER BY 'maingroup','secondgroup', 'code_no'
LIMIT 0, 100 ") ;
```

Running Two Versions Together If we can run both the ASP and PHP version in parallel, an attacker would be unable to use this specific SQL injection vulnerability to compromise both variants, since the attack can only succeed in PHP version.

To launch an injection attack exploiting this vulnerability, an attacker will attempt to inject raw SQL statement into the “search" box, e.g., through the following.

```
http://www.example.com/search_list.php?chose=item&searchstring=asus%' UNION
SELECT null,CreditCard, ExpDate, null,null,null,null,null FROM card_payment
where PaymentMethod LIKE '%visa
```

For the PHP version, this will generate the database query:

```
select * from products WHERE item LIKE '%asus%' UNIONSELECT null,CreditCard,
ExpDate, null,null, null,null,null FROM card_payment where PaymentMethod
LIKE '%visa%' ORDER BY 'maingroup','secondgroup','code_no' LIMIT 0, 100
```

As the first apostrophe is enclosed by injected apostrophe, "union select" will be executed; consequently, the attacker would be able to retrieve the unauthorized credit card information from the "card_payment" table.

On the other hand, in the ASP version, the query is:

```
SELECT * FROM products where " item " LIKE '%asus%'' UNION SELECT null,
CreditCard, ExpDate, null,null, null,null,null FROM card_payment where
PaymentMethod LIKE ''%visa%' ORDER BY maingroup, secondgroup, code_no
```

As the user input apostrophe is replaced by double apostrophe, the injection code would be treated as a normal string. That is, the database will not execute the union select SQL command and the attacker will not be able to obtain any unauthorized information.

Lesson Learned This specific attack example shows that when the same user input induces different, or more precisely, "not equivalent" behavior on different variants of the diversity system, it

potentially signals an attack . Therefore, we can use different versions of the same web application to construct a diversity system, using the different behaviors on the variants to detect possible attacks.

However, this is an example showing one vulnerability of a specific application does not exist on the other version of the same application, making detection possible. In other words, it only shows the possibility of employing diversity to defend against injection attacks. But to prove the general applicability of this idea, we need to show, for other applications with different versions, most of their vulnerabilities do not exist on other versions. In order to show this, we perform an exhaustive search on known vulnerabilities in Section 4.2.

4.2 Exhaustive Search in CVE Database

The above motivating example only demonstrates that diversity may help security in one specific case. To expand this into a general idea, we perform an exhaustive search among all injection vulnerabilities appearing in well-known vulnerability databases. Although these databases certainly cannot cover all existing vulnerabilities, it is "comprehensive with respect to all publicly known vulnerabilities and exposures" [17]. The goal of this exhaustive search is to confirm the hypothesis that different versions of the same Web application [1] rarely share common injection vulnerabilities, and hence the opportunistic diversity will be useful for detecting injection attacks.

Specifically, we first search for web applications that have different versions, then search for known vulnerabilities of these applications. If most of these vulnerabilities do not exist in other

[1] Here by different versions of the same Web application, we mean versions written in different script languages; we do not consider different upgraded versions since they likely have less diversity.

versions of the same application, we can demonstrate that diversity can help defending attacks related to these vulnerabilities.

4.2.1 Preparation of Exhaustive Search

First problem of the exhaustive search is determining the data source. The most common way to find as much vulnerabilities as possible is through vulnerability databases. Several well-known vulnerability databases such as National Vulnerability Database (NVD) [30], the Open Source Vulnerability Database (OSVDB) [79] and Common Vulnerabilities and Exposures (CVE) are candidates, we choose CVE as our data source. The background introduction of CVE database is given in Section 2.3.

The second problem comes to the definition of *different versions of the same application*. It is common for web applications to have different versions, but the differences between them is generally insignificant, and thus made them not appealing to be used in diversity approach. As result, we need these different versions to be written in different script languages to provide sufficient diversity. In the rest of the paper, we use the term *variants* to represent web applications written in different script languages that can run in parallel and provide similar services.

4.2.2 Searching Applications with Variants

In the following exhaustive search, we

- First list applications involved with vulnerabilities we are interested in

- Then find applications with variants among the result.

Listing the Vulnerabilities Following the motivating example, we focus on SQL injection vulnerabilities here. We use keyword "SQL injection" to search in the master copy of CVE. Totally 5870 vulnerability entries are found; these entries contain all the SQL injection vulnerabilities exist in CVE. Sample vulnerability entry is as follows:

CVE-2012-5912	Multiple SQL injection vulnerabilities in PicoPublisher 2.0 allow remote attackers to execute arbitrary SQL commands via the id parameter to (1) page.php or (2) single.php.

Generally, it contains the information of vulnerability type, application name, location of the vulnerability (parameter name and file name). Therefore, we can map each entry to a Web application and find common entries shared by different variants based on the entries.

Searching for Applications with Variants Considering the large amount of vulnerability entries (5870), it would be very time-consuming to perform the search manually. Fortunately, following observations allow us to conduct the search in a semi-automatic fashion.

- First, the majority of involved Web applications are written in four script languages, ASP, PHP, JSP, and ASP.NET; applications written in other script languages rarely have a variant appearing in the list.
- Second, since each CVE vulnerability entry contains the file name in which this vulnerability occurs, the file extensions, such as ".asp",".php",".jsp", and ".aspx" will indicate the script language.

These allow us to group the involved Web applications into four categories, according to the script languages they are written in.

The grouping result is summarized in Table 3. The number of PHP applications is estimated as being over 1000, and the reason we do not include them will be clear shortly.

Application Language	ASP	JSP	ASP.NET
The Number of Applications	488	42	30

Table 3: Grouping of Applications

As it shows, JSP and ASP.NET applications are the minority among nearly 6000 SQL injection vulnerability entries (30 to 40 of each). ASP applications have a relatively larger number, 488. But PHP applications are the majority, estimating over 1000.

As the results show, the four categories have very different population sizes. This observation helps us to determine an optimal search order among them. That is, we should begin with a category with less applications. In addition, we observe that applications written in JSP are less likely to have variants written in other languages. On the other hand, since ASP.NET is an improved version of ASP, many ASP.NET applications would have an older ASP version, in which case they generally do not have a corresponding PHP version. Finally, the most common case would be either ASP or ASP.NET applications with a PHP variant. With such considerations and observations, we complete the search in three steps.

With this knowledge and the statistical data, we complete the search in following three steps:

- Step 1: List all ASP.NET applications, search for their ASP, PHP, JSP variants.

- Step 2: List all JSP applications, search for their ASP, PHP variants.

- Step 3: List all ASP applications, search for their PHP variants.

In this searching, we use the name of the applications as keyword and search for its vulnerability entries in CVE, so the results are not limited to SQL injection vulnerabilities only. After the results are listed, we then search for file extensions:".asp", ".php", ".jsp", ".aspx" in these entries. If more than one file extensions exist, the application may have variants and will be considered as candidate. We will verify it manually.

The Result As we finished this part of search, there are totally 16 applications in CVE database that has at least one variant. They are listed in Table 4.

- In Step 1, we found five ASP.NET applications with variants, all variants are written in ASP.
- In Step 2, none is found.
- In Step 3, we found 12 PHP applications with variants, all variants are written in ASP as well.

Note there is an overlap between these steps for application "DVBBS", which has three variants, in ASP, PHP, and ASP.NET.

4.2.3 Matching Common Vulnerabilities

For the 16 aforementioned applications, we need to determine whether two variants of the same application have similar vulnerabilities, which is denoted as "common vulnerabilities".

We give the definition of common vulnerabilities, which is based on equivalent parameters.

Definition 3. Equivalent parameters *Given parameter m in application A and parameter n in application B, we say m and n are* equivalent parameters *if the following condition is satisfied:*

Application/SQL injection vulnerability entries	ASP	PHP	ASP.NET	JSP
Active Bids	o	o		
BlogMe	o	o		
Brooky eStore	o	o		
Dvbbs	o	o	o	
fipsGallery	o	o		
Innovative CMS (ICMS, formerly Imoel-CMS)	o	o		
Jbook	o	o		
MaxCMS/PIPICMS	o	o		
MidiCart	o	o		
myNewsletter	o	o		
Pre Classified Listings	o	o		
WmsCms	o	o		
Absolute News Manager(.NET)	o		o	
Active Price Comparison	o		o	
WebEvents (Online Event Registration Template)	o		o	
Xigla Absolute Banner Manager (.NET)	o		o	

Table 4: Web Application with Variants and SQL Injection Vulnerability

- *A and B are variants of the same application.*
- *m and n receive input value from equivalent input field.*

The definitions of equivalent parameters is based on the assumption that user had designate the same input to the parameters in different variants according to the same functionalities. For most cases of building a diversity system, this process has to be done manually.

Definition 4. Common vulnerabilities *Given SQLIA vulnerability x involved with parameter m and vulnerability y involved with parameter n, we say x and y are* common vulnerabilities *if m and n are equivalent parameters.*

Among these 16 web applications, there are totally 34 SQL injection vulnerability entries in CVE, which are listed in the Appendix. Approximately, each application has two entries on average. Each application has at least one entry (Brooky eStore, etc.), and at most four entries (Pre

Classified Listings). With this relatively small amount of entries per application, it is easy to search for common vulnerability between variants.

In practice, the existence of equivalent parameters can be categorized into four cases:

1, Same parameter name and same file name (ignoring extensions).

2, Same parameter name and different file names.

3, Different parameter names and same file name.

4, Different parameter names and different file names.

These cases are listed from most common to most uncommon. Most equivalent parameters between variants still share the same name.

For example, in our motivating example given earlier, the “searchstring” parameter is an equivalent parameter in above Case 1 (same parameter names and same file names), but the vulnerability is not common, since it is only applicable to the PHP version.

Matching Common Vulnerabilities Since the common vulnerability is based on equivalent parameters, to match common vulnerabilities among the result of Section 4.2.2, we need to search for equivalent parameters related to these vulnerability entries. In order to simplify the task of matching common vulnerabilities, we first assume Case 1 for all equivalent parameters, such that we can automatically search for equivalent parameters by names. We then manually verify the results, and if Case 1 does not apply, we will manually find the equivalent parameter matching other cases.

In the end, we only find common vulnerabilities in one application named “Midicart", as detailed below.

CVE-2006-6209 (ASP)	CVE-2005-1503 (PHP)
Multiple SQL injection vulnerabilities in MidiCart ASP Shopping Cart and ASP Plus Shopping Cart allow remote attackers to execute arbitrary SQL commands via the (1) id2006quant parameter to (a) item_show.asp, or the (2) maingroup or (3) secondgroup parameter to (b) item_list.asp. NOTE: the code_no parameter to Item_Show.asp is covered by CVE-2005-2601.	Multiple SQL injection vulnerabilities in MidiCart PHP Shopping Cart allow remote attackers to execute arbitrary SQL commands via the (1) searchstring parameter to search_list.php, the (2) maingroup or (3) secondgroup parameters to item_list.php, or (4) code_no parameter to item_show.php.

Table 5: Common Vulnerabilities of Midicart

In both variants, "maingroup", "secondgroup", and "code_no" are all equivalent parameters in Case 1 (same parameter name and same file name). Attackers can exploit such vulnerabilities in both variants, and hence they cannot be detected through diversity alone (note, however, we do not rely on diversity alone for detection in this paper). For all other applications, no common vulnerability is found.

Those findings confirm our previous hypothesis that different variants of the same application rarely share common vulnerabilities, and hence opportunistic diversity may indeed assist the detection of attacks. On the other hand, the case of common vulnerability shown above indicates that opportunistic diversity by itself may not be sufficient for this purpose. Therefore, we will combine opportunistic diversity with anomaly detection in the rest of the paper.

4.2.4 Summary

By completing the exhaustive search on CVE vulnerability database, we:

- Searched for nearly 6000 SQL injection vulnerability entries related to 2000 web applications.

- Found 16 applications that have variants written in other languages with 34 vulnerability entries.
- Defined the term equivalent parameters and common vulnerabilities.
- Found common vulnerabilities in two vulnerability entries, related to one application.

There are some limitations in our results. The 16 applications we found may not cover all applications with variants in CVE database. However, for those applications with variants we missed, they would not have common vulnerabilities in CVE database, since only one version of them has vulnerability entry.

Thus the search result still fulfills our expectations. Only two out of 34 vulnerability entries have common vulnerability pairs belonging to variants of the same application, which proves our previous claim: for most SQL injection vulnerabilities, they do not exist in their variants. The low rate of common vulnerability existences indicates diversity can significantly improve applications' security, preventing attackers from exploiting most vulnerabilities in individual applications.

With the effectiveness of improving security through diversity demonstrated, an actual model of utilizing diversity for attack detection can be proposed. We will discuss this in Chapter 5.

Chapter 5

The Methodology

In this section, we present our diversity-based attack detection system. Specifically, we will describe the details of multiple stages comparison among variants and the final result production.

5.1 Overview

To employ opportunistic diversity for preventing attacks, we monitor the interaction between a user and multiple variants of an application together with their database backends. Anomaly detection is performed based on features extracted from different stages of such interaction. Partial results obtained at different stages and from different variants are combined through a learning-based approach to reach a decision of whether allowing the result to be returned to the user.

Our attack detection model is constituted by three major components: a controller, a monitor, and multiple variants with their database backends. Figure 6 illustrates the architecture of the model. We will briefly explain each of these components in this section and their details would be

given later.

As Figure 6 shows, the user interacts with one variant of the application as usual. The controller, monitor, and other variants are transparent to the user. The controller is responsible for extracting user inputs from the first variant and distributing them to the other variants. The monitor extracts features from different stages of the data flow, conducts anomaly detection, and finally combines partial detection results to reach a final decision. Based on the decision, the controller will either allow the first variant to return the result to the user, or deny it and return nothing to the user.

Arrows between different components in Figure 6 represent the input and output dataflow. Note that each monitoring process is based on each user input instead of each generated SQL query; it starts when the application receives an user input, and ends when the output related to this user input is produced. Thus although there is only a single pair of dataflow arrow between each application variant and database, in practice it is possible that multiple queries are induced by a single user input.

5.2 The Controller

The controller works as a medium between user and application variants with two major modules: the user input distribution module and output unifying module.

The working process of the controller is shown in Figure 7: when it receives a user input, it properly distributes that data to equivalent parameters of all variants under its control. If no attack is detected by monitor, the controller will return an unified result back to the user, which can be

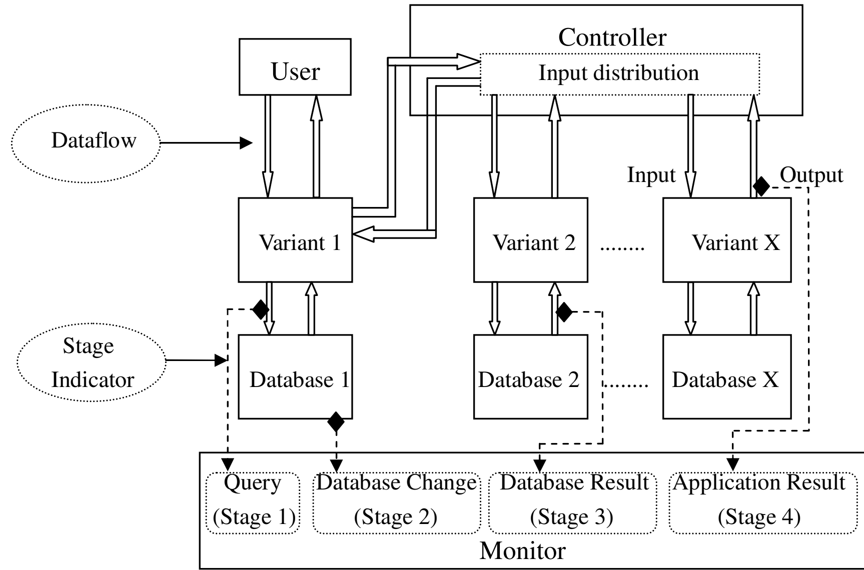


Figure 6: The Model

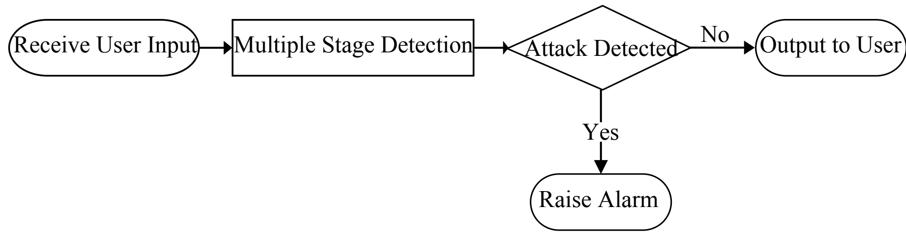


Figure 7: The Working Process of the Controller

done by choosing one of the results from the variants. Otherwise, attack alarm would be raised and the output is blocked.

5.3 The Monitor

The monitor is the component in charge of attack detection in our model. It first compares the behavior between variants in multiple stages, then combines results of multiple stages to a final detection result. Alarm would be raised if the final detection result indicates an attack.

Multiple Stages of Comparison Our approach does not solely depend on examining SQL queries to detect attacks as most existing work do. Having multiple stages of detection can produce a more precise detection result; it can also utilize more aspects of diversity employed in our approach. As mentioned above, the major differences among variants in our system is web applications written in different languages and use different databases. Thus besides monitoring the SQL queries, we can also expect different behaviors in terms of database activities and the output results.

As result, we use these four stages of monitoring:

- SQL queries generated by application.
- Changes in database.
- Query result returned by database.
- Result returned to the user by applications.

Each stage has one or more *features* to score the differences in details. For example, we use a feature “tree edit distance of abstract syntax tree obtained from query" in the first stage to score structural difference of the queries. Features of each stage would be explained separately later in the subsections.

We expect to detect certain difference in one or more of these stages among variants when the user input is purely attack-free. But when an attack is performed, larger difference generally appears on multiple stages. On the other hand, even when an attack is exploiting common vulnerability, we may still detect noticeable difference on certain stages. Thus introducing multiple stages of comparison is effective on mitigating both false positives and false negatives.

Methodology of Comparison The intuitive way of detecting different behavior among variants is to directly compare the result of each stage among variants. However, in some applications, this may not be a practical approach. Although the variants are functionally similar, sometimes the inherent differences between them can make direct comparison infeasible. For example, one variant may use several queries for a service, but the other variant may use stored procedure for the same service. In this case, direct comparison on the first stage, query string, would generate large amount of false positives.

Different from many existing work, we do not compare directly the features extracted from different variants for detecting attacks, but to compare the partial anomaly detection results obtained based on such features. The reason is twofold. First, as we have shown through the case study, opportunistic diversity alone may not always be sufficient for detecting attacks. Second, although different variants are functionally equivalent, sometimes they may exhibit significant differences in terms of implementation details, which renders a direct comparison infeasible.

We approach this problem by first performing learning-based anomaly detection at each variant, and then compare the detection results between different variants. Specifically, for features that have significant differences among variants, we first follow the learned-based method proposed in [6] and introduced in Section 2.4.4 to establish an anomaly detection model for those features by learning secure profiles through training with attack-free data. The runtime features are then compared to the learned profiles to obtain anomaly detection scores. This *anomaly detection phase* will produce scores in uniform formats for different variants, which can then be compared in the next *diversity detection phase* to further improve the detection accuracy.

Finally, the comparison result from each stage would be correlated by decision making tools to produce a final detection result. In this approach, we use decision tree learning to produce the final result.

Compared to the previous approach in [6], our approach has the following contributions:

- We do not detect attack from SQL queries only, but also from the detection result of other stages.
- We correlate the detection result of multiple stages to produce a more accurate detection result.
- We use application variants to employ diversity in our approach, so that the result does not base solely on anomaly detection. This further reduces false alarm rate.

The following sections provide a detailed description of each stage. We have chosen a small number of features and left enriching the collection of features as a future work. The details of final result production with decision tree will be introduced in Section 5.4.

5.3.1 Stage 1: SQL Query

At this stage, we utilize the opportunistic diversity originated from different source code writing of Web applications among different variants. Specifically, when different variants filter user inputs in different ways, the queries generated for the same user input may vary. This can happen in two cases as follows.

- The value of a parameter is filtered properly by one variant but not filtered at all by the other. An attacker can attack the vulnerable variant by simply injecting raw SQL string. For the other variant, the special characters in the injected string, such as apostrophe, will be escaped and the attack becomes ineffective.

- The parameter is filtered by both variants, but one of them filters it in a vulnerable way, e.g. allowing encoding or decoding after filtering. An attacker can still inject a SQL string with encoded special characters to bypass the vulnerable filtering scheme so the special characters can be restored once passing the filtering. For the other variant, the encoded special characters will not be restored which makes the attack ineffective.

In both cases, the query produced from the same malicious input becomes different among variants. Namely, the one without proper filtering becomes effective injection code, the other remains secure.

Example 4. *Suppose a login action generates the query*

```
SELECT * FROM admin WHERE username = 'XXX' AND password = 'YYY'
```

Suppose the "username" parameter in variant A is not filtered, and variant B will escape any user input apostrophe to "%2527". Assume an attacker injects "username" with string ' or '1'='1;.

1. *In variant A, the attack is successful with query*

   ```
   SELECT * FROM admin WHERE username = ' ' or '1'='1' AND password = 'YYY'
   ```

2. *In variant B, the attack fails with query*

```
SELECT * FROM admin WHERE username = '%2527 or %25271%2527=%25271' AND
password = 'YYY'
```

□

Example 4 belongs to the first case of improper input filtering. As it shows, with different input filtering, injection attack can produce different SQL queries, which makes it detectable in Stage 1.

Features for Detection

In this stage, we utilize the diversity at the query string level to detect injection attacks. Therefore, the features for detection in this stage should be able to characterize the difference between normal SQL queries and malicious queries.

Intuitively, using string models, like the length and character distribution model suggested in [6] is good for measuring significant string level deviation for the queries. However, we find through our case study that, while such string models are good candidates for the anomaly detection phase, they are usually ineffective for the diversity detection phase, since different variants are usually implemented with very different queries, and such differences can easily outweigh the difference between attacks and normal queries.

We approach this problem in the structural level of the SQL query, while leaving other features as future work. Before the SQL queries being executed in database, they are first parsed and represented as *Abstract Syntax Tree(AST)*, in which the nodes represent their structural characters. While normal user input usually generates similar ASTs, the ASTs yield from malicious input can have a lot of structural deviation from the normal ones. If such deviation is detected on one variant

but not others, it indicates possible injection attack.

Consequently, we compare different variants based on three features.

- The first feature is *the edit distance of ASTs* [80]. In the anomaly detection phase, we calculate the *tree edit distance* between runtime ASTs and corresponding ASTs inside the previously learned profiles. In this approach, we employ a tree distance metric, *Robust Algorithm for the Tree Edit Distance(RTED)* [80], to measure the difference between ASTs. A higher value of tree edit distance indicates the runtime query has larger structural deviation from query in profile.

  Next, in the diversity detection phase, the calculated edit distances are compared across different variants to produce the final score for this feature.

- The second feature is *the list of involved tables*. Most SQL parsers can retrieve the name of database tables involved in a query while parsing it into AST. If the involved tables do not match the tables in corresponding profiles, it is a significant sign of injection attacks. We use a binary score to describe the result from each variant. “0" means involved tables in runtime queries are identical to those in corresponding profiles; “1" means otherwise.

  The binary score from each variant is added up together to produce the final score for this feature. Thus if the final score is 1, it means one variant’s runtime query involves different tables than profile and vice versa.

- The third feature is *the number of non-parsed queries*, which is utilized directly in the diversity detection phase. A SQL parser can only output ASTs from queries with a legitimate

grammar and cannot work with incorrect grammars. In many injection attack scenarios, the queries will likely have incomplete parenthesis or apostrophe, and thus cannot be parsed.

Since this feature will always lead to a zero value in the training phase, the anomaly detection phase may be omitted and we can directly compare the feature across different variants to produce the final score.

We give an example of SQLIA that can be detected in this stage.

Example 5. *The standard AST generated from the queries in Example 4 is shown in Fig 8.*

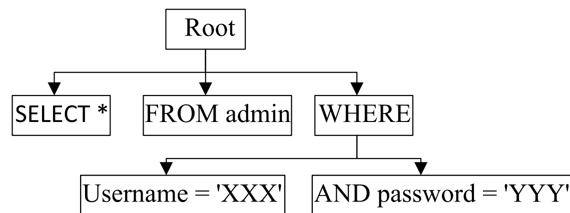


Figure 8: Standard AST of Example 4

The AST generated from user input in variant A and B is shown in Fig 9.

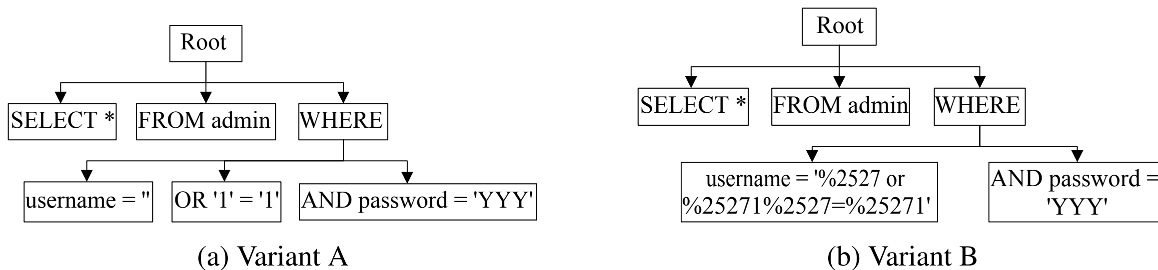


Figure 9: The Generated ASTs from Both Variants of Example 4

The AST in Fig 9(a) has 3.0 tree edit distance with standard AST, while the AST in Fig 9(b) has 0 distance with standard AST. As it shows, ASTs are able to characterize queries with different semantic structures.

⊡

Limitations For some malicious input, we may still observe benign results from the features of Stage 1 because of, but are not limited to, the following reasons:

- The related input filtering of all variants are improper and thus have common vulnerabilities.
- The query after injection is mistakenly matched with wrong profile.

To detect attacks that cannot be detected solely based on SQL queries, we introduce following three stages.

5.3.2 Stage 2: Changes to Database

In this stage, we utilize the diversity of different databases among the variants, which may arise due to three factors as follows.

- Unique characteristics of specific database products. For example, many stored procedures or commands are proprietary to one database product, and may not exist, or are under different names in other databases. The names of data types in different database products may also vary.
- Different requirements for enclosed bracket or parenthesis. For example, some variants require a pair of enclosed brackets or parentheses for certain input fields, while others do not have such requirements.

- Different database schemas (e.g., table names and attribute names) in different variants' databases. In many case, the difference lies in the prefix or suffix of the name.

When SQL injection is performed, attacker usually is required to enclose the apostrophe or parenthesis at the beginning of the injection code. If one variant puts a left parentheses("(") before the field the attacker injects, while the other variant does not have the parentheses. The injected query can only run on one variant successfully, since there would either be unclosed left parentheses on one variant or a surplus right parentheses on the other variant.

Furthermore, SQL injection codes often have specific names inside, whether it is database table name or column name. Since the database design may be slightly different between variants, an injection code of this kind may lead to different behaviors among the databases.

Example 6. *Suppose the parameter "username" causes an injection vulnerability on both variants due to the lack of proper filtering. In such a case, diversity among queries (Stage 1) will not help to detect the attack. The following shows how the attack may be detected at Stage 2.*

1. *Suppose in variant A, the query is*

```
SELECT * FROM admin WHERE( username = 'XXX')
```

2. *In variant B, the query is*

```
SELECT * FROM admin WHERE username = 'XXX'
```

Assume the attacker injects the "username" parameter with string `');drop table admin-`

1. *In variant A, the query becomes*

```
SELECT * FROM admin WHERE( username = ‘’);drop table admin--
```

2. *In variant B, the query becomes*

```
SELECT * FROM admin WHERE username = ‘’);drop table admin--
```

Clearly, the query in variant B will not be executed because of the extra right parenthesis.

⊡

As both examples show, when detection in Stage 1 fails, some attacks can cause different behaviors in database among variants, which make them detectable in Stage 2.

Features for Detection

The detection features of this stage are aimed for monitoring application variants making different changes to databases. Our first intuition is to use the database schema similarity score, such as the existing approaches to database schema matching [4,33,77], to measure the difference between databases at runtime and previous saved database schema. However, in our study, we found that simpler features may also be sufficient, because most legitimate activities only involve selection queries and do not modify the database schema. Thus more simplified features are sufficient.

Consequently, we use these two features to evaluate the difference among variants in Stage 2.

- The first feature is *the existence of changes to database schema*. This feature is directly compared among variants. Since changes to database schema are very uncommon for legitimate activities, we use a binary score for this feature. If any modification is applied to a variant's database schema, the score would be “1”; otherwise, it is “0”.

The binary score from each variant is added up together to produce the final score for this feature. Thus if the final score is 1, that means one variant's database schema is changed and so on.

- The second feature is the *number of modified records*. This feature is also directly compared among variants. It records the number of records in a database that are changed as a result of the queries. In contrast to the previous feature, this feature is more fine-grained and will more likely record a non-zero result. However, in practice we found that for legitimate queries, they usually change same or similar number of rows in database between different variants.

  Therefore, the number of rows can be compared directly across variants to produce the final score of this feature. A higher value of final score indicates a bigger deviation among variants on number of rows changed in database.

Example 7. *For the attack in Example 6, the first feature would score "1" since database schema is only changed on one of the variants. The second feature would have a score equal to the size of table "admin", because "admin" table is deleted by variant A but it stays intact for variant B.*

⊡

5.3.3 Stage 3: Database Result

Stage 3 is complementary to Stage 2 by utilizing the same diversity in databases but, instead of monitoring different changes made to databases, it is based on the results returned by the database to the application.

As mentioned in Stage 2, because of different databases used, the same injection string can be successfully executed on some variants but not others. If a malicious query does not have result set, difference among variants can only be observed by the change of database (Stage 2). However, if a result set exists, we can also observe difference in the result set among variants. The reason for difference in result set is similar to Stage 2, including unique characteristics of database products, enclosed parentheses, apostrophes handling, and customized names in databases.

Example 8. *Suppose attacker injects "username" parameter in the application variants same as Example 6 with following string, both variants have table named "products":*

```
') union SELECT * FROM products WHERE 't' = 't
```

The injected query will only succeed in variant A, but not variant B because of the surplus parenthesis. As result, the output result set of variant A contains all the rows from table "admin" and "products", while in variant B the query is not executed and there is no result set at all.

⊡

As Example 8 shows, some injection attacks that lead to different behavior in the database may not change the database itself. However, we can monitor this difference by comparing the result set among variants.

Features for Detection

The features of this stage should be able to characterize differences among result sets. Typical database result set is shown below, which is identical to a database table. Each column has a unique type of data and each row is one record.

	Column 1	Column 2	...	Column X
Row 1				
Row 2				
...				
Row X				

Since a result set under the relational model is mostly a relation, we can use similar features as introduced in Stage 2 but apply them to the result set relation instead of database relations. Consequently, we use these two features in Stage 3, in respect to the "database schema" and "number of records" features of Stage 2.

- The first feature is the *type of data in result set* (which is slightly different from the "database schema" feature in Stage 2). This feature will be used in the anomaly detection phase. The data type of each column of the runtime result set is compared to those in the learned profiles. We use a binary score to represent the result at each variant to indicate whether the data type of a column matches the profile. If the data type of any column does not match with the profile, the score would be "1", otherwise, it is "0".

  The binary score from each variant is added up together to produce the final score for this feature. Thus if the final score is 1, that means one variant's data type of result set deviates from the profile and so on.

- The second feature is the *number of rows in result set* (similar to the "number of records" feature in Stage 2). Since a "SELECT" query would return different numbers of records depending on the "WHERE" clause, with the same user input, different variants will likely return a similar number of rows in the result set.

Thus the final score of this feature is produced by directly comparing the number of rows among variants. A higher value indicates bigger difference.

Example 9. *For the attack in Example 8, the first feature would score "1" since variant B does not have a result set, and thus the data type does not match the profile. The second feature would score the number of the rows of record in table "admin" and "products", since the result set of variant A has this number of rows while variant B has zero rows.*

⊡

5.3.4 Stage 4: Application Result

In this stage, we utilize the diversity in the result returned to user by applications. In most cases, the application result is a HTML page.

We introduce this stage in order to monitor the distinct behavioral difference among variants. As mentioned in previous stages, injection attacks often involve enclosed apostrophe and parenthesis problems, which leads to an non-executable SQL query. Despite that AST production of Stage 1 can also spot SQL query that cannot be executed, checking application result can add some tolerance in our detection model. A HTML page with error message is usually returned to user in this situation. Apparently, it has big difference with HTML page that contains normal result.

Example 10. *Suppose the SQL queries in variant A and B are the same as Example 6, attacker inject "username" parameter with following string:*

```
') UNION SELECT * FROM products
```

Suppose the table "products" has zero record on both variants, we cannot observe any difference from features of Stage 2. However, the HTML page returned to user in variant A is a page with empty result. In variant B, it would be an error page since the related query cannot be executed.

⊡

As Example 10 shows, some attacks can lead to different HTML pages returned to user, which makes them detectable in Stage 4.

Features for Detection

To measure difference between HTML pages, many features are selectable, including size of the page, title of the page, etc. However, the detection accuracy of fine grained features in this stage may largely depend on specific applications. If the inherent difference among variants of the application is big, using these features for detection can be very inaccurate. On the other hand, focusing on monitoring error message in HTML page is sufficient for detecting most difference caused by attack in this stage.

Consequently, we use one feature in Stage 4:

- The feature is *existence of error message*, this feature is directly compared among variants since the profile obtained from attack-free data usually has no error message. We check the existence of multiple error messages that would not normally appear in regular use in runtime pages. A binary score is used for recording the result. If error message exist on the page, the score would be "1", otherwise, it is "0".

Stage	Feature	Type
Stage 1	Edit distance of ASTs	Anomaly
	List of involved database tables	Anomaly
	Number of not parsed queries	Direct comparison
Stage 2	Changes to database schema	Direct comparison
	Number of modified records	Direct comparison
Stage 3	Type of data in result set	Anomaly
	Number of rows in result set	Direct comparison
Stage 4	Existence of error page	Direct comparison

Table 6: Attack Detection Features of each Stage

The binary score from each variant is added up together to produce the final score for this feature. Thus if the final score is 1, that means the web page returned to user on one variant has error message and so on.

Example 11. *For the attack in Example 10, the final score of this feature is "1", since error message is only spotted on variant A.*

⊡

5.4 Result Correlation

We list all the features for attack detection on each stage in Table 6, along with their types (anomaly or direct comparison). Note that these are only features we selected for attack detection, and there are other candidate features in each stage that might be more efficient in certain practices.

Although each feature in Table 6 can produce detection results separately, correlating such results will further improve the detection accuracy and reduce false alarms. Consequently, we correlate the result of each stage to make final decision on attack detection.

Decision Tree Learning Approach The partial detection results obtained at different stages may be correlated in many ways for better detection accuracy. In this work, we employ decision tree learning to correlate scores of different features to make decision of attack detection.

As we reviewed the basic concept of decision tree in Section 2.4.4, it can be used to find the best strategy of using various attributes to partition the input data into certain classes. To be exact, the strategy is to determine the priority of attributes in terms of decision making. Namely, attributes with closer relevance to data partitioning have higher priority and would be the upper nodes in the decision tree.

Figure 10 illustrates our approach to correlating partial results using decision tree learning. Features at each stage are used as attributes for the decision tree, and training data are collected for different user inputs. For each user input, the anomaly detection phase and the diversity detection phase together will produce a result table as depicted in the figure. Those tables are used to build a decision tree, which will be applied to runtime inputs to classify them into two classes: "attack" or "normal".

Decision Tree Learning Algorithm We choose the *C4.5 Algorithm* [81] as the decision tree learning algorithm in our approach. Compare to its predecessor *ID3 (Iterative Dichotomiser 3) Algorithm* [82], C4.5 has several advantages including: allow continuous attributes and missing attribute values, and the trees can be pruned after creation.

The attribute splitting criterion of C4.5 is the normalized *information gain*, whose definition is given in Section 2.4.4. The attribute with highest information gain is chosen to make decision in every recursive run. The data set is partitioned into sublists and the C4.5 algorithm then recurses

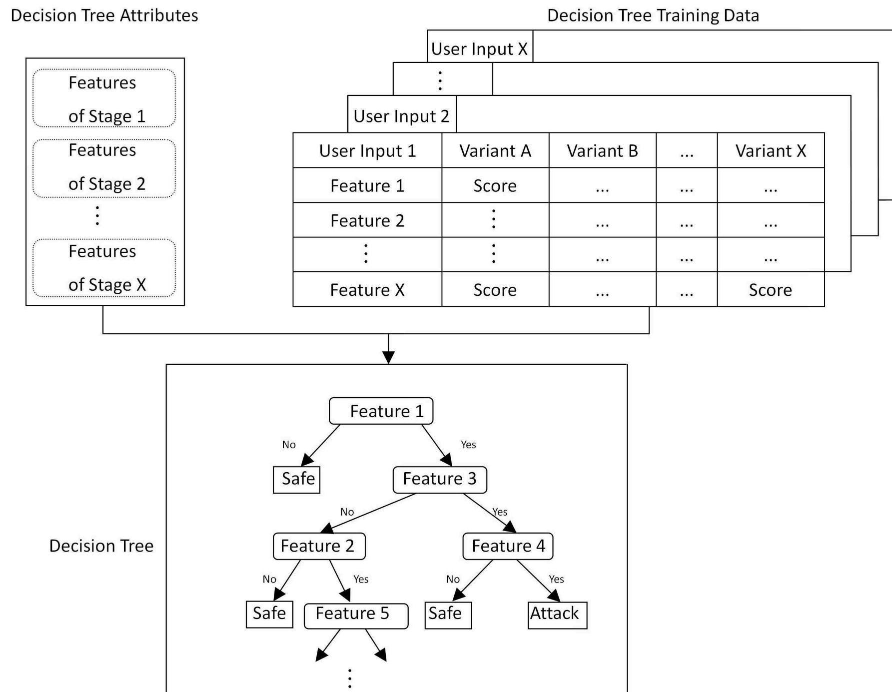


Figure 10: Final Decision Making

on them.

The Attributes As mentioned previously, the score of eight features across four stages are used as attributes for decision tree. For the features whose final score is computed by adding up binary scores in each variant, we can treat them as discrete attributes, since there are a limited number of values. For example, if the system has two variants, possible score values of this kind of features are 0, 1 and 2.

For the other features, their scores are treated as continuous attributes. We can take advantage of the C4.5 algorithm, which is able to handle both continuous and discrete attributes. The continuous attributes are discretized with best partition thresholds automatically by the C4.5 algorithm.

Training Data Set In order to have an accurate detection result. The training data for decision tree learning need to thoroughly cover most situations. We use three types of training data in this approach.

- Attack-free user input, which is the normal request from user. Large amount of attack-free data can be obtained by scripts that simulate user activity, e.g. thread spam bots for forum application and auto-registration bots.
- Attack input that will succeed in all variants; this can be obtained by attacking the common vulnerabilities of all variants.
- Attack input that will succeed in at least one, but not all variant, this can be obtained from attacks that exploit independent vulnerability in one of the variants.

The Result Classes In this approach, the data is partitioned into two classes: Attack or Attack-free.

Cross Validation We follow the *10-fold cross validation* [83] to produce the final decision tree. The training data is randomly partitioned into ten folds and the decision tree training process is recursed ten times. In each run, nine folds of data are used for training the decision tree. Then we validate its accuracy by applying the trained tree to the remaining one fold of data. At last, the decision tree with best accuracy among ten produced trees is selected as final result.

5.5 Summary

In this chapter, we:

- Proposed an attack detection model utilizing diversity in multiple stages.
- Gave attack examples that can be detected on certain stage.
- Selected detection features for all four stages and methods of result score production are given.
- Correlated result from multiple stages with decision tree learning to produce final detection result.

Employing diversity gives our attack detection model significant advantages over traditional anomaly detection approaches, since our detection does not solely rely on learned attack-free profiles. By comparing anomaly detection results across variants, we can reduce certain amount of false alarms.

Using multiple stages of comparison also gives our model advantage over previous diversity detection approaches that only use a single feature for detection. By correlating detection results in multiple stages, we can further detect attacks that cannot be detected by a single feature, which reduces false negatives. Meanwhile, certain attack-free data may be classified as attacks by certain features, thus correlating multiple features can also help to reduce false positives.

As the attack detection model is given, we will evaluate our approach on actual web applications. This will be discussed in Chapter 6.

Chapter 6

Implementation and Experiments

In this chapter, we implement our proposed attack detection model on a real web application and perform experimental detection using decision tree learning. We evaluate our approach by comparing our detection model to several other options.

6.1 Model Implementation

6.1.1 Multi-Stage Comparison

In this section, we will describe the detailed implementation of each stage separately, including challenges we encountered and corresponding solutions.

Stage 1

As described in Section 5.3.1, for Stage 1, we need to monitor the runtime SQL queries of all variants, generate ASTs for the queries and compute the tree edit distances. We implement this

event_time	user_host	thread_id	server_id	command_type	argument
2014-03-05 13:37:47	pma[pma] @ localhost [127.0.0.1]	1390	1	Query	SELECT MAX(version) FROM \`phpmyadmin\`.\`pma_trackin...
2014-03-05 13:37:47	pma[pma] @ localhost [127.0.0.1]	1390	1	Query	SELECT tracking_active FROM \`phpmyadmin\`.\`pma_trac...
2014-03-05 13:37:47	root[root] @ localhost [127.0.0.1]	1391	1	Quit	

Figure 11: Mysql Log

EventClass	TextData	LoginName	SPID	StartTime	BinaryData
65534	NULL	NULL	NULL	10:07.9	NULL
13	SELECT DISTINCT maingroup FRO	mon	54	15:28.2	NULL
13	SELECT DISTINCT secondgroup F	mon	61	15:28.3	NULL
13	SELECT DISTINCT secondgroup F	mon	61	15:28.4	NULL
13	SELECT DISTINCT secondgroup F	mon	61	15:28.4	NULL
13	SELECT DISTINCT secondgroup F	mon	54	15:28.4	NULL
13	SELECT * FROM products where	mon	54	16:36.7	NULL

Figure 12: SQL Server Log

stage with following steps:

- Extract runtime SQL queries from applications.

  1, For Mysql database, we use the event log function to save runtime queries to “general_-log" table in “mysql" database. Fig.11 shows the data of this table.

  2, For SQL Server database, we use Microsoft SQL Profiler to create a trace and save runtime queries to a user designated table. Fig.12 shows the data of this table.

  3, On both variants, we use a unique user name in their connection string to the database. This is to distinguish queries originated from our web application and others.

  4, We only monitor the records in the log belongs to type “query”. For Mysql, “command_type” field need to be “Query”. For SQL Server, “EventClass” field need to be “13” or “10”, which are the code for type “SQL:BatchStarting” and “RPC:Completed”

respectively.

5, We use time as a factor to distinguish queries from different actions in the log. We found that the logged queries aggregated in short time intervals generally belongs to the same action. The time interval we set is 3 seconds, which is appropriate for our experiment. In a heavier load application, the time interval can be set smaller. However, this method certainly has limitations when the web application has simultaneous actions.

- Generate ASTs, calculate edit distance and retrieve table names from SQL queries.

1, We use the tool "General SQL Parser JAVA" [84] to parse the retrieved queries into ASTs. Example 12 shows the AST format produced from it.

Example 12. *We parsed the following SQL query into AST with "General SQL Parser JAVA".*

```
SELECT * FROM products where text LIKE '%asus%' and code_no>0
```

The parsed AST is represented by:

```
SELECT *
FROM   products
WHERE  TEXT LIKE '%asus%'
       AND code_no > 0
```

Each row represents a node in AST, the spacing at beginning of the a row represents the hierarchy of the node.

□

2, We use the JAVA program from *Robust Algorithm for the Tree Edit Distance(RTED)* [80] to calculate the edit distance between ASTs. However, this tool only accepts bracket notation of trees.

We wrote a function to transform the AST format from "General SQL Parser JAVA" (as shown in Example 12) to bracket notations, according to the rows and spacing at beginning of each row. For example, the AST in Example 12 would be transformed into:

```
{x{null}{null}{null{null}}}
```

Note that "x" is a manually added root node to the AST; it does not affect edit distance calculation.

3, We use the "getTable()" function in "General SQL Parser JAVA" to retrieve database table names involved in the queries, this is done simultaneously with AST generation.

4, Since SQL parser cannot parse stored procedure into AST, for certain types of stored procedure that is commonly seen in the application we used, we wrote a function to extract SQL query out of them.

Example 13. *A stored procedure belongs to "sp_cursoropen":*

```
exec sp_cursoropen @p1 output,N'SELECT * FROM products
where code_no = '1006'',@p3 output,@p4 output,@p5 output
```

The SQL query extracted from it would be:

```
SELECT * FROM products where code_no = '1006'
```

$\boxdot$

5, For the other queries that cannot be successfully parsed into ASTs, we keep a record of their numbers.

The Scores As we use two variants in our experiment, the score of features in Stage 1 are calculated by following method.

- Edit distance of ASTs:

  Suppose variant a has m queries and variant b has n queries for the same action, Da_i and Db_i are the edit distance of AST between the ith runtime query and its corresponding profile on two variants respectively. The score of this feature is defined as:

$$Score = |\sum_{i=1}^{m} Da_i - \sum_{i=1}^{n} Db_i|;$$

- List of involved database tables:

  Suppose Da and Db are the anomaly score of this feature in variant a and b respectively, it values "1" if the involved database tables of runtime query are different from corresponding profile; otherwise values "0". The score of this feature is defined as:

$$Score = Da + Db;$$

- Number of not parsed queries:

Suppose Da and Db are the number of not parsed queries in variant a and b respectively. The score of this feature is defined as:

$$Score = |Da - Db|;$$

Stage2

As described in Section 5.3.2, for Stage 2, we need to monitor the change of databases when runtime queries are being executed. We implement this stage with following steps:

- To monitor the change of database at runtime, we first set up a table recording the initial state of the database. The information we record includes: name of the table, number of records in the table and schema of the table. They are used as "reference" for future detection. These information is retrieved by:

  1, For Mysql, we retrieve the information from "tables" table in database "information_-schema".

  2, For SQL Server, we use the query "*select * from sysobjects where xtype = 'u'*" to retrieve the information.

- Since we already retrieved the involved database table names of queries, during runtime, we use the same method to retrieve current information of these tables, and compare it to its reference, so that any change would be detected.

1, For the number of records, we compare the number of changed rows in database between variants.

2, For database schema, we first compare the schema of individual tables, then compare the total number of tables in database.

The Scores We evaluate the change of database with two features:

- The number of modified records.

  Suppose variant a has m involved tables and variant b has n; Da_i and Db_i are the runtime number of rows in the ith involved tables of two variants respectively, Sa_i and Sb_i are the reference row count. Score of this feature is defined as:

$$Score = ||\sum_{i=1}^{m} Da_i - \sum_{i=1}^{m} Sa_i| - |\sum_{i=1}^{n} Db_i - \sum_{i=1}^{n} Sb_i||;$$

- Changes to database schema.

  Suppose if database schema of variant a is not changed, Da is defined as "0"; otherwise is "1". Db is its counterpart in variant b. The score of this feature is defined as:

$$Score = Da + Db;$$

Stage3

As described in Section 5.3.3, for Stage 3, we need to monitor the result from database. We implement this stage with following steps:

- At initialization of the detection system, we create a duplicate database for each variant which is the same as the one used by variant. During runtime, queries executed in original databases will also be executed in the duplicate databases. Thus we can retrieve result set of executed queries from the duplicate ones.

- We save the data type of result set as a string and compare to corresponding profile. The number of records in result set would be directly compared between variants.

The Scores We evaluate the database result set with two features:

- The number of rows in result set.

  Suppose variant a has m queries and variant b has n; Da_i and Db_i are the number of records for ith query in two variants respectively. The score of this feature is defined as:

$$Score = |\sum_{i=1}^{m} Da_i - \sum_{i=1}^{n} Db_i|;$$

- Type of data in result set.

  Suppose there are m result sets in variant a and n result sets in variant b. If the ith result set's data type of variant a are same as corresponding profile, Da_i is defined as "0", otherwise is "1". Db_i is its counterpart in variant b. The score of this feature is defined as:

$$Score = |\sum_{i=1}^{m} Da_i - \sum_{i=1}^{n} Db_i|;$$

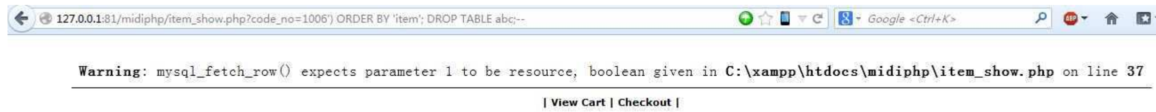


Figure 13: An Error Web Page Example

Stage 4

As described in Section 5.3.4, for Stage 4, we need to monitor the result returned by application. Namely, the web page returned to user by application.

In this experiment, we use a keyboard and mouse mimic software "Keyboard Simulator" [85] to script a "save-to-file" action. This script is added at the end of each data training script(which would be discussed in Section 6.2.2), so that the web page is automatically saved to a ".txt" file.

Since we use "existence of error page" as feature for Stage 4, after we obtained the txt file, we will check for multiple error information in the file, such as the warning information shown in Fig 13.

This method may not be practical in practice. We can either instrument the application or take an alternative approach similar to Stage 3: setting up a duplicate browser on server side which would act exactly the same as user. We would be able to retrieve the web page from the server side browser.

The Scores The score of feature for this stage, "existence of error page", is defined by:

If error web page exists in variant a, Da is defined as "1", otherwise is "0". Db is its counterpart is variant b:

$$Score = Da + Db;$$

6.1.2 Anomaly Profile

In this section we will discuss the setup of profiles for anomaly features in our model and the profile matching method.

Profile Setup As described in Section 5.3, some of the features used in our model are anomaly features, which requires to setup some profiles so that the runtime values can be compared with. These anomaly features include:

- Edit distance of ASTs from SQL query, Stage 1.
- Involved tables of SQL queries, Stage 1.
- Data type of result set, Stage 3.

We can use the same feature score calculation method as introduced in Section 6.1.1 to setup the profile. The difference is, in Section 6.1.1 the final score is computed by combining the feature score of each variant, while in profile setup, we save the feature score in separate profiles for each variant.

Table 7 shows an example of the profile.

Profile ID	AST	Involved Table	Data Type	Profile Length
0	{x{}{}{}{{}{}}}	products	12 12 12 12 -1 12 12 12	1
1	{x{}{}{}}	products	12	2
1	{x{{}}{}{}{}}	products	12 12	2

Table 7: Anomaly Profile Example

- Profile ID: The unique id number of each profile.

- AST: Bracket notation of the AST parsed from SQL query.

- Involved Table: Names of the database table involved in a query.

- Data Type: Data type of all columns in the result set; each number represents a unique data type.

- Profile Length: Number of records for the profile. For example, profile with ID “1” in Table 7 has two records.

We use attack-free data that covers most regular actions of the application to generate the profile. The profile setup stops when no profile with new AST is generated. Then the produced profile would be used for later detection.

Profile Matching For anomaly detection, we need to match runtime result to the profile. We accomplish profile matching in two steps:

- Select profile from the learned profiles which has the same number of queries with retrieved ones.

- Match runtime ASTs to selected profiles, the profile with least cumulative AST edit distance with runtime data would be selected.

If no profile is matched with, the system would use the first profile by default.

The details of profile matching are given in the algorithm below:

Algorithm 1 Profile Matching

Retrieve runtime queries which executed in limited time interval as $runtime_query$.
for each $profile \in profile_list$ **do**
  if $runtime_query$ has same number of records as $profile$ **then**
    $profile \in candidate_profile$
  end if
end for
for each $profile_i \in candidate_profile$ **do**
  for each $runtime_query \in runtime_profile$ **do**
    for each $query \in profile_i$ **do**
      $curr_distance = AST(runtime_query, query)$
      if $curr_distance < min_distance$ **then**
        $min_distance$=$curr_distance$
      end if
    end for
    $total_distance_i = total_distance_i + min_distance$
  end for
end for
if $total_distance_i$ is $min(total_distance)$ **then**
  $runtime_query$ matched with $profile_i$
end if

6.2 Experiments

In this section, we use our implemented detection model to perform detection experiments on a real web application. We evaluate our approach by its detection performance in the experiments.

6.2.1 The Web Application and Vulnerabilities

The Application *Midicart* [78] is an online shopping cart application that has multiple versions. The interface of it is shown in Fig.14. In this implementation, we use Midicart ASP version with Microsoft SQL Server as database and PHP version with Mysql as database.

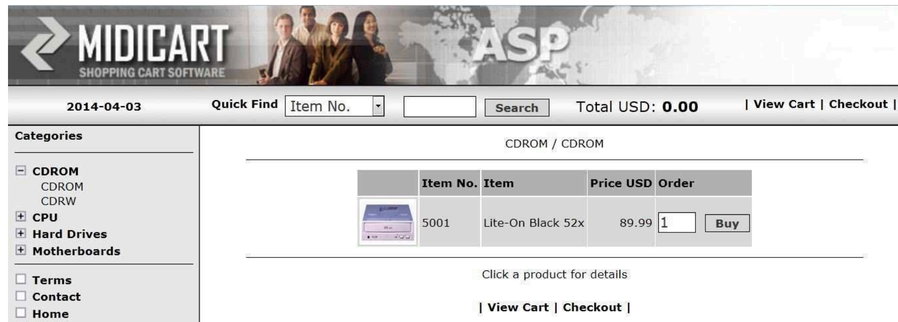


Figure 14: Midicart Interface

Vulnerabilities The reason we choose Midicart for implementation is because, according to our study of CVE vulnerability database in Chapter 4.2, this application has both common vulnerabilities and vulnerabilities existing only on one variant, which is good for demonstrating our proposed model's ability to detect attacks that exploits either kind of vulnerability. The CVE entries regarding Midicart are listed in Table 8.

CVE-2006-6209	Multiple SQL injection vulnerabilities in MidiCart ASP Shopping Cart and ASP Plus Shopping Cart allow remote attackers to execute arbitrary SQL commands via the (1) id2006quant parameter to (a) item_show.asp, or the (2) maingroup or (3) secondgroup parameter to (b) item_list.asp. NOTE: the code_no parameter to Item_Show.asp is covered by CVE-2005-2601.
CVE-2005-2601	SQL injection vulnerability in MidiCart allows remote attackers to execute arbitrary SQL commands via the code_no parameter to (1) Item_Show.asp or (2) search_list.asp.
CVE-2005-1503	Multiple SQL injection vulnerabilities in MidiCart PHP Shopping Cart allow remote attackers to execute arbitrary SQL commands via the (1) searchstring parameter to search_list.php, the (2) maingroup or (3) secondgroup parameters to item_list.php, or (4) code_no parameter to item_show.php.

Table 8: Midicart Vulnerability Entries in CVE

As Table 8 shows, "maingroup", "secondgroup", "code_no" parameter in multiple files have injection vulnerability on both ASP and PHP versions. However, "searchstring" parameter is only

vulnerable in PHP version, but not in ASP. We checked this parameter in file "search_list.asp" and confirmed it is properly filtered.

In order to run the detection experiment, we use the old version of Midicart that was produced in 2006, in which the vulnerabilities listed above exist.

Normal Activities For regular users(customers), the functionality of Midicart is relatively simple, normal activities of the application include:

- Browse main page.
- List commodities by their categories.
- Search commodities by item number or description.
- Register for payment.

6.2.2 Training Data

In this implementation, our attack training data is obtained by exploiting the vulnerabilities in Table 8. The attack-free data is obtained by performing the four types of normal activities introduced previously.

The Attacks

Attack 1. ***Vulnerability***:*"code_no" parameter in item_show.asp, item_show.php*

Is Common Vulnerability: *Yes.*

- ***Post Data1***:*?code_no=1006 ' UNION select * from products where code_no = '1005*

  Result1:*(ASP): SELECT * FROM products where code_no = '1006' UNION select * from products where code_no = '1005' ORDER BY item*

  *(PHP): select * from products where(code_no = '1006 ' UNION select * from products where code_no = '1005') ORDER BY 'item'*

- ***Post Data2***: *?code_no=1006 ' and '1' = '1*

  Result2:*(ASP): SELECT * FROM products where code_no = '1006 ' and '1' = '1' ORDER BY item*

  *(PHP): select * from products where(code_no = '1006 ' and '1' = '1') ORDER BY 'item'*

This attack is to exploit the non-filtered "code_no" parameter on both versions, the union select SQL injection of Post Data 1 can only succeed in one of the versions since attacker cannot construct an injection string which can enclose the parenthesis in both versions. However, tautology attack of Post Data 2 can succeed on both versions.

Attack 2. ***Vulnerability***:*"maingroup" and "secondgroup" parameter in item_list.asp, item_list.php*

Is Common Vulnerability: *Yes.*

- ***Post Data1***:*?maingroup=CPU&secondgroup=Socket-A' UNION SELECT null, null, maingroup, secondgroup,null, null,null,null FROM products where code_no='1001*

  Result1:*(ASP): SELECT * FROM products where maingroup = 'CPU' AND secondgroup =*

'Socket-A' UNION SELECT null, null, maingroup, secondgroup,null, null,null,null FROM products where code_no='1001' ORDER BY code_no

*(PHP): select distinct * from products WHERE maingroup = 'CPU' AND secondgroup = 'Socket-A' UNION SELECT null, null, maingroup, secondgroup,null, null,null,null FROM products where code_no='1001' ORDER BY code_no*

- ***Post Data2**: ?maingroup=CPU&secondgroup=Socket-A' and '1'='1*

  ***Result2**:(ASP): SELECT * FROM products where maingroup = 'CPU' AND secondgroup = 'Socket-A' and '1'='1' ORDER BY code_no*

  *(PHP): select distinct * from products WHERE maingroup = 'CPU' AND secondgroup = 'Socket-A' and '1'='1' ORDER BY code_no*

This attack is to exploit the non-filtered "maingroup" and "secondgroup" parameter in both versions, the union select SQL injection can succeed in both versions since this is a common vulnerability and there is no enclosed parenthesis problem.

⊡

Attack 3. ***Vulnerability**:"searchstring" parameter in search_list.asp, search_list.php*

***Is Common Vulnerability**: No.*

- ***Post Data**:asus%' UNION SELECT null, null,CreditCard, ExpDate,null, null,null,null FROM card_payment where posted LIKE '%111*

  ***Result**:(ASP): SELECT * FROM products where code_no LIKE '%asus%'' UNION SELECT null, null,CreditCard, ExpDate,null, null,null,null FROM card_payment where posted LIKE*

"%111%' ORDER BY maingroup, secondgroup, code_no

*(PHP): select * from products WHERE code_no LIKE '%asus%' UNION SELECT null, null,CreditCard, ExpDate,null, null,null,null FROM card_payment where posted LIKE '%111%' ORDER BY 'maingroup','secondgroup','code_no' LIMIT 0, 100*

This attack is to exploit the "searchstring" parameter in both versions. However, while PHP version has SQLIA vulnerability in this parameter, in ASP version this parameter properly escaped special characters like apostrophe. Thus the union select SQL injection can only succeed on PHP version.

As shown in the result, with proper input filtering, the ASP version would simply search for string: *"%asus%" UNION SELECT null, null,CreditCard, ExpDate,null, null,null,null FROM card_payment where posted LIKE "%111"*

⊡

Attack-free Data

To aquire sufficient amount of attack-free training data, we traverse all the functionalities of the Midicart application, both manually and through an automated tool. We use keyboard&mouse mimic software to create scripts which can simulate normal user activities.

The software we employ is "Keyboard Simulator" [85], we show a piece of script code created by this software below, in which the functionality is to open URL "http://127.0.0.1:81/midiphp/", select "code_no" tag and search string "1005".

```
[Script]
ProcessID=Plugin.Web.Bind("wqm.exe")
```

```
...
Call Plugin.Web.Tips("running")
Call Plugin.Web.SetSize(1366,784)
Call Plugin.Web.Go("http://127.0.0.1:81/midiphp/")
Call Plugin.Web.ScrollTo(0,0)
Call Plugin.Web.HtmlSelect("code_no","name:chose&frame:4")
Call Plugin.Web.HtmlInput("1005","name:searchstring&frame:4")
Call Plugin.Web.LeftClick(733, 90)
...
```

⊡

Each simulation script for data training contains URL opening actions on both variants and clicks on certain elements on the web page if applicable. Apart from these actions, we also attached a "save-to-file" action at the bottom of every script, so that the opened web page would be saved to a txt file. This is used for the implementation of Stage 4 as mentioned in Section 6.1.1.

Training Data Summary

Table 9 listed the summary of training data we used in the experiment with Midicart. Totally 608 data are tested, in which 54 (8.88%) are attacks. We believe this is an appropriate percentage for attacks in the dataset. Also, over 600 of training data should be sufficient for testing Midicart which has relatively simple functionalities.

Dataset	Attack 1	Attack 2	Attack 3	Attack-free	Total
Numbers	22	12	20	554	608

Table 9: Training Data Summary

6.2.3 Decision Tree Learning and Result Evaluation

As discussed in Section 5.4, our attack detection model uses decision tree learning to correlate multiple stages detection results. In this section, we will discuss the implementation of decision tree learning and evaluation of final detection results.

Decision Tree Learning

We use the standard C4.5 C program [81] for decision tree learning in our experiment. It requires a ".data" file to record the value of training data, and a ".names" file to record the attributes.

Since we have two variants, features that use binary score have limited values in their final score. Consequently, their attributes are set as 0, 1 and 2. Other features will be set as continuous so that C4.5 algorithm can automatically discretize them. We set the attributes for these eight features as follows:

- Tree edit distance of AST, {continuous}.

- Involved database table, {0, 1, 2}.

- Number of not parsed queries, {continuous}.

- Number of rows changed in database, {continuous}.

- Change of database schema, {0, 1, 2}.

- Type of data in result set, {continuous}.

- Number of rows in result set, {continuous}.

- Existence of error page, {0, 1, 2}.

Finally, the classification labels we have are: Attack and Safe.

Cross Validation We follow the ten-fold cross validation method [83] in our experiment with these steps:

1, Randomize the order of 608 training data and equally divide it into ten folds.

2, Run decision tree learning process ten times. In each run, one fold of training data is used as test data, while other nine folds are used for learning the decision tree.

3, The decision tree with the best accuracy for the testing data will be selected.

Result and Evaluation

Among all ten trees we learned from our training data, the one shown in Fig.15 has the best detection accuracy for the testing data. It uses four features from three stages as decision nodes:

1, Stage 1, tree edit distance of AST.

2, Stage 4, existence of error page.

3, Stage 1, number of not parsed queries.

4, Stage 3, type of data in result set.

Since both attacks and attack-free data in this experiment do not modify the database, no feature from Stage 2 (changes in database) is used.

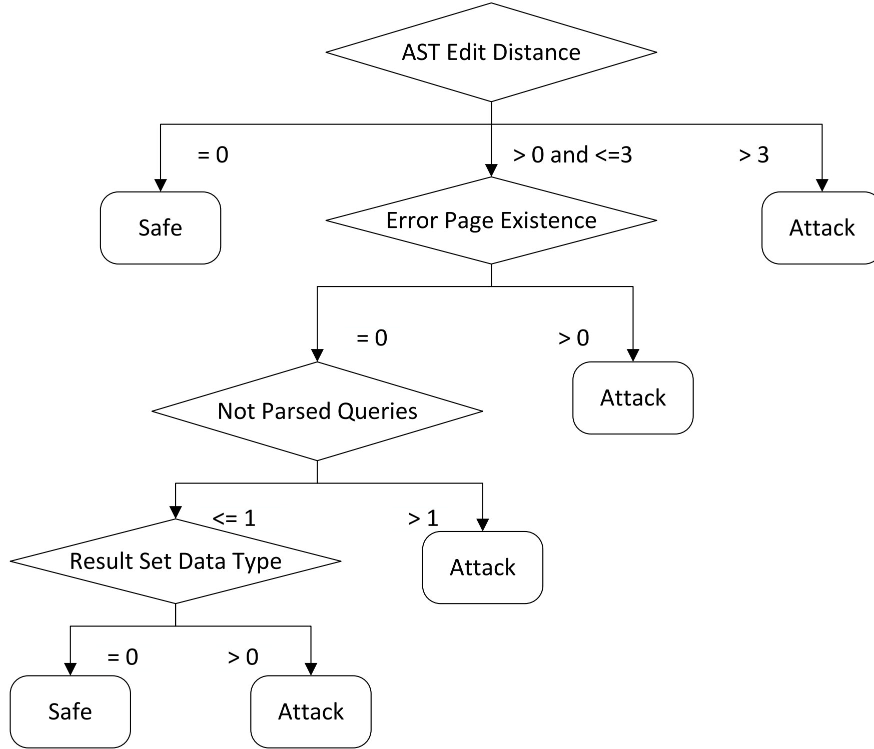


Figure 15: The Learned Decision Tree

We apply this decision tree to all the training data we have, and evaluate its detection performance through three benchmarks: *Accuracy (AC), False Positive Rate (FPR), False Negative Rate (FNR)*, which are defined as:

$$FPR = \frac{FP}{FP + TN}, \qquad FNR = \frac{FN}{FN + TP}$$

$$AC = \frac{TP + TN}{TP + TN + FP + FN}$$

Table 10 shows detection performance of the learned decision tree:

FP	FN	TN	TP	FPR	FNR	AC
0	14	554	40	0%	25.93%	97.7%

Table 10: Detection Performance of Decision Tree

As shown by the FPR in Table 10, the decision tree can classify all attack-free data in training data correctly. Simple functionality of application Midicart is the main reason of 100% FPR. We should be able to observe some FPs if implementing this approach in an application with more complex functionality or more inherent differences between variants. But this still shows our approach can effectively mitigate FPR.

We recorded 14 FNs totally, which means 14 out of 554 attack data are misclassified. Overall, the detection accuracy is 97.7%.

Analysis of FNR The FNR of 25.93% in the result is relatively high. However, note that we have chosen the Midicart application for evaluation, particularly because it has common vulnerabilities between different variants, as discovered in our case study. That is, this is essentially the worst case scenario in terms of FNs. For all other applications for which we have searched for injection vulnerabilities in CVE (totally around 2000), the amount of FNs will be significantly lower.

More than half of attack training data are from attacks exploiting common vulnerability. As mentioned in Section 6.2.2, we use Attack 1, 2 and 3 to generate attack training data, in which Attack 1 and 2 are exploiting common vulnerability while Attack 3 is not.

We revisit our training data and connect each FN to each of the three attacks. All 14 FNs come from Attack 1 and 2, which means 20 attack data generated from Attack 3 are all correctly

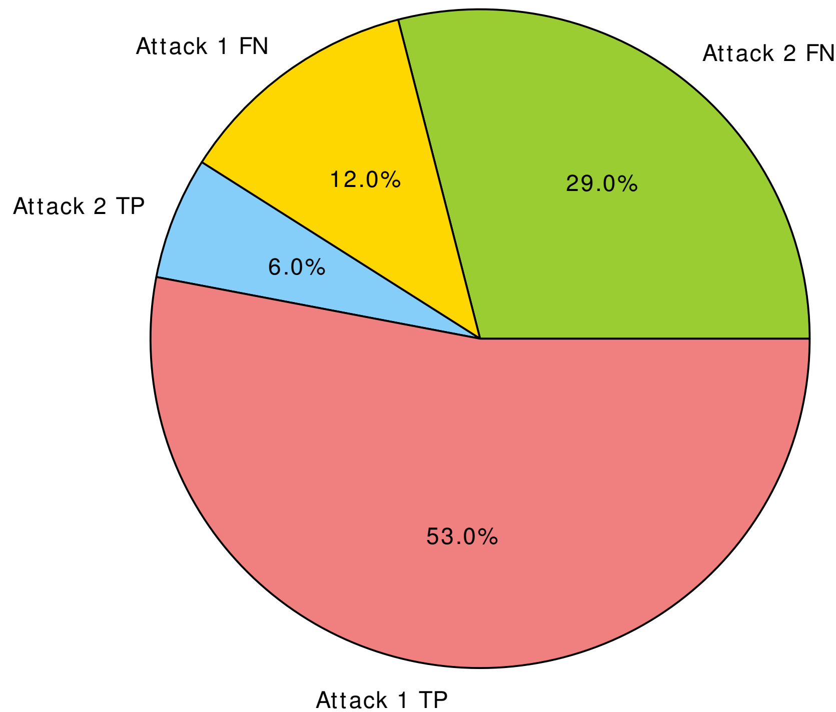


Figure 16: FPR and TNR for Detecting Attack on Common Vulnerabilities

classified. Attack 3 exploits the vulnerability that only exist on PHP variant. It shows by employing diversity, our approach has good detection performance on attacks exploiting this kind of vulnerabilities. Table 11 shows FN and TP numbers for Attack 1, 2 and 3.

Attack 1 FN	Attack 1 TP	Attack 2 FN	Attack 2 TP	Attack 3 FN	Attack 3 TP	Total
4	18	10	2	0	20	54

Table 11: Detection Performance of Attack 1 ,2 and 3

The decision tree can still detect a portion of attacks that exploit common vulnerability (Attack 1 and 2). As Fig 16 shows, 20 out of 34 attack data from exploiting common vulnerabilities can be correctly detected. The reason our approach can detect this kind of attacks include:

1, Some injection strings contain specific database elements that differs across variants, same

situation as Example 7, this happens on both Attack 1 and 2.

2, The unenclosed parenthesis problem, same situation as Example 6, this only happens on Attack 1.

Consequently, we can detect most attacks which belong to Attack 1, while only a portion of attacks in Attack 2 can be detected. Nevertheless, the ability of detecting attacks from exploiting common vulnerabilities demonstrates multi-stage detection's advantage over single stage.

Comparison with Purely Anomaly-based Approach In order to evaluate the improvement of our approach over traditional anomaly detection. We compared the detection accuracy of our approach to a purely anomaly-based detection that does not have diversity employed.

The feature we selected for anomaly detection is "AST edit distance" from Stage 1, since this feature has highest *information gain* during decision tree learning, which means it can "best" classify the data. We use the ASP variant for anomaly detection.

In the purely anomaly-based detection, we compute the edit distance between runtime AST and AST in corresponding profile. The PHP variant is not involved. We then set threshold to classify the data into "Attack" and "Safe". If the distance is lower than threshold, it is classified as "Safe", otherwise is "Attack".

The thresholds we used is 0.5, 3.5, 6.5, 9.5. Fig. 17 shows the detection accuracy using each of the thresholds, which ranges from 92.11% to 95.23%, using 3.5 as threshold has best accuracy. We can easily see that our approach has better detection performance(AC 97.7%) over anomaly-based detection. Note that, due to the relatively simple functionalities and small number of attack types,

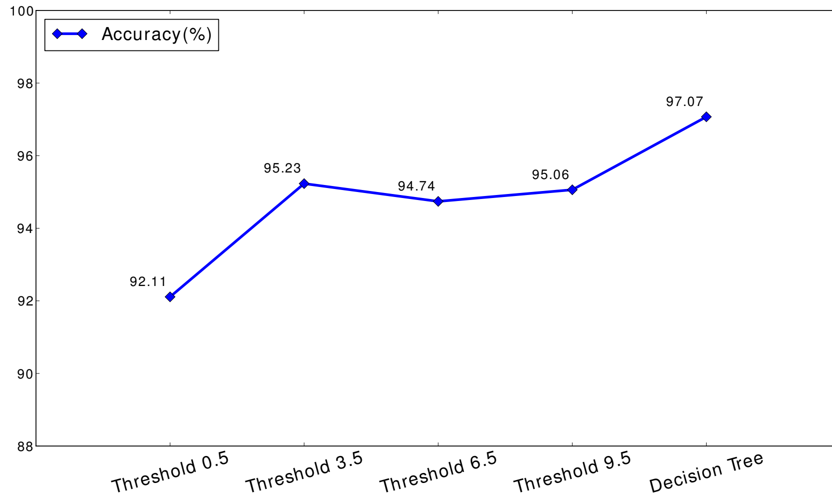


Figure 17: Detection Accuracy Comparison between Anomaly Approach and Decision Tree

the improvements may not seem very significant for this particular application.

Comparison with Single-Stage Diversity Detection Approach To evaluate the contribution of using multiple stages to detection accuracy, we compare our approach to detections using single stage.

The features we chose are those which were used as nodes in the learned decision tree, and a threshold is set for each feature. Training data is classified by comparing its score of the feature to the threshold. Note that diversity is involved here; training data's score is computed across variants, instead of just from ASP variant like the one in previous evaluation.

Table 12 shows the features and thresholds for single-stage detection.

Fig.18 and 19 show the FPR and FNR comparison across different single-stage detection and decision tree approach. Decision tree approach has the best FPR (0%); detection using AST edit distance with threshold 3.5 and detection using result set data type also has 0% FPR. If threshold of AST edit distance is set to 0.5, the FPR increased a bit to 1.62%. FPR of detection using error

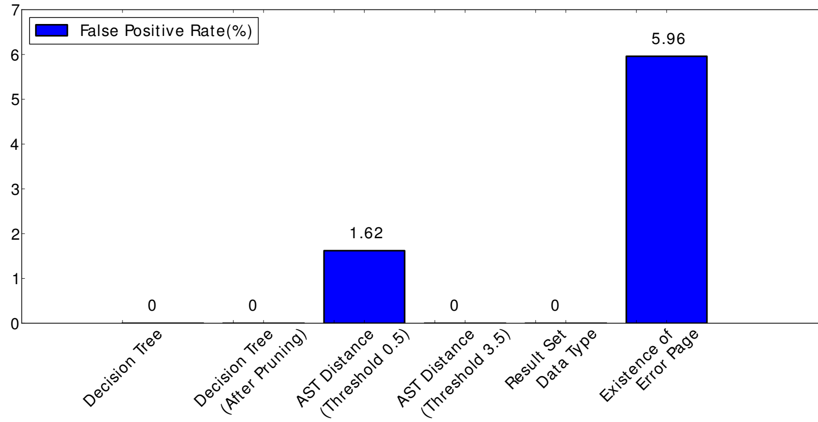


Figure 18: FPR Comparison of Detection with Single Feature and Decision Tree

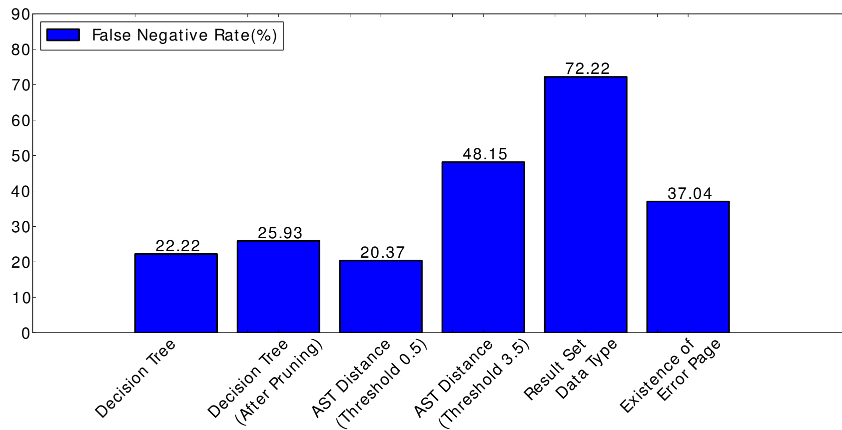


Figure 19: FNR Comparison of Detection with Single Feature and Decision Tree

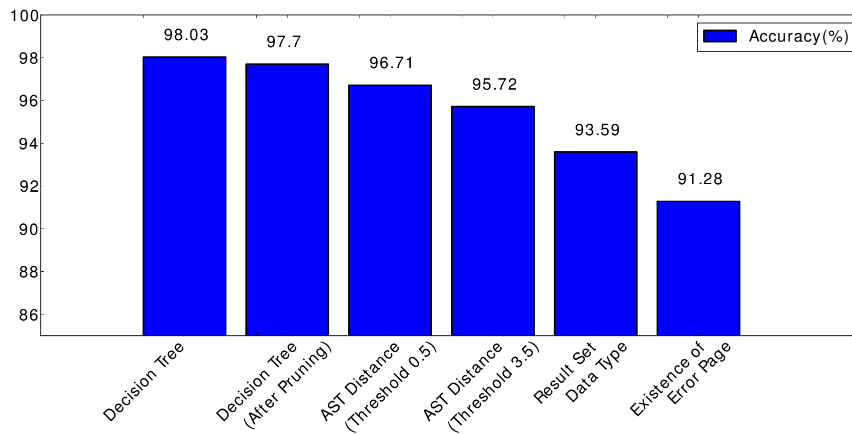


Figure 20: Detection Accuracy Comparison with Single Feature and Decision Tree

Feature	Threshold	Classification
AST Edit Distance	< 0.5	Safe
	> 0.5	Attack
AST Edit Distance	< 3.5	Safe
	> 3.5	Attack
Result Set Data Type	< 0.5	Safe
	> 0.5	Attack
Existence of Error Page	0	Safe
	1 or 2	Attack

Table 12: Features and Thresholds for Single-Stage Detection

page existence is relatively high (5.96%).

For the FNR, decision tree approach (25.93%) and AST edit distance with threshold 0.5 (20.37%) are at the same level. Using other three features would result a high FNR. The substantial FNRs are mainly affected by the amount of common vulnerability exploits in training data, as analyzed previously. We expect the FNR to drop if the percentage of common vulnerability exploits in training data is lower.

Overall, decision tree approach still has better detection accuracy than any other single-stage detection as shown in Fig.20. It proves correlating detection result from features of multiple stages can actually increase detection accuracy.

Moreover, comparing the detection accuracy of AST edit distance in Fig. 20 to the accuracy using the same feature in Fig.17, the one in Fig. 20 which uses two variants has higher accuracy than the one in Fig.17 which only uses ASP variant. This again proves employing diversity can help improving attack detection accuracy.

6.3 Summary

In this section, we:

- Described the problems and solutions on implementing each stage of our attack detection model.
- Collected training data and performed detection experiment on a real web application.
- Produced decision tree and evaluated its detection performance by comparing it with traditional anomaly-based approach and single-stage diversity detection approach.

The analysis of results from our experiments shows our approach has good detection accuracy overall. Detection with the produced decision tree has low FPR and FNR comparing to other approaches. It can detect most attacks that exploit vulnerability which only exists on specific variant, and can also detect substantial amount of attacks that exploit common vulnerability.

Comparing our approach to several purely anomaly-based detections, our approach has the best accuracy among them. This proves employing diversity can improve detection performance. Comparing our approach to detections with single feature, our approach also has the best accuracy among them. This proves correlating features of multiple stages can also improve detection performance. Overall, the evaluation of detection experiment result demonstrates the advantage of our approach.

Chapter 7

Conclusion

In this thesis, we proposed a multi-stage attack detection system employing opportunistic diversity.

Firstly, we evaluated the feasibility and effectiveness of this approach through a study in CVE vulnerability database. After a thorough search in CVE, we identified the web applications that have sufficient diversity among different versions, we also discovered the rare existence of common vulnerability in these applications. This proves feasibility and effectiveness of our approach.

Secondly, we designed the attack detection model, using four stages with eight features. We gave attack examples of each feature to show the type of attack it can detect; detailed score calculation for each feature is also introduced. Multiple features are correlated by decision tree learning, we use the produced decision tree to detect attack.

Finally, we evaluated our approach by implementing it on a real web application with two variants. The result shows our approach has good detection accuracy. We demonstrated the advantage of our approach over purely anomaly-based detection or single-stage detection with our lower

false positive rate, false negative rate, and thus better accuracy. The detection of exploits on common vulnerability also shows the advantage of using multiple stages. Overall, this implementation proves the contribution of our proposed detection system on previous approaches.

For future work, we will consider applying our approach to web applications with more complex functionalities. Furthermore, additional features can also help to enrich our proposed attack detection model.

Appendix A

CVE Entries for Applications with Common Vulnerability

Application	CVE Identifier	CVE Entry
Active Bids	CVE-2009-4229	Multiple SQL injection vulnerabilities in ActiveWebSoftwares Active Bids allow remote attackers to execute arbitrary SQL commands via (1) the catid parameter in the PATH_INFO to the default URI or (2) the catid parameter to default.asp. NOTE: this might overlap CVE-2009-0429.3. NOTE: the provenance of this information is unknown; the details are obtained solely from third party information.
	CVE-2009-0429	Multiple SQL injection vulnerabilities in Active Bids allow remote attackers to execute arbitrary SQL commands via the (1) search parameter to search.asp, (2) SortDir parameter to auctionsended.asp, and the (3) catid parameter to wishlist.php.
	CVE-2008-5640	SQL injection vulnerability in bidhistory.asp in Active Bids 3.5 allows remote attackers to execute arbitrary SQL commands via the ItemID parameter.

BlogMe	CVE-2008-2175	SQL injection vulnerability in comments.php in Gamma Scripts BlogMe PHP 1.1 allows remote attackers to execute arbitrary SQL commands via the id parameter.
	CVE-2007-2661	SQL injection vulnerability in archshow.asp in BlogMe 3.0 allows remote attackers to execute arbitrary SQL commands via the var parameter, a different vector than CVE-2006-5976.
	CVE-2006-5976	Multiple SQL injection vulnerabilities in admin_login.asp in BlogMe 3.0 allow remote attackers to execute arbitrary SQL commands via the (1) Username or (2) Password field. NOTE: some of these details are obtained from third party information.
Brooky eStore	CVE-2003-0585	SQL injection vulnerability in login.asp of Brooky eStore 1.0.1 through 1.0.2b allows remote attackers to bypass authentication and execute arbitrary SQL code via the (1) user or (2) pass parameters.
DVBBS	CVE-2009-4470	SQL injection vulnerability in boardrule.php in DVBBS 2.0 allows remote attackers to execute arbitrary SQL commands via the groupboardid parameter.
	CVE-2008-5222	SQL injection vulnerability in login.asp in Dvbbs 8.2.0 allows remote attackers to execute arbitrary SQL commands via the username parameter.
fipsGallery	CVE-2006-6117	SQL injection vulnerability in index1.asp in fipsGallery 1.5 and earlier allows remote attackers to execute arbitrary SQL commands via the which parameter.
Innovative CMS (ICMS, formerly Imoel-CMS)	CVE-2005-4397	SQL injection vulnerability in RunScript.asp iCMS allows remote attackers to execute arbitrary SQL commands via the Event_ID parameter.
JBOOK	CVE-2008-6391	SQL injection vulnerability in main.asp in Jbook allows remote attackers to execute arbitrary SQL commands via the username (user parameter).
	CVE-2008-6376	SQL injection vulnerability in main.asp in Jbook allows remote attackers to execute arbitrary SQL commands via the password (pass parameter).
	CVE-2006-1743	Multiple SQL injection vulnerabilities in form.php in JBook 1.4 allow remote attackers to execute arbitrary SQL commands via the (1) nom or (2) mail parameters. NOTE: the provenance of this information is unknown; the details are obtained solely from third party information.

MAXCMS	CVE-2009-1818	SQL injection vulnerability in admin/admin_manager.asp in MaxCMS 2.0 allows remote attackers to execute arbitrary SQL commands via an m_username cookie in an add action.
	CVE-2009-1764	SQL injection vulnerability in inc/ajax.asp in MaxCMS 2.0 allows remote attackers to execute arbitrary SQL commands via the id parameter in a digg action.
MidiCart	CVE-2006-6209	Multiple SQL injection vulnerabilities in MidiCart ASP Shopping Cart and ASP Plus Shopping Cart allow remote attackers to execute arbitrary SQL commands via the (1) id2006quant parameter to (a) item_show.asp, or the (2) maingroup or (3) secondgroup parameter to (b) item_list.asp. NOTE: the code_no parameter to Item_Show.asp is covered by CVE-2005-2601.
	CVE-2005-2601	SQL injection vulnerability in MidiCart allows remote attackers to execute arbitrary SQL commands via the code_no parameter to (1) Item_Show.asp or (2) search_list.asp.
	CVE-2005-1503	Multiple SQL injection vulnerabilities in MidiCart PHP Shopping Cart allow remote attackers to execute arbitrary SQL commands via the (1) searchstring parameter to search_list.php, the (2) maingroup or (3) secondgroup parameters to item_list.php, or (4) code_no parameter to item_show.php.
myNewsletter	CVE-2008-1295	SQL injection vulnerability in archives.php in Gregory Kokanosky (aka Greg's Place) phpMyNewsletter 0.8 beta 5 and earlier allows remote attackers to execute arbitrary SQL commands via the msg_id parameter.
	CVE-2006-2887	Multiple SQL injection vulnerabilities in myNewsletter 1.1.2 and earlier allow remote attackers to execute arbitrary SQL commands via the UserName parameter in (1) validatelogin.asp or (2) adminlogin.asp.
Pre Classified Listings	CVE-2010-1370	SQL injection vulnerability in detailad.asp in Pre Classified Listings ASP allows remote attackers to execute arbitrary SQL commands via the siteid parameter.
	CVE-2010-1369	SQL injection vulnerability in signup.asp in Pre Classified Listings ASP allows remote attackers to execute arbitrary SQL commands via the email parameter.
	CVE-2008-6887	SQL injection vulnerability in detailad.asp in Pre Classified Listings 1.0 allows remote attackers to execute arbitrary SQL commands via the siteid parameter.
	CVE-2007-2675	SQL injection vulnerability in search.php in Pre Classifieds Listings 1.0 allows remote attackers to execute arbitrary SQL commands via the category parameter.

WmsCms	CVE-2010-2317	Multiple SQL injection vulnerabilities in WmsCms 2.0 and earlier allow remote attackers to execute arbitrary SQL commands via the (1) search, (2) sbr, (3) pid, (4) sbl, and (5) FilePath parameters to default.asp; and the (6) sbr, (7) pr, and (8) psPrice parameters to printpage.asp.
Absolute News Manager(.NET)	CVE-2007-6269	Multiple SQL injection vulnerabilities in xlaabsolutenm.aspx in Absolute News Manager.NET 5.1 allow remote attackers to execute arbitrary SQL commands via the (1) z, (2) pz, (3) ord, and (4) sort parameters.
	CVE-2008-2757	SQL injection vulnerability in search.asp in Xigla Absolute News Manager XE 3.2 allows remote authenticated administrators to execute arbitrary SQL commands via the orderby parameter.
Active Price Comparison	CVE-2008-5975	SQL injection vulnerability in links.asp in Active Price Comparison 4.0 allows remote attackers to execute arbitrary SQL commands via the linkid parameter. NOTE: the provenance of this information is unknown; the details are obtained solely from third party information.
	CVE-2008-5974	Multiple SQL injection vulnerabilities in login.aspx in Active Price Comparison 4.0 allow remote attackers to execute arbitrary SQL commands via the (1) password and (2) username fields.
	CVE-2008-5638	Multiple SQL injection vulnerabilities in Active Price Comparison 4 allow remote attackers to execute arbitrary SQL commands via the (1) ProductID parameter to reviews.aspx or the (2) linkid parameter to links.asp.
WebEvents	CVE-2007-4108	SQL injection vulnerability in sign_in.aspx in WebEvents (Online Event Registration Template) allows remote attackers to execute arbitrary SQL commands via the Password parameter.
Xigla Absolute Banner Manager (.NET)	CVE-2008-2760	SQL injection vulnerability in searchbanners.asp in Xigla Absolute Banner Manager XE 2.0 allows remote authenticated administrators to execute arbitrary SQL commands via the orderby parameter.
	CVE-2007-6291	SQL injection vulnerability in abm.aspx in Xigla Absolute Banner Manager .NET 4.0 allows remote attackers to execute arbitrary SQL commands via the z parameter.

Bibliography

[1] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. *N-variant systems: A secretless framework for security through diversity*. Defense Technical Information Center, 2006.

[2] Vipin Swarup Cliff Wang X. Sean Wang Sushil Jajodia, Anup K. Ghosh. *Moving Target Defense*. Advances in Information Security.

[3] Sandeep Nair Narayanan, Alwyn Roshan Pais, and Radhesh Mohandas. Detection and prevention of sql injection attacks using semantic equivalence. In *Computer Networks and Intelligent Computing*, pages 103–112. Springer, 2011.

[4] Jayant Madhavan, Philip A Bernstein, and Erhard Rahm. Generic schema matching with cupid. In *Proceedings of the International Conference on Very Large Data Bases*, pages 49–58, 2001.

[5] Rakesh Agrawal and Ramakrishnan Srikant. Privacy-preserving data mining. *ACM Sigmod Record*, 29(2):439–450, 2000.

[6] Fredrik Valeur, Darren Mutz, and Giovanni Vigna. A learning-based approach to the detection of sql attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 123–140. Springer, 2005.

[7] WG Halfond, J. Viegas, and A. Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, pages 65–81. IEEE, 2006.

[8] The heartbleed bug. `http://www.heartbleed.com/`.

[9] B. Littlewood and L. Strigini. Redundancy and diversity in security. *Computer Security–ESORICS 2004*, pages 423–438, 2004.

[10] D. Gao, M. Reiter, and D. Song. Behavioral distance measurement using hidden markov models. In *Recent Advances in Intrusion Detection*, pages 19–40. Springer, 2006.

[11] B.G. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX Annual Technical Conference*, pages 287–292, 2008.

[12] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro. Os diversity for intrusion tolerance: Myth or reality? In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 383–394. IEEE, 2011.

[13] S. Bhatkar, D.C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX security symposium*, volume 120. Washington, DC., 2003.

[14] The pax team. `http://pax.grsecurity.net/`.

[15] G.S. Kc, A.D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM, 2003.

[16] S. Bhatkar and R. Sekar. Data space randomization. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22, 2008.

[17] Common vulnerabilities and exposures (cve) database. `http://cve.mitre.org/`.

[18] Jose Fonseca, Marco Vieira, and Henrique Madeira. Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 365–372. IEEE, 2007.

[19] Sql injection. `https://www.owasp.org/index.php/SQL_Injection`.

[20] Christoph Wehrmann. Cross site scripting.

[21] Kevin Spett. Cross-site scripting. *SPI Labs*, 2005.

[22] Daniel Geer, Rebecca Bace, Peter Gutmann, Perry Metzger, Charles P Pfleeger, John S Quarterman, and Bruce Schneier. Cyberinsecurity: The cost of monopoly. *Computer and Communications Industry Association (CCIA)*, 2003.

[23] Mark Stamp. Risks of monoculture. *Communications of the ACM*, 47(3):120, 2004.

[24] F. Majorczyk and J.C. Demay. Automated instruction-set randomization for web applications in diversified redundant systems. In *Availability, Reliability and Security, 2009. ARES'09. International Conference on*, pages 978–983. IEEE, 2009.

[25] Marco Casassa Mont, Adrian Baldwin, Yolanta Beres, Keith Harrison, Martin Sadler, and Simon Shiu. Towards diversity of cots software applications: Reducing risks of widespread faults and attacks. *HP Laboratories, Bristol, UK HPL-2002-178*, 2002.

[26] Eric Totel, Frédéric Majorczyk, and Ludovic Mé. Cots diversity based intrusion detection and application to web servers. In *Recent Advances in Intrusion Detection*, pages 43–62. Springer, 2006.

[27] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 33–46. ACM, 2009.

[28] M. Franz. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 workshop on New security paradigms*, pages 7–16. ACM, 2010.

[29] The security content automation protocol (scap). `http://scap.nist.gov/`.

[30] Us national vulnerability database(nvd). `http://nvd.nist.gov/`.

[31] Cve identifier. `http://cve.mitre.org/cve/identifiers/index.html`.

[32] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *ACM SIGPLAN Notices*, volume 41, pages 372–382. ACM, 2006.

[33] Hong-Hai Do and Erhard Rahm. Coma: a system for flexible combination of schema matching approaches. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 610–621. VLDB Endowment, 2002.

[34] Fausto Giunchiglia, Pavel Shvaiko, and Mikalai Yatskevich. Semantic schema matching. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, pages 347–365. Springer, 2005.

[35] Fausto Giunchiglia, Pavel Shvaiko, and Mikalai Yatskevich. *S-Match: an algorithm and an implementation of semantic matching*. Springer, 2004.

[36] Dennis Shasha, Jason TL Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 39–52. ACM, 2002.

[37] Natalya F Noy and Mark A Musen. Anchor-prompt: Using non-local context for semantic matching. In *Proceedings of the workshop on ontologies and information sharing at the international joint conference on artificial intelligence (IJCAI)*, pages 63–70, 2001.

[38] Debin Gao, Michael K Reiter, and Dawn Song. Behavioral distance for intrusion detection. In *Recent Advances in Intrusion Detection*, pages 63–81. Springer, 2006.

[39] John Mingers. An empirical comparison of selection measures for decision-tree induction. *Machine learning*, 3(4):319–342, 1989.

[40] Ron Kohavi. Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid. In *KDD*, pages 202–207, 1996.

[41] Usama M Fayyad and Keki B Irani. On the handling of continuous-valued attributes in decision tree generation. *Machine learning*, 8(1):87–102, 1992.

[42] John T Kent. Information gain and a general measure of correlation. *Biometrika*, 70(1):163–173, 1983.

[43] Open web applications security project (owasp). `https://www.owasp.org/`.

[44] O. Maor and A. Shulman. Sql injection signatures evasion. *Imperva, Inc., Apr*, 2004.

[45] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting sql injection vulnerabilities. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 87–96. IEEE, 2007.

[46] C. Gould, Z. Su, and P. Devanbu. Jdbc checker: A static analysis tool for sql/jdbc applications. In *Proceedings of the 26th International Conference on Software Engineering*, pages 697–698. IEEE Computer Society, 2004.

[47] W.G.J. Halfond and A. Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183. ACM, 2005.

[48] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. Candid: preventing sql injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 12–24, New York, NY, USA, 2007. ACM.

[49] G. Buehrer, B.W. Weide, and P.A.G. Sivilotti. Using parse tree validation to prevent sql injection attacks. In *Proceedings of the 5th international workshop on Software engineering and middleware*, pages 106–113. ACM, 2005.

[50] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *ACM SIGPLAN Notices*, volume 41, pages 372–382. ACM, 2006.

[51] S. Boyd and A. Keromytis. Sqlrand: Preventing sql injection attacks. In *Applied Cryptography and Network Security*, pages 292–302. Springer, 2004.

[52] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, pages 124–145. Springer, 2006.

[53] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium*, volume 14, pages 18–18, 2005.

[54] William R Cook and Siddhartha Rai. Safe query objects: statically typed objects as remotely executable queries. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 97–106. IEEE, 2005.

[55] David Scott and Richard Sharp. Abstracting application-level web security. In *Proceedings of the 11th international conference on World Wide Web*, pages 396–407. ACM, 2002.

[56] Michael Martin, Benjamin Livshits, and Monica S Lam. Finding application errors and security flaws using pql: a program query language. In *ACM SIGPLAN Notices*, volume 40, pages 365–383. ACM, 2005.

[57] R.A. McClure and I.H. Kruger. Sql dom: compile time checking of dynamic sql statements. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 88–96. IEEE, 2005.

[58] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 12th international conference on World Wide Web*, pages 148–159. ACM, 2003.

[59] 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 *Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM, 2004.

[60] B. Salamat, C. Wimmer, and M. Franz. Synchronous signal delivery in a multi-variant intrusion detection system. Technical report, Technical report, School of Information and Computer Sciences, University of California, Irvine, 2009.

[61] Joni Fraga and David Powell. A fault-and intrusion-tolerant file system. In *Proceedings of the 3rd International Conference on Computer Security*, volume 203. Ireland, 1985.

[62] Alysson Neves Bessani. From byzantine fault tolerance to intrusion tolerance. 2012.

[63] A. Gorbenko, V. Kharchenko, O. Tarasyuk, and A. Romanovsky. Using diversity in cloud-based deployment environment to avoid intrusions. *Software Engineering for Resilient Systems*, pages 145–155, 2011.

[64] M. Garcia, N. Neves, and A. Bessani. Diversys: Diverse rejuvenation system.

[65] J. Antunes and N. Neves. Diveinto: Supporting diversity in intrusion-tolerant systems. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 137–146. IEEE, 2011.

[66] Algirdas Avizienis and Liming Chen. On the implementation of n-version programming for software fault tolerance during execution. In *Proc. IEEE COMPSAC*, volume 77, pages 149–155, 1977.

[67] A. Avizienis. The n-version approach to fault-tolerant software. *Software Engineering, IEEE Transactions on*, SE-11(12):1491 – 1501, dec. 1985.

[68] M.R. Lyu, J.H. Chen, and A. Avizienis. Software diversity metrics and measurements. In *Computer Software and Applications Conference*, pages 69–78, 1992.

[69] S. Mitra, N.R. Saxena, and E.J. McCluskey. A design diversity metric and analysis of redundant systems. *IEEE Trans. Comput.*, 51(5):498–510, May 2002.

[70] B. Littlewood, P. Popov, and L. Strigini. Modeling software design diversity: A review. *ACM Comput. Surv.*, 33(2):177–208, June 2001.

[71] R.A. Maxion. Use of diversity as a defense mechanism. In *Proceedings of the 2005 Workshop on New Security Paradigms*, NSPW '05, pages 21–22, New York, NY, USA, 2005. ACM.

[72] B. Salamat, T. Jackson, G. Wagner, C. Wimmer, and M. Franz. Runtime defense against code injection attacks using replicated execution. *Dependable and Secure Computing, IEEE Transactions on*, 8(4):588–601, 2011.

[73] A. Nguyen-Tuong, D. Evans, J.C. Knight, B. Cox, and J.W. Davidson. Security through redundant data diversity. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 187–196. IEEE, 2008.

[74] Shuo Chen, Jun Xu, Emre C Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th conference on USENIX Security Symposium*, volume 14, pages 12–12, 2005.

[75] Y. Yang, S. Zhu, and G. Cao. Improving sensor network immunity under worm attacks: a software diversity approach. In *Proceedings of the 9th ACM international symposium on Mobile ad hoc networking and computing*, pages 149–158. ACM, 2008.

[76] Alessandro Orso, Wenke Lee, and Adam Shostack. Preventing sql code injection by combining static and runtime analysis. Technical report, DTIC Document, 2008.

[77] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 117–128. IEEE, 2002.

[78] Midicart official site. `http://www.midicart.se`.

[79] The open source vulnerability database (osvdb). `http://www.osvdb.org/`.

[80] Mateusz Pawlik and Nikolaus Augsten. Rted: a robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment*, 5(4):334–345, 2011.

[81] John Ross Quinlan. *C4. 5: programs for machine learning*, volume 1. Morgan kaufmann, 1993.

[82] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[83] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, volume 14, pages 1137–1145, 1995.

[84] General sql parser java official site. `http://www.sqlparser.com/sql-parser-java.php`.

[85] Keyboard simulator official site. `http://www.anjian.com/`.