EAI Endorsed Transactions
on Security and Safety

Research Article EAI.EU

Opportunistic Diversity-Based Detection of Injection Attacks in Web Applications

Wenyu Qu[1], Wei Huo[2], Lingyu Wang[2,*]

[1]School of Computer Software, Tianjin University, Tianjin, China
[2]Concordia Institute For Information Systems Engineering, Concordia University, Montreal, Canada

Abstract

Web-based applications delivered using clouds are becoming increasingly popular due to less demand of client-side resources and easier maintenance than desktop counterparts. At the same time, larger attack surfaces and developers' lack of security proficiency or awareness leave Web applications particularly vulnerable to security attacks. On the other hand, diversity has long been considered as a viable approach to detecting security attacks since functionally similar but internally different variants of an application will likely respond to the same attack in different ways. However, most diversity-by-design approaches have met difficulties in practice due to the prohibitive cost in terms of both development and maintenance. In this work, we propose to employ opportunistic diversity inherent to Web applications and their database backends to detect injection attacks. We first conduct a case study of common vulnerabilities to confirm the potential of opportunistic diversity for detecting potential attacks. We then devise a multi-stage approach to examine features extracted from the database queries, their effect on the database, the query results, as well as the user-end results. Next, we combine the partial 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.

Received on 21 November 2018; accepted on 04 December 2018; published on 11 December 2018

Keywords: Diversity, SQL Injection, Web Application Security, Intrusion Detection

doi:10.4108/eai.11-12-2018.156032

1. Introduction

Web-based applications are becoming increasingly popular in an age of cloud computing, mobile devices, and Internet of things. 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. For instance, the widespread panic about the Heart Bleed vulnerability [1] 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 only works for known attacks, whereas anomaly detection is well known to suffer from inaccuracy.

In a slightly different context, software diversity has traditionally been regarded as a promising mechanism for improving the robustness of a software system against unknown attacks [2]. More recently, diversity has found new applications in cloud computing security [3], Moving Target Defense (MTD) [4], network security [5], and network routing [6] (a more detailed review of related work will be given in Section 2). By comparing outputs [7] or behaviors [8] of multiple software replicas with diverse implementation details, security attacks may be detected and tolerated as

*Corresponding author. Email: wang@ciise.concordia.ca

Byzantine faults [9]. 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 [10], or automatically generating diversity through randomization of address space [11, 12], instruction set [13], or data space [14]. Nonetheless, a dilemma faced by most existing works on detecting attacks through diversity is that, diversity is either too costly (as in the case of diversity-by-design), or not sufficient to be used as a stand-alone means for attack detection (as in the case of opportunistic diversity).

In this paper, unlike most existing works, our key idea is to employ opportunistic diversity in Web applications and database backends to *assist*, instead of *replacing*, traditional anomaly detection methods, in order to improve the overall detection accuracy. This approach allows us to avoid the prohibitive cost of diversity-by-design, while having the best of both worlds from anomaly detection and opportunistic diversity. Specifically, we propose a multi-stage approach to employ the opportunistic diversity found in Web applications and their database backends for reducing the false alarms generated by anomaly detection, as follows.

- First, we conduct a case study on real world vulnerabilities to confirm the effectiveness of the proposed approach. Specifically, we perform an extensive study of almost 6,000 Common Vulnerabilities and Exposures (CVE) Web injection vulnerabilities [15] 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 implies that 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 experimental 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 diversity-based detection based on a single stage.

The main contribution of this paper is twofold. First, this is among the first efforts that employ opportunistic diversity for improving the accuracy of anomaly detection in the specific context of Web applications; 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 single-stage detection, and thus leads to a promising direction towards practical detection solutions; although we have focused on injection attacks in this paper, the methodology can potentially be extended to detect other attacks.

The rest of the paper is organized as follows. We first review related work in Section 2. We then present a case study in Section 3. We propose the attack detection model in Section 4 and present the experimental results in Section 5. Finally, we conclude the paper in Section 6.

2. 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.

2.1. Defense of Injection Attacks

Despite many years of research, various forms of injection attacks still remain a major threat to Web applications (e.g., injection attack has been ranked number one in the top ten critical Web application security risks by the Open Web Application Security Project (OWASP) [16]). Defensive coding is one of the common defense approaches, with mechanisms like [17] 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 SQL injection attacks (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. propose *SAFELI* [18], a mechanism that uses static analysis to detect SQLIAs in

Web applications during compilation. Similarly, *JDBC Checker* [19] is a practical tool implementing static analysis. Such static code checkers can verify the correctness of dynamically-generated SQL queries but cannot handle SQLIA queries that contains correct type and syntax.

There also exist approaches that involve the combination of static and dynamic analysis. *AMNESIA* [20] is a hybrid solution that combines static and dynamic analysis which uses static analysis to build a model for web applications and intercepts the run-time queries to verify if they match the model. *CANDID* [21] is another dynamic analysis approach which dynamically runs web applications with candidate inputs and detect SQLIAs by comparing them to the structure of candidate queries. *SQLGuard* [22] and *SQLCheck* [23] check SQLIAs based on parse tree models generated by static code analysis where the runtime queries' structures are compared to the model and only the matched ones would be sent to database. Most such approaches require code modification of web applications and their accuracy is still largely based on static analysis. Another promising SQLIA defense approach is *SQLrand* [24], which uses a randomized instruction set of queries to prevent SQLIAs where the SQL keywords are different from normal so that the injection code with normal SQL keyword cannot have command effect to database. This technique is generally very effective unless if the secret key is compromised, and the interpretation of proxy filter may also significantly increase the computation cost.

A SQLIA detection technique is proposed in [25] where a standard safe query statement for the web application is first generated with known safe user input strings. Detection is achieved by parsing standard SQL statements and 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)* [26] is a more fine grained analysis where, 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 [27] has similar ideas where only strings originated from external source is considered to be untrusted and syntax evaluation is applied to those strings and the query will only execute if the pattern matching has positive result. The work in [28] prevents more general injection attacks through analyzing the parse tree of query strings. A standard grammar of specific application is pre-defined and 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 with the syntactic constraints.

2.2. Diversity in Security

Using design diversity for fault tolerance has been investigated for a long time. Many approaches in this field implement *Byzantine Fault Tolerant(BFT)* replication as fault tolerance solution. In [29], many practical issues in doing so are mentioned, e.g., BFT replication performance, recovery, effectiveness of diversity, confidentiality, etc. In [9], some of the challenges are further discussed in a single machine environment, such as achieving both effectiveness and efficiency while combining OS and binary randomization techniques. In [30], an intrusion-avoidance architecture is built based on cross-platform JAVA technologies and 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. 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)* [31] 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. When diversity is increased, the problem of incompatibility between replicas also emerges. *DiveInto* [32] 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.

There exist many diversity systems which run multiple variants and compare their results. M.Garcia et al. [10] evaluates opportunistic diversity in OS and its effectiveness for intrusion tolerance. They obtain vulnerability data of various OSes from NIST National Vulnerability Database (NVD), classify them and search for common vulnerabilities across different OSes. The result is promising as only for a very small number of non-applications, remotely exploitable common vulnerabilities are found, which proves diversity system can significantly improve the security of OS. They also evaluated various OS pairs and announced the pairs with best performance on intrusion tolerance. Besides opportunistic diversity, another common approach to diversity is by applying randomization to the low level code of web servers/web

applications. In [33], the authors propose *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 reduce the cost of implementation, making it possible to deploy enough number of COTS servers to fulfill intrusion tolerance requirement. Another randomization approach is *address space randomization (ASR)* which obfuscates the location of data and code instead of randomizing the program code itself. In [11], an ASR technique is proposed to mitigate various code injection attacks on executable files, such as stack smashing and format string attacks. Apart from ISR and ASR, another randomization technique, the *Data Space Randomization (DSR)*, is proposed in [14] which randomizes the representation of different data objects; using different masks on different objects, attackers can no longer determine if the intended value is overwritten into the code. Those randomization techniques allow systems running multiple variants in parallel to be built. However, the problem of synchronization and related false alarm issues emerge in this kind of systems are raised in [34] and a synchronization technique is proposed based on majority voting.

N-Version Programming and N-Variant Systems [7, 35, 36] are typical examples of diversity systems that utilize diversity to defend various attacks. 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 [37, 38]. The main limitation of using diversity for fault tolerance lies in the high complexity of creating different versions, which may not justify the benefit [39]. Babak Salamat et al. proposed *Orchestra* [40] with similar idea, which 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 introduces a notion: *synchronization point*, which is instantiated by invocation of a system call to evaluate if the variants are in conforming states. In another work by Babak Salamat et al. [41], the problem of false positives/negatives is also discussed, i.e., race condition may occur when third party trying to manipulate a file under synchronization, causing divergence between variants and false positives. Anh Nguyen-Tuong et al. [42] proposes a diversity system using the N-Variant system framework where they formally re-define some of the key attributes including normal *equivalence* and *detection*.

There are also diversity systems which distribute different variants to individual users to prevent attacks. *E unibus pluram* [43] 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 attackers 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. Another work in [44] 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 achieves better isolation between adjacent variants while using only a limited number of diversified applications. Finally, the software assignment problem, i.e., how to optimally assign diverse software to different hosts in a network in order to improve the network's resilience to security threats like worms, is addressed in [45] by considering practical constraints, and a similar issue is formulated and solved as a multi-objective optimization problem in [46].

3. The Case Study

In this section, we first use an example to demonstrate how opportunistic diversity may improve the security of a web application. We then perform an extensive study of injection vulnerabilities based on the NVD database in order to evaluate the sharing of vulnerabilities among different versions of Web applications.

3.1. An Example

Midicart [47] is an online shopping cart application with both ASP and PHP versions. We demonstrate how opportunistic diversity may help detecting injection attacks through an example based on a SQL injection vulnerability, which is found in the Midicart PHP version but not in the ASP version.

The ASP Application Line 20 of the "search_list.asp" file 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.

The PHP Application In contrast, 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','second
group', 'code_no' LIMIT 0, 100 ") ;
```

Injection Attacks 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&searchs
tring=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 following 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 will be the following.

```
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.

Observations This example shows that, there may exist opportunistic diversity between different versions of the same Web application, which will cause the same user input to induce different behaviors in those different variants of the same application. Therefore, we can potentially employ different versions of the same web application to construct a diversity system and rely on the different behaviors to help detect possible injection attacks.

On the other hand, to show the general applicability of this idea, we need to study the sharing of vulnerabilities between the different versions of Web applications in general. In order to achieve this, we will perform an extensive study of injection vulnerabilities in Section 3.2.

3.2. Study of Shared Injection vulnerabilities in CVE

We perform an extensive study of shared injection vulnerabilities between different variants of the same Web application (here variants mean multiple versions of the application written in different script languages, instead of upgraded versions) based on the CVE database which is generally considered as comprehensive with respect to all publicly known vulnerabilities and exposures [15]. The goal of this study is to evaluate the hypothesis that different versions of the same Web application rarely share common injection vulnerabilities and hence the opportunistic diversity among those versions will be useful for detecting injection attacks.

Applications with Variants. We first search the CVE entries for Web applications that have multiple variants written in different languages in two steps as follows.

1. We first find Web applications with injection vulnerabilities.
2. Among the result, we then find applications with multiple variants.

Applications with Injection Vulnerabilities We apply search keywords like "SQL injection" to find applications involving injection vulnerabilities in the master copy of the CVE database. Totally 5,870 entries are found to be related to SQL injection vulnerabilities in CVE. A 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, such an entry contains the information of vulnerability type, application name, location of the vulnerability (parameter name and file name). Such information may be used to map each entry to a Web application and find common entries shared by different variants.

Applications with Variants Considering the large amount of vulnerability entries (5870), it would be a tedious process to perform the search manually. Fortunately, following observations allow us to conduct the search in a semi-automated 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 1.

Language	ASP	JSP	ASP.NET	PHP
# of Applications	488	42	30	1000+

Table 1. Grouping of Applications

As the table shows, JSP and ASP.NET applications are the minority (30 to 40 of each) among nearly 6,000 SQL injection vulnerability entries. ASP applications have a relatively larger number, 488, whereas PHP applications are the clear majority, estimated as over 1000. The four categories have very different population sizes. This observation helps us to derive some useful heuristics to reduce the effort as follows. First, 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 the following three steps.

1. Find all ASP.NET applications, and search for their ASP, PHP, JSP variants.
2. Find all JSP applications, and search for their ASP, PHP variants.
3. Find all ASP applications, and search for their PHP variants.

During the study, we use the name of the applications as the keyword to 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. Finally, we verify the results manually.

The Result There are totally 16 applications in CVE database that involve some injection attacks and have multiple variants, which are listed in Table 2. Specifically,

Application	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	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	o		o	
Xigla Absolute Banner Manager	o		o	

Table 2. Web Applications with Variants and Injection Vulnerabilities

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

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

Matching Common Vulnerabilities. Among the 16 applications, we need to determine whether two variants of the same application have "common" vulnerabilities, as explained in the following.

- For any two variants A and B of the same application, we say parameter m in A and parameter n in B are *equivalent parameters* if m and n take equivalent input values from equivalent input fields. Equivalent parameters usually need to be identified based on the same functionalities between variants.
- We say two SQLIA vulnerabilities x and y are *common vulnerabilities* if they involve equivalent parameters m and n, respectively.

Our study shows that the existence of equivalent parameters can be categorized into four cases listed below. Those cases are from the most common to the most uncommon, and most equivalent parameters between variants will still share the same name.

1. The same parameter name and same file name (ignoring extensions).

2. The same parameter name and different file names.
3. Different parameter names and the same file name.
4. Different parameter names and different file names.

For example, in our motivating example given earlier, the "searchstring" parameter is an equivalent parameter in above Case 1 (the same parameter name and same file names), even though this vulnerability is not a common vulnerability since it only exists in the PHP version.

Among these 16 web applications, there are totally 34 SQL injection vulnerability entries in CVE, which are listed in the Appendix. Each application has at least one entry and at most four entries (Pre Classified Listings). With such a relatively small amount of entries per application, it is easy to manually identify common vulnerability between variants.

Matching Common Vulnerabilities We now examine the 34 vulnerabilities to search for common vulnerabilities based on equivalent parameters. In order to simplify the task, we first assume the 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 3. Common Vulnerabilities of Midicart

In both variants, "maingroup", "secondgroup", and "code_no" are all equivalent parameters in Case 1 (the 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 will not rely on diversity alone for detection in this paper). For all other applications, no common vulnerability is found.

Summary. Our key findings are as follows.

- There exist 5,870 SQL injection vulnerability entries related to around 2,000 web applications.
- Among those, there exist 16 applications that have multiple variants written in different languages.
- Those applications involve 34 vulnerability entries.
- Among those, only one application contains one pair of common vulnerabilities.

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

There are some limitations in our study. First of all, the CVE database is certainly not supposed to be comprehensive enough to cover all known vulnerabilities. Also, the manual analysis of common vulnerabilities may not be perfectly accurate. Finally, the 16 applications we found may not cover all applications with variants in CVE database since if an application will be missed in our search if only one of its versions has vulnerability entries. However, this may be acceptable since those missed would not have common vulnerabilities anyway (since only one version has vulnerability entries in the CVE database).

4. The Methodology

In this section, we present our multi-stage diversity-based attack detection method.

4.1. 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 architecture is composed of three major components, i.e., a controller, a monitor, and multiple variants with their database backends,

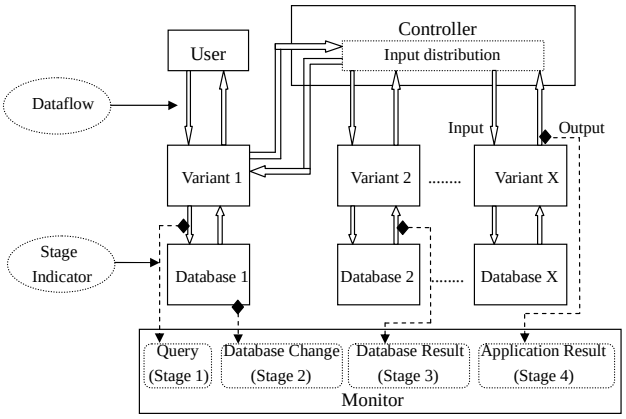


Figure 1. The Model

as shown in Figure 1. Arrows between different components in the figure 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.

As shown in Figure 1, the user interacts with one variant of the application. 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. Those will be detailed in the following.

4.2. Detection

The controller works as a medium between user and application variants with two major modules: the user input distribution module and output unifying module. When it receives a user input, it distributes this input to equivalent parameters of all variants. If no attack is detected by the monitor, the controller will return the result back to the user based on one of the results received from the variants. Otherwise, an alarm would be raised and the output is blocked.

The monitor is the component in charge of attack detection. It first compares the behavior between variants in multiple stages, then combines partial results to reach a final detection result. This is detailed in the following.

Multiple Stages of Detection Our approach does not solely depend on examining SQL queries to detect attacks as in most existing works. Having multiple stages of detection can produce a more reliable detection result; it can also utilize more aspects of the diversity. Since the variants of Web applications are written in different languages and based on different databases, we can expect different behaviors in terms of not only SQL queries, but also database activities, their output results, and the user-side results. Therefore, we employ four stages of monitoring as follows.

- SQL queries generated by the application.
- Changes made to the database.
- Query results returned by the database.
- Final results to be returned to the user.

Each stage has one or more features to score the differences in details. For example, we use the 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.

We expect to detect certain difference among variants in one or more of these stages when the user input is attack-free. When an attack is performed, more significant difference will generally appear in multiple stages. On the other hand, even when an attack is exploiting common vulnerabilities, we may still detect noticeable difference in certain stages. Thus introducing multiple stages of comparison may be effective for mitigating both false positives and false negatives.

Detection Methodology The straightforward way of detecting different behaviors 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 a stored procedure for the same service. In this case, direct comparison on the first stage, query string, would generate a large amount of false positives.

Therefore, different from many existing works, 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, as mentioned above, different variants may exhibit significant differences in terms

of implementation details which prevent a direct comparison.

Specifically, for features that have significant differences among variants, we first apply a learned-based method [48] to establish an anomaly detection model by learning secure profiles through training with attack-free data (note the work in [48] does not involve multi-stage detection or diversity-based detection). 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 paper, we use decision tree learning to produce the final result. The following provides a detailed description of each stage.

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 a 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 1. 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'
   ```

The different input filtering means 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 [48] 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. Therefore, we focus on the structure of the SQL query. The SQL queries 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. Consequently, we compare different variants based on three features.

- The first feature is *the edit distance of ASTs* [49]. 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)* [49], 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 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 2. The standard AST generated from the queries in Example 1 is shown in Fig 2.

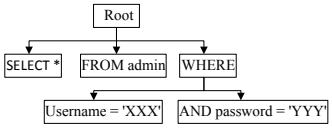
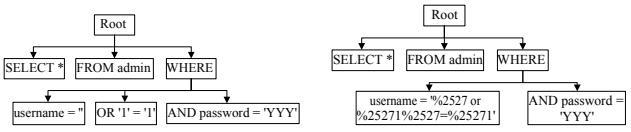


Figure 2. Standard AST of Example 1

The AST generated from user input in variant A and B is shown in Fig 3.



(a.) Variant A (b.) Variant B

Figure 3. The Generated ASTs from Both Variants

The AST in Fig 3(a) has 3.0 tree edit distance with standard AST, while the AST in Fig 3(b) has 0 distance with standard AST. As it shows, ASTs are able to characterize queries with different 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.

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 cases, the difference lies in the prefix or suffix of the name.

When SQL injection is performed, the attacker usually is required to enclose the apostrophe or parenthesis at the beginning of the injection code. If one variant has 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 3. 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 may still cause different behaviors in the database among variants, which makes 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 [50–52], to measure the difference between databases at runtime and previously 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. Consequently, we rely on following 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.

- 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 the number of rows changed in the database.

Example 4. For the attack in Example 3, the first feature would score "1" since the 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.

□

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 5. Suppose attacker injects "username" parameter in the application variants similarly as in Example 3 with following string, assuming both variants have tables named "products":

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

The injected query will only succeed in variant A 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 to be returned to the application.

□

As Example 5 shows, some injection attacks that lead to different behaviors in the database may not change the database itself. However, we can capture 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. 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.

- 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 6. For the attack in Example 5, 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.

□

Stage 4: Application Result. In this stage, we utilize the diversity in the result to be returned to the user by the application, which in most cases is an HTML page. Injection attacks often involve enclosed apostrophe and parenthesis which may lead to an non-executable SQL query in some variants. A HTML page with error message is usually returned to users in this case. Although the AST-based Stage 1 can also detect SQL queries that cannot be executed, checking the application results can provide additional detection opportunity.

Example 7. Suppose the SQL queries in variant A and B are the same as in Example 3 when the attacker injects "username" parameter with following string:

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

Suppose the table "products" has zero record on both variants so 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. This difference makes the attack detectable in Stage 4.

□

Features for Detection To measure difference between HTML pages, many features are available, including size of the page, title of the page, etc. However, the detection accuracy of using such fine grained features in this stage may largely depend on the applications, since the inherent differences in those features among variants may be significant even for normal results. Our study shows that focusing on monitoring error messages in HTML pages would be sufficient for detecting the difference in most cases. Consequently, we use one feature in Stage 4, i.e., the *existence of error messages*, which 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. 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". The binary score from each variant is added up to produce the final score for this feature.

Stage	Feature	Application
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 4. Detection Features

4.3. Result Correlation

Table 4 summarizes the features used in each stage along with the way of applying those features (in anomaly detection or through direct comparison). Since the behavior of an application in different stages is usually inter-dependent, correlating the results from different stages may further improve the detection accuracy and reduce false alarms. Consequently, we correlate the results of different stages to make a final decision on attack detection.

The partial detection results obtained at different stages may be correlated in many ways. In this work, we employ decision tree learning to correlate scores of different features to make the final decision of attack detection. 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. The collection of such results tables for all inputs are used to build a decision tree, which will be applied to runtime inputs to classify them into two classes: "attack" or "normal".

We choose the *C4.5 Algorithm* [53] as the decision tree learning algorithm. Compared to its predecessor *ID3 (Iterative Dichotomiser 3) Algorithm* [54], C4.5 has several advantages including allowing continuous attributes and missing attribute values, and the trees can be pruned after creation. As mentioned previously, the score of all the features across the 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 and are discretized using the C4.5 algorithm.

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

- Attack-free user input, which is the normal request from a user. The attack-free data is obtained using scripts that simulate normal user activity.
- Attack input that will succeed in all variants; this is obtained by attacking the common vulnerabilities of all variants.
- Attack input that will succeed in at least one, but not all variant, this is obtained from attacks that exploit non-common vulnerabilities in the variants.

We follow the ten-fold cross validation [55] to produce the final decision tree. The training data is randomly partitioned into ten folds and in each run, nine folds of data are used for training the decision tree and the trained tree is applied to the remaining one fold of data for testing.

5. Implementation and Experiments

In this section, we describe our implementation based on a real web application and perform experiments to compare our detection model to several other options.

5.1. Implementation

The Application. Our implementation is based on *Midicart* [47], which is an online shopping cart application that has multiple variants. We use Midicart ASP version with Microsoft SQL Server as database and PHP version with Mysql as database. The main reason we choose Midicart for implementation is that, according to our study of shared vulnerabilities in Section 3.2, this application has both common vulnerabilities and vulnerabilities existing only on one variant, which is necessary for evaluating our proposed method in different cases. The CVE entries regarding Midicart are listed in Table 5. In the table, parameters "maingroup", "secondgroup", and "code_no" in multiple files have injection vulnerabilities in both the ASP and PHP versions. However, "searchstring" parameter is only vulnerable in the PHP version (all the vulnerable versions were produced in 2006).

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 5. Midicart Vulnerability Entries in CVE

For regular users(customers), the functionality of Midicart is relatively simple, normal activities of the application include the following.

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

Detection. As described in Section 4.2, 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 stage in two steps. First, we extract runtime SQL queries from applications as follows. For mysql database, we use the event log function to save runtime queries to "general_log" table. For SQL Server database, we use Microsoft SQL Profiler to create a trace and save runtime queries to a user designated table. On both variants, we use a unique user name in their connection string to the database. This is to distinguish queries originated by our web application from others. We only monitor the records in the log belonging to type "query". For Mysql, the "command_type" field needs to be "Query". For SQL Server, the "EventClass" field needs to be "13" or "10", which are the code for type "SQL:BatchStarting" and "RPC:Completed" respectively. We use time as a factor to distinguish queries from different actions in the log.

Second, we generate ASTs, calculate edit distance and retrieve table names from SQL queries as follows. We use a tool *General SQL Parser JAVA* [56] to parse the retrieved queries into ASTs. We use another tool *Robust Algorithm for the Tree Edit Distance(RTED)* [49] to calculate the edit distance between ASTs. However, since this tool only accepts bracket notation of trees, we have developed a script to transform the generated ASTs to the bracket notations according to the rows and spacing at beginning of each row. 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.

For Stage 2, we monitor changes made to the database at runtime by collecting information from the data dictionary about the state of the database including the table names, the number of records in the tables, and the schema of the tables. For the number of records, we compare the number of changed rows in database between variants. For database schema, we first compare the schema of individual tables, then compare the total number of tables in database. Suppose variant a has m involved tables and variant b has n; Da_i and Db_i are the number of rows in the ith involved tables of two variants at runtime, respectively; Sa_i and Sb_i are the reference row count. Score of this feature is defined as $Score = \|\sum_{i=1}^{m} Da_i - \sum_{i=1}^{m} Sa_i| - |\sum_{i=1}^{n} Db_i - \sum_{i=1}^{n} Sb_i\|$. Finally, for changes made to database schema, if the schema of variant a is not changed, Da is "0"; otherwise, it is "1". Let Db be the counterpart in variant b. The score of this feature is defined as $Score = Da + Db$;.

For Stage 3, we need to monitor the result from database. We implement this stage by retrieving the result set of executed queries from the database. We save the data type of result set as a string and compare to the corresponding profile. The number of records in the result set would be directly compared between variants. We calcuate two scores for the database result set, i.e., the number of rows in result set, $Score = |\sum_{i=1}^{m} Da_i - \sum_{i=1}^{n} Db_i|$; (variant a has m queries and variant b has n; Da_i and Db_i are the number of records for ith query in the two variants, respectively), and the type of data in result set, $Score = |\sum_{i=1}^{m} Da_i - \sum_{i=1}^{n} Db_i|$; (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 the same as the corresponding profile, Da_i is "0" and "1" otherwise; Db_i is the counterpart in variant b).

For Stage 4, we need to monitor the result returned by the application. We use a keyboard and mouse mimic software "Keyboard Simulator" [57] to script a "save-to-file" action. This script is added at the end of each training script such that the web page is automatically saved to a text file. Since we use "existence of error page" as the feature for Stage 4, we can later parse the text file to retrieve multiple error messages in order to calculate the score of this stage as either "1" if the error page exists in variant a, or 0 otherwise, for each variant.

For result correlation, we employ the C4.5 algorithm with discrete attributes for features using binary score and continuous attributes for other features. The following shows the case of two variants.

1. Tree edit distance of AST, {continuous}.
2. Involved database table, {0, 1, 2}.
3. Number of not parsed queries, {continuous}.
4. Number of rows changed in database, {continuous}.
5. Change of database schema, {0, 1, 2}.
6. Type of data in result set, {continuous}.
7. Number of rows in result set, {continuous}.
8. Existence of error page, {0, 1, 2}.

5.2. Experiments

Dataset. Our training data is consisted of both attack-free data and attacks. The attack-free data is obtained by performing the four types of normal activities mentioned in Section 5.1. To obtain a sufficient amount of attack-free training data, we traverse all the functionalities of the Midicart application, both manually and through an automated tool to create scripts which can simulate normal user activities. The software we employ is the "Keyboard Simulator" [57]. The following shows an example script created using this software in which the functionality is to open the URL "http://127.0.0.1:81/midiphp/", select "code_no" tag and search for 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 training contains URL opening actions at both variants and clicks on certain elements on the web page if applicable. Apart from these actions, we also attach a "save-to-file" action at the bottom of every script such that the opened web page would be saved as part of the implementation of Stage 4 as mentioned in Section 5.1.

For attacks, we have implemented the following attacks which exploit common vulnerabilities and vulnerabilities only applicable to one variant, respectively.

Attack 1 This attack is to exploit the non-filtered "code_no" parameter in both variants; 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 variants. However, the tautology attack of Post Data 2 can succeed on both variants.

- **Exploited 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'

Attack 2 This attack is to exploit the non-filtered "maingroup" and "secondgroup" parameter in both variants; the union select SQL injection can succeed in both variants since this is a common vulnerability and there is no enclosed parenthesis problem.

- **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

Attack 3 This attack is to exploit the "searchstring" parameter in both variants. However, while the PHP variant has the SQLIA vulnerability in this parameter, in the ASP variant this parameter properly escapes special characters like apostrophe. Thus the union select SQL injection can only succeed on the PHP variant.

- **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

The Results.

Decision Tree Figure 4 shows the decision tree which yields the best detection accuracy for our experiments. It uses four features from three stages as the decision nodes, as detailed below. Since both attacks and attack-free data in this particular application do not modify the database, no feature from Stage 2 (changes in database) is used.

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.

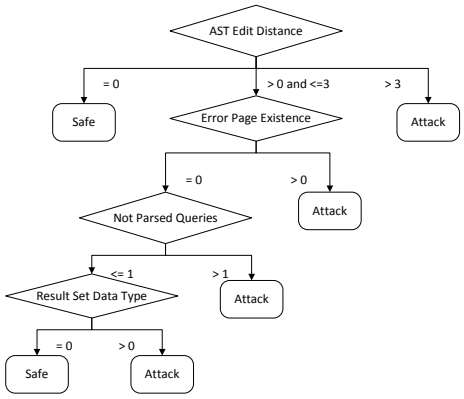


Figure 4. The Learned Decision Tree

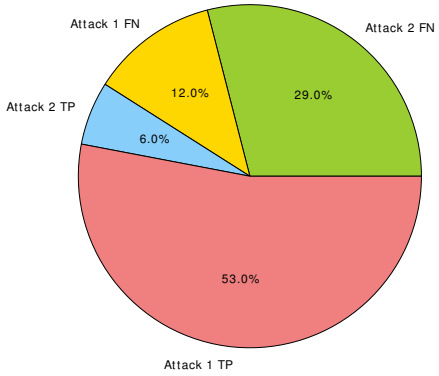


Figure 5. FPR and TNR for Detecting Attacks Exploiting Common Vulnerabilities

Detection Performance We evaluate the detection performance using three metrics, the Accuracy (AC), False Positive Rate (FPR), and False Negative Rate (FNR), which are defined as: $FPR = \frac{FP}{FP+TN}$, $FNR = \frac{FN}{FN+TP}$, and $AC = \frac{TP+TN}{TP+TN+FP+FN}$, respectively. Table 6 shows the detection performance results.

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

Table 6. The Detection Performance

As shown in Table 6, the decision tree can classify all attack-free data in training data correctly. The relatively simple functionality of the application is the main reason for the 100% FPR. We expect to observe FPs when implementing this approach in other applications with more complex functionality or more significant differences between variants. The 14 FNs recorded means 14 out of 554 attack data are misclassified and therefore the detection accuracy is 97.7%.

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

Next, by revisiting the training data, we map each FN to each of the three attacks mentioned in Section 5.2. Recall that Attack 1 and 2 are exploiting common vulnerabilities while Attack 3 only works on one variant. In our dataset, more than half of the attack data are from attacks exploiting common vulnerabilities. All 14 FNs come from Attack 1 and 2, which means the 20 attack instances generated from Attack 3 are all correctly classified, which shows employing diversity does improve the detection performance despite the FNs. Table 7 shows a breakdown of the FN and TP among the three attacks.

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

Table 7. Detection Performance Breakdown among Attacks

More importantly, we can still detect a portion of the attacks that exploit the common vulnerabilities (Attack 1 and 2) despite the fact that such attacks work on both variants. As Figure 5 shows, 20 out of 34 attack instances exploiting common vulnerabilities can be correctly detected. The reason is the following.

1. Some injection strings contain specific database elements that differs across variants (similar to Example 4), which applies to both Attack 1 and 2.

2. The parenthesis is unenclosed (similar to Example 3), which only applies to Attack 1.

Comparison to Anomaly Detection In order to evaluate the benefit of diversity, we compare the detection accuracy of our diversity-based detection approach to anomaly-based detection that does not involve diversity.

We focus on the feature "AST edit distance" from Stage 1 since this feature has the highest *information gain* during decision tree learning and can "best" classify the data. We use the ASP variant for anomaly detection. In the anomaly detection, we compute the edit distance between runtime ASTs and ASTs in the corresponding profile. The PHP variant is not involved since anomaly detection does not require diversity. We then choose different thresholds to classify the data into attacks and normal cases, including 0.5, 3.5,

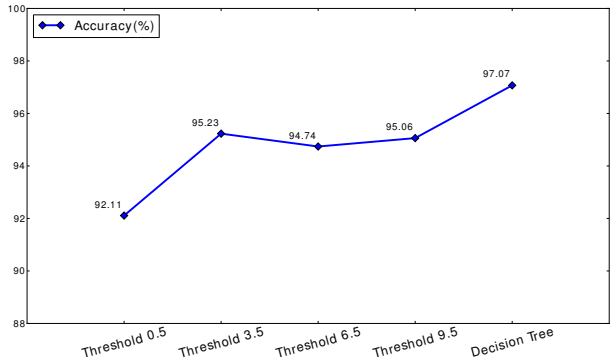


Figure 6. Detection Accuracy Comparison with Anomaly Detection

6.5, 9.5. Figure 6 shows the detection accuracy using each of these thresholds, which ranges from 92.11% to 95.23%, and using 3.5 as the threshold gives the highest accuracy. In contrast, we can see that the diversity-based approach (labeled as "Decision Tree") has better detection performance (97.7%) than the anomaly-based detection. Note that the relatively simple functionalities and the small number of attack types involved in the application have led to the relatively high accuracy for the anomaly detection, and hence there is limited room for improvements using diversity.

Comparison with Single-Stage Diversity-Based Detection To evaluate the contribution of using multiple stages to detection accuracy, we compare our approach to diversity-based detection using a single stage. We use the same features as those used in the learned decision tree (but without result correlation using the decision tree), and a threshold is set for each feature. Specifically, the features include the AST Edit Distance (threshold 0.5 and 3.5), the Error Page Existence (threshold 1), and the Result Set Data Type (threshold 0.5). Training data is classified by comparing its score of the feature to the threshold. Note that, unlike the previous experiments, diversity is involved here since the score is computed across both variants.

Figures 7 show the FPR comparison between our approach and different single-stage detection approaches. Our approach has the best FPR (0%); detection using the AST edit distance with threshold 3.5 and detection using the result set data type also have 0% FPR. When the threshold of the AST edit distance is set to 0.5, the FPR increases to 1.62%. The FPR of detection using the error page existence is relatively high (5.96%).

For the FNR, as shown in Figures 8, our approach (25.93%) and AST edit distance with threshold 0.5 (20.37%) are at similar levels. Using other three features result in a high FNR. As mentioned before, the FNRs are mainly due to the common vulnerabilities existing

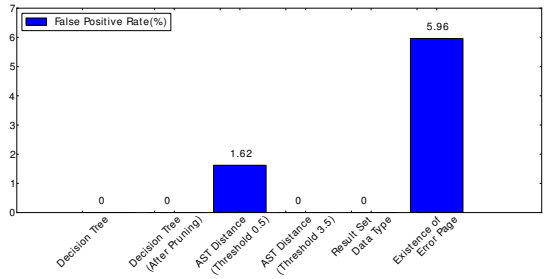


Figure 7. FPR Comparison with Single Stage Detection

in both variants, which is relatively rare as shown in our case study.

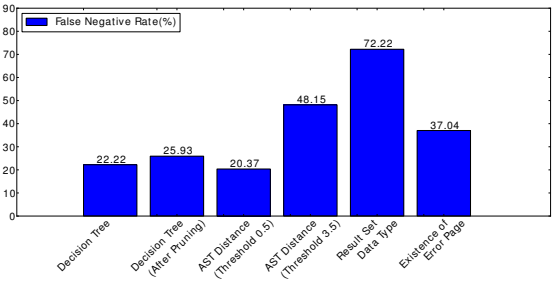


Figure 8. FNR Comparison with Single Stage Detection

As shown in Figure 9, overall our approach has a higher detection accuracy than any other single-stage detection approach, which shows the benefit of correlating results from multiple stages of detection. Moreover, by comparing the detection accuracy of the AST edit distance feature in Figure 9 and in Figure 6, we can see that the same feature when assisted by diversity (Figure 9) leads to a higher accuracy. This again demonstrates the effectiveness of diversity in improving the detection accuracy.

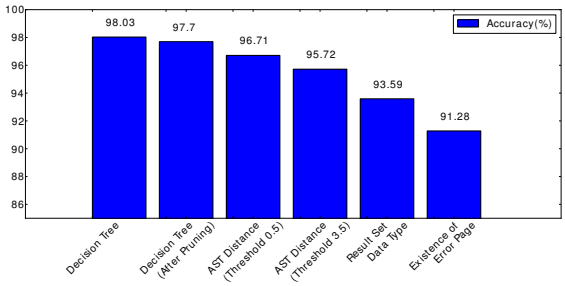


Figure 9. Detection Accuracy Comparison with Single Stage Detection

6. Conclusion

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

First, we have evaluated the feasibility of this approach through an extensive study of injection vulnerabilities shared by different variants of the same Web application based on the CVE database. We have identified the Web applications with multiple variants and demonstrated the rare existence (in only one application) of common vulnerabilities shared between such variants. Second, we have designed an attack detection model to employ diversity between multiple variants while using features extracted from four stages of a Web application. We either compared the features directly, or compared their anomaly detection results between different variants, and the partial results from different stages are correlated using decision tree learning. Finally, we implemented and evaluated our approach based on a real Web application with common vulnerabilities shared between two variants. The results show our approach has better detection performance than both anomaly detection only, and single-stage diversity-based detection. In addition, combining diversity with anomaly detection has helped to detect the exploits of common vulnerabilities which would evade detection using diversity alone. Future directions include applying the approach to Web applications with more complex functionalities, expanding the scope of attacks and enriching the selection of features.

Acknowledgements

Authors with Concordia University were partially supported by the Natural Sciences and Engineering Research Council of Canada under Discovery Grant N01035.

References

[1] "The heartbleed bug." http://www.heartbleed.com/.

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

[3] K. Yang, X. Jia, K. Ren, B. Zhang, and R. Xie, "Dac-macs: Effective data access control for multiauthority cloud storage systems," *Information Forensics and Security, IEEE Transactions on*, vol. 8, no. 11, pp. 1790–1801, 2013.

[4] S. Jajodia, A. Ghosh, V. Swarup, C. Wang, and X. Wang, *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*. Springer, 1st ed., 2011.

[5] M. Zhang, L. Wang, S. Jajodia, A. Singhal, and M. Albanese, "Network diversity: A security metric for evaluating the resilience of networks against zero-day attacks," *IEEE Transactions on Information Forensics and Security*, vol. 11, pp. 1071–1086, May 2016.

[6] J. Caballero, T. Kampouris, D. Song, and J. Wang, "Would diversity really increase the robustness of the routing infrastructure against software defects?," *Department of Electrical and Computing Engineering*, p. 40, 2008.

[7] 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.

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

[9] B. Chun, P. Maniatis, and S. Shenker, "Diverse replication for single-machine byzantine-fault tolerance," in *USENIX Annual Technical Conference*, pp. 287–292, 2008.

[10] 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*, pp. 383–394, IEEE, 2011.

[11] S. Bhatkar, D. 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*, vol. 120, Washington, DC., 2003.

[12] "The pax team." http://pax.grsecurity.net/.

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

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

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

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

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

[18] 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*, vol. 1, pp. 87–96, IEEE, 2007.

[19] 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*, pp. 697–698, IEEE Computer Society, 2004.

[20] W. 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*, pp. 174–183, ACM, 2005.

[21] S. Bandhakavi, P. 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, (New York, NY, USA), pp. 12–24, ACM, 2007.

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

[23] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *ACM SIGPLAN Notices*, vol. 41, pp. 372–382, ACM, 2006.
[24] S. Boyd and A. Keromytis, "Sqlrand: Preventing sql injection attacks," in *Applied Cryptography and Network Security*, pp. 292–302, Springer, 2004.
[25] S. N. Narayanan, A. R. Pais, and R. Mohandas, "Detection and prevention of sql injection attacks using semantic equivalence," in *Computer Networks and Intelligent Computing*, pp. 103–112, Springer, 2011.
[26] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *Recent Advances in Intrusion Detection*, pp. 124–145, Springer, 2006.
[27] A. Orso, W. Lee, and A. Shostack, "Preventing sql code injection by combining static and runtime analysis," tech. rep., DTIC Document, 2008.
[28] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *ACM SIGPLAN Notices*, vol. 41, pp. 372–382, ACM, 2006.
[29] A. N. Bessani, "From byzantine fault tolerance to intrusion tolerance," 2012.
[30] A. Gorbenko, V. Kharchenko, O. Tarasyuk, and A. Romanovsky, "Using diversity in cloud-based deployment environment to avoid intrusions," *Software Engineering for Resilient Systems*, pp. 145–155, 2011.
[31] M. Garcia, N. Neves, and A. Bessani, "Diversys: Diverse rejuvenation system,"
[32] J. Antunes and N. Neves, "Diveinto: Supporting diversity in intrusion-tolerant systems," in *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pp. 137–146, IEEE, 2011.
[33] F. Majorczyk and J. Demay, "Automated instruction-set randomization for web applications in diversified redundant systems," in *Availability, Reliability and Security, 2009. ARES'09. International Conference on*, pp. 978–983, IEEE, 2009.
[34] B. Salamat, C. Wimmer, and M. Franz, "Synchronous signal delivery in a multi-variant intrusion detection system," tech. rep., Technical report, School of Information and Computer Sciences, University of California, Irvine, 2009.
[35] A. Avizienis and L. Chen, "On the implementation of n-version programming for software fault tolerance during execution," in *Proc. IEEE COMPSAC*, vol. 77, pp. 149–155, 1977.
[36] A. Avizienis, "The n-version approach to fault-tolerant software," *Software Engineering, IEEE Transactions on*, vol. SE-11, pp. 1491 – 1501, dec. 1985.
[37] M. Lyu, J. Chen, and A. Avizienis, "Software diversity metrics and measurements," in *Computer Software and Applications Conference*, pp. 69–78, 1992.
[38] S. Mitra, N. Saxena, and E. McCluskey, "A design diversity metric and analysis of redundant systems," *IEEE Trans. Comput.*, vol. 51, pp. 498–510, May 2002.
[39] B. Littlewood, P. Popov, and L. Strigini, "Modeling software design diversity: A review," *ACM Comput. Surv.*, vol. 33, pp. 177–208, June 2001.
[40] 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*, pp. 33–46, ACM, 2009.
[41] 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*, vol. 8, no. 4, pp. 588–601, 2011.
[42] A. Nguyen-Tuong, D. Evans, J. Knight, B. Cox, and J. Davidson, "Security through redundant data diversity," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pp. 187–196, IEEE, 2008.
[43] M. Franz, "E unibus pluram: massive-scale software diversity as a defense mechanism," in *Proceedings of the 2010 workshop on New security paradigms*, pp. 7–16, ACM, 2010.
[44] 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*, pp. 149–158, ACM, 2008.
[45] C. Huang, S. Zhu, and R. Erbacher, "Toward software diversity in heterogeneous networked systems," in *IFIP Annual Conference on Data and Applications Security and Privacy*, pp. 114–129, Springer, 2014.
[46] C. Huang, S. Zhu, and Q. Guan, "Multi-objective software assignment for active cyber defense," in *2015 IEEE Conference on Communications and Network Security (CNS)*, pp. 299–307, Sept 2015.
[47] "Midicart official site." http://www.midicart.se.
[48] F. Valeur, D. Mutz, and G. Vigna, "A learning-based approach to the detection of sql attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 123–140, Springer, 2005.
[49] M. Pawlik and N. Augsten, "Rted: a robust algorithm for the tree edit distance," *Proceedings of the VLDB Endowment*, vol. 5, no. 4, pp. 334–345, 2011.
[50] H.-H. Do and E. Rahm, "Coma: a system for flexible combination of schema matching approaches," in *Proceedings of the 28th international conference on Very Large Data Bases*, pp. 610–621, VLDB Endowment, 2002.
[51] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity flooding: A versatile graph matching algorithm and its application to schema matching," in *Data Engineering, 2002. Proceedings. 18th International Conference on*, pp. 117–128, IEEE, 2002.
[52] J. Madhavan, P. A. Bernstein, and E. Rahm, "Generic schema matching with cupid," in *Proceedings of the International Conference on Very Large Data Bases*, pp. 49–58, 2001.
[53] J. R. Quinlan, *C4. 5: programs for machine learning*, vol. 1. Morgan kaufmann, 1993.
[54] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.
[55] R. Kohavi *et al.*, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *IJCAI*, vol. 14, pp. 1137–1145, 1995.
[56] "General sql parser java official site." http://www.sqlparser.com/sql-parser-java.php.
[57] "Keyboard simulator official site." http://www.anjian.com/.

Appendix: The CVE Entries of Injection Vulnerabilities in Applications with Multiple Variants

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.