2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-7281-9630-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/SANER50967.2021.00040

SQLIFIX: Learning Based Approach to Fix SQL Injection Vulnerabilities in Source Code

Mohammed Latif Siddiq[§], Md. Rezwanur Rahman Jahin[§], Mohammad Rafid Ul Islam,
Rifat Shahriyar, Anindya Iqbal
Bangladesh University of Engineering and Technology
Email: 1505069.mls@ugrad.cse.buet.ac.bd, 1505065.mrrj@ugrad.cse.buet.ac.bd, rafid@openrefactory.com,
rifat@cse.buet.ac.bd, anindya@cse.buet.ac.bd

***Abstract*—SQL Injection attack is one of the oldest yet effective attacks for web applications. Even in 2020, applications are vulnerable to SQL Injection attacks. The developers are supposed to take precautions such as parameterizing SQL queries, escaping special characters, etc. However, developers, especially inexperienced ones, often fail to comply with such guidelines. There are quite a few SQL Injection detection tools to expose any unattended SQL Injection vulnerability in source code. However, to the best of our knowledge, very few works have been done to suggest a fix of these vulnerabilities in the source code. We have developed a learning-based approach that prepares abstraction of SQL Injection vulnerable codes from training dataset and clusters them using hierarchical clustering. The test samples are matched with a cluster of similar samples and a fix suggestion is generated. We have developed a manually validated training and test dataset from real-world projects of Java and PHP to evaluate our language-agnostic approach. The results establish the superiority of our technique over comparable techniques. The code and dataset are released publicly to encourage reproduction.**

***Index Terms*—SQL Injection, Prepared Statement, Automatic Fix**

I. Introduction

Structured query language (SQL) is the language to interact with relational databases. The interaction is performed using different SQL statements. The SQL injection (SQLI) attack exploits inputs of SQL statements. The attacks are usually accomplished by contaminating SQL queries with special characters or keywords. The attacker tries to alter the logic of the statement to read confidential data, to modify database records, and to corrupt/delete data. They sometimes take control of administrative operations or issue malicious commands to the operating system.

SQLI attacks are among the oldest and most deadly threats to web applications since the dawn of web databases. Even some reputed applications of large organizations fall victim to this. In August 2014, computer security company Hold Security found that SQL injections can be successfully used to disclose confidential information from almost 420,000 websites [1]. Seals [2] shows that SQLI was behind stealing the personal details of 156,959 customers from a British Telecommunications company in 2015. Even in 2020, an SQLI attack was used to access information from the server of Link, a start-up founded on Stanford campus [3]. In the first decade of this century, quite a few incidents of credit card information theft and millions of dollars of fraudulent purchases caused by SQLI were reported.

Automatic Program Repair (APR) is an increasingly popular research area to provide automatic fix suggestions for different types of errors/bugs. These techniques suggest some candidate changes which may repair a given fault. These techniques are expected to reduce significant tedious effort and time for bug fixing by the developers in the future. Recent improvements in advanced machine learning, especially deep learning, and the availability of large numbers of patches enable learning-based repair in addition to previous rule based approaches. In this work, our objective is to develop an APR tool to mitigate SQLI vulnerabilities with high quality.

The common suggestions to prevent SQLI include parameterizing queries instead of directly embedding user input in them, escaping the characters that have a special meaning in SQL, and checking the pattern of the parameters. However, developers often fail to comply with such guidelines, especially when there is a tight deadline. For new/inexperienced developers, this is not unexpected. To expose any unattended vulnerability, there are quite a few SQL injection detection tools. On the other hand, to the best of our knowledge, very few works have been done to help the developers fix these vulnerabilities quickly.

The early approaches [4]–[7] have proposed solutions for mitigating SQLI attacks. However, none of these solutions address the actual SQLI vulnerabilities that exist in the source code. Only a few existing works [8], [9] attempted to generate fix suggestions for SQLI by removing vulnerabilities from source code. These approaches propose changes in source code to remove SQLI vulnerabilities, whereas previous mitigating approaches would try to fortify against the SQLI attacks. [9] proposed to remove SQLI vulnerabilities from SQL statements by replacing them with secure prepared statements. However, their rule based approach is language-specific and not adaptive to future changes in the language. Their work on Java fails to cover advanced programming constructs such as Lambda expression, inline methods, etc. Also, they cannot fix multi-line queries (concatenated together). In this paper, we propose a learning based approach that overcomes these limitations. We have developed a solution for Java and PHP using this

[§]These authors contributed equally to this work.

978-1-7281-9630-5/21/$31.00 ©2021 IEEE
DOI 10.1109/SANER50967.2021.00040

language-agnostic approach. An essential requirement from APR techniques is a natural change that sustains the readability of changed code [10]. Our solution provides good results in this consideration as well.

Our approach, which is highly inspired by Bader et al. [10], is based on a hierarchical clustering where the higher nodes in the hierarchy are more abstract than the lower ones. It creates a dendrogram of edit patterns (the pattern we need to apply to the tainted model to fix vulnerability) where only the leaves contain concrete edits. When we get a new vulnerable code, we start matching the AST (Abstract Syntax Tree) of the given code with the "before_tree" (elaborated in Section III) of an edit pattern starting from the root of the cluster tree. We go down the dendrogram until we find a match for our AST and suggest the fix pattern if we find a match. We use 21 Java projects mined from GitHub, where 14 projects are used for training, and 7 projects are used for testing our model. In addition, we use 6 PHP projects mined from GitHub to train and 4 projects to test our model.

Specifically, the contributions in this paper are as follows:

- Designing an automated SQL injection fixing tool (SQLIFIX) that significantly outperforms the other available similar techniques.
- Adopting a learning based approach (unlike previous rule based ones) that is language agnostic, covers a wider range of injections and does not degrade readability of the source code.
- Generating training dataset (1200 for Java and 350 for PHP) and test dataset (300 for Java and 150 for PHP) manually from real-world projects. We release data and code to encourage reproduction: https://github.com/RRJahin/SQLIFIX.

Our approach generates solutions for 67.52% cases for Java, tested with 7 real-world Java projects. We compare our approach with the comparable approach [9], which generates correct solutions for 23.67% cases on the same test set. We extend our approach for PHP, which generates 41.33% correct solutions on an independent test set.

II. Motivating Examples

In this section, we present some examples that depict the objective of SQLIFIX. In Table I and II, we illustrate an example demonstrating the utility of our approach. There is an SQL query with multiple conditions and another select query is embedded in a condition. This query is vulnerable to SQL Injection in several ways. For example, if the input for *NAME* is **'John' OR 1=1**, this query will provide *IDs* of all students from the *STUDENTS* table instead of the information for the row with *NAME = 'John'* assuming that all other conditions are satisfied.

This vulnerability can be removed by using *PreparedStatement* class instead of using *Statement* class from JDBC. Because for *PreparedStatement* class, the database engine doesn't combine the bound variables into the SQL statement and then parse the whole thing; the bound variables are kept separate and never parsed as a generic SQL statement. In the corresponding changed code presented in Table II, i.e., code snippet after applying the fix suggestion, the vulnerability is removed without reducing naturalness or readability.

TABLE I
SOURCE CODE

```
String sql = "SELECT ID FROM STUDENTS WHERE"
      + " NAME = "+ name + " AND PATRONYMIC ="
      + patro + "AND DATE_OF_BIRTH = "
      + dob + " AND GROUP_ID = "
      + "(SELECT ID FROM GROUPS WHERE NUMBER = "
      + groupNum+ ")";
Statement ps = con.createStatement();
rs = ps.executeQuery(sql);
```

TABLE II
CHANGED CODE

```
String sql = "SELECT ID FROM STUDENTS WHERE "
      + "NAME = ? AND PATRONYMIC = ? "
      + "AND DATE_OF_BIRTH = ? "
      + "AND GROUP_ID = (SELECT ID FROM GROUPS "
      + "WHERE NUMBER = ?)";
PreparedStatement ps = con.prepareStatement(sql);
ps.setObject(1,name);
ps.setObject(2,patro);
ps.setObject(3,dob);
ps.setObject(4,groupNum);
rs = ps.executeQuery();
```

The target of SQLIFIX is to generate fix suggestions from this type of vulnerable code, keeping readability unchanged. We present some more examples generated for real-world Java and PHP projects in Table III and IV. These cover different types of injections that we can successfully address.

- **Source Code 1**: In this example, there is a single payload, *response*. By varying payload for this variable, we can simulate different types of SQLI. For example: if the payload is **105'**, it causes an error but hackers can gather information by this **Error-Based SQLI** before launching the actual attack.
- **Source Code 2**: In this example, there are multiple scopes for launching SQLI. If the payload for the variable *initial* is **Mr. −−**, then the SQLI is occurred for UPDATE Query. If the payload for *countryname* is **X) OR 1=1; −−**, then this is an example of SQLI by **Tautologies**.
- **Source Code 3**: In this example, there are multiple ways to launch SQLI. If the payload of the variable *loc* is **X)); DROP Users; −−** , hackers can launch SQL Injection with **Piggybacked Query**.
- **Source Code 4**: In this example, it is an **SQLI by LIKE query and column number mismatch**. For example, if the payload for *productName* is **UNION ALL SELECT 9,9, 9,' Text', 9 FROM SysObjects WHERE '' = ''**, then result set of this query will show all the rows in the SysObjects table and will also show constant row values for each row in the SysObjects table defined in the query [11].

- **Source Code 5**: In this example, there is a **Second Order SQLI**. If the payload for the variable *uname* is **X; DROP Users; −−**, then the first query safely stores the input in the database. But the second query causes an SQLI. By using payload **X UNION select * from CreditDetails**, it can also cause **Union-Based SQLI**.
- **PHP Source Code 1 and 2**: In those example source codes for PHP, various types of SQLI such as, **Error-Based SQLI**, SQLI by **Piggybacked Queries** etc. can occur.

III. Methodology

Our approach for SQLIFIX can be divided into three main components.

1) The data collection and preparation phase,
2) Model building and learning phase,
3) Prediction phase.

Here, we will present an overview of the above three phases and elaborate on each component in subsequent subsections. The conceptual flow of the whole approach is shown in Figure 1. In the data collection phase, we mined various Java projects from GitHub and extracted the functions that have SQL injection vulnerability to generate our training dataset. We also created corresponding hand-crafted fixes for the SQLI vulnerable code snippets. Then, we feed the training data to a hierarchical clustering based model in the learning phase. It would cluster the similar type of changes together and help to match the structure of a given code to previously seen structures and do accurate predictions later in the prediction phase. We relied on static analysis warnings to prepare the training data and evaluate our predicted fixes.

A. Data Collection and Preparation

We mined 40 random Java projects from GitHub using their API[1]. Then, we used SpotBugs[2] to identify the codes that have SQL injection vulnerability. From the 40 projects mined from GitHub, we found 21 projects that were vulnerable to SQL injections using SpotBugs. We used these vulnerable projects to create our initial corpus. Then, we separated them into *two* groups, i.e., training set and testing set. A detailed description of the *two* groups is given below.

1) Training Set: The training set consists of 14 projects (Table - V) out of the 21 projects which we found to be vulnerable to SQL injections. To prepare these projects for our model, we created a java file for each vulnerable method that exist in training data and paired them with their "Prepared Statement" solution by following the OWASP guidelines [12]. The vulnerable code and the fix code are dubbed as "before_code" and "after_code", respectively throughout the whole paper. In total,we got 1200 training data (tainted methods) from the 14 vulnerable real world projects. This number is sufficient for training a hierarchical clustering based model as we made sure that we covered all possible patterns of SQL injection.

2) Test Set: The test set is entirely made from the remaining 7 vulnerable projects (Table VII). As test data, we have about 300 files having about 430 infected methods detected by SpotBugs. After the training is done, we evaluated our model using this data. As the test data came from projects which were not in the training set, there is no scope for data leakage.

B. Data Preprocessing

We used a tree-differencer, GumTree (Falleri et. al. 2014 [13]), to determine lines that are different in the vulnerable code and its fix code. These lines were parsed using JavaParser[3] to make an AST (Abstract Syntax Tree). We saved both the AST of vulnerable code snippet and that of the patch into JSON files which became the input for our training mechanism. We also used the SQL parser, GSP[4], to parse the SQL query. Next, we replaced the Java representation of the SQL query with the AST received from GSP into the input JSON files.

These prepared input files contain the following information.

- **before_code:** The vulnerable code snippet that gets modified to be fixed.
- **after_code:** The code snippets that have been modified to fix the vulnerability.
- **before_tree:** The AST representation of the *before_code*.
- **after_tree:** The AST representation of the *after_code*.

We created a similar dataset for PHP to show that our learning model is language agnostic. There are 350 training data and 150 test data for PHP. The evaluation results for both datasets are elaborated in Section IV.

C. Training Model

SQLIFIX uses complete-linkage clustering, an agglomerative hierarchical clustering algorithm that creates a hierarchy of edit patterns. The higher we go in the hierarchy, the more generalized the edit patterns become. We achieve this generalization by merging the edit patterns through anti-unification (Kutsia et. al. [14]) which is an approach of generalization among different symbolic expressions. Our work is inspired by GetaFix (Bader et. al. [10]).

We adopted nearest neighbor chaining algorithm to speed up the clustering process. As mentioned before, we used anti-unification to merge two nodes to achieve abstraction for the higher order nodes. First, we anti-unify two before_tree nodes and after_tree nodes to obtain a new edit pattern containing a more abstract before_tree and after_tree. We introduced "holes" (pattern variables) in place of "variables" and other expressions in the merged pattern. These holes represent the parts of the tree where concrete edits differ in the original trees.

We used these holes to determine the distance between two nodes in the hierarchical structure. As the complete-linkage hierarchical clustering algorithm warrants the merging

[1]https://developer.github.com/v3/
[2]https://spotbugs.github.io/
[3]https://javaparser.org/
[4]http://www.sqlparser.com/

TABLE III
SOME EXAMPLES OF FIXATIONS GENERATED FOR JAVA

Possible Types of SQLI: Error-Based SQLI

Source Code 1	Changed Code 1

`String sql = "select accounts.* ,fName, lName "`
`+ "from users inner join "`
`+ "account_holders as ah on ah.user_id = "`
`+ "users.id inner join accounts on "`
`+ "accounts.id = ah.a where account.id = "`
`+ response +";";`
`Statement ps = con.createStatement();`
`rs = ps.executeQuery(sql);`
	`String sql = "select accounts.* ,fName, lName "`
	`+ "from users inner join "`
	`+ "account_holders as ah on ah.user_id = "`
	`+ "users.id inner join accounts on "`
	`+ "accounts.id = ah.a where account.id = ?;"`
	`PreparedStatement ps = con.prepareStatement(sql);`
	`ps.setObject(1,response);`
	`rs = ps.executeQuery();`

Possible Types of SQLI: SQLI by Tautologies

Source Code 2	Changed Code 2

`Statement ps = con.createStatement();`
`rs = ps.executeUpdate("Update Users set firstname = "+ initial+"+ firstname where countryid = (select countryid from Country where countryname = "+ countryname+");");`
	`PreparedStatement ps = con.prepareStatement("Update Users set firstname = ? firstname where countryid = (select countryid from Country countryname = ?);");`
	`ps.setObject(1,initial);`
	`ps.setObject(2,countryname);`
	`rs = ps.executeUpdate();`

Possible Types of SQLI: SQLI with Piggybacked Query

Source Code 3	Changed Code 3

`String sql = "Insert into User(uname,upass,ulocid)"`
`+ "values("+name+","+pass+",(select locid "`
`+ "from location where loc="+loc+"));";`
`Statement ps = con.createStatement();`
`rs = ps.executeUpdate(sql);`
	`String sql = "Insert into User(uname,upass,ulocid)"`
	`+ "values(?,?,(select locid "`
	`+ "from location where loc=?));";`
	`PreparedStatement ps = con.prepareStatement(sql);`
	`ps.setObject(1,name);`
	`ps.setObject(2,pass);`
	`ps.setObject(3,loc);`
	`rs = ps.executeUpdate();`

Possible Types of SQLI: SQLI with LIKE query and column number mismatch

Source Code 4	Changed Code 4

`String sql = "SELECT productName FROM allProduct "`
`+ "WHERE productName LIKE "+productName+";";`
`Statement ps = con.createStatement();`
`rs = ps.executeUpdate(sql);`
	`String sql = "SELECT productName FROM allProduct "`
	`+ "WHERE productName LIKE ?;"`
	`PreparedStatement ps = con.prepareStatement(sql);`
	`ps.setObject(1,productName);`
	`rs = ps.executeUpdate();`

Possible Types of SQLI: Second Order SQLI and Union-Based SQLI

Source Code 5	Changed Code 5

`String sql = "Insert into Users (username,email) values(?,?");";`
`PreparedStatement ps = con.prepareStatement(sql);`
`// No problem to store in DB`
`ps.setObject(1,uname);`
`ps.setObject(2,uemail);`
`ps.executeUpdate();`
`Statement sta = con.createStatement();`
`// Possible Scope for SQLI`
`ResultSet rs = sta.executeQuery("Select * from Users Where username = "+uname+";");`
	`String sql = "Insert into Users (username,email) values(?,?");";`
	`PreparedStatement ps = con.prepareStatement(sql);`
	`// No problem to store in DB`
	`ps.setObject(1,uname);`
	`ps.setObject(2,uemail);`
	`ps.executeUpdate();`
	`PreparedStatement sta = con.prepareStatement("Select * from Users Where username = ?;");`
	`// Protected against SQLI`
	`sta.setObject(1,uname);`
	`ResultSet rs = sta.executeQuery();`

TABLE IV
SOME EXAMPLES OF FIXATIONS GENERATED FOR PHP

PHP Source Code 1	PHP Changed Code 1

`$name = $_POST['name'];`
`$conn->query("SELECT * FROM myTable WHERE name='$name'");`
`$conn->close();`
	`$name = $_POST['name'];`
	`$stmt = $conn->prepare("SELECT * FROM myTable WHERE name= ?");`
	`$stmt->bind_param("s", $name);`
	`$stmt->execute();`
	`$stmt->close();`

PHP Source Code 2	**PHP Changed Code 2**

`$conn->query("INSERT INTO userTable (name, email) VALUES ('{$_POST['name']}', '{$_POST['email']}')");`
`$conn->query("UPDATE guestTable SET name = '{$_POST['name']}' WHERE email = '{$_POST['email']}'");`
	`$stmt1 = $conn->prepare("INSERT INTO userTable (name, email) VALUES (?, ?)");`
	`$stmt2 = $conn->prepare("UPDATE guestTable SET name = ? WHERE email = ?");`
	`$stmt1->bind_param("ss", $_POST['name'], $_POST['email']);`
	`$stmt2->bind_param("ss", $_POST['name'], $_POST['email']);`
	`$stmt1->execute();`
	`$stmt2->execute();`
	`$stmt1->close();`
	`$stmt2->close();`

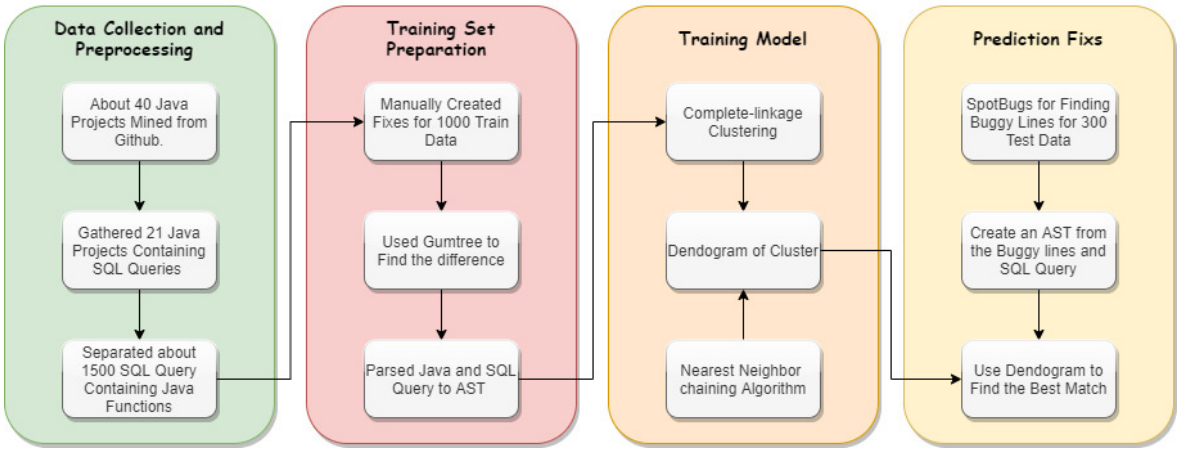

Fig. 1. Conceptual model of our methodology

TABLE V
PROJECTS USED FOR JAVA TRAINING SET

https://github.com/MahnuelO/InventarioWeb
https://github.com/hencjo/summer-migration
https://github.com/mariadb-corporation/mariadb-connector-j
https://github.com/Minal11/UserVerificationSystem
https://github.com/zxybazh/Online-Book-Store-System
https://github.com/zj386018/zjjava
https://github.com/zooncool/crawledemo
https://github.com/xingkuan/vSync
https://github.com/victormartor/TFG_PC
https://github.com/VeePeeK/JavaGit
https://github.com/tmwsiy2012/fiftyfiftystockscreener
https://github.com/texas-mule/banking-app-devkala48
https://github.com/sashanksridhar/Ecomerce-
https://github.com/princekm/GUI-DBMS

TABLE VI
PROJECTS USED FOR PHP TRAINING SET

https://github.com/mmockelyn/GC2i
https://github.com/yashkrsingh/Pool-Your-Cab
https://github.com/Shivanithakur94/newfilesj
https://github.com/cumbach/ParadataChrisUmbach
https://github.com/cumbach/ParadataChrisUmbach
https://github.com/flowerszhong/hfx

of two nodes with the smallest distance. We achieved this by calculating the maximum information gain and coverage between two nodes. We considered two nodes to have more information if they have similar sub-tree structure, but higher hole count after merging. The result of this clustering process is a dendrogram containing the abstract edit patterns as the middle nodes and the concrete edits as leaves.

To optimize further, if the new edit pattern is similar to any of the two before_tree nodes from which the new edit patter is generalized, that before_tree node is considered the parent

TABLE VII
PROJECTS USED FOR JAVA TESTING SET

https://github.com/Ronak6892/iTrust-v23
https://github.com/gschmidt1/Momow1
https://github.com/IraDaniel/java-homework-10
https://github.com/danielmmy/Naegling-GUI
https://github.com/rmariscal13/Java_BBDD
https://github.com/vicarmocanu/Hotel-Management---Java
https://github.com/npafitis/EPL441Clinic

TABLE VIII
PROJECTS USED FOR PHP TESTING SET

https://github.com/NukeVietCMS/module-shops
https://github.com/Ricky7/butik_online
https://github.com/urki/urki-test-project
https://github.com/jbennett122/JBENNETT_CS418

of the other one which is dissimilar. This helps to reduce the tree height as all the nodes producing similar pattern after anti-unification are children of a single node.

Since hierarchical clustering algorithms cluster the different patterns in a hierarchical structure, it does not need more data like the usual learning algorithms. Instead, it relies heavily on the variation of patterns in data.

D. Fixer

After the training is done, the model is saved as a large tree with a common root. In the dendrogram, every node represents an edit pattern which were the abstract edit result of the merging of its children. The leaves represent concrete edits. When we get a test code (vulnerable Java method), we use Static Analyzer to spot all the vulnerable queries present in the method and then use JavaParser to track all the lines associated with each query. After that, we generate separate AST for each of them and start matching those one at a time with every node of the dendrogram, starting from the root. We make our way down the tree and match the given AST with the "before_tree" of the abstract edit patterns. We stop checking the children of a node if that node's "before_tree" does not match with the given AST. For the matching mechanism, we followed the matching criteria from GetaFix. When we reach to a leaf node in this way, that leaf's "after_tree" gives us the pattern that will fix our vulnerable test code. Here we have used a slightly modified anti-unification method to identify the variables that serve the same purpose with a different name as these variables need to be replaced to generate the actual suggestion. The anti-unification used in this case goes on recursive call if only type and number of children are same, whereas, for the general version the variable name also needs to be same along with the other two to do the same.

IV. RESULTS

In this section, we discuss the experimental results and relevant findings of our work.

A. SQLIFIX for Java

As discussed earlier, we developed SQLIFIX by training with the dataset from Java projects and then from PHP projects following the same approach. We took 300 Java files from 7 projects as the testing dataset that were not included in the training dataset. The list of projects can be found in Table V and Table VII. From there, 430 code segments containing vulnerable queries were separated. Our model generated solutions for 291 of these 430 code segments (67.52%). Then we replaced the buggy codes with fix suggestions and again tested them with SpotBugs. This time 78.69% of the vulnerabilities previously detected were considered benign by SpotBugs. We manually inspected the remaining suggestions. Out of these, 19.93% of the suggestions needed a little tweak in variable names and then got accepted. Only 1.37% of the suggestions were found irrelevant. Here is an example of a suggestion that required little modification:

```
Statement stmt = connector.createStatement();
String snewer = "select userid from usersr
                    where username='" + s + "'";
ResultSet rs = stmt.executeQuery(snewer);
```

Input Code Segment

```
String snewer = "select userid from ?
                    where username='?'";
PreparedStatement stmt =
              connector.prepareStatement(snewer);
stmt.setObject(1,usersr);
stmt.setObject(2,s);
ResultSet rs = stmt.executeQuery();
```

Generated Suggestion

In the query of the generated suggestion, 'usersr' should not be replaced with a place holder. Hence, it requires some attention from the developer.

	Number	Percentage
Vulnerable Code Segment	430	
Solution Found	291	67.52%
Accurate (Among the Solutions)	229	78.69%
Close (Among the Solutions)	58	19.93%
Unrelated (Among the Solutions)	4	1.37%

B. SQLIFIX for PHP

To show that our model is language agnostic, we created a dataset of 350 training data from 6 projects and 150 test data from 4 PHP projects. The list of projects can be found in Table VI and Table VIII. Our model generated suggestions for 130 test samples (86.67%). We analyzed our result and it provided 62 accepted fixations (41.33% of total test data). Our tool has some difficulties in providing solutions for PHP as the string structure can be different in PHP than in Java. For example, PHP supports the single-quoted string where Java does not.

C. Comparison with the PSR algorithm

Prepared statement replacement algorithm (PSR) [9], a rule based SQLIV fixer, gathers information from source code

containing SQLIVs and generates secure prepared statement code that maintains functional integrity. We have collected the source code of the PSR algorithm from the authors. Our SQLIFIX generates solutions for 67.52% cases for Java, tested with 7 real-world Java projects. PSR generates solutions for 23.67% cases on the same test set. So, our SQLIFIX performs significantly better than PSR. We also extend SQLIFIX for PHP, which generates 41.33% correct solutions on an independent test set.

D. Limitations of the PSR algorithm

We found the following limitations in the PSR algorithm that were overcome by SQLIFIX.

- **Style Dependencies**: The PSR algorithm has some style dependency issues. For example:

```
public ResultSet getUser(String id)
{
    Connection conn = DB.getConnection();
    Statement st = conn.createStatement();
    String q = "select * from user where ID ="+id;
    ResultSet rs = st.executeQuery(q);
    ResultSet rs;
}
```

If the vulnerability is encapsulated as the previous code format, the PSR algorithm can not provide a solution. The correct style format should be:

```
public ResultSet getUser(String id) {
    Connection conn = DB.getConnection();
    Statement st = conn.createStatement();
    String q = "select * from user where ID ="+id;
    ResultSet rs = st.executeQuery(q);
    return rs;
}
```

- **Missing Access Modifier**: The vulnerability container function and class should have an access modifier, i.e., public, private, and protected. Otherwise, the PSR algorithm can not provide any solution. For example:

```
  class UserDB {
    void updateUser(String init, String id) {
        Connection conn = DB.getConnection();
        Statement st = conn.createStatement();
        String q = "Update userTable "+
            "Set userName = "+init+" userName "+
            "where userID = "+id;
        st.executeUpdate(q);
    }
  }
```

The PSR algorithm could not provide a solution for the previous code because there is no access modifier before class (or function).

- **Annotation Before Function**: To refactor the vulnerable code, there should be nothing between the class name and function name, which contains the vulnerabilities. Otherwise, the PSR algorithm can not fix them. For example, if there is an annotation before the function name, it can not provide a solution.

```
class UserDB {
    @override
    public ResultSet getUsersInfo(String response) {
        Connection conn = DB.getConnection();
        String sql = "select fName, lName "
            + "from users inner join "
            + "acc_holders as ah on ah.userID = "
            + "users.id inner join accounts on "
            + "accounts.id = ah.a "
            + "where account.id = "
            + response +";";
        Statement ps = conn.createStatement();
        return ps.executeQuery(sql);
    }
}
```

In the previous code snippet, no solution was available because there is an annotation between class and function name.

- **Multi-line String**: The PSR algorithm fails to provide a solution if the SQL query string is a multi-line string. For example:

```
void removeUser(Connection conn, String id) {
    Statement st = conn.createStatement();
    String sql = "Delete from userTable "
        + "where userID = "+id;
    st.executeUpdate(sql);
}
```

In the previous example, String variable q is declared with multiple lines of code, which caused the failure.

- **Elimination of Comments**: The PSR algorithm can not keep single line comments in solution and can not find a solution if there are multi-line comments. For example:

```
public void getUser(String id) {
    // This comment will not be in the solution
    /*
        For this comment, PSR algorithm
        can not find a solution.
    */
    Connection conn = DB.getConnection();
    Statement st = conn.createStatement();
    String q = "select * from user "
        + "where id = "+id;
    st.executeQuery(q);
}
```

E. Limitations of SQLIFIX

We could not find solutions for 139 code segments. Here are the major reasons behind our failure to generate a solution:

- **Presence of query modifiers like SQL_CALC_FOUND_ROWS, LIMIT, etc.**: General SQL Parser (GSP), the tool used for parsing SQL queries, can not parse if these modifiers are present.
- **Erroneous Code or Query**: JavaParser or GSP cannot parse.
- **Batch Query**: there is no real equivalent implementation of the batch query in PreparedStatement.

These hindrances are either due to the limitation of the tools we used for parsing (GSP, JavaParser) or because of the structure of PreparedStatement (batch query). Correspondingly, JavaParser cannot parse if there is a presence of any error in the Java code. If these obstacles could be overcome, the performance would have improved.

Conversely, there are some limitations of the current version of SQLIFIX. SQLIFIX will not generate a solution for the following cases:

- If any query generation requires project-level knowledge, i.e., knowledge from other files or classes.
- Query prepared using string concatenation from multiple string variables containing a portion of the query.
- Lack of coverage of all kinds of patterns in our training data.

```
String query = table_name.getText();
String s1 = "select * from ";
query = s1.concat(query);
```

- Use of ternary operator for query selection.

```
String query = x < y ?
    "select * from X" : "select * from Y";
```

In the future release of SQLIFIX, we have plans to introduce project-level knowledge to our model. Thus, we will be able to suggest solutions even if related code segments from other functions or classes other than the one containing the query are needed. The training data set is also being enriched consistently.

V. Related Work

In this work, we have developed a technique to provide an automated fix suggestion for removing SQLI vulnerabilities. It would use an existing approach for the detection of SQLI vulnerability in the code. Hence, we consider SQLI detection approaches, APR techniques, and the few automatic SQLI repair techniques relevant to our study.

A. Automated Program Repairing Approaches

APR is an emerging area of research with a high prospect of benefiting software developers. Researchers have been trying to automatically repair software codes by generating an actual fix for more than two decades [15]. Among the well-studied works, Genprog [16]–[18] is a Genetic Algorithm based technique using test suites. SemFix [19] and Angelix [20] used symbolic execution to repair programs. Samimi et al. [21] worked specifically for PHP programs. Reference [22] parameterized a manually written bug report and extracted necessary values from the report to repair the programs.

Application of the learning-based approaches to detect and fix bugs have shown promising result in recent years. Among the state-of-the-art techniques, [23], [24] presented by Tufano *et al.* successfully applied Neural Machine Translation (NMT) for program repair. Although the accuracy achieved (9%) [23] is not acceptable, it divulges a new research avenue. They also applied the seq2seq model with attention mechanism [25] for repairing Java functions within a specified length [24]. SequenceR [26] used a sequence-to-sequence model to replace a buggy code segment with a correct one performing similar work. They used Copy mechanism [27] to solve infinite variable problem [28] for program repair. Another recent research CODIT [29] applied seq2seq model with attention [25] and copy mechanism for program repair. ENCORE [30] used an ensemble of multiple CNN based Neural Machine Translation models on improving the performance. Getafix [10] uses static analysis warnings to detect possible bugs and suggests fixes of the same bug category. It splits a given set of example fixes into AST-level edits and learns recurring fix patterns from these using a hierarchical clustering technique. It finds the cluster with a close match, ranks candidate fixes of that cluster, and selects the best match as a fix suggestion.

Facebook's Getafix (Bader et al. [10]) is a tool that learns from developers' past code fixes to suggest new concrete fixes for current bugs in the code. Our work is highly inspired by this tool. Getafix proposes a new hierarchical clustering algorithm for pattern mining, combining with anti-unification (an well-known method for the generalization of different symbolic expressions). It creates a dendrogram of possibly related tree differences and uses the fix patterns representing the most common code transformation. These patterns can be abstract, containing "holes" (abstract symbols to represent concrete patterns) where program transformations differ. In the final step, it takes buggy source code and the dendrogram created in the pattern mining step to produce concrete fix patterns. It also employs a ranking technique to select the most relevant fixes for a particular bug.

B. SQL Injection Detection Approaches

The SQLI vulnerability detection approaches can be grouped into static and dynamic testing approaches. In this work, we have adopted static approaches for the detection and localization of vulnerable code snippets. Hence, while discussing related works, our major focus would be on such techniques.

SOFIA [31] has been proposed as a programming-language and source-code independent tool which parses SQL statements and creates parse trees that are then fed to a clustering algorithm. Then the tree edit distance is used to measure the distance among the parse trees. Shar and Tan [32] proposed to detect SQLI vulnerable code by characterizing input function into a pattern of code attributes. It used a vulnerability prediction model to predict vulnerable code. This approach was implemented for PHP applications. Shahriar and Zulkernine [33] computed the entropy of SQL statements in source code. When an SQL statement is invoked, they will compute the entropy again to identify any change in the entropy measure for that query. Then they consider that dynamic queries with attack inputs result in an increased or decreased level of entropy, whereas a dynamic query with benign inputs does not result in any change of entropy value. Zhang and Wang [34] proposed another static analysis based technique to detect XSS and SQL vulnerabilities by using morphological analysis. However, these works cannot detect unknown patterns of SQL injection attacks.

AMNESIA [4] proposed a method based on program analysis to implement non-deterministic finite automata-based models for all the benign queries. The scheme validates each SQL query by finding an accepting path in the automaton. Failure to do so would identify the query as an attack. Another notable

work CANDID [35] dynamically mined the programmer-intended query structure on any input and detected attacks by comparing it against the structure of the actual query issued.

Another notable work by Islam et el. [36] performed the automatic detection of database injections for NoSQL utilizing several supervised learning techniques and showed that these algorithms detect injections better than the traditional rule based systems.

C. Automatic Fix for SQL injection

SQLCHECK, proposed by Su and Wassermann [7], performs a static analysis of an SQL statement parse tree and wraps the generated input validation code statement. It stopped all 18424 SQLIAs without generating any false positives on five open-source web application projects. This parse tree approach effectively identifies SQL statements structure and detects potential SQLIAs by comparing these structures. The parse tree approach focuses on the structure of the attacks instead of the removal of the SQLIVs.

WebSSARI, proposed by Huang et al. [37], performs a statical analysis of source code to find potential vulnerabilities, including SQLIVs. This tool inserts runtime guards into the source code that sanitizes input. It is effective to prevent general input manipulation attacks by sanitizing input on 230 open source web applications. This solution focuses on white and blacklisting on input rather than removal of the vulnerability.

SQLGuard, proposed by Buehrer et al. [38], secure vulnerable SQL statements by comparing the statement at the runtime with the parse tree of the original statement and allowing execution if they match. It stopped the four SQLIA types mentioned by their papers without any false positives in one student created web application. SQLGuard does add a computational overhead of dynamic SQL statement validation.

AMNESIA, proposed by Halfond and Orso [4], secure vulnerable SQL statements through static analysis, statement generation, and runtime monitoring. AMNESIA generates a generalized statement structure model for a vulnerable SQL statement by analyzing it and then allows/denies the statement based on comparing the model at runtime. Their solution stopped all of the SQLIAs in their attack set without generating any false positives on five open-source web applications. AMNESIA adds a computational overhead due to an additional process that has to be integrated into the runtime environment.

TAPS, proposed by Prithvi et al. [39], is a symbolic execution technique for query parameterization to make it safe for SQL injection. The main assumption behind the tool is it requires the web application to be transformed, to not perform content processing or inspection of partial query string variables. It analyzes the parsed structure of the SQL statements and identifies data arguments for the parameterized query. Then it traverses the program backwards to the program statements to generate these arguments, and substitutes the arguments with placeholders (i.e., the symbol "?"). For the three largest applications(WarpCMS, Utopia NewsPro, and AlmondSoft), TAPS transformed 93%, 99%, and 81% of the analyzed control flows. However, it requires developer intervention if either one of the following conditions hold: (i) the main assumption of the tool is violated, (ii) a well-formed SQL query cannot be constructed statically (e.g., use of reflection, library callbacks, etc.), (iii) the SQL query is malformed because of infeasible paths that cannot be determined statically, (iv) conflicts are detected along various paths, and (v) query is constructed in a loop that cannot be summarized.

Aharon et al. [40] presents Code-Motion for API Migration, an algorithm that performs API migration (statement is replaced by PreparedStatement), moving code as necessary to preserve functionality while changing the structure as little as possible. The proposed algorithm is language specific (designed for Java only) and it is confined to special scenarios such as having similar type and same number of variables for conditional queries, no inter-procedural building of query, etc. In the paper they have grouped SQLI vulnerabilities into six categories, and the proposed algorithm is claimed to be applicable for half of them. However, only one of the six is implemented and evaluated.

Prepared statement replacement algorithm (PSR), proposed by Thomas et al. [9], remove SQLI vulnerabilities by replacing SQL statements with prepared statements. The static structure of prepares statements prevents SQLIA from changing the logical structure of a prepared statement. The generator generates automated fixes using the PSR algorithm. It can correctly replace 94% of the SQLIAs of 4 open source projects. They focused on removing SQLIVs instead of just mitigation and chose automated generation to prepared statements, whereas the manual conversion to prepared statements is tedious, complex, and error-prone.

VI. Conclusion

In this paper, we have presented a learning-based SQL injection fix tool, SQLIFIX. It prepares abstraction of SQL Injection vulnerable codes from the training dataset and clusters them using hierarchical clustering. The test samples are matched with a cluster of similar samples and a fix suggestion is generated. We have prepared a manually validated training and test dataset from real-world projects of Java and PHP to evaluate our language-agnostic approach. The results establish the superiority of our technique over comparable techniques by 3 times. In the future, we plan to generate a large set of synthetic training data and try to improve the performance of the model further. Also, since NoSQLInjection can also be detected by machine learning based approaches [36], we would like to explore if our approach works for fixing them as well.

Acknowledgements

The research was partially supported by the grant 'Code Review Quality Measurement' granted to the Department of Computer Science and Engineering of Bangladesh University of Engineering and Technology from Samsung Research Bangladesh.

REFERENCES

[1] D. Poeter, "Close-Knit russian hacker gang hoards 1.2 billion id creds," PC Magazine, 2014, accessed: 2021-01-11.

[2] T. Seals, "TalkTalk: SQL injection possible vector for isp breach," Infosecurity Magazine, 2015, accessed: 2021-01-11.

[3] S. Catania, "Vulnerability in link website may have exposed data on stanford students' crushes," The Stanford Daily, 2020, accessed: 2021-01-11.

[4] W. G. J. Halfond and A. Orso, "Amnesia: Analysis and monitoring for neutralizing sql-injection attacks," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 174–183. [Online]. Available: https://doi.org/10.1145/1101908.1101935

[5] W. G. J. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter sql injection attacks," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: Association for Computing Machinery, 2006, p. 175–185. [Online]. Available: https://doi.org/10.1145/1181775.1181797

[6] W. G. J. Halfond, J. Viegas, and A. Orso, "A classification of SQL injection attacks and countermeasures," in *International Symposium on Secure Software Engineering*, 2006. [Online]. Available: https://www.cc.gatech.edu/~orso/papers/halfond.viegas.orso.ISSSE06.pdf

[7] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 372–382. [Online]. Available: https://doi.org/10.1145/1111037.1111070

[8] F. Dysart and M. Sherriff, "Automated fix generator for SQL injection attacks," in *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, 2008, pp. 311–312. [Online]. Available: https://doi.org/10.1109/ISSRE.2008.44

[9] S. Thomas, L. Williams, and T. Xie, "On automated prepared statement generation to remove SQL injection vulnerabilities," *Information and Software Technology*, vol. 51, no. 3, pp. 589 – 598, 2009. [Online]. Available: https://doi.org/10.1016/j.infsof.2008.08.002

[10] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360585

[11] K. Ahmad, J. Shekhar, and K. Yadav, "Classification of sql injection attacks," *VSRD Technichal & Non-technical Journal*, vol. I, no. 4, pp. 235–242, 2010.

[12] "OWASP cheat sheet series - SQL injection prevention cheat sheet," https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html, accessed: 2021-01-11.

[13] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 313–324. [Online]. Available: https://doi.org/10.1145/2642937.2642982

[14] T. Kutsia, J. Levy, and M. Villaret, "Anti-unification for unranked terms and hedges," *Journal of Automated Reasoning*, vol. 52, p. 155–190, 2014. [Online]. Available: https://doi.org/10.1007/s10817-013-9285-6

[15] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, p. 34–67, Jan. 2019. [Online]. Available: https://doi.org/10.1109/TSE.2017.2755013

[16] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 947–954. [Online]. Available: https://doi.org/10.1145/1569901.1570031

[17] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM*, vol. 53, no. 5, p. 109–116, May 2010. [Online]. Available: https://doi.org/10.1145/1735223.1735249

[18] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. USA: IEEE Computer Society, 2009, p. 364–374. [Online]. Available: https://doi.org/10.1109/ICSE.2009.5070536

[19] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 772–781. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606623

[20] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 691–701. [Online]. Available: https://doi.org/10.1145/2884781.2884807

[21] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren, "Automated repair of html generation errors in php applications using string constraint solving," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 277–287. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227186

[22] C. Liu, J. Yang, L. Tan, and M. Hafiz, "R2fix: Automatically generating bug fixes from bug reports," in *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ser. ICST '13. USA: IEEE Computer Society, 2013, p. 282–291. [Online]. Available: https://doi.org/10.1109/ICST.2013.24

[23] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 4, Sep. 2019. [Online]. Available: https://doi.org/10.1145/3340544

[24] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 25–36. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00021

[25] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.0473

[26] Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019. [Online]. Available: https://doi.org/10.1109/TSE.2019.2940179

[27] J. Gu, Z. Lu, H. Li, and V. O. Li, "Incorporating copying mechanism in sequence-to-sequence learning," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1631–1640. [Online]. Available: https://www.aclweb.org/anthology/P16-1154

[28] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 763–773. [Online]. Available: https://doi.org/10.1145/3106237.3106290

[29] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020. [Online]. Available: https://doi.org/10.1109/TSE.2020.3020502

[30] T. Lutellier, L. Pang, H. V. Pham, M. Wei, and L. Tan, "ENCORE: ensemble learning using convolution neural machine translation for automatic program repair," *CoRR*, vol. abs/1906.08691, 2019. [Online]. Available: http://arxiv.org/abs/1906.08691

[31] M. Ceccato, C. D. Nguyen, D. Appelt, and L. C. Briand, "Sofia: An automated security oracle for black-box testing of SQL-injection vulnerabilities," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 167–177. [Online]. Available: https://doi.org/10.1145/2970276.2970343

[32] L. K. Shar and H. B. K. Tan, "Predicting common web application vulnerabilities from input validation and sanitization code patterns," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: Association for Computing Machinery, 2012, p. 310–313. [Online]. Available: https://doi.org/10.1145/2351676.2351733

[33] H. Shahriar and M. Zulkernine, "Information-theoretic detection of SQL injection attacks," in *2012 IEEE 14th International Symposium*

on High-Assurance Systems Engineering, 2012, pp. 40–47. [Online]. Available: https://doi.org/10.1109/HASE.2012.31

[34] X. Zhang and Z. Wang, "Notice of retraction: A static analysis tool for detecting web application injection vulnerabilities for asp program," in *2010 2nd International Conference on E-business and Information System Security*, 2010, pp. 1–5. [Online]. Available: https://doi.org/10.1109/EBISS.2010.5473561

[35] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "Candid: Dynamic candidate evaluations for automatic prevention of SQL injection attacks," *ACM Transactions on Information and System Security*, vol. 13, no. 2, Mar. 2010. [Online]. Available: https://doi.org/10.1145/1698750.1698754

[36] M. R. Ul Islam, M. S. Islam, Z. Ahmed, A. Iqbal, and R. Shahriyar, "Automatic detection of nosql injection using supervised learning," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, 2019, pp. 760–769. [Online]. Available: https://doi.org/10.1109/COMPSAC.2019.00113

[37] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 40–52. [Online]. Available: https://doi.org/10.1145/988672.988679

[38] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using parse tree validation to prevent SQL injection attacks," in *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, ser. SEM '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 106–113. [Online]. Available: https://doi.org/10.1145/1108473.1108496

[39] P. Bisht, A. P. Sistla, and V. N. Venkatakrishnan, "Taps: Automatically preparing safe SQL queries," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 645–647. [Online]. Available: https://doi.org/10.1145/1866307.1866384

[40] A. Abadi, Y. A. Feldman, and M. Shomrat, "Code-motion for api migration: Fixing SQL injection vulnerabilities in java," in *Proceedings of the 4th Workshop on Refactoring Tools*, ser. WRT '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1–7. [Online]. Available: https://doi.org/10.1145/1984732.1984734