Extracting Knowledge from Open Source Projects to Improve Program Security

Fitzroy Nembhard, Marco Carvalho, and Thomas Eskridge
School of Computing, Florida Institute of Technology
Melbourne, FL 32901
fnembhard2012@my.fit.edu, mcarvalho,teskridge@cs.fit.edu

***Abstract*—Open source repositories contain a wealth of unstructured and unlabeled data from which useful knowledge can be extracted. This knowledge can be applied in a wide range of applications such as discovering how programmers improve their programs over time and finding patterns to detect and mitigate vulnerabilities. In this work, we propose to use text mining and machine learning to extract knowledge from open source code in order to categorize and structure source code. By mining a subset (over 600,000 Java files) of a 2011 dataset that contains over 70,000 open source projects, we present a case study showing that useful patterns can be extracted from source code and that these patterns can be used to create a recommender system to help programmers avoid unsafe practices. We demonstrate the utility of our proposed techniques by applying them to the detection of SQL Injection.**

***Index Terms*—Cybersecurity, software engineering, code analysis, data mining**

I. Introduction

Source code repositories contain a wealth of information that can be mined and used in a variety of applications. Existing research has mainly been focused on the use of topic models, association rules, and heuristics to mine source code repositories for application to traceability, extraction of tactics[1], and monitoring changes in source code [2, 3, 4, 5]. It is interesting to investigate the mining of code repositories from the standpoint of knowledge extraction with the focus of helping programmers improve the security of their programs.

The security of computer programs is very important and necessary in order to safeguard against potential attacks. Research and funding to mitigate cyber-related threats have been on the rise for approximately two decades [6]. However, programmers often ignore security or wait until their projects are near completion to start thinking about security or adding secure modules to their programming projects. This is partly due to the belief that adding secure modules to code is a laborious and time-consuming process. Researchers and software engineers have proposed the use of static, dynamic, and hybrid code analysis techniques to detect vulnerabilities in program code [7, 8, 9]. However, many approaches are not mitigative and have high false positive and false negative rates that affect their adoption into the software engineering community [10, 11, 12, 13]. Because of this, most common responses to threats are reactive and take the form of patches and system modifications, which are much more costly [14].

This work proposes a solution to the secure coding problem that makes it easier for programmers to follow secure practices by offering recommendations to programmers during development. The premise is that source code projects can be mined to create a knowledge base in order to build a recommender system that provides advise to programmers, which, if followed, can help programmers write more secure code. If the security problem is tackled throughout the system development life-cycle, many security issues will be mitigated at a much lower cost [14].

To that end, this work features a model that analyzes source code and uses data from the National Vulnerability Database (NVD) to create feature sets for recognizing a set of vulnerabilities. Specifically, source code is represented using an Abstract Syntax Tree (AST), which provides the ability to extract statements, imported modules, and methods, etc, to provide the necessary information for detecting vulnerabilities. Unlike static analysis tools that find vulnerabilities and present a set of warnings, this work also proposes the extraction of code samples from open source repositories to present to the user a ranked set of methods that implement fixes for each vulnerability. To validate the model, a portion of the *Sourcerer 2011 dataset* [15] was processed to create a labeled corpus of 7,793 instances for detecting SQL Injection (SQLI). Using the corpus, a set of classifiers were trained and evaluated and the results are presented. The main contributions of this work are as follows:

- The development of a methodology that utilizes data from the National Vulnerability Database and open source projects to detect vulnerabilities in source code
- The utilization of patterns in source code to automatically label a dataset for SQLI detection
- Development and use of a MapReduce algorithm to process thousands of Java files and extract features for detecting and mitigating vulnerabilities

The paper is organized as follows: related work is discussed in Section II followed by the proposed approach in Section III. In Section IV, the model building phase, which focuses on the dataset, data representation, and the MapReduce algorithm for

[1] a means of satisfying a quality-attribute-response measure by manipulating some aspect of a quality attribute model [1]

978-1-5386-6133-8/18/$31.00

mining and extracting features from source code, is discussed. The methodology is evaluated using a case study and the results and discussion are presented in Section V followed by the conclusion in Section VI.

II. Related Work

Several works exist in the area of source code mining. The most relevant works can be classified as follows: extracting and monitoring architectural tactics [4], extracting and evaluating topic models [16, 17], improving bug finding techniques [18], predicting source code changes [5], and the use of an intermediate language to automatically detect web-based vulnerabilities [19].

Mirakhorli and Cleland-Huang presented an approach that addresses the problem of architectural degradation that is focused on keeping developers informed of underlying architectural decisions [4]. They mined and pre-classified code segments to create inputs that represent architectural tactics that help to map parts of code to relevant patterns. These patterns are used to monitor code and inform developers of architectural changes [4].

Ying et al. presented an approach that uses data mining techniques to detect change patterns from a system's source code change history [5]. First, they extracted data from a software configuration management system and performed preprocessing. Next, they used an association rule mining algorithm to form change patterns, which are used to assist developers in deciding which source files to change when modifying a programming project.

Thomas [17] was one of the first to propose the use of Latent Dirichlet Allocation (LDA) topic models to combat many problems in the field of software engineering such as understanding and maintaining the source code of a software project. He discussed challenges such as choosing the right parameters for topic modeling and making results of topic models easier to interpret [17].

Gopalakrishnan et al.[16] presented a bottom-up approach that recommends architectural tactics based on topics discovered from source code projects. They used a classifier in addition to a recommender system to predict where tactics should be placed in a programming project to improve the quality, but not security, of the code.

In [19], Medeiros et al. presented the DEKANT tool that automatically detects web-based vulnerabilities using hidden markov models (HMM). First, the tool extracts code slices from source code and translates these slices into an intermediate slice language (ISL). It then analyses the representation to determine the presence of vulnerabilities in code written in PHP.

None of the existing works utilizes a recommender system to help programmers improve the security of their programs. The uniqueness of this research lies in the use of observed vulnerability patterns and a set of classifiers that are trained from knowledge extracted from raw program code to detect the presence or absence of secure patterns and make security recommendations based on the knowledge base. While Medeiros et al. used a machine learning technique to detect vulnerabilities in PHP projects, the tool developers are required to use an intermediate language to annotate each tainted function [19]. This work, on the other hand, does not need an intermediate language. Instead, the proposed model works directly with the parse tree of the source code to detect patterns for automatic annotation, detection and classification.

III. The Approach

The solution follows two phases: model building and application. Figure 1 captures the solution framework. As shown in the figure, a model is first created to analyze data from the NVD and open source programs in order to create a feature set for training a set of classifiers to detect a select set of Common Vulnerabilities and Exposures (CVE) in user programs. A recommender system then uses this information to present to the user a ranked set of methods that fixes these vulnerabilities. A brief explanation is now provided of each phase in the proposed approach:

1) *Mining and labeling open source projects*
   Because source code is unstructured, it is necessary to organize and label the program files in the repository in order to train and evaluate supervised machine learning algorithms. Once source code is labeled, it can be used to train a set of classifiers that categorizes program code as secure or insecure.
2) *Building the model*
   An AST is used to represent source code. This representation allows for the use of a visitor pattern to extract statements and methods from source code into a model that detects vulnerability patterns. Section IV-B provides more details on data representation.
3) *Designing and evaluating the recommender system*
   The generated model will be used to create a recommender system to provide secure recommendations to programmers during development. A set of classifiers will be used to evaluate the knowledge base of the recommender system using standard metrics within the machine learning community such as precision, recall, and F1 score.

IV. Model Building

In this section, data representation and steps followed to create a corpus for classification and evaluation are discussed.

A. The Dataset

The dataset used for this study is Sourcerer 2011, which contains over 70,000 open source projects written in the Java programming language (≈368GB of compressed data). The dataset is divided into four TAR archives, identified as *aa* to *ad*. Each of these archives contains varying numbers of projects, which are numbered in a sequential manner. Each project is then organized into a cache of important files, the source code, which follows the organization system used by the developers, and a *project.properties* file, which contains information such as the repo URL and author. This work

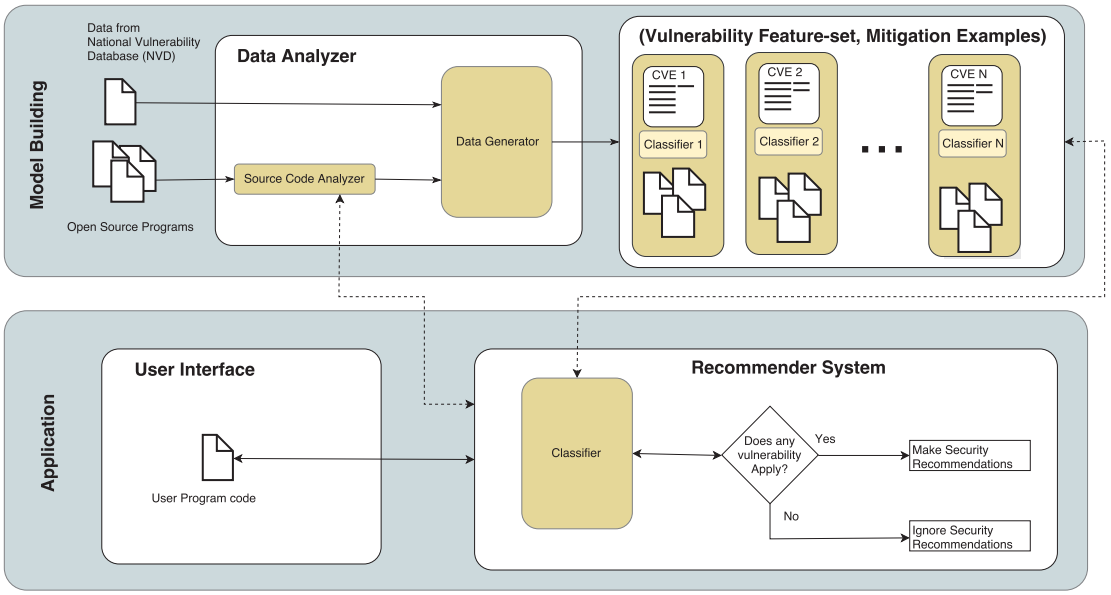

Fig. 1: Solution Framework

focused on approximately 6,800 open source projects from part *aa* of the dataset (605,809 Java files).

B. Data Representation

In this work, each Java file is modeled as an AST, which is a hierarchical intermediate representation of a program that presents source code structure according to the grammar of a given programming language [20]. It is a reduced parse tree where nodes are connected through parent-child relationships. The construction of an AST begins with a node that represents the entire translation/compilation unit followed by a number of intermediate levels, then simple language constructs such as type name, identifier name, or operator as the leaf nodes [20].

The JavaParser[2] library was utilized to construct and traverse an AST from Java source code. JavaParser is an open source library that allows native Java interaction with an AST generated from Java source code [21]. JavaParser was chosen due to its ease of use when compared with other parsers [22].

C. Using MapReduce to Extract Features for Categorizing the Dataset

Apache Hadoop[3] was used as a MapReduce framework to process the dataset. Experiments were conducted on a small cluster that was created using the Virtual Infrastructure for Network Emulation (VINE) [23]. The cluster comprises a Hadoop namenode (server of 4TB space, 16GB RAM, and 8 CPUs) and 10 Hadoop datanodes (virtual machines, each of 100GB disk space, 8GB RAM, 4 CPUs).

MapReduce is a programming model and an associated implementation for processing and generating large datasets [24]. The concept behind MapReduce is the specification of computations in terms of a map and a reduce function that can be executed in parallel across large-scale clusters of machines. The map and reduce functions interact with key value pairs as follows:

$$map(k1, v1) \rightarrow list(k2, v2)$$
$$reduce : (k2, list(v2)) \rightarrow list(k3, v3)$$

The main goal of the MapReduce paradigm is to distribute tasks across multiple nodes in order to decrease the effect of machine failures and schedule inter-machine communication to make efficient use of the network and disks [24]. The Apache Hadoop software library is one of the most popular implementations of the MapReduce methodology. It is a framework that allows for the distributed processing of large data sets across clusters of computers using a simple programming model [25].

The proposed MapReduce algorithm is shown in Algorithm 1. Considering that the projects in Sourcerer 2011 have inconsistent organization structure, each project must be processed to ensure that files follow a uniform structure that can be submitted to the Hadoop Distributed File System (HDFS) for processing. First, a *Bash* script was used to parse each *project.properties* file within each project in the repository and information was extracted to create a more uniform file structure. Java files were reorganized such that there is one directory of files for each project. The filenames were later used as keys for the MapReduce framework. To create a unique filename, each repository URL and a universally unique identifier (UUID) were prepended to the original file name of the Java files within each project directory. For example, given a repo project identified by the url: `google-api-translate-java.googlecode.com`

[2]https://github.com/javaparser/javaparser
[3]http://hadoop.apache.org/

and a Java file with the name `DBHelper.java`, the fileID (key) would resemble the following: `google-api-translate-java.googlecode.com~74b000bc-06ba-11e8-83ee-000c29bc3179~DBHelper.java`. An added benefit of this convention is to provide support for validating automatically-labeled data and querying the dataset during the application phase of the project.

Hadoop utilizes blocks and input splits for data processing. Blocks refer to physical storage locations and the default block size is 64MB [26]. The default split size is equal to a block size. Users can write custom input split functions to specify how files should be handled in HDFS. Because each Java file is modeled as an AST, it is imperative that HDFS process each file without splitting it. To ensure this, a custom record reader was written to read each Java file and submit it to the mappers for processing as shown in the algorithm. Datanodes then run the map and reduce functions, respectively, to process the data and output the results.

Algorithm 1: MapReduce Algorithm for Mining Features from Java Code

input : repository_path: path to repository dataset
output: a set of features for a certain vulnerability

```
foreach project ∈ repository_path do
    javaDataFiles = selectJavaFiles()
    createCustomRecordReader() // record reader to read full java program file

    foreach javaFile ∈ javaDataFiles do
        Function map(javaFile):
            key = getFileName(javaFile)
            value = extractText(javaFile) // using customRecordReader
            addToIntermediateList(key, value)
            emit(intermediateList)
                        /* each reduce is a vulnerability detector that emits a set of features for identifying a certain vulnerability */
        Function reduce(intermediateList):
            foreach pair ∈ intermediateList do
                outkey = intermediateList.key
                inValue = intermediateList.value
                outValue = buildFeatureSet(inValue)
                    // based on abstract syntax tree
                emitFinal(outKey, outValue)
```

V. Case Study: CWE-89 (SQL Injection)

This section provides a discussion on the application of the proposed model in detecting the SQL Injection vulnerability.

A. Understanding SQLI

An SQLI attack occurs when an attacker provides specially crafted input to an application that employs database services such that the provided input results in a different database request than was intended by the application programmer [27]. SQLI has been a common vulnerability for many years, securing the top position on the CWE 2011 [28], OWASP 2010 [29], and OWASP 2017 [30] vulnerability lists. Susceptible applications (e.g. web-apps) generally accept user input without validation or neutralization, which are then used as parameters in SQL database requests. SQLI is a serious vulnerability because it could lead to unauthorized access to sensitive data, cause severe updates to or deletions from a database, and even result in devastating shell command execution [31]. Listing 1 is a sample code that could potentially lead to SQL Injection. This is due to the fact that the `doLogin` function accepts the parameters *username* and *password* in plaintext and includes them in the SQL query string without neutralization. The use of the `PreparedStatement` class from JDBC or J2EE is recommended by the curators of NVD as a fix for SQL Injection [28]. This class allows for the use of a wildcard ("?" character) to create a parametric query that escapes potentially tainted user input. Listing 2 shows a compliant query that uses the `PreparedStatement` class to mitigate SQL Injection.

Listing 1 Example Java Code that could potentially lead to SQL Injection

```
1  import java.sql.*;
2
3  class Login {
4        public boolean doLogin(String username,
             ↪ String pwd) {
5
6         String sqlString = "SELECT * FROM db_user
              ↪  WHERE username = '" +
7         username + "' AND password = '" +
8         pwd + "'";
9        }
10 }
```

Listing 2 Example Java Code that mitigates SQL Injection

```
1  import java.sql.PreparedStatement;
2
3  class Login {
4       public boolean doLogin(String username,
            ↪ String pwd) {
5
6         String sqlString = "select * from db_user
              ↪  where username=? and password=?";
7         PreparedStatement pStmt = connection.
              ↪ prepareStatement(sqlString);
8         pStmt.setString(1, username);
9         pStmt.setString(2, pwd);
10        ResultSet rs = pStmt.executeQuery();
11       }
12 }
```

B. Extracting features for Detecting SQLI

If Listing 1 is compared with Listing 2, a number of differences can be noted between the two samples of code that can be used to create a feature set for recognizing the SQL injection vulnerability. First, the unsafe query string in Listing 1 contains explicit apostrophes, and no methods are used to sanitize the user input. Consequently, six main features for detecting and classifying the vulnerability were identified. These features are presented in Table I. Two of the features are multivalued attributes (*source* and *sink*) while the others are boolean attributes. A source, in the context of SQLI (and other taint-style vulnerabilities), refers to an untrusted data source from which user input is received [32]. A sink is a security-sensitive function (e.g. the `java.sql.Statement.executeQuery` function) [32]. A *source-sink* error arises if a value constructed at a location designated as a *source* reaches a location designated as a sink [33]. Data was selected from online resources [34, 35, 36] to create a set of Java source and sink functions.

Since the main cause of SQLI is unsanitized input and un-escaped apostrophes, the features *quoted_variables_found* and *potentially_sanitized* are crucial in detecting the vulnerability. However, simply checking the presence of an apostrophe is not acceptable because a query string such as `iper.update("insert into people values('namtesss','sdfjlsfjls')", conn)`, which was found in a unit test of the *goldriver.googlecode.com* open source project in the Sourcerer 2011 dataset, is a safe SQL statement. It is the quoted portion of a concatenated query string that should not contain single quotes as discussed before and depicted in Listing 1. In addition, *prepared_statement_imported* and *all_queries_parameterized* can help to measure the utility of the `PreparedStatement` class in mitigating SQLI across the projects in the Sourcerer 2011 dataset. Moreover, the *METADATA* attribute contains statements and methods to help validate the automatic labeling of data and also provide ranked samples to the user to mitigate the vulnerability.

C. Results and Discussion

Of the 605,809 Java files submitted to the Hadoop Distributed File System for processing, 7,793 files were flagged by our Data Generator as containing SQL statements. These files were used to create the ground truth of 6,164 safe and 1,629 unsafe instances. A random sample of 100 instances from the dataset was manually reviewed to ensure that the algorithm was correctly labeling the dataset for SQL injection. Using the sinks attribute, it was determined that 56% of the flagged programs did not utilize the `PreparedStatement` class to create parametric queries, which is recommended for mitigating SQL injection.

A sample of 80% of the dataset was selected for training and 20% for testing. Three classifiers (Decision Tree, Random Forest, and SVM) from the WEKA API [37] were employed to evaluate the proposed methodology. For the testing phase, ten-fold cross-validation was performed. All three classifiers achieved the same results. The confusion matrix is shown in Table II and performance results in Table IV. The following metrics are reported, and their respective formulas shown in Table III: true positive rate (TPR), false positive rate (FPR), precision (PREC), recall (REC), f-measure, Matthews correlation coefficient (MCC), receiver operating characteristic (ROC) area, and precision recall (PRC) area. Precision, also known as positive predictive value (PPV), measures the exactness of the classifiers. Recall refers to the completeness of a classifier. The F1-score (F-measure) is the harmonic mean that considers both precision and recall to give a better explanation of a classifier's accuracy than basic accuracy. Additionally, the ROC curves for the testing phase of the classifiers are shown in Figure 2. As can be seen from the results, all three classifiers performed significantly well, with average precision and recall of 100%. Further, the false positive rates average 0%. These results support the premise that open source projects can be mined to provide knowledge for training classifiers to detect vulnerabilities and for classifying source code as secure or insecure.

The reasons hand-coded features were selected for use in this study are because of simplicity of programming, the ability to ensure data correctness, and to create the end-to-end processing framework. This is an essential step in creating datasets that are verifiably correct. This provides a baseline on which to judge the performance of the methodology. Future work will investigate automatic feature creation such as with deep learning or other approaches.

VI. Conclusion

In this work, a methodology is presented for extracting knowledge from open source projects to create a model that identifies unsafe practices in user projects and provides recommendations to improve program security. While source code is unstructured and unlabeled, useful knowledge can be extracted to detect and mitigate vulnerabilities. The utility of the proposed methodology is demonstrated through a case study that involved mining a dataset of approximately 6,800 open source projects (605,809 Java files) and the preparation of a labeled corpus of over 7,000 records that were used to detect SQL Injection. The labeled corpus was used to train and evaluate three classifiers, which achieved an average recall of 100% and 0% false positive rate. These results support the premise that open source projects can be mined to provide knowledge for training classifiers to detect vulnerabilities and for classifying source code as secure or insecure.

TABLE I: Features for Recognizing SQL Injection

Feature	Data Type	Possible Values	Description
source	multi-valued	{getPathInfo, getResource, getName, getServletPath, getRemoteHost, getLocalAddr, getParameterMap, getRealPath, getServerName, getPathTranslated, getInitParameterNames, getHeader, getCookies, getPath, getComment, getParameter, getParameterValues, getRequestURL, getHeaders, getRequestURI, getResourceAsStream, getRequestDispatcher, getQueryString, getResourcePaths, getDomain, getValue, getLocalName, getInitParameter, getRemoteUser, getHeaderNames, getContentType, getParameterNames, concatenateWhere, getNamedDispatcher}	The method that accepts or processes potentially tainted user input
sink	multi-valued	{executeLargeUpdate, updateWithOnConflict, setGrouping, queryForList, batchUpdate, update, buildQuery, prepareStatement, delete, buildUnionSubQuery, queryWithFactory, rawQueryWithFactory, nativeSQL, queryForInt, blobFileDescriptorForQuery, longForQuery, sqlRestriction, newQuery, executeInsert, createQuery, queryForMap, queryForLong, apply, execSQL, queryForRowSet, query, stringForQuery, buildQueryString, <init>, addBatch, execute, executeQuery, createSQLQuery, createNativeQuery, setFilter, appendWhere, queryForObject, newPreparedStatementCreator, as, compileStatement, createDbFromSqlStatements, buildUnionQuery, rawQuery, executeUpdate, prepareCall}	The method that creates, modifies, or executes an SQL query
quoted_variables_found	boolean	{true, false}	Tells whether explicit apostrophes were used to formulate an SQL query string
potentially_sanitized	boolean	{true, false}	Tells whether user inputs were passed to untainted functions before being used in SQL strings
prepared_statement_imported	boolean	{true, false}	Specifies whether the recommended PreparedStatement class was imported
all_queries_parameterized	boolean	{true, false}	Specifies whether the question-mark wildcard was used as variable placeholders in query strings
METADATA	String	–	Data (encoded in base 64) containing SQL statements and methods found in each Java file to assist with verification of classification
Class	binary	{safe, unsafe}	The target variable

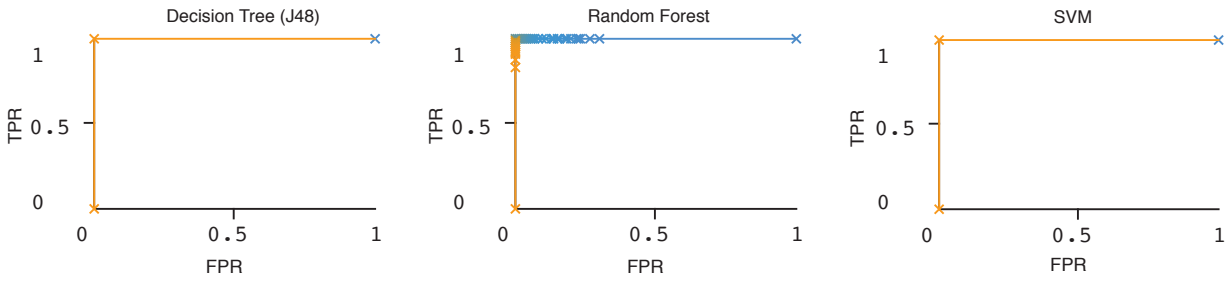

Fig. 2: ROC Curves for the Testing Phase

TABLE II: Confusion Matrix for Decision Tree, Random Forest, and SVM

	Training Phase		Testing Phase	
	Safe	Unsafe	Safe	Unsafe
Safe	4192	0	1252	0
Unsafe	0	1322	0	307

TABLE III: Classifier Performance Metrics

Measure	Formula
FPR	FP / (FP + TN)
SN, TPR, REC	TP / (TP + FN)
PREC, PPV	TP / (TP + FP)
MCC	(TP * TN - FP * FN) / ((TP + FP)(TP + FN)(TN + FP)(TN + FN))$^{1/2}$
F_1	2 * PREC * REC / (PREC + REC)

TABLE IV: Performance for Decision Tree, Random Forest, and SVM

	TPR	FPR	PREC	REC	F-1	MCC	ROC	PRC
safe	1.000	0.000	1.000	1.000	1.000	1.000	1.000	1.000
unsafe	1.000	0.000	1.000	1.000	1.000	1.000	1.000	1.000
Weighted Avg.	1.000	0.000	1.000	1.000	1.000	1.000	1.000	1.000

REFERENCES

[1] F. Bachmann, L. Bass, and M. Klein, "Deriving architectural tactics: A step toward methodical architectural design," Carnegie-Mellon Univ., Pittsburgh, PA, Software Engineering Inst., Tech. Rep., 2003.

[2] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova, "Best practices for automated traceability," *Computer*, vol. 40, no. 6, pp. 27–35, Jun. 2007.

[3] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, Oct 2002.

[4] M. Mirakhorli and J. Cleland-Huang, "Detecting, tracing, and monitor-

ing architectural tactics in code," *IEEE Trans. Software Eng.*, vol. 42, pp. 205–220, 2016.

[5] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, Sept 2004.

[6] J. Mirkovic, T. V. Benzel, T. Faber, R. Braden, J. T. Wroclawski, and S. Schwab, "The DETER project: Advancing the science of cyber security experimentation and test," in *2010 IEEE International Conference on Technologies for Homeland Security (HST)*, Nov 2010, pp. 1–7.

[7] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, 2003, pp. 24–27.

[8] A. G. Bardas, "Static code analysis," *Journal of Information Systems & Operations Management*, vol. 4, no. 2, pp. 99–107, 2010.

[9] A. Aggarwal and P. Jalote, "Integrating static and dynamic analysis for detecting vulnerabilities," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, vol. 1, Sept 2006, pp. 343–350.

[10] J. Bleier, E. Poll, H. Xu, and J. Visser, "Improving the usefulness of alerts generated by automated static analysis tools," Master's thesis, Radboud University Nijmegen, 2017.

[11] OWASP, "Source code analysis tools," https://www.owasp.org/index.php/Source_Code_Analysis_Tools, 2017, accessed: 2017-07-07.

[12] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation exploitation in error ranking," in *ACM SIGSOFT Software Engineering Notes*, ser. SIGSOFT '04/FSE-12. New York, NY, USA: ACM, 2004, pp. 83–93.

[13] S. Kim and M. D. Ernst, "Which warnings should I fix first?" in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 45–54.

[14] A. Castro Lechtaler, J. C. Liporace, M. Cipriano, E. García, A. Maiorano, E. Malvacio, and N. Tapia, "Automated analysis of source code patches using machine learning algorithms," in *XXI Congreso Argentino de Ciencias de la Computación (Junín, 2015)*, 2015.

[15] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An infrastructure for large-scale collection and analysis of open-source code," *Science of Computer Programming*, vol. 79, pp. 241 – 259, 2014.

[16] R. Gopalakrishnan, P. Sharma, M. Mirakhorli, and M. Galster, "Can latent topics in source code predict missing architectural tactics?" in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 15–26.

[17] S. W. Thomas, "Mining software repositories using topic models," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 1138–1139.

[18] C. C. Williams and J. K. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 466–480, June 2005.

[19] I. Medeiros, N. Neves, and M. Correia, "DEKANT: a static analysis tool that learns to detect web application vulnerabilities," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 1–11.

[20] R. Software, "Abstract syntax tree (AST)," https://support.roguewave.com/documentation/klocwork/en/10-x/abstractsyntaxtreeast/, 2018, accessed: 2017-01-25.

[21] N. Smith, D. van Bruggen, and F. Tomassetti, *JavaParser: Visited; Analyse, transform and generate your Java code base*. British Columbia, Canada: Leanpub, 2017.

[22] M. Badalíková, "System for analysis of java language," Master's thesis, Czech Technical University in Prague, 2016.

[23] T. C. Eskridge, M. M. Carvalho, E. Stoner, T. Toggweiler, and A. Granados, "VINE: A cyber emulation environment for MTD experimentation," in *Proceedings of the Second ACM Workshop on Moving Target Defense*, ser. MTD '15. New York, NY, USA: ACM, 2015, pp. 43–47.

[24] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[25] A. B. Patel, M. Birla, and U. Nair, "Addressing big data problem using hadoop and map reduce," in *2012 Nirma University International Conference on Engineering (NUiCONE)*, Dec 2012, pp. 1–5.

[26] D. DeRoos, P. Zikopoulos, B. Brown, R. Coss, and R. B. Melnyk, *Hadoop for dummies*. John Wiley & Sons, Incorporated, 2014.

[27] 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: ACM, 2005, pp. 106–113.

[28] MITRE, "2011 cwe/sans top 25 most dangerous software errors," http://cwe.mitre.org/top25/, Sep 2011, accessed: 2017-12-6.

[29] J. Williams and D. Wichers, "Owasp top 10-2010 the ten most critical web application security risks," The OWASP Community, Tech. Rep., 2010.

[30] ——, "The ten most critical web application security risks," *rc1,//OWASP Foundation*, 2017.

[31] B. Livshits, "Improving software security with precise static and runtime analysis," Ph.D. dissertation, Stanford University, Stanford, California, 2006.

[32] J. Clarke-Salt, *SQL injection attacks and defense*. Elsevier, 2009.

[33] H. Zhu, T. Dillig, and I. Dillig, "Automated inference of library specifications for source-sink property verification," in *Asian Symposium on Programming Languages and Systems*. Springer, 2013, pp. 290–306.

[34] B. Flood, "Find-sec-bugs injection sinks," https://github.com/find-sec-bugs/find-sec-bugs/tree/master/plugin/src/main/resources/injection-sinks, Nov 2017, accessed: 2018-2-8.

[35] L. Sampaio, "Which methods should be considered "sources", "sinks" or "sanitization"?" http://thecodemaster.net/methods-considered-sources-sinks-sanitization/, 2014, accessed: 2018-2-8.

[36] OWASP, "Searching for code in j2ee/java," https://www.owasp.org/index.php/Searching_for_Code_in_J2EE/Java, Apr 2016, accessed: 2018-2-8.

[37] F. Eibe, M. Hall, I. Witten, and J. Pal, "The weka workbench," *Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*, vol. 4, 2016.