EURASIP Journal on Information Security

RESEARCH Open Access

Towards the application of recommender systems to secure coding

Fitzroy D. Nembhard*, Marco M. Carvalho and Thomas C. Eskridge

Abstract

Secure coding is crucial for the design of secure and efficient software and computing systems. However, many programmers avoid secure coding practices for a variety of reasons. Some of these reasons are lack of knowledge of secure coding standards, negligence, and poor performance of and usability issues with existing code analysis tools. Therefore, it is essential to create tools that address these issues and concerns. This article features the proposal, development, and evaluation of a recommender system that uses text mining techniques, coupled with IntelliSense technology, to recommend fixes for potential vulnerabilities in program code. The resulting system mines a large code base of over 1.6 million Java files using the MapReduce methodology, creating a knowledge base for a recommender system that provides fixes for taint-style vulnerabilities. Formative testing and a usability study determined that surveyed participants strongly believed that a recommender system would help programmers write more secure code.

Keywords: Secure coding, Vulnerability detection, Code analysis, Data mining, Secure systems, Intellisense, Big data, Knowledge extraction, Software engineering, Cybersecurity

1 Introduction

Data breaches continue to plague organizations across the globe. The *2017 Cost of Data Breach Study* conducted by the Ponemon Institute shows that the average total cost of a data breach is US$3.62 million [1]. One of the main causes of data breaches is code-level vulnerabilities [2, 3]. A 2017 report by Tricentis shows that for 11 months in 2016, news articles reported at least 3 software failures per month that were caused by code-level vulnerabilities [4]. These statistics emphasize the need for improved security analytics techniques. Compounding the problem is the fact that many developers are skeptical of using existing code analyzers because of high false-positive rates, the time required to investigate inactionable alerts, and usability issues [5, 6]. Further, a significant number of existing code analysis tools are designed to find bugs or vulnerabilities in program code, but many of these tools do not offer mitigation support to help programmers write secure code. If data breaches and other security-related issues are to be resolved, it is imperative that developers have useful and effective tools at their disposal to help them write secure code.

To address the secure coding problem, this research presents a recommender system that detects the presence of insecure program code and offers live recommendations that include fixes for vulnerabilities based on common practices in the security field, to make it easier for programmers to write more secure code. Recommender systems are software tools and techniques that provide suggestions for items that are most likely of interest to a particular user [7]. Traditionally, recommender systems have been applied to commodities such as books, CDs, etc. Ricci et al. [7] noted that the attributes of the items recommended by classic content-based recommendation techniques are keywords extracted from the descriptions of the items [7].

The methodology presented in this work uses source code mining to extract hand-selected features that are used to detect vulnerabilities in program code and to select code examples that mitigate certain vulnerabilities. First, a repository of more than 14,000 open-source projects is mined, and features are extracted based on vulnerability descriptions provided in the National Vulnerability Database (NVD). Next, using the extracted features, datasets containing safe, and unsafe examples are prepared and used as knowledge for a recommender system, which currently detects and assists

*Correspondence: fitzroy@ieee.org
College of Engineering and Sciences, Florida Institute of Technology, 150 W. University Blvd., Melbourne, FL 32901, USA

with mitigating taint-style vulnerabilities. The recommender system was designed by taking into account input from participants in a knowledge elicitation survey. The classic recommendation approach is used to present code examples to the programmer that are most similar to the code being developed instead of using generic examples, which is the traditional practice.

The research question is that a recommender system built using text-mining techniques can assist programmers with detection and mitigation of vulnerabilities as they type code during development. Targeting and correcting unsafe practices as programmers type code will help to catch bugs earlier than using traditional static and dynamic approaches. This work makes the following major contributions:

- The design, implementation, and evaluation of a recommender system that uses text mining techniques, coupled with IntelliSense technology, to recommend fixes for potential vulnerabilities in program code. The implemented system uses code running on Apache Hadoop to extract knowledge from a large body of open-source projects to provide features for detecting taint-style vulnerabilities
- The use of a knowledge elicitation survey to determine the current use of code analyzers among programmers and to elicit their views on the design of the proposed system
- A bipartite evaluation (scalability and usability) of the proposed system along with a discussion on the statistical significance of the usability results.

The article is organized as follows: related work is presented in Section 2 followed by an overview of the approach in Section 3. Section 4 provides a thorough discussion of modeling and detection. Section 5 discusses the methods followed to design and implement the proposed system. This section also presents a discussion on a knowledge elicitation survey that was conducted to obtain information that affect the design of the system as well as a usability study that ascertains the usability and usefulness of recommender systems in helping programmers write more secure code. Results and discussion of the user study and a scalability evaluation are presented in Section 6 followed by conclusions and future work in Section 7.

2 Related work

In this section, a discussion is provided on works that are closely related to this work in the area of automated coding support, particularly in static analysis, and Dynamic Application Security Testing (DAST) or dynamic analysis and auto-fixing of programming errors.

2.1 Static analyzers

2.1.1 Lightweight analyzers

Splint is a heuristics-based tool that finds potential vulnerabilities by checking to see that source code is consistent with the properties implied by annotations [8]. Splint is limited to American National Standards Institute (ANSI) C code and does not offer the functionalities required in agile and data-driven development environments.

FindBugs is Java-based static analysis tool that is intended to find coding defects that developers will want to review and remedy [9]. The concept is based on bug patterns that can be found based on Java byte code [9]. FindSecBugs is a FindBugs plugin, which is geared towards security audits of Java web applications [10].

Alenezi and Javed proposed the Developer Companion framework to help developers produce secure web applications [11]. Developer Companion uses several static analysis tools to analyze program code, cross-references the results against the Common Weakness Enumeration (CWE) and NVD, and presents to developers a recommendation based on the aggregated data [11].

2.1.2 Tools that improve static analysis warnings/alerts

Some researchers have proposed tools and frameworks to prioritize alerts or vulnerabilities to make it easier for developers and managers to address the more critical issues [12, 13].

The tool proposed in [12] is known as Autobugs, which gathers historic alert data from static analysis tools and combines the alert-data with complexity metrics to build a classifier that predicts the actionability of an alert from data and unit properties [12]. Unfortunately, the author reported that models based on historic alert data could potentially mislead developers to believe they have no security issues [12].

In [13], a vulnerability management strategy, known as VULCON, is proposed to prioritize vulnerabilities for patching. Using two metrics (total vulnerability exposure and time-to-vulnerability remediation), the framework ingests vulnerability scan reports from code analyzers such as Nessus [14] and outputs security exposure metrics and vulnerability management plans to managers, operators, analysts and engineers, so they can decide on which vulnerabilities to mediate [13].

2.1.3 Static analyzers based on source code mining

Gopalakrishnan et al. [15] presented a bottom-up approach that recommends architectural tactics (a quality-attribute-response) 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 [16], 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 analyzes the representation to determine the presence of vulnerabilities in code written in PHP.

2.1.4 *Vulnerable code pattern recognition using machine learning*

In a survey of software vulnerability analysis and discovery using machine learning and data mining techniques, Ghaffarian and Shahriari categorized approaches into four main areas [17]. Of these four areas, the area most closely related to this work is "Vulnerable Code Pattern Recognition." Under this category, the work by Yamaguchi et al. [18] is related. In [18], the authors proposed a method that assists a security analyst with auditing source code. Abstract Syntax Trees (ASTs) are extracted form source code (C-code) and then embedded in a vector space, such that techniques from machine learning can be applied to analyze the code. Further, latent semantic analysis is used to determine dominant directions (structural patterns) in the vector space from which code similar to a known vulnerability is identified and used to detect vulnerabilities.

In addition, Shar and Tan [19] produced a series of papers [19–22] on vulnerability detection and mitigation, each improving upon their previous work. The most related paper in their work is [19]. In [19], 20 static code attributes based on data-flow analysis of PHP web applications are proposed for predicting program statements that are vulnerable to SQL-injection (SQLI) and cross-site scripting. The authors extracted control-flow (CFG) and data-flow graphs (DFG) of a given PHP program and performed backward data-flow analysis on target sink statements that may reach certain input source statements [17]. The extracted attributes are used to create vectors, which are coupled with their known vulnerability status to train classifiers to predict the vulnerability status of unseen sink statements [19]. A source refers to an untrusted data source from which user input is received and a sink is a security-sensitive function [23].

2.2 Dynamic analyzers

A plethora of tools [24, 25] exist in the dynamic analysis domain, the majority of which are commercial. Interestingly, a great deal of focus in DAST is devoted to web applications [26–28]. Huang et al. proposed a crawler that allows for a black-box, dynamic analysis of web applications [26]. Using reverse engineering (to identify all possible points of attack within a web application) and a fault injection process, the tool attempts to determine the most vulnerable points within an application [26]. In addition, Petukhov and Kozlov proposed an extended tainted[1] mode model that incorporates the advantages of penetration testing and dynamic analysis to widen the scope of the web application being covered during testing [28].

Since dynamic analysis involves testing application behavior, some researchers believe it is a more realistic approach than static analysis [29]. However, the main challenge with dynamic tools is identifying the source of a bug [6]. Bugs often manifest themselves as program crashes and this makes them difficult to mitigate.

2.2.1 *Dynamic analyzers based on AI/machine learning*

In [30], the authors described a tool, known as HACKAR, that uses an improved version of Java PathFinder (JPF) to execute Java programs and identify vulnerabilities. The tool is a dynamic analyzer that formulates a problem using Satisfiability Modulo Theory (SMT) and uses symbolic execution to determine program paths that may lead to vulnerabilities. In addition, HACKAR uses a goal regression[2] technique proposed by [31] to learn the semantics of tasks based on program traces in order to produce a knowledge base for providing advice to programmers on how to fix vulnerabilities.

2.3 Automated code repair and auto-completion

Several works exist in the area of automated code repair and code completion. In 2009, the first tools, ClearView [32] and GenProg [33], that perform automated code repair on real-world programs were demonstrated [34]. Since that time, focus on automated code repair has grown steadily with several other tools being developed, each either proposing an improvement on an existing methodology or a unique algorithm (e.g., SPR [35], Kali [36], AE [34], and Prophet [37]). In 34, Weimer et al. categorized existing code repair tools into two categories: those that use stochastic search or produce multiple candidate repairs, which are validated using test cases (e.g., GenProg, PAR [38], AutoFix-E [39], ClearView, Debroy, and Wong [40]), and techniques that use synthesis (e.g., SemFix [41]) or constraint solving (and symbolic execution) to produce a single patch that is correct by construction (e.g., AFix [42], FUZZBUSTER [43], FUZZBALL, and FUZZBOMB [44]). Since many of these tools require test cases to operate, they fit well in the area of dynamic analysis.

Raychev et al. proposed an approach that learns a probabilistic model from existing annotated program data and uses this model to predict properties of new, unseen programs [45]. The authors also created a scalable prediction engine called JSNICE that predicts names of identifiers and type annotations of variables. That is, given an optimized minified JavaScript code, JSNICE generates

JavaScript code that is annotated with types and identifier names.

In 46, Gupta et al. proposed the DeepFix algorithm that uses a multi-layered sequence-to-sequence neural network to fix common programming errors (e.g., missing declarations or statements, missing identifiers, and undeclared variables) in C code [46]. The neural network comprises an encoder recurrent neural network (RNN) to process the input and a decoder RNN with attention that generates fixes using an iterative process [46].

There are also linters[3] (e.g., SonarLint [47]), code quality analyzers (e.g., ASIDE [48] and code-clone detection tools (XIAO [49]) that attempt to improve the quality of code within integrated development environments (IDEs). XIAO is a tool that helps to deal with the issue of code-cloning where programmers may have repetitious code within their coding project. The premise is that detecting code clones can be useful in finding similar security bugs and also improves the quality of code through refactoring of code clones [49]. Baset and Denning showed that SonarLint and many existing IDE-based tools (e.g., ESVD [50]) present short description of common programming errors, but do not provide example fixes for security-related vulnerabilities [51].

In [52], Raychev et al. presented an approach to code completion based on a novel combination of program analysis with statistical language models. Given a codebase, their system first extracts abstract histories in the form of sentences from the data. Then, these sentences are fed to a language model such as an n-gram model or recurrent neural network model that learns probabilities for each sentence.

Also, in [53], the authors described an architecture that allows library developers to introduce interactive and highly specialized code generation interfaces, called palettes, directly into the editor. Both of these code completion approaches are based on system design and sentence suggestion and have not been applied to vulnerability detection and mitigation.

2.4 Difference between the proposed approach and existing approaches

The methodology proposed and implemented in this research couples text mining algorithms and IntelliSense techniques to analyze program code as the programmer types, compares the user's code with a knowledge base of unsafe practices to determine the presence of unsafe code and recommends fixes by providing ranked example code to the programmer during development. IntelliSense, also known as code-completion or code-hinting, refers to productivity features that help programmers learn about their code by keeping track of parameters and providing the ability to add properties to code during development. While [30] uses goal regression to learn about the user program, it requires that the program be symbolically executed in order to find vulnerabilities. As discussed in the literature [54], symbolic execution suffers from path explosion, path divergence and challenges with complex path constraints, especially on real world problems. This presents challenges with the generalizabilty of the solution, as confirmed by the authors [30].

In [16], an intermediate language is required to annotate tainted functions in the code. In contrast, the proposed model in this research works directly with the parse tree of the source code to detect patterns for automatic detection and classification of vulnerabilities based on descriptions and fixes recommended by the NVD. Further, the proposed approach mines a large code base and uses the safe examples to provide not only advice but also example fixes to the programmer.

The proposed approach differs significantly from the generate-and-path approaches discussed in the preceding section because patches often work for a given set of test cases, but fail to generalize to other programming projects. For example, in 36, Qi et al. analyzed reported patches for GenProg, RSRepair, and AE, and found that the overwhelming majority of reported patches did not produce correct outputs even for the inputs in the validation test suite [36]. GenProg was reported to find patches for 37 out of 55 defects in a validation suite. However, the researchers found that patches did not produce correct output. Likewise, AE was reported to find patches for 27 out of 54 defects, but did not produce correct outputs in the evaluation conducted by Qi et al. Further reruns by the authors confirmed that GenProg found correct patches for only 2 out of 105 defects. Qi et al. attributed the poor performance of these tools to weak proxies (bad acceptance tests), poor search spaces that do not contain correct patches, and random genetic search that does not have a smooth gradient for the genetic search to traverse to find a solution [36]. These weaknesses highlight the challenge with generate-and-patch or generate-and-validate approaches.

Further, unlike the works that propose stand-alone static analysis tools [18, 19], the proposed work augments static analysis with IntelliSense to drive the mitigation process within IDEs as the programmer types code. As discussed in the literature [5, 55] and confirmed by the knowledge elicitation survey conducted in this work (see Section 5.2), a majority of developers surveyed do not take advantage of stand-alone static analysis tools. Even though these tools may perform well, they require the extra time of going outside of the development environment to perform scans and explore mitigation approaches. However, this new proposed methodology of coupling vulnerability scanning with IntelliSense provides live scanning and mitigation without significantly affecting the developer's coding experience. In addition, by using a recommender system, this work shows that providing the programmer with a ranked set of examples that are most similar to the code being developed allows the programmer to better understand vulnerabilities as they relate to their projects. Other auto-fixing approaches (e.g., DeepFix and generate-and-patch) that automatically transform program code do not provide the programmer with examples that are very similar to the code being developed. Moreover, the unique presentation of information in the form of recommendations has the added benefit of educating programmers on how to avoid certain vulnerabilities in future projects.

3 Proposed approach

The approach consists of two main phases (modeling and application) and two main components (the data analyzer and the recommender system) as shown in Fig. 1. Here, each component is described. A more thorough discussion of the modeling phase is provided in Section 4 while Section 5 covers the application phase (system design and implementation).

The first phase in the proposed approach is the modeling phase. This phase involves analyzing data collected from the National Vulnerability Database (NVD) in addition to open-source programs to identify features for detecting a set of vulnerabilities. These features are then used by a data analyzer to process program code using simple and effective, data-driven vulnerability detectors to detect each vulnerability. The approach currently focuses on the Java programming language but is general enough to apply to other programming languages.

The second phase involves capturing code as the programmer types and transferring it to the recommender system that executes vulnerability detectors, which in turn categorizes the program code based on the knowledge of the recommender system and outputs recommendations that include examples for fixing each vulnerability.

3.1 The data analyzer

The data analyzer consists of feature extractors that are designed based on vulnerability descriptions and fixes from the NVD. The analyzer accepts as input open-source program code and outputs feature sets for detecting a set of vulnerabilities. Open-source projects are mined and source code is categorized in order to provide knowledge to the recommender system for detecting and mitigating each vulnerability. Recommender systems require sufficient data in order to effectively provide useful recommendations to users. Therefore, a distributed framework such as MapReduce is proposed to extract features from a large collection of code repositories to drive the data labeling process. Labeled datasets are used to train the recommender system to provide to the programmer safe code examples that fix a set of vulnerabilities.

3.2 The recommender system

The recommender system incorporates vulnerability detectors that are designed using key insights about vulnerabilities based on data provided by NVD and CWE. It accepts the user's code and utilizes the data analyzer to create a feature set/data object from the given program code. The feature set is used to determine the classification of the data object. If the data object is unsafe, a recommendation that includes a warning that contains a list of unsafe method(s) and variable(s) found in the

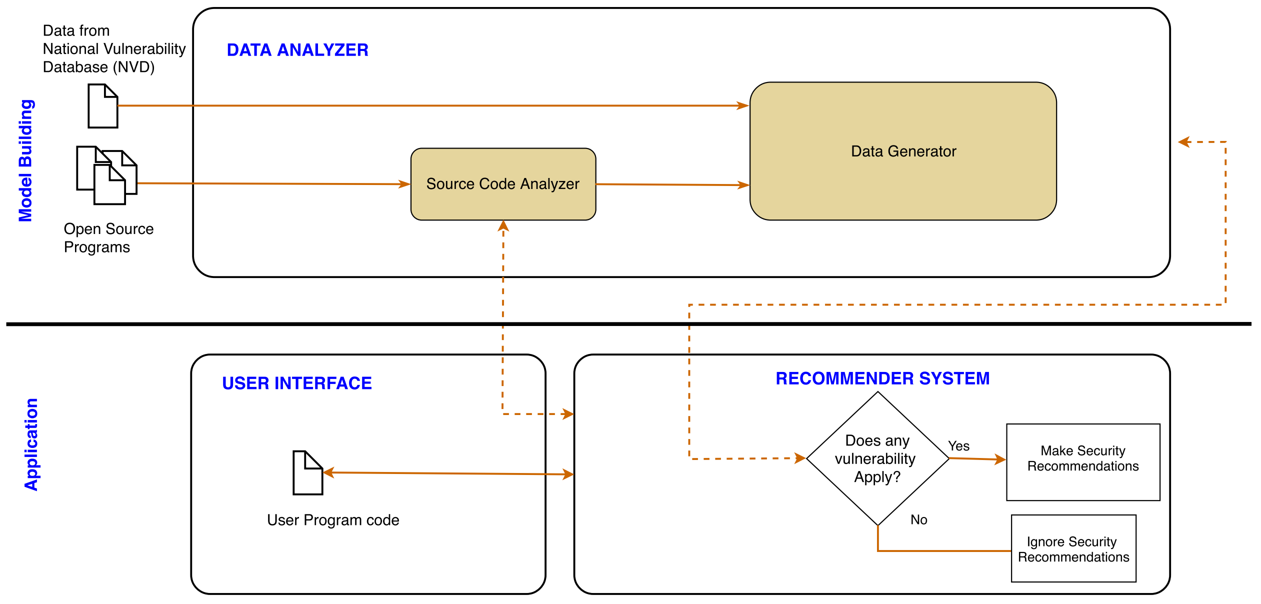


Fig. 1 System architecture

user's code is displayed to the user. The recommendation will also include ranked fixes for each vulnerability. Fixes are ranked using text similarity schemes in order to display a list of examples that resemble the code being developed. IntelliSense technology is used to initiate the recommender system as the programmer types in order to help the programmer mitigate potential vulnerabilities as soon as possible.

4 Modeling and detection

This section discusses the modeling and vulnerability detection phase of the work. It provides a detailed explanation on data representation and feature extraction. Included is a discussion on the feature extraction algorithms and the steps followed to prepare the knowledge base for the recommender system.

4.1 Datasets

Two main datasets (The National Vulnerability Database/Common Vulnerabilities and Exposures (NVD/CVE) and Sourcerer 2011) are used in this work to provide vulnerability descriptions that are important for feature extraction and source code from which feature sets and mitigation examples can be extracted.

The National Vulnerabilities Database (NVD/CVE) CVE is a dictionary of common identifiers for publicly known cybersecurity vulnerabilities, which is hosted by the MITRE Corporation[56]. CVE submissions are made after vulnerabilities are identified in widely used software applications. Each submission is reviewed by a team of experts and is assigned a unique identifier (CVE ID) by a CVE Numbering Authority (CNA), a description, and references. The US National Vulnerability Database is a "comprehensive cybersecurity vulnerability database that integrates all publicly available US Government vulnerability resources and provides references to industry resources" [56]. NVD is provided by the National Institute of Standards and Technology (NIST). NVD enhances the information in CVE to deliver more details for each CVE entry such as fix information, severity scores, and impact ratings according to a Common Vulnerability Scoring System (CVSS)[57].

The Sourcerer 2011 The Sourcerer 2011 dataset is a collection of artifacts based on over 70,000 Java projects and approximately 100,000 Java ARchive (jar) files that were collected from Apache, Google Code and Sourceforge in 2011 [58]. 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 content, which follows the organization system used by the developers, and a *project.properties* file, which contains information such as the repo URL and author. The Java files are processed and used to create the ground-truth for classification and to provide mitigation examples.

4.2 Data representation

Each Java file used in this work is modeled as an Abstract Syntax Tree. An Abstract Syntax tree is an hierarchical intermediate representation of a program that presents source code structure according to the grammar of a given programming language [59]. It is a reduced parse tree in which 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 [59].

The JavaParser library is used 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 [60].

4.3 Feature extraction

Features for detecting vulnerabilities were identified after careful manual analysis of vulnerability descriptions provided by the NVD/CVE. Apache Hadoop was utilized as a MapReduce environment running custom code to process the Sourcerer dataset in order to extract features for detecting the vulnerabilities. MapReduce is a programming model and an associated implementation for processing and generating large datasets [61]. The Apache Hadoop software library is one of the most popular implementations of the MapReduce methodology that allows for the distributed processing of large data sets across clusters of computers using a simple programming model [62].

4.3.1 MapReduce algorithm for feature extraction

The MapReduce algorithm that was implemented for execution in Apache Hadoop is shown in Algorithm 1. Based on the structure of the Sourcerer dataset, it was necessary that the repository be organized before processing using Hadoop. Bash scripts were used to parse each *project.properties* within each project in the repository to extract information about each project in order to create a more uniform file structure. Java files were reorganized such that there is one directory for each project. The filenames were later used as keys for the MapReduce framework. Since Hadoop splits data files according to a default block size, a custom record reader was employed, as shown in the algorithm (line 3), to read each Java file without splitting it. This enabled complete and accurate creation of an AST from each file. Moreover, each vulnerability requires a different *buildFeatureSet* procedure (shown on line 16 of the algorithm). This procedure is discussed below for each of the vulnerabilities evaluated in this work.

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)
        return
                /* 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)
            end
        return
    end
end
```

4.3.2 *Extracting features for detecting taint-style vulnerabilities*

This work uses two taint-style vulnerabilities (SQL Injection and Command Injection) to evaluate the proposed methodology. These vulnerabilities were chosen due to their high CWE severity score and frequency in the 2017 version of the NVD as shown in Fig. 2. Taint-style vulnerabilities are caused by the lack of input/output validation and are traditionally modeled as *source-sink* problems.

CWE-89—improper neutralization of special elements used in an SQL command ('SQL injection'). An SQL injection (SQLI) attack is one that 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 [63]. SQLI has been a common vulnerability for many years, securing position number one on the Open Web Application Security Project (OWASP) 2010 [64], 2017 [65], and the CWE 2011 [66] lists. Applications (e.g., web-apps) generally accept user input, which are then used in executing database requests. These requests are typically SQL statements.

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 [67]. Listing 1 features sample code that could potentially result in SQLI. This is because the programmer is incorporating unsanitized variables in the creation of a query string.

The use of the `PreparedStatement` class from Java Database Connectivity (JDBC) or Java Enterprise Edition (J2EE) is often recommended as a fix for SQL injection [68]. This class allows for the use of a placeholder ("?" character) to create a parametric query that escapes potentially tainted user input. Using these clear descriptions of the vulnerability and how it can be mitigated, six main hand-selected features for detecting and classifying SQLI can be identified. These features are described in Table 1, and the algorithm used to build the feature set is presented as Algorithm 2. A list of known Java sources and sinks was obtained from online resources [69–71]. These known sources and sinks are used as a point of reference along with static dataflow analysis of the user program to identify potentially tainted variables. A variable is considered *potentially_sanitized* if it is passed to a function that is not in the list of known tainted sources. Three techniques are used to check for potential sanitization throughout a given program file: inline (sanitization done during the creation of a query string), in-method (sanitization done as soon as a parameter is passed to a method), and before-use (sanitization of parameters before they are passed to methods that invoke query functions). By using the list of generated features, a data instance in the dataset is automatically labeled as safe if the boolean feature *quoted_variables_found* is false, the incoming variables are *potentially_sanitized*, and parameterized queries are used to create the SQL statements. Contrariwise, it is labeled as unsafe. A random sample of 100 labeled instances was tested and no errors were found, giving a 99% confidence, which indicates the effectiveness of the selected features. Live detection

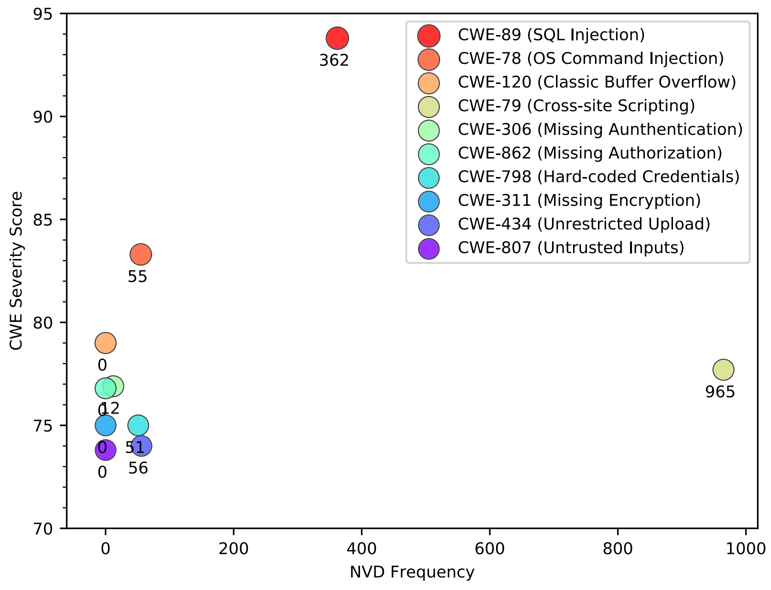


Fig. 2 Number of vulnerabilities in the NVD 2017 List that were caused by the top 10 SANS/CWE of 2011. The plot also shows the CWE severity score for each CWE

of SQLI is done in conjunction with Algorithm 2 as follows:

1 Create an AST from the Java program file.
2 Extract import statements, SQL statements, method calls, sources, and sinks from the AST.
3 For all SQL statements in the program, check if variables are potentially sanitized using static dataflow analysis by comparing the sources and sinks in the program with a knowledge base of known sources and sinks and checking if apostrophes and/or parameterized queries are properly used.
4 If these checks show that data is not properly sanitized and parameterized queries are not properly used, then consider the program susceptible to SQLI and use the recommender system to recommend the most appropriate fix that is most similar to the project being developed.

Listing 1 Example Java code that could potentially lead to SQL injection

```
import java.sql.*;

class Login {
        public boolean doLogin(String username, String
                pwd) {
                  String sqlString = "SELECT * FROM
                        db_user WHERE username = '" +
                     ↪ username + "' AND password = '"
                        + pwd + "'";
                  Statement stmt = connection.
                        createStatement();
                  ResultSet rs = stmt.executeQuery(
                        sqlString);
        }
}
```

CWE-78—improper neutralization of special elements used in an operating system (OS) command ('OS command injection'). Command injection is an attack in which the goal of the attacker is to execute arbitrary commands on the host operating system via a vulnerable application [72]. As the name suggests, these commands are typically targeted to the command shell, which is a software program that provides direct communication between the user and the operating system [73]. The commands supplied by the attacker are usually executed with the same privileges of the vulnerable application.

In Java applications, calls to the `Runtime.exec(...)` method could be exploited to allow an attacker to run arbitrary commands on the host operating system. Listing 2 shows example code that is vulnerable to command injection. This is because it utilizes the Windows command shell (`cmd.exe`) to execute the `dir` command without proper sanitization. After careful analysis of this vulnerability, four main features have been manually selected for detection and classification. Table 2 describes each feature while Algorithm 3 outlines the *buildFeatureSet* procedure. From the feature set, the following heuristic can be used to automatically categorize the dataset for command injection: if shell commands are present and unsanitized, arguments/variables are used in the command string or any faulty characters are used in the command string, label the data instance as unsafe. Otherwise, label the instance as safe. A random sample of 100 labeled instances was also tested, showing no errors (99% confidence) in the labels assigned to the command injection dataset.

Algorithm 2: Procedure for building the feature set for detecting SQL injection

```
input  : ast: abstract syntax tree of Java code
output: a set of features for detecting SQL
        Injection
1  Procedure buildFeatureSet (ast)
2      Initialize featureSet parameters as safe
3      imports = Get list of ImportDeclaration from ast
4      sqlStatements = Extract all statements containing SQL Commands from ast
5      methodCalls = Get list of MethodCallExpr from ast
6      sources = Get list of all tainted sources from imports
7      sinks = Get list of all sinks from methodCalls
8      Set feature sources = sources
9      Set feature sinks = sinks
10     foreach sqlStatement ∈ sqlStatements do
11         if sqlStatement is concatenated string then
12             Create stmtArray from sqlStatement
13             foreach item ∈ stmtArray do
14                 if item is functionCallExpr && item ∈ taints then
15                     Set feature potentially_sanitized = false
16                 end
17                 if item is variable && item not passed to potential sanitizer function then
18                     Set feature potentially_sanitized = false
19                 end
20                 if item is string && item contains apostrophes then
21                     Set feature quoted_variables_found = true
22                 end
23             end
24         end
25     end
26     if all sqlStatements parameterized then
27         Set feature all_queries_parameterized = true
28     end
29     if preparedStatement class found ∈ imports then
30         Set feature prepared_statement_imported = true
31     end
```

Listing 2 Example of unsafe Java code that uses runtime exec

```
import java.io.*;
class ChangeDir {
      public static void main(String[] args) {

          Runtime runtime = Runtime.getRuntime();
          String[] cmd = new String[3];
          cmd[0] = "cmd.exe" ;
          cmd[1] = "/C";
          cmd[2] = "dir " + args[0];
          Process proc = runtime.exec(cmd);
      }
}
```

Algorithm 3: Procedure for building the feature set for detecting command injection

```
input  : ast: abstract syntax tree of Java code
output: a set of features for detecting Command
        Injection
1  Procedure buildFeatureSet (ast)
2      Find (all ExpressionStatements ∋ the exec method) ∈ ast
3      foreach cmd parameter ∈ exec statement do
           // the cmd parameter is the 1st parameter based on the exec method signature
4          if cmd parameter is concatenated string then
5              updateFeatureSet (cmd)
6          else
7              Find all occurrences of cmd variable in ast
8              if any occurrence is string then
9                  updateFeatureSet (cmdOccurrence)
10             end
11         end
12     end
13 Function updateFeatureSet (cmdString):
14     if concatenated variables ∈ cmdString ¬ potentially sanitized then
15         Set feature unsanitized_args_processed = true
16     end
17     if cmdString ∋ a call to a shell command then
18         Set feature shell_command_present = true
19     end
20 return
```

4.3.3 Results of the text-mining process

To prepare data for the knowledge base within the recommender system, the MapReduce algorithm was implemented in Java and executed in Apache Hadoop. The

Table 1 Features for detecting SQL injection

Feature	Data type	Possible values	Description
Sources	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
Sinks	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 a 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 prepared statement 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

data in Part *aa* of the Sourcerer 2011 dataset was used to create the knowledge base. Table 3 summarizes the distribution of the projects within the subset of the dataset that was analyzed. Specifically, the Sourceforge projects and Google Code projects were processed to create training data and test data, respectively. Table 4 shows the breakdown of the training and testing samples.

5 Methods

This section describes the methods followed in designing and implementing the system. First, initial ideas on the requirements and design of a useful and effective code analysis tool are delineated. Next, the steps involving a knowledge elicitation survey that was conducted to empirically ascertain the current use of code analyzers among programmers and to elicit their views on the design of the proposed system are presented. Finally, a discussion on the impact of the survey on the final design of the system is provided.

5.1 Initial system design

Due to the observation that many programmers are skeptical of using existing code analyzers, the following

Table 2 Features for detecting OS command injection

Feature	Data type	Possible values	Description
Shell_command_present	Boolean	{True, false}	Tells whether a shell command is supplied to runtime.exec. Shell commands include command.com, cmd.exe, /bin/sh /bin/csh, /bin/ksh, /bin/bash, /bin/tcsh, /bin/zsh, /bin/rc, /bin/es
Unsanitized_args_processed	Boolean	{True, false}	Specifies whether the programmer passes potentially tainted user arguments to the runtime.exec method
Faulty_characters_present	Boolean	{True, false}	Specifies whether faulty characters are present in the command passed to the runtime.exec method
File_permission_imported	Boolean	{True, false}	Tells whether the recommended Java File permission class is imported to prevent command injection
Metadata	String	–	A field containing runtime examples and methods found in each Java file
Class	Binary	{Safe, unsafe}	The target variable

requirements are worth considering during the design of a new system:

- The system must be a part of the IDE to enable effective scanning and mediation
- The warnings should be brief and actionable (links to more detailed information should be provided for interested users)
- Emphasis should be placed on fixing the potential vulnerabilities and encouraging good programming practice
- The fixes should not be generic but as specific as possible to the project being developed
- Scanning of vulnerabilities should be done such that the programmer's productivity is not negatively impacted

By using this inexhaustive list of requirements, a mockup of the proposed system was created (see Fig. 3). The proposed tool is called VulIntel, short for Vulnerability IntelliSensor. The tool is intended to be part of the IDE and uses IntelliSense technology to scan code as the programmer types. A list is populated with the names/IDs of potential vulnerabilities. Clicking on a vulnerability in the list displays a brief description of the vulnerability including a reference to the unsafe method and variables involved. Further, a ranked list of examples is presented to the user to help with mitigation.

Table 3 Distribution of projects in part "aa" of the Sourcerer dataset

Repository	Number of projects	Number of java files
Google Code	6865	605809
Sourceforge	7511	1015732
Miscellaneous	—	625302
Total	*14376*	*2246843*

5.2 Knowledge elicitation survey

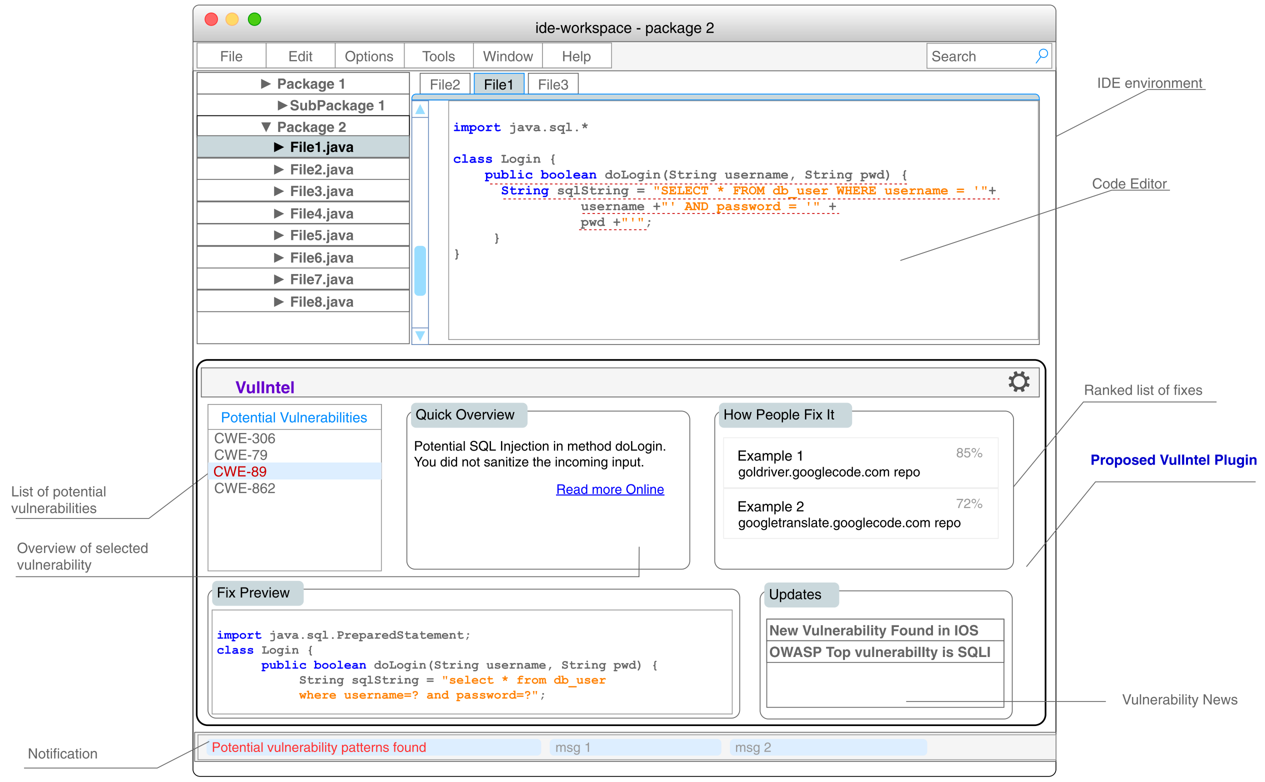
It is important to solicit feedback for any system design to satisfy usability requirements as well as to answer questions that will assist with development. Consequently, an online knowledge elicitation survey was conducted with the main goal of obtaining formative feedback on the design of the proposed interface and the views of programmers about a tool that utilizes IntelliSense technology to find vulnerabilities in program code and provides recommended fixes for detected vulnerabilities. Approval[4] to conduct the study was obtained from the Institutional Review Board at Florida Institute of technology. The results from the survey are summarized below and the survey questions are included in Appendix A.1.

5.2.1 Participants

To recruit a diverse population of participants, invitation emails with a link to the survey were sent to individuals of various experience levels in industry and academia. The list consisted of more than 10 organizations from countries that included the USA, Brazil, Germany, and the UK. The main criteria for participants was that they have at

Table 4 Breakdown of data within the knowledge base of the recommender system

Knowledge base			
SQLI corpus			
Repository	*Safe*	*Unsafe*	*Total*
Google Code	6164	1629	7793
Sourceforge	9459	2584	12043
Total	*15623*	*4213*	*19836*
Command injection corpus			
Google Code	482	19	501
Sourceforge	2250	70	2320
Total	*2732*	*89*	*2821*



```
import java.sql.*

class Login {
    public boolean doLogin(String username, String pwd) {
      String sqlString = "SELECT * FROM db_user WHERE username = '"+
                username +"' AND password = '" +
                pwd +"'";
    }
}
```

Potential SQL Injection in method doLogin.
You did not sanitize the incoming input.
Read more Online

Example 1 85%
goldriver.googlecode.com repo

Example 2 72%
googletranslate.googlecode.com repo

```
import java.sql.PreparedStatement;
class Login {
      public boolean doLogin(String username, String pwd) {
           String sqlString = "select * from db_user
           where username=? and password=?";
```

New Vulnerability Found in IOS
OWASP Top vulnerabillty is SQLI

Potential vulnerability patterns found

Fig. 3 Mockup of proposed system as incorporated in an IDE

least 3 years experience with an object-oriented programming language such as Java, C#, or C++. A total of 104 participants completed the survey (44 graduate students, 39 industry professionals, 11 undergraduate students, 7 professors, and 3 others).

5.2.2 Familiarity with programming languages and IDEs
Participants were asked to select their familiarity with a set of programming languages from a list that uses a 5-point Likert scale[5]. Participants's main language of choice was the Java Programming language, with 40% indicating that they are "very familiar" with it and 25% claiming to be "experts" (see Fig. 5). The IDE that scored the highest in use frequency (84.62%) among participants was Eclipse. This was followed by Visual Studio with 73.08% and Netbeans with 61.52%. The results are summarized in Figs. 4 and 5.

5.2.3 Results and discussion
The answers to four overarching questions that the survey was designed to address are discussed below along with a summary of themes that emerged from the survey.

(1) To what extent are programmers using code analyzers? To answer this question, participants were asked whether they performed static and/or dynamic analysis on their code and how useful they found the given recommendations. 13.46% of the participants stated that they used a static analyzer such as FindBugs, 3.85% used a dynamic analyzer such as Java PathFinder, 9.62% used both dynamic and static analyzers, and 56.73% reported that they did not scan their code for vulnerabilities.

(2) How useful are the advice/recommendations provided by existing tools? This question was presented to participants who indicated that they currently take advantage of existing code analyzers. 25.81% of this group of participants described as "helpful" the recommendations they received from the scanners they used and 67.74% reported that the advice given was "somewhat helpful" in fixing vulnerabilities.

(3) Would programmers utilize a tool that uses IntelliSense technology to find and suggest fixes for vulnerabilities? Participants were first asked if they currently take advantage of IntelliSense technology. Sixty-eight participants (68%) reported that they currently utilize the technology while 32 (32%) did not; 4 participants skipped the question. In addition, the participants were asked their opinion about the application of IntelliSense technology

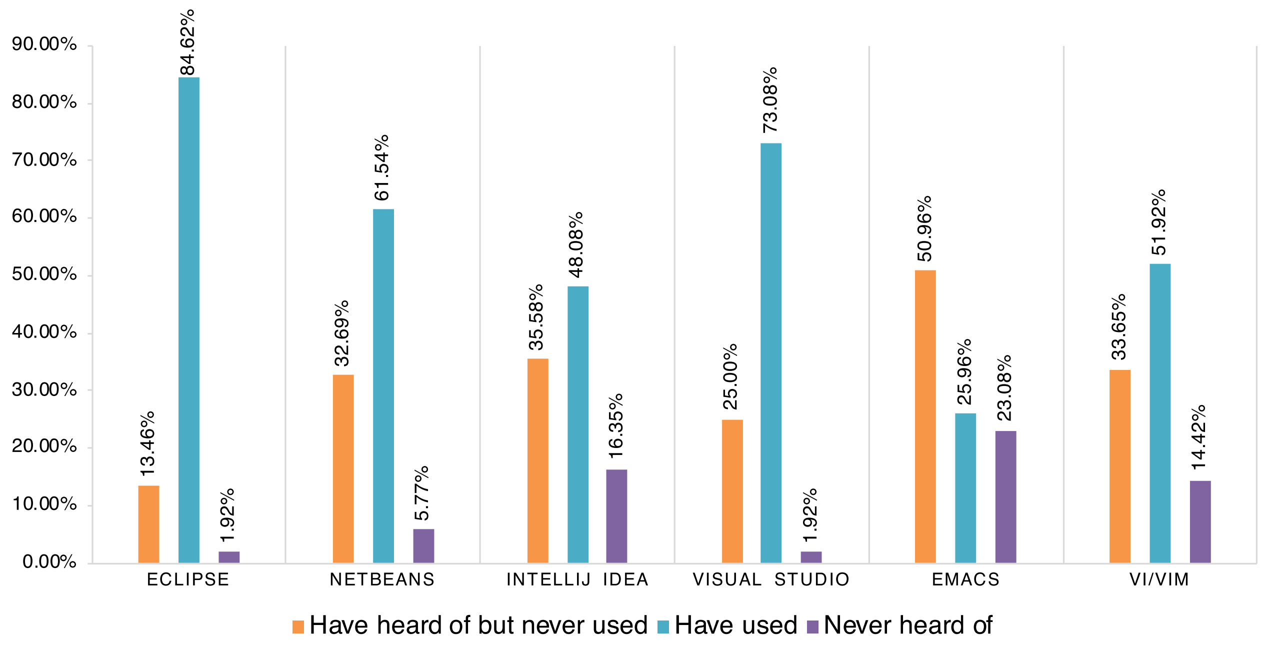


Fig. 4 Participants' familiarity with IDEs

to vulnerability detection. 87 of the participants (87%) intimated that they would appreciate a system that can scan their code for vulnerabilities as they code; 10 (10%) were not interested in the technology, but believe other programmers may be interested; 3 participants did not believe it would be a good idea to apply IntelliSense to vulnerability detection, and 4 skipped the question.

(4) What are the design criteria and expectations for a tool that scans code for vulnerabilities and presents fixes to the user? The participants were then shown the mockup (see Fig. 3) of the proposed tool and asked in what situations and for what types of projects they would utilize it. The responses are summarized in Figs. 6 and 7. Moreover, they were asked their opinion about

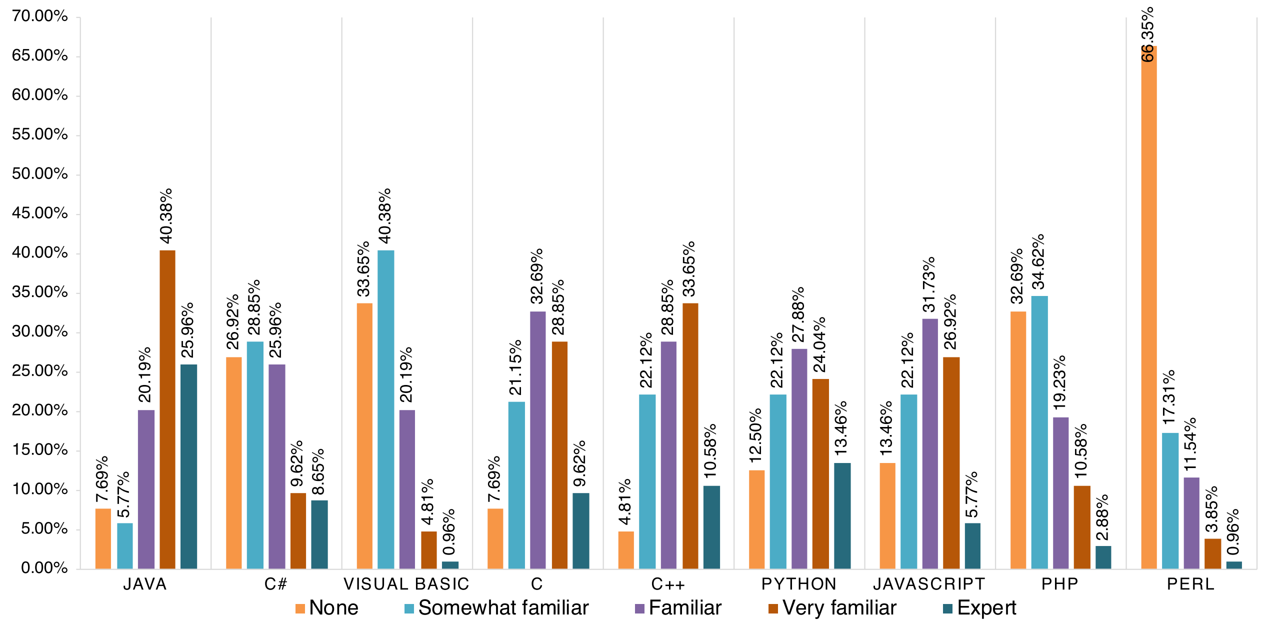


Fig. 5 Participants' familiarity with programming languages

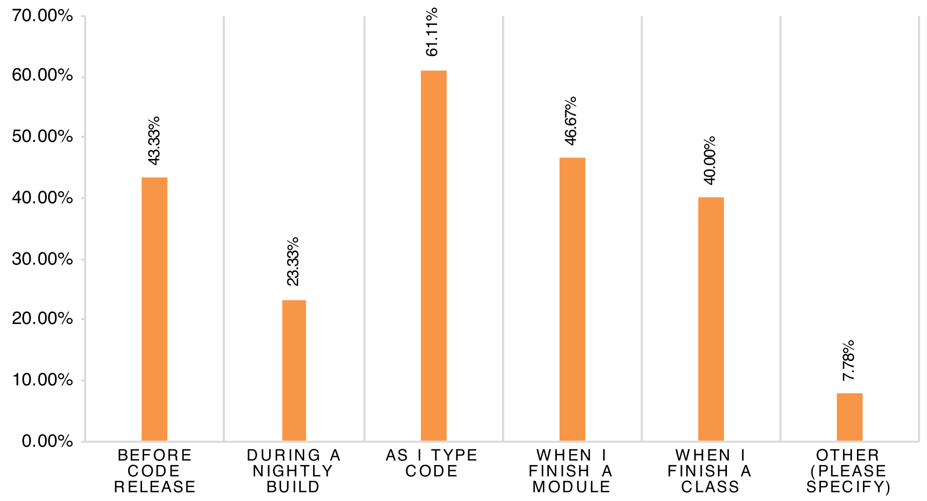


Fig. 6 Situations under which programmers would use the proposed plugin

what they (dis)liked about the interface and what types of vulnerabilities they would like to detect using the tool.

Themes that emerged from the survey

Several important themes stood out in the responses provided by participants in the knowledge elicitation survey as evaluated using the grounded theory approach [74]. From the list of vulnerabilities provided by the participants, SQL injection, buffer overflows, and the OWASP list of vulnerabilities are well-known and important to programmers. However, there are other vulnerabilities that are often overlooked by programmers but could pose significant risks. For example, Fig. 2 shows that hard-coded credentials (CWE-789) and missing encryption (CWE-311) account for dozens of vulnerabilities in the 2017 NVD release, yet these vulnerabilities were not mentioned by any participant.

Three main themes emerged from the open-ended responses that were provided by the participants:

Theme 1: usability Some participants were concerned about the number of objects on the proposed UI. They suggested that while updates are important, the "news updates" panel adds clutter to the interface and should be minimized if possible.

Theme 2: performance While some participants were in favor of scanning being done in the background, a few of them were concerned about the impact this may have on the code editor. For example, one participant submitted the following response:

"I like that it tells you security vulnerabilities as you type. I am a little concerned about how efficient

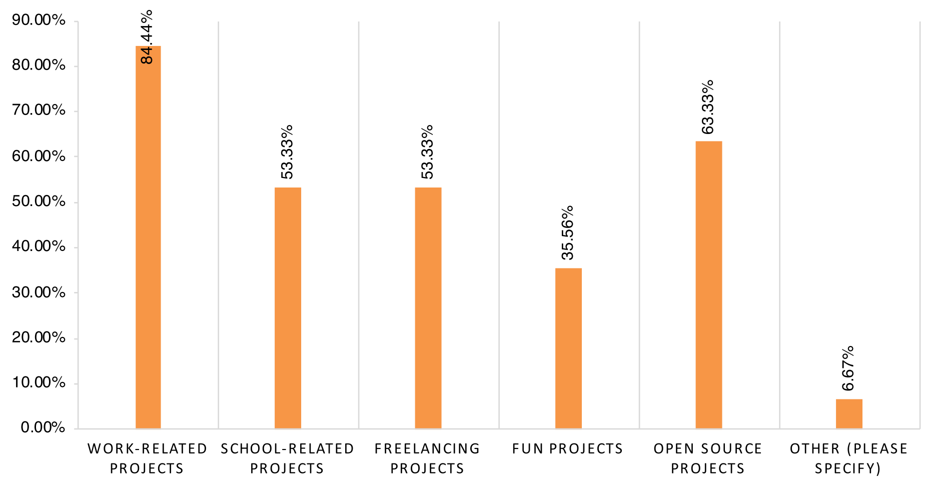


Fig. 7 Types of projects for which programmers would use the proposed plugin

scanning for these vulnerabilities might be. I would most likely stop using it if it slowed down my editor."

Theme 3: fixing vulnerabilities A number of participants commented on the plugin's proposed ability to provide fixes for the vulnerabilities that it finds. One participant provided the following feedback:

"Really helpful as it provides you with multiple fixes and examples and visually appealing."

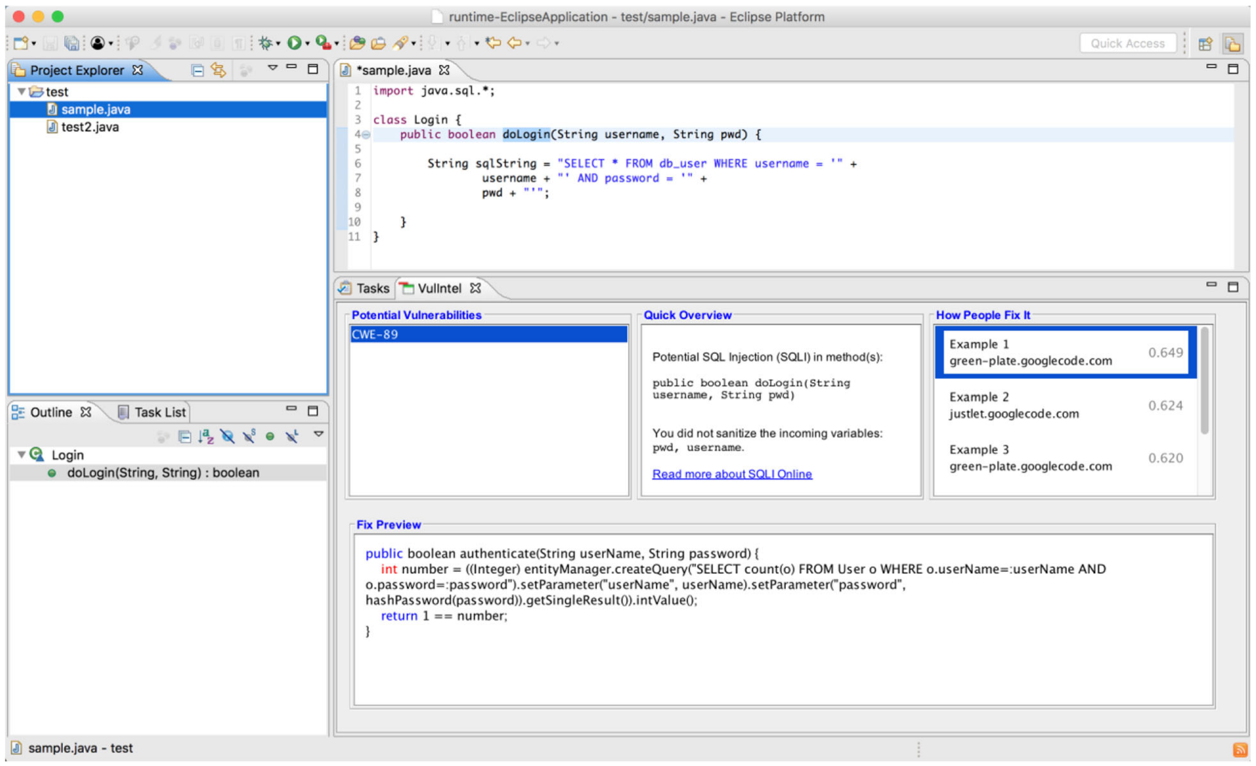
5.3 Final system design

The aforementioned themes were used to influence the design of the final system. For example, the theme of usability helped to declutter the interface. First, the knowledge base within the recommender system was updated with knowledge from open-source projects as discussed in Section 4.3. The model was serialized and imported into an Eclipse plugin. The Eclipse IDE was chosen because of its familiarity among surveyed programmers as discussed earlier. Figure 8 shows a screenshot of the final system as an Eclipse plugin. The design of the plugin was influenced by the responses received in the knowledge elicitation survey. IntellliSense technology was utilized by extending the Eclipse Code Recommenders [75] system, which is a fundamental component within the Eclipse intelligent code completion framework. The IntelliSense system was programmed to initiate the scanner after the user enters or removes at least five characters, excluding spaces. This behavior was chosen after experimenting with options such as after method completion or after entering or removing at least 10 characters.

5.4 Recommending fixes for vulnerabilities

It is of interest to use the vulnerability-safe (negative) examples from the labeled corpora to provide recommendations to help programmers fix the detected vulnerabilities. Several questions arise in determining a similarity scheme that finds code that is similar to the user's code but is safe against the vulnerabilities found in the user's code. For example, what is the best trade-off between the time taken to find similar code that is not only syntactically relevant but also semantically helpful to the user? To answer this and other questions, experiments were conducted using three text similarity schemes (cosine similarity, MinHash, and SimHash) in order to select one that takes the least amount of time to find *relevant* examples.

The cosine similarity between two vectors (or two programs) is a measure that calculates the cosine of the angle between them irrespective of the magnitude of the vectors. In this work, the vectors represent the term frequencies of terms that are common between two programs (methods). The vectors were created by using Apache



```
import java.sql.*;

class Login {
    public boolean doLogin(String username, String pwd) {

        String sqlString = "SELECT * FROM db_user WHERE username = '" +
                username + "' AND password = '" +
                pwd + "'";

    }
}
```

Potential SQL Injection (SQLI) in method(s):

`public boolean doLogin(String username, String pwd)`

You did not sanitize the incoming variables: `pwd, username`.

Read more about SQLI Online

```
public boolean authenticate(String userName, String password) {
   int number = ((Integer) entityManager.createQuery("SELECT count(o) FROM User o WHERE o.userName=:userName AND
o.password=:password").setParameter("userName", userName).setParameter("password",
hashPassword(password)).getSingleResult()).intValue();
   return 1 == number;
}
```

Fig. 8 Screenshot of the final design of the plugin as incorporated in the Eclipse environment

Lucene [76] to tokenize the Java code and remove Java keywords and other English stop words from the code.

Minhash is a Locality Sensitive Hashing (LSH) technique based on the min-wise independent permutations of sets. The goal of MinHash is to estimate the Jaccard similarity quickly without explicitly computing the intersection and union of the sets. Jaccard is the ratio of the number of elements in the intersection of two sets to the number of elements in the union.

SimHash is also a LSH for the cosine similarity measure that maps high-dimensional vectors to small fingerprints [77]. It is based on the concept of Signed Random Projections (SRP) that transforms a multi-dimensional vector into a binary string and stores only the sign of the random projection values.

Figure 9 presents the results from an experiment that compares the three similarity approaches. First, the figure shows a sample user code that is vulnerable to SQLI. Next, the most similar code that fixes the vulnerability, as returned by each algorithm, is presented. The figure also shows the similarity score and the time taken to search a dataset of 18,842 safe instances for code that is similar to the user's code. As can be seen from the results, all three algorithms finished the search in under 2 s. Moreover, the returned samples suggest that cosine similarity produced a more semantically similar piece of code to the user's code.

5.5 Usability study

The study[6] followed the A/B testing format where participants used two tools to complete two tasks and provide feedback based on the experience they had while using both tools. While A/B tests are traditionally used to compare the performance of or user preferences regarding two different versions of a particular tool or design, it is used in this work to compare two different tools with two different

User's Unsafe Example Code

```java
import java.sql.*;
    class Login {
        public boolean doLogin(String username, String pwd) {
        String sqlString = "SELECT * FROM db_user WHERE username = '" + username + "' AND password = '" + pwd + "'";
    }
}
```

Safe Code that is Similar to User's Code

Metric | Similar Sample Code | Similarity Score | Avg Time (sec)

Cosine Similarity

```java
public boolean authenticate(String userName, String password) {
    int number = ((Integer) entityManager.createQuery("SELECT count(o) FROM User o
    WHERE o.userName=:userName AND o.password=:password")
    .setParameter("userName", userName).setParameter("password", hashPassword(password))
    .getSingleResult()).intValue();
    return 1 == number;
}
```

0.65 | 1.63

Sim Hash

```java
public synchronized Document query_GetNewMessagesFromDB(String userID, String messageID) {
    // this is hard-coded now, could have set this via argument if >1 chats
    // per userID were needed
    final String chatID = "21";
    if (datasource == null) {
        connect();
    }
    // get XML style date format: 2006-11-15T14:44:09
    String command = "SELECT message_id, user_name, message,
    date_format(post_time,'%Y-%m-%dT%H:%i:%s') as post_time" +
     " FROM message WHERE user_id = ? AND chat_id = ? AND message_id > ?
    ORDER BY 'message_id' ;";
    PreparedStatement ps = null;
    ResultSet result = null;.
    ...
}
```

0.62 | 1.85

Min Hash

```java
public List findAskAnswerByAsk(Integer askId) {
    try {
        String queryString = "SELECT DISTINCT model FROM AskAnswer AS model WHERE model.ask = ? ";
        org.hibernate.Query query = getSession().createQuery(queryString);
        query.setParameter(0, askId);
        return query.list();
    } catch     (RuntimeException re) {
        log.error("find by property name failed", re);
        throw re;
    }
}
```

0.80 | 1.66

Fig. 9 Finding safe code that is most similar to the user's code

interfaces geared towards vulnerability detection and mitigation. The proposed tool uses an IntelliSense approach to detect vulnerabilities while the second tool (FindBugs) does not use IntelliSense. FindBugs was chosen as the second tool due to its coverage in the literature [78, 79], its adoption by major companies such as Google [80], its open-source nature, and its target language being Java.

First, the goal of the study is outlined, followed by the methodology, which includes a description of the participants, the apparatus and materials used, and the methods employed in the study. The results of the study are then presented along with a discussion on their significance.

5.5.1 Study goal

The overall goal of the study was to ascertain the usefulness and usability of a recommender system in helping programmers write more secure code.

5.5.2 Participants

Fourteen participants completed the study (1 professor, 1 industry professional, 4 researchers, 3 undergraduate students, 1 master's student, and 4 PhD students). These participants were recruited using a combination of convenience and snowball sampling via email and word of mouth. Nine subjects were in the age group 18–29, four between 30–49, and one between 50–64. Participants ranged in coding experience with 13 people having at least 3 years experience and 1 person between 0–2 years. Subjects were asked to select their primary programming languages and 9 of them selected Java and Python as their languages of choice.

5.5.3 Apparatus and materials

All participants used a Dell Latitude 3550 laptop (Intel Core i3 - 1.70 GHz CPU, 64-bit, 8 GB of RAM) to complete the tasks. The study took place in a classroom in the Harris Institute for Assured Information at Florida Institute of Technology, with one participant and one experimenter per interview; each session lasted 30 to 45 min. The Eclipse IDE (version Oxygen.3a 4.7.3) was installed on the computer beforehand. The VulIntel plugin and the FindBugs plugin (version 3.0.1) were also installed before the study started. To have a fair comparison of tools, FindBugs, which includes the FindSecBugs plugin, was configured to target only security Bugs. This was done to minimize the effect of unrelated issues on the scanning time or presentation of errors to the participants because FindBugs is able to find bugs related to bad practice, correctness, performance, etc., while VulIntel currently scans for security-related vulnerabilities.

5.5.4 Methods

First, the experimenter presented the participant with an Informed Consent Form. The experimenter reviewed the contents of the form and gave participants a randomly assigned ID that was used to refer to the participant throughout the study. After reviewing the contents of the consent form and the required tasks for the study, the participant was given the option to withdraw or to proceed by signing the form. The study then began with a short demographic-style questionnaire (see Appendix A.2.1) that was designed using Google Forms. After signing the consent form, the interviewer told the participant the order of the tools they would be using. Tool order was alternated to avoid learning bias (i.e., 7 participants used FindBugs first before using VulIntel while 7 used VulIntel before using FindBugs). The interviewer then explained to participants how to use the first tool to scan their code for vulnerabilities and how to use the information the tool provided to fix any potential vulnerabilities. Participants were told that they should use only the information provided by the tool, and no other resources, to fix any reported vulnerabilities.

Next, the experimenter activated screen-capturing (and audio-recording) software, stepped aside, and allowed the participant to complete the two tasks using the first tool. After completing the tasks using the first tool, the participant was then given a questionnaire (see Appendix A.2.2) followed by an interview (see Appendix A.2.3) based on their experience using the tool to scan and fix the given code of potential vulnerabilities. If a participant was unable to fix the vulnerabilities using the tool, the experimenter allowed the participant to proceed with the next tool. The screen-capturing software was closed and the same experiment was given for the second tool.

Tasks Each participant was given two tasks related to the top two taint-style vulnerabilities discussed earlier (see Section 4.3.2). Each task consisted of the user typing preselected sample code into the text editor of the Eclipse IDE while the code scanner window was open and the scanner activated. Two Java classes containing sample methods were created prior to the experiment with vulnerable portions of the code removed, so the participant could type, observe the behavior of the scanner, and use the information provided by the scanner to fix the vulnerability.

The code used for Task 1 (SQL injection) is a modified version of an example provided by the Software Engineering Institute at the Carnegie Mellon University [81] while the code used for Task 2 (Command Injection) was obtained from the OWASP website [82].

6 Results and discussion

This section presents the results and discussion of the usability study and scalability analysis of the proposed tool.

Figure 10 provides a frequency summary of participants' responses to four main questions asked on the question-

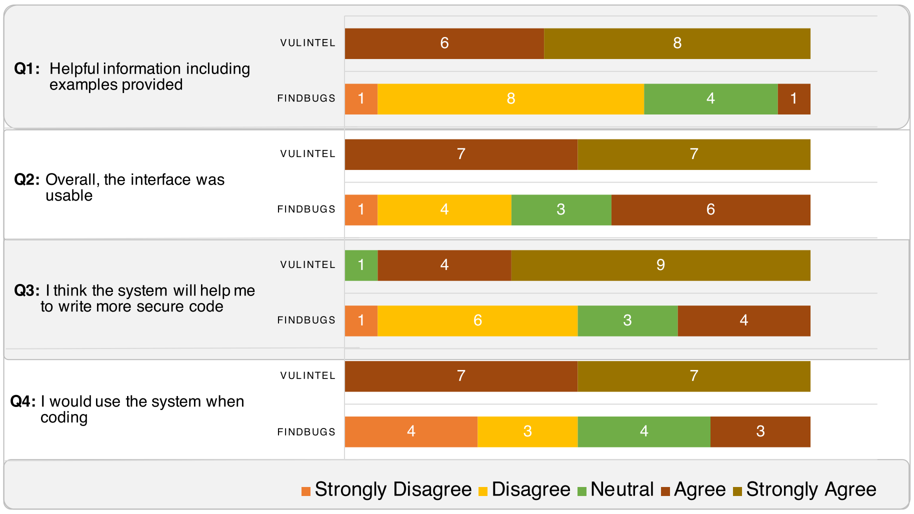


Fig. 10 Summary of participants' responses to four main questions

naire (see Appendix A.2.2) for each tool after participants completed the tasks. All four questions were presented using a 5-point Likert scale[7]. As can be seen from the Fig. 10, more people agreed with VulIntel satisfying these questions positively than those who agreed that FindBugs did the same. If the Likert scale is collapsed into two categories (agree and disagree) by removing the neutral responses, the following can be concluded:

- Fourteen participants agreed that VulIntel provided helpful information including fixes for the two tasks given whereas only 1 participant agreed that FindBugs provided the same.
- Fourteen participants agreed that the VulIntel interface was usable whereas only 6 agreed that the FindBugs interface was usable.
- Thirteen participants indicated that they think VulIntel would help them write more secure code while only 4 participants think that FindBugs would help them to write more secure code.
- All participants stated that they would use the VulIntel system when coding while only 3 participants would use FindBugs.

6.1 Scalability

While the goal of this work is to couple text mining techniques with IntelliSense technology to create a recommender system that detects and mitigates vulnerabilities in user programs, it is also of interest to determine the scalability of the proposed methodology on projects of various sizes. To do so, a random sample of 10 Google Code projects in the dataset was selected and processed for SQL injection. Table 5 shows the time taken to classify these projects for SQLI by using a Macbook Pro laptop (16GB of RAM, 3 GHz Intel Corei7 processor). The experiment was done while other processes were running on

Table 5 Time taken to detect SQLI in various open-source projects

Repository name	Number of java files	Total LLOC	Total classification time (Sec)	SQLI found
gwtspeechbubble.googlecode.com	3	82294	0.009	FALSE
ov2java.googlecode.com	4	133122	0.010	FALSE
xmlui.googlecode.com	4	110426	0.044	FALSE
permutationcombination.googlecode.com	12	327806	0.043	FALSE
org2hash.googlecode.com	26	434457	0.136	FALSE
teknoatolye.googlecode.com	39	666449	0.147	FALSE
grimwepa.googlecode.com	43	1315067	0.668	TRUE
lambdacore.googlecode.com	56	969340	0.177	FALSE
gracedm.googlecode.com	266	5951640	2.544	FALSE
oxygensoftwarelibrary.googlecode.com	545	4534779	7.173	FALSE

the machine in order to mimic the environment of a typical developer/programmer. The table also provides other information on the experiment such as the total LLOC (logical lines of code) for each project and the number of files in each one. LLOC was computed by counting the expression statements (an expression followed by a semicolon) in each AST.

The results show that the approach scales very well by being able to scan a project of over 4.5 million lines of code for SQLI in under 8 s while projects of up to 1 million lines of code take under a second. Even though the experiment was only done for one vulnerability, the scanning process can be parallelized through the use of threads to maintain this performance while scanning for other vulnerabilities. This parallelization is feasible since threads are already being used by the tool to find similar example code that mitigates vulnerabilities.

6.2 Discussion

Statistical significance Four paired sample t tests and analysis of variance (ANOVA) tests were conducted for the four questions discussed previously. t tests are used to determine whether the mean difference between two sets of observations is equal to zero (that there is no difference between the groups being explored). ANOVA tests were done to check whether the choices of participants depended on the order of the tools presented during the study (i.e., whether there is interaction between tool-order and participants' agreement). To obtain numeric data for carrying out the tests, the Likert scale was converted to an ordinal scale[8]. All t tests were two-tailed and defined as follows: ($H_0 : \mu_d = 0$ and $H_1 : \mu_d \neq 0$). The results are summarized in Table 6.

As can be seen from the table, the p values are statistically significant for the paired-sample t tests on all four factors regarding participants' agreement. Therefore, the null hypotheses are rejected and the conclusion that the proposed tool was more usable and helpful than FindBugs in helping programmers write more secure code are supported. Additionally, the p values for the ANOVA tests show that the null hypothesis that states that there is no interaction between tool order and participants' agreement cannot be rejected. Therefore, the conclusion is that tool order did not affect the choices of participants. These results confirm the hypothesis that surveyed participants strongly believe that a recommender system built using text mining techniques can help programmers write more secure code.

6.2.1 Study limitations

The convenience sampling done for the usability study conducted in this work poses a few limitations.

Sample size The number of participants, which were limited to professional code developers, though relatively diverse in experience, was small ($N = 14$). There is the potential of obtaining different results with a larger sample. However, since it is typical in the usability community to conduct studies with focus groups between 6 and 10 participants [83], the results presented in this initial study are acceptable. Further, the statistical significance reported helps to strengthen the conclusions.

Gap between tool age The gap between the age of both tools is also worth mentioning. FindBugs was originally released in 2006, with its most recent release in 2015. The proposed tool in this study has not yet been released to the public. Therefore, age difference between the two tools may have some effect on the results.

Experimenter demand effects Demand effects could also pose a limitation. However, this limitation may be very minimal, since none of the participants involved in the study has ever seen or worked with the featured tools and tool order was alternated during the study.

7 Conclusions and future work

In this work, a methodology is proposed, designed, and evaluated to help programmers fix potential vulnerabili-

Table 6 Results from paired-sample t-tests and one-way ANOVA tests for four tool factors. A/B test represents participants who used FindBugs before using VulIntel whereas B/A test represents the opposite

Paired sample t-Test					One-way ANOVA test based on tool order			
					A/B test		B/A Test	
Tool factor	*t* statistic	DF	*p* value	95% confidence interval	*F* value	Pr(>*F*)	*F* value	Pr(>*F*)
Helpfulness *(Q1)*	10.3333	13	1.228e-07	[1.7514, 2.6772]	0.1200	0.735	1.0909	0.3169
Usability *(Q2)*	6.5655	13	1.81e-05	[1.0064, 1.9936]	2.7000	0.1263	0.2500	0.6261
Secure coding ability *(Q3)*	7.3202	13	5.83e-06	[1.3091, 2.4052]	1.1707	0.3005	0.0000	1.0000
Adoption *(Q4)*	6.1085	13	3.729e-05	[1.3389, 2.8040]	2.0769	0.1751	0.2500	0.6261

ties as they type code during development. The proposed methodology employs the use of text mining techniques to extract features from code repositories in order to categorize code and use data-driven vulnerability detectors to detect vulnerabilities. The vulnerability detectors work in unison with a recommender system to provide the programmer during development with ranked code examples that resemble the project being developed in order to mitigate a set of vulnerabilities. This work advocates the use of a recommender system that uses similarity metrics to recommend a set of example fixes instead of using the traditional approach of automatically fixing the user's code. Providing the user a set of similar examples that are safer than the code being developed not only allows the user to fix vulnerabilities but also educates the program on how to avoid the errors that lead to vulnerabilities in future projects.

A usability study showed that all 14 participants involved agreed that the proposed system was more usable than the FindBugs system, and it provided more helpful advice including fixes for the tasks they completed using the system. In addition, all but one participant indicated that the proposed system would help them to write more secure code. The results were statistically evaluated, and paired sample *t* tests and ANOVA suggest that there is statistical significance, confirming the applicability of recommender systems to secure coding.

7.1 Future work

Future directions for this work include the following:

- The use of deep learning and other methods to determine the features for detecting vulnerabilities instead of using hand-selected features. While the features proposed in this work are engineered to a degree, they provide 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 and provides a baseline on which to judge the performance of the methodology. Automatic extraction of features will allow for the addition of machine learning algorithms to the methodology.
- Expanding the work by detecting and correcting more vulnerabilities/weaknesses in the SANS/CWE 2011 list of Most Dangerous Software Errors. The analysis in this work showed that by correcting the two featured vulnerabilities, 1300 out of 1500 vulnerabilities in the 2017 NVD release could be avoided.
- Further improving the user interface based on the responses received from participants in the usability study
- Expanding the tool to support more programming languages and IDEs
- Collecting reports from users on their awareness about secure coding based on tool usage and tracking error reduction based on recommendations provided by the tool
- Performing A/B testing of the features within the proposed tool

Endnotes

[1] Data that is unchecked or unsanitized

[2] A backward-reasoning style proof theory for plan synthesis that considers an action that would achieve a goal under some specified circumstances and tries to find a way to achieve the goal by performing the action [84].

[3] A code analyzer that flags code based on programming errors, bugs, stylistic errors, and suspicious constructs

[4] IRB#: 18-006

[5] "None," "somewhat familiar," "familiar," "very familiar," "expert"

[6] IRB#:18-006

[7] "Strongly disagree," "disagree," "neutral," "agree," "strongly agree"

[8] "Strongly disagree": 1, "disagree" : 2, "neutral" : 3, "agree" : 4, "strongly agree" : 5

Appendix A: Appendices

A.1 Knowledge elicitation survey questions

1 What is your occupation? (a) Undergraduate student (b) Graduate Student (c) Professor (d) Industry Expert (e) Freelancer (f) Other

2 How would you describe your level of familiarity with the following programming languages? Use the Likert Scale: ("None", "Somewhat familiar", "Familiar", "Very familiar", "Expert") (1) Java (2) C# (3) Visual Basic (4) C (5)C++ (6)Python (7) JavaScript (8) PHP (9) Perl

3 Please indicate your familiarity with the following source code editors and/or integrated development environments (IDEs) using the same Likert scale for the question above. (1) Eclipse (2) Netbeans (3)IntelliJ IDEA (4) Visual Studio (5) Emacs (6) Vi/Vim (7) Other (please specify)

4 How important or unimportant is it for you to develop secure code? (a) Very important (b) Important (c) Unimportant (d) Very unimportant

5 How do you currently scan your code for weaknesses (vulnerabilities) or unsafe practices? (a) I use a static analyzer such as FindBugs (b) I use a dynamic analyzer such as Java PathFinder (c) I use both static and dynamic analyzers (d) I write code and another

party scans it for vulnerabilities (e) I currently do not scan my code for vulnerabilities (f) Other

6 If you answered (a) - (c) in question 5, how helpful do you find the warnings/advice provided by the selected scanner? (a) The advice is very helpful in fixing vulnerabilities (b) The advice given is somewhat helpful in fixing vulnerabilities (c) The advice provided does not help in fixing vulnerabilities

7 Do you currently utilize IntelliSense technology (also known as "code completion" or "code hinting" during coding? (a) Yes (b) No

8 What is your opinion about detecting vulnerabilities using IntelliSense technology? (a)I would appreciate a system that can scan my code for vulnerabilities as I code (b) I do not care about such technology, but I believe other programmers would appreciate a system that utilizes this technology (c) I do not think this would be a good idea

We will now show you a mockup (See Fig. 3) of a tool we are developing that is designed to help programmers find and fix vulnerabilities as they code. The tool will be created using machine learning techniques and implemented as a plugin in common IDEs such as Eclipse.

9 Consider situations where you are writing code. In what situations would you utilize this plugin? (a) Before code release (b) During a nightly build (c) As I type code (d) When I finish a module (e) When I finish a class (f) Other

10 What do you like or dislike about the plugin featured in the mockup?

11 For what types of project would you use this plugin? (multiple responses can be selected) (a) work projects (b) school projects (c) While freelancing (d)Fun projects (e) Open-source projects (f) Other

A.2 Usability study questions

A.2.1 Pre-task completion questionnaire

1 What is your age group? (a) 18-29 years old (b) 30-49 years old (c) 50-64 years old (d) 65 years and over

2 What is your occupation?

3 Select your primary programming languages (1) Java (2) C# (3) Visual Basic (4) C (5)C++ (6)Python (7) JavaScript (8) PHP (9) Perl (10) Other

4 How many years of coding experience do you have? (a) 0-2 (b) 3-5 (c) 6-8 (d) 9-11 (e) 12-14 (f) 15-20 (g) Over 20 years

A.2.2 Post-task completion questionnaire

The following Likert scale was used for the following questions: "Strongly Disagree", "Disagree", "Neutral", "Agree", "Strongly Agree"

1 provided me with helpful information including examples on how to fix vulnerabilities

2 Overall, the ________ interface was usable

3 I think the ________ system will help me to write more secure code

4 I would use the ________ system when coding

5 I think using the ________ system will allow me to fix vulnerabilities faster than other tools

6 It is easier for me to fix vulnerabilities based on a deeper understanding of the vulnerabilities rather than examples of other fixes.

A.2.3 Post-task completion interview

1 Were you able to complete all the tasks given to you on the ________ system ? Why or why not?

2 What did you like about using the code analyzer on the ________ system?

3 What did you dislike about using the code analyzer on the ________ system?

Abbreviations

ANOVA: Analysis of variance; ANSI: American national standards institute; AST: Abstract syntax tree; CFG: Control-flow graph; CNA: CVE numbering authority; CVE: Common vulnerabilities and exposures; CVSS: Common vulnerability scoring system; CWE: Common weakness enumeration; DAST: Dynamic application security testing; DFG: Data-flow graph; HM: Hidden Markov models; IDE: Integrated development environment; ISL: Intermediate slice language; LLOC: Logical lines of code; NIST: National institute of standards and technology; NVD: National vulnerability database; OWASP: Open web application security project; RNN: Recurrent neural network; SMT: Satisfiability modulo theory; SQLI: Structured query language injection

Acknowledgements

The authors would like to thank the anonymous reviewers for their careful reading of our manuscript and their many insightful comments and suggestions that greatly contributed to improving the final version of the paper. They would also like to thank the Editors for their generous comments and support during the review process.

Funding

Publication of this article was funded in part by the Open Access Subvention Fund and the Florida Tech Libraries.

Availability of data and materials

The datasets used and/or analyzed during the current study are available from the corresponding author on reasonable request.

Authors' contributions

The majority of the work was done by the first author (FDN) as a part of the first author's Ph.D. dissertation. FDN organized the studies, designed the interview questions, designed and created the recommender system featured in the work, and undertook all participant meetings and interviews. FDN also performed all quantitative statistical analyses. Both MMC and TCE guided the first author in all aspects of the research and made suggestions that improved the clarity and neutrality of the interview questions as well as the overall quality of the manuscript. All authors read and approved the final manuscript.

Authors' information

Fitzroy D. Nembhard is currently working as a Post-Doctoral Researcher in the Harris Institute for Assured Information at Florida Institute of Technology. He received the Ph.D. in Computer Science from Florida Institute of Technology in 2018 under the advisorship of Marco M. Carvalho. He also received a MS in Bioinformatics and BS in Computer Science from Morgan State University in 2012 and 2009, respectively. Prior to completing his doctoral studies, Dr. Nembhard worked as an adjunct faculty in the Computer Science Department at Morgan State University. His interests include using Machine Learning and Data Mining to solve problems in Cyber Security and Bioinformatics (particularly in designing and improving sequence-based algorithms using

parallel and distributed methodologies), designing tools to improve software and information security, and designing visualizations for cyber specialists to solve critical problems in the cyber domain.
Marco M. Carvalho is a Professor at the Florida Institute of Technology (Florida Tech) and a Research Scholar/Scientist at the Institute for Human and Machine Cognition. At Florida Tech, Dr. Carvalho serves as the Dean of the College of Engineering and Sciences, Director of Research of the Harris Institute for Assured Information, and Director of the Intelligent Communications and Information Systems Laboratory. Dr. Carvalho graduated with a M.Sc. in Mechanical Engineering with specialization in dynamic systems and control theory from the University Brasilia (UnB–Brazil). He also holds a M.Sc. in Computer Science from the University of West Florida and a Ph.D. in Computer Science from Tulane University, with specialization in Machine Learning and Data Mining. Dr. Carvalho currently leads several research efforts in the areas of cyber security, moving target defense, critical infrastructure protection, and tactical communication systems, primarily sponsored by the Department of Defense, the U.S. Army Research Laboratory, the U.S. Air Force Research Laboratory, ONR, the National Science Foundation, DoE and Industry. He is the Principal Investigator of a DoD/AFRL sponsored project focused on Systems Behavior Approach to Moving Target Command and Control, and the Principal Investigator of an AFRL sponsored effort on Resilient Airborne Networks. Dr. Carvalho's research interests include resilient distributed systems, multi-agent systems and emergent approaches to systems optimization and security.
Thomas C. Eskridge is an Associate Professor of Information Assurance and Cybersecurity in the Harris Institute for Assured Information at the Florida Institute of Technology. Dr. Eskridge has a Ph.D. in Philosophy from Binghamton University and an MS and BS in Computer Science from Southern Illinois University. His research focuses on amplifying human performance through intelligent assistance and innovative visualizations, both of which require developing a deep understanding of operator goals and mental task models to represent, reason, and visually display. He is currently developing tools that enable software agents and human operators to collaboratively represent and reason about networks, user actions, and cyber security events. Previous projects include developing a hybrid connectionist-symbolic knowledge ărepresentation system to model human analogical reasoning, case-based reasoning systems supporting milling-machine operators, formal knowledge representation editors, distributed multi-agent systems, fixed-wing and rotary-wing cockpit displays, visualizations for cyber situation awareness, defense posture, and mission management.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note
Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 15 November 2018 Accepted: 8 May 2019
Published online: 13 June 2019

References

1. Ponemon Institute LLC, 2017 cost of data breach study. Ponemon Institute and IBM Security (2017). https://www.securityupdate.net/SU/IBMSecurity/IBM-Security-Cost-of-Data-Breach-Study.pdf, Accessed 30 May 2018
2. L. B. Othmane, G. Chehrazi, E. Bodden, P. Tsalovski, A. D. Brucker, Time for addressing software security issues: Prediction models and impacting factors. Data Sci. Eng. **2**(2), 107–124 (2017). https://doi.org/10.1007/s41019-016-0019-8
3. G. McGraw, *Software security: building security in, vol 1*. (Addison-Wesley Professional, Boston, 2006)
4. Tricentis, Software fail watch: 2016 in review (2017). https://tricentis-com-tricentis.netdna-ssl.com/wpcontent/uploads/2017/01/20161231SoftwareFails2016.pdf, Accessed 30 May 2018
5. B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, *Why don't software developers use static analysis tools to find bugs?* (IEEE Press, Piscataway, 2013), pp. 672–681. ICSE '13
6. T. Kremenek, K. Ashcraft, J. Yang, D. Engler, in *ACM SIGSOFT Software Engineering Notes*. Correlation exploitation in error ranking (ACM, New York, 2004), pp. 83–93. SIGSOFT '04/FSE-12, https://doi.org/10.1145/1029894.1029909
7. F. Ricci, L. Rokach, B. Shapira, *Introduction to Recommender Systems Handbook*. (Springer US, Boston, 2011), pp. 1–35
8. D. Evans, D. Larochelle, Improving security using extensible lightweight static analysis. IEEE Softw. **19**(1), 42–51 (2002). https://doi.org/10.1109/52.976940
9. N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, W. Pugh, Using static analysis to find bugs. IEEE Softw. **25**(5), 22–29 (2008). https://doi.org/10.1109/MS.2008.130
10. F. Nembhard, M. Carvalho, T. Eskridge, in *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*. A hybrid approach to improving program security, (2017), pp. 1–8. https://doi.org/10.1109/SSCI.2017.8285247
11. M. Alenezi, Y. Javed, Developer companion: A framework to produce secure web applications. Int. J. Comput. Sci. Inf. Secur. **14**(7), 12 (2016)
12. J. Bleier, *Improving the usefulness of alerts generated by automated static analysis tools*. (Radboud University Nijmegen, Master's thesis, 2017)
13. K. A. Farris, A. Shah, G. Cybenko, R. Ganesan, S. Jajodia, VULCON: A system for vulnerability prioritization, mitigation, and management. ACM Trans. Priv. Secur. **21**(4), 16:1–16:28 (2018). https://doi.org/10.1145/3196884
14. Tenable, Nessus professional (2018). https://www.tenable.com/products/nessus/nessus-professional, Accessed 15 Feb 2018
15. R. Gopalakrishnan, P. Sharma, M. Mirakhorli, M. Galster, in *Proceedings of the 39th International Conference on Software Engineering*. Can latent topics in source code predict missing architectural tactics? (IEEE Press, Piscataway, 2017), pp. 15–26. ICSE '17, https://doi.org/10.1109/ICSE.2017.10
16. I. Medeiros, N. Neves, M. Correia, in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. DEKANT: a static analysis tool that learns to detect web application vulnerabilities (ACM, 2016), pp. 1–11. https://doi.org/10.1145/2931037.2931041
17. S. M. Ghaffarian, H. R. Shahriari, Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Comput. Surv. **50**(4), 56:1–56:36 (2017). https://doi.org/10.1145/3092566
18. F. Yamaguchi, M. Lottmann, K. Rieck, in *Proceedings of the 28th Annual Computer Security Applications Conference*. Generalized vulnerability extrapolation using abstract syntax trees (ACM, New York, 2012), pp. 359–368. ACSAC '12, https://doi.org/10.1145/2420950.2421003
19. L. K. Shar, H. B. K. Tan, Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns. Inf. Softw. Technol. **55**(10), 1767–1780 (2013). https://doi.org/10.1016/j.infsof.2013.04.002
20. L. K. Shar, H. B. K. Tan, in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. Predicting common web application vulnerabilities from input validation and sanitization code patterns (IEEE, 2012), pp. 310–313. https://doi.org/10.1145/2351676.2351733
21. L. K. Shar, H. B. K. Tan, L. C. Briand, in *Proceedings of the 2013 International Conference on Software Engineering*. Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis (IEEE Press Piscataway, NJ, USA, 2013), pp. 642–651
22. L. K. Shar, L. C. Briand, H. B. K. Tan, Web application vulnerability prediction using hybrid program analysis and machine learning. IEEE Trans. Dependable Secure Comput. **12**(6), 688–707 (2015). https://doi.org/10.1109/TDSC.2014.2373377
23. F. Nembhard, *A recommender system for improving program security through source code mining and knowledge extraction*. (Florida Institute of Technology, PhD thesis, 2018)
24. M. Curphey, R. Arawo, Web application security assessment tools. IEEE Secur. Priv. **4**(4), 32–41 (2006). https://doi.org/10.1109/MSP.2006.108
25. OWASP, Category:vulnerability scanning tools (2017). https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools, Accessed 07 July 2017
26. Y. W. Huang, C. H. Tsai, T. P. Lin, S. K. Huang, D. Lee, S. Y. Kuo, A testing framework for web application security assessment. Comput. Netw. **48**(5), 739–761 (2005). https://doi.org/10.1016/j.comnet.2005.01.003
27. J. Bau, E. Bursztein, D. Gupta, J. Mitchell, in *Security and Privacy (SP) year=2010 IEEE Symposium on*. State of the art: Automated black-box web application vulnerability testing (IEEE, 2010), pp. 332–345. https://doi.org/10.1109/SP.2010.27
28. A. Petukhov, D. Kozlov, *Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing*. (Department of Computer Science, Moscow State University, 2008)

29. J. P. Jonkergouw, *Effectiveness of automated security analysis using a uniface-like architecture*. (Master's thesis, Universiteit van Amsterdam, 2014)
30. U. Kuter, M. H. Burstein, J. Benton, D. Bryce, J. T. Thayer, S. McCoy, in *AAAI*. HACKAR: Helpful advice for code knowledge and attack resilience (Twenty-Seventh IAAI Conference, Austin, 2015), pp. 3987–3992
31. R. Reiter, The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. Artif. Intell. Math. Theory Comput. Papers Honor John McCarthy. **27**, 359–380 (1991). Academic Press Professional, Inc. San Diego, CA, USA, https://doi.org/10.1016/B978-0-12-450010-5.50026-8
32. J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, et al., in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. Automatically patching errors in deployed software (ACM, 2009), pp. 87–102. https://doi.org/10.1145/1629575.1629585
33. C. Le Goues, T. Nguyen, S. Forrest, W. Weimer, Genprog: A generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1), 54 (2012). https://doi.org/10.1109/TSE.2011.104
34. W. Weimer, Z. P. Fry, S. Forrest, in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Leveraging program equivalence for adaptive program repair: Models and first results, (2013), pp. 356–366. https://doi.org/10.1109/ASE.2013.6693094
35. F. Long, M. Rinard, in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. Staged program repair with condition synthesis (ACM, New York, 2015), pp. 166–178. ESEC/FSE, 2015
36. Z. Qi, F. Long, S. Achour, M. Rinard, in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems (ACM New York, NY, USA, 2015), pp. 24–36
37. F. Long, M. Rinard, Automatic patch generation by learning correct code. SIGPLAN Not. **51**(1), 298–312 (2016). https://doi.org/10.1145/2837614.2837617
38. D. Kim, J. Nam, J. Song, S. Kim, in *Proceedings of the 2013 International Conference on Software Engineering*. Automatic patch generation learned from human-written patches (IEEE Press Piscataway, NJ, USA, 2013), pp. 802–811
39. Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, A. Zeller, in *Proceedings of the 19th International Symposium on Software Testing and Analysis*. Automated fixing of programs with contracts (ACM, New York, 2010), pp. 61–72. ISSTA '10, https://doi.org/10.1145/1831708.1831716
40. V. Debroy, W. E. Wong, in *2010 Third International Conference on Software Testing, Verification and Validation*. Using mutation to automatically suggest fixes for faulty programs, (2010), pp. 65–74. https://doi.org/10.1109/ICST.2010.66
41. H. D. T. Nguyen, D. Qi, A. Roychoudhury, S. Chandra, in *2013 35th International Conference on Software Engineering (ICSE)*. Semfix: Program repair via semantic analysis (IEEE, 2013), pp. 772–781. https://doi.org/10.1109/ICSE.2013.6606623
42. G. Jin, L. Song, W. Zhang, S. Lu, B. Liblit, in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. Automated atomicity-violation fixing (ACM, New York, 2011), pp. 389–400. PLDI '11, https://doi.org/10.1145/1993498.1993544
43. D. J. Musliner, J. M. Rye, D. Thomsen, D. D. McDonald, M. H. Burstein, P. Robertson, in *2011 Fifth IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops*. Fuzzbuster: Towards adaptive immunity from cyber threats, (IEEE, 2011), pp. 137–140. https://doi.org/10.1109/SASOW.2011.26
44. D. J. Musliner, S. E. Friedman, M. Boldt, J. Benton, M. Schuchard, P. Keller, S. McCamant, in *Fourth International Conference on Communications, Computation, Networks and Technologies (INNOV)*. Fuzzbomb: Autonomous cyber vulnerability detection and repair (Fourth International Conference on Communications, University of Bonn, Bonn, 2015), p. 2015
45. V. Raychev, M. Vechev, A. Krause, in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Predicting program properties from "big code" (ACM, New York, 2015), pp. 111–124. POPL '15, https://doi.org/10.1145/2676726.2677009
46. R. Gupta, S. Pal, A. Kanade, S. Shevade, in *Thirty-First AAAI Conference on Artificial Intelligence*. Deepfix: Fixing common c language errors by deep learning (Thirty-First AAAI Conference on Artificial Intelligence, San Francisco, 2017)
47. SonarSource, Get the power to write better code (2019). https://www.sonarlint.org/features/, Accessed 15 Feb 2019
48. J. Xie, B. Chu, H. R. Lipford, J. T. Melton, in *Proceedings of the 27th Annual Computer Security Applications Conference*. Aside: IDE support for web application security (ACM, New York, 2011), pp. 267–276. ACSAC '11, https://doi.org/10.1145/2076732.2076770
49. Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, T. Xie, in *Proceedings of the 28th Annual Computer Security Applications Conference*. Xiao: tuning code clones at hands of engineers in practice (ACM, 2012), pp. 369–378. https://doi.org/10.1145/2420950.2421004
50. S. Micheelsen, B. Thalmann, *A static analysis tool for detecting security vulnerabilities in python web applications*. (Aalborg University, Master's thesis, 2016)
51. A. Z. Baset, T. Denning, in *Security and Privacy Workshops (SPW) 2017 IEEE*. IDE plugins for detecting input-validation vulnerabilities (IEEE, 2017), pp. 143–146. https://doi.org/10.1109/SPW.2017.37
52. V. Raychev, M. Vechev, E. Yahav, in *Acm Sigplan Notices, vol 49*. Code completion with statistical language models (ACM, 2014), pp. 419–428. https://doi.org/10.1145/2594291.2594321
53. C. Omar, Y. Yoon, T. D. LaToza, B. A. Myers, in *Proceedings of the 34th International Conference on Software Engineering*. Active code completion (IEEE Press Piscataway, NJ, USA, 2012), pp. 859–869
54. S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. **86**(8), 1978–2001 (2013). https://doi.org/10.1016/j.jss.2013.02.061
55. A. Z. Baset, T. Denning, in *Security and Privacy (SP) 2017 IEEE Symposium on*. Ide plugins for detecting input-validation vulnerabilities, (2017), pp. 143–146. https://doi.org/10.1109/SPW.2017.37
56. The MITRE Corporation, Common vulnerabilities and exposures (CVE) (2017). https://cve.mitre.org/about/, Accessed 29 Dec 2017
57. S.tandards.a.nd.T.echnology.(.NIST). National Institute for, National vulnerability database (2017). https://nvd.nist.gov/home, Accessed 12 Dec 2017
58. S. Bajracharya, J. Ossher, C. Lopes, Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. Sci. Comput. Prog. **79**, 241–259 (2014). https://doi.org/10.1016/j.scico.2012.04.008
59. RogueWave Software, Abstract syntax tree (AST) (2018). https://docs.roguewave.com/en/klocwork/current/abstractsyntaxtreeast, Accessed 25 Jan 2018
60. N. Smith, D. van Bruggen, F. Tomassetti, *JavaParser: Visited; Analyse, transform and generate your Java code base*. (Leanpub, British Columbia, 2017)
61. J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters. Commun. ACM. **51**(1), 107–113 (2008). https://doi.org/10.1145/1327452.1327492
62. A. B. Patel, M. Birla, U. Nair, in *2012 Nirma University International Conference on Engineering (NUiCONE)*. Addressing big data problem using hadoop and map reduce, (2012), pp. 1–5. https://doi.org/10.1109/NUICONE.2012.6493198
63. G. Buehrer, B. W. Weide, P. A. G. Sivilotti, in *Proceedings of the 5th International Workshop on Software Engineering and Middleware*. Using parse tree validation to prevent SQL injection attacks (ACM, New York, 2005), pp. 106–113. SEM '05, https://doi.org/10.1145/1108473.1108496
64. J. Williams, D. Wichers. OWASP top 10-2010 the ten most critical web application security risks. Tech. rep. (OWASP, Bel Air, 2010). https://www.owasp.org/images/0/0f/OWASP_T10_-_2010_rc1.pdf
65. J. Williams, D. Wichers, *The ten most critical web application security risks*. (OWASP Foundation, Bel Air, 2017)
66. MITRE, 2011 CWE/SANS top 25 most dangerous software errors (2011). http://cwe.mitre.org/top25/, Accessed 06 Dec 2017
67. B. Livshits, *Improving software security with precise static and runtime analysis. PhD thesis*. (Stanford University, Stanford, 2006)
68. MITRE, CWE-89: Improper neutralization of special elements used in an SQL command ('SQL injection') (2018). http://cwe.mitre.org/top25/, Accessed 8 Feb 2018
69. B. Flood, Find-sec-bugs injection sinks (2017). https://github.com/find-sec-bugs/find-sec-bugs/tree/master/findsecbugs-plugin/src/main/resources/injection-sinks, Accessed 8 Feb 2018

70. L. Sampaio, Which methods should be considered "sources", "sinks" or "sanitization"? (2014). http://thecodemaster.net/methods-considered-sources-sinks-sanitization/, Accessed 8 Feb 2018
71. OWASP, Searching for code in J2EE/Java (2016a). https://www.owasp.org/index.php/Searching_for_Code_in_J2EE/Java, accessed: 8 Feb 2018
72. OWASP, Command injection (2016b). https://www.owasp.org/index.php/Command_Injection, Accessed 24 Feb 2018
73. M.IP. Center, Windows commands (2016). https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/windows-commands, Accessed 6 Mar 2018
74. A. Strauss, J. Corbin, *Basics of qualitative research techniques 2nd Edition*. (SAGE publications Thousand Oaks CA, Thousand Oaks, 1998)
75. F.oundation. The Eclipse, Code recommenders: The intelligent development environment (2017). http://www.eclipse.org/recommenders/, Accessed 15 Nov 2017
76. The Apache Software Foundation, Welcome to apache lucene (2018). https://lucene.apache.org/, Accessed 9 Feb 2018
77. M. Sadegh Riazi, B. Chen, A. Shrivastava, Wallach D., Koushanfar F., *Sub-linear privacy-preserving near-neighbor search with untrusted server on large-scale datasets. arXiv preprint*. (Cornell University, Hurston Ave., Ithaca, 2016). arXiv:1612.01835, 2016
78. N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, Y. Zhou, in *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. Using FindBugs on production software (ACM, New York, 2007), pp. 805–806. OOPSLA '07, https://doi.org/10.1145/1297846.1297897
79. A. Vetro, A. Morisio, A. Torchiano, An empirical validation of findbugs issues related to defects. IET Conf. Proc. **9**, 144–153 (2011). https://doi.org/10.1049/ic.2011.0018
80. N. Ayewah, W. Pugh, in *Proceedings of the 19th International Symposium on Software Testing and Analysis*. The google FindBugs fixit (ACM, New York, 2010), pp. 241–252. ISSTA '10, https://doi.org/10.1145/1831708.1831738
81. M. Dhruv, Ids00-j. prevent sql injection (2017). https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection, Accessed 15 Nov 2017
82. OWASP, Command injection in java (2017). https://www.owasp.org/index.php/Command_injection_in_Java, accessed: 15 Nov 2017
83. R. L. Mack, J. Nielsen, *Usability inspection methods*. (Wiley, New York, 1994)
84. J. L. Pollock, The logical foundations of goal-regression planning in autonomous agents. Artif. Intell. **106**(2), 267–334 (1998). https://doi.org/10.1016/S0004-3702(98)00100-3