A Hybrid Approach to Improving Program Security

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

***Abstract*—The security of computer programs and systems is a very critical issue. With the number of attacks launched on computer networks and software, businesses and IT professionals are taking steps to ensure that their information systems are as secure as possible. However, many programmers do not think about adding security to their programs until their projects are near completion. This is a major mistake because a system is as secure as its weakest link. If security is viewed as an afterthought, it is highly likely that the resulting system will have a large number of vulnerabilities, which could be exploited by attackers. One of the reasons programmers overlook adding security to their code is because it is viewed as a complicated or time-consuming process. This paper presents a tool that will help programmers think more about security and add security tactics to their code with ease. We created a model that learns from existing open source projects and documentation using machine learning and text mining techniques. Our tool contains a module that runs in the background to analyze code as the programmer types and offers suggestions of where security could be included. In addition, our tool fetches existing open source implementations of cryptographic algorithms and sample code from repositories to aid programmers in adding security easily to their projects.**

***Index Terms*—Cybersecurity, secure, software, programs**

I. Introduction

The cyber-related challenges that plague modern computer systems require innovative solutions. Techniques in machine learning and text mining can be used to mitigate many of the attacks that are launched on computer networks and systems. Recently, researchers in software engineering have created models based on supervised machine learning algorithms to address the problem of architectural degradation through keeping developers informed of underlying architectural decisions [31]. While these general approaches to software engineering can help programmers follow certain design patterns, they do not emphasize the importance of writing secure code. To date, we have found no specific tool that makes it easy for programmers to incorporate security into their projects from the initial stages of software design to software maintenance. Unfortunately, there is a huge deficit of and demand for tools that offer simple security to the tech community. Emphasis is often placed more on productivity than on security [29]. For example, auto-complete features and predictive typing have become ubiquitous in smart devices and modern integrated development environments(IDEs) to help users become more productive in less time. Additionally, programmers enjoy reusing code and obtaining support for their projects. It is reported that in 2016, every 8 seconds, a developer asks a question on Stack Overflow [2]. Still, many programmers ignore security in order to deliver a program that solves a problem and meets a deadline.

To address the problem by encouraging programmers to write more secure code, we present a solution that detects the presence of insecure program code, reminds programmers when they forget to include important security modules in their code (or when a cryptographic package is deprecated), offers recommendations based on common practices in the security field, and provides snippets of code to make it easier for programmers to implement a secure functionality.

This paper is organized as follows: in Section II, we discuss related work in the area of code analysis and in Section III, we discuss our approach. We present a case study in Section IV followed by our conclusion in Section V.

II. Related Work

Code analysis tools fall into three categories: Static Application Security Testing (SAST) or static analysis tools, Dynamic Application Security Testing (DAST) or dynamic analysis tools, and Interactive Application Security Testing (IAST) tools.

A. Static Application Security Testing (SAST) Tools

Static analysis tools are designed to analyze source code and/or compiled versions of code to help find security flaws [35]. The most common tools [7], [14] that detect flaws in program code are based on static analysis. These tools attempt to locate vulnerabilities such as buffer overflows, memory-access vulnerabilities, resource leaks, and format bugs. Splint is a heuristic-based tool that finds potential vulnerabilities by checking to see that source code is consistent with the properties implied by annotations [14]. Unfortunately, Splint is limited to ANSI C code and does not offer the functionalities required in modern development environments.

FindBugs[1] is another static analysis tool, which supports the Java programming language and is used for finding coding defects [7]. It was designed to effectively identify low-hanging fruit, which involves cheaply detecting defects that developers will want to review and remedy [7]. FindBugs does not provide support for mitigation of vulnerabilities [8]. FindSecBugs[2] is a FindBugs plugin, which is geared towards security audits of Java web applications. As of July 7, 2017, it is reported that FindSecBugs can detect 113 different vulnerability types with over 689 unique API signatures [6].

[1]http://findbugs.sourceforge.net/
[2]https://find-sec-bugs.github.io/

Alenezi and Javed proposed a framework, called Developer Companion, to help developers produce secure web applications [4]. Their idea is to enable developers and testers to find security problems in web applications during implementation. While code is being written, the proposed framework uses several static analysis tools to analyze the code. The code is then cross-referenced against the Common Weakness Enumeration (CWE) and National Vulnerability Database (NVD), and eventually a recommendation based on the collected data is given to the developer [4].

Archie is an open-source plugin that helps to automate the creation and maintenance of architecturally-relevant trace links between code, architectural decisions, and related requirements [32]. It has been used to find architectural tactics such as heartbeat, resource pooling, and role-based access control [32].

Some of the weaknesses of common static code analysis tools include excessive false positives [11], [24], [35] and inability to detect authentication problems, access control issues, and insecure use of cryptography [35]. Additionally, static analysis tools often report lengthy and numerous alerts, which causes a lack of acceptance by developers and businesses [11], [21]. Thus, [11] proposed a framework, called Autobugs, for improving the alerts reported by common static analysis tools. Autobugs gathers historic alert data from static analysis tools and combines the alert-data with complexity metrics to build a classifier that predicts the actionability[3] of an alert from data and unit properties [11]. Unfortunately, the author reported that models based on historic alert data could potentially mislead developers to believe they have no problems [11]. Compounding these weaknesses is the understanding that different tools have different objectives such as speed at returning results (sacrificing quality), deep semantic analysis of code (sacrificing speed), focus on one class of weaknesses (limitations) [8], and installation and configuration of tools (time [25].

B. Dynamic Application Security Testing (DAST) tools

DAST tools are also known as dynamic analyzers and are used to test an application or software product in an operating state [41]. A plethora of tools [13], [34] exist in this domain, the majority of which are commercial. Interestingly, a great deal of focus in DAST is devoted to web applications [9], [19], [36]. Huang et al proposed a crawler, which allows for a black-box, dynamic analysis of web applications [19]. 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 [19]. In addition, Petukhov et al proposed an extended tainted mode model that incorporates the advantages of penetration testing and dynamic analysis to widen the scope of the web application being covered during testing [36].

Since dynamic analysis involves testing application behavior, some researchers believe it is a more realistic approach than static analysis [22]. However, the main challenge with dynamic tools is identifying the source of a bug [24]. Bugs often manifest themselves as program crashes and this makes them difficult to mitigate. Because the goal of this work is to encourage programmers to write more secure code, and dynamic analysis requires executable code, this type of program analysis is not applicable to this work.

C. Interactive Application Security Testing (IAST) tools

Pioneered by Quotium in 2011, IAST is an emerging form of application security which enables a fully automatic code analysis that ensures no code vulnerabilities are introduced during development [3], [37]. Consisting of active and passive approaches, IAST involves monitoring code in memory during execution in order to find specific events that could cause vulnerabilities such as database queries, file system access, web service calls, and input validations [3], [43]. Though the term ”interactive” is used in its nomenclature, pioneers of IAST tout that it does not require any manual work or customization [3], which makes it unclear to developers how this class of tools are expected to work. Fortify PTA (Program Trace Analyzer) is an IAST tool, which was offered by HP [38], but was made obsolete in 2012 [18]. Aspect Security also released the Contrast Interactive Application Security Testing (IAST) tool, which finds vulnerabilities in custom code as well as libraries [39]. While Contrast is able to locate vulnerabilities such as unsafe encryption algorithms and weak data validation [39], its purpose is not to encourage programmers to write secure code.

Comparison Between Our Hybrid Approach and Existing Approaches

Table I shows a comparison of popular tools[4] in the area of static, dynamic, and hybrid code analysis. While $\approx 45\%$ of the tools are proprietary, thus limiting disclosure of their design techniques, it can be seen that none of the existing tools utilizes the machine learning and text mining techniques we employ in our approach (See Section III for a thorough discussion on our approach). However, some tools [15] utilize machine learning as a technique to improve the actionability of alerts generated by some popular tools, but they are not predictive or mitigative and do not learn from the program code to detect missing security modules. In contrast to these tools, our approach uses machine learning and text mining techniques to determine when code is missing secure modules. While we utilize collected data, this is done in the model-building phase of our project, and reliance on databases is done only for checking for deprecation. Further, our tool does not rely only on vulnerabilities present in the code to make recommendations; instead, using a predictive, classification-based model, our tool is intended to help programmers think about security from the initial stages of development to project maintenance. With the added ability to offer code snippets

[3]An alert is actionable if a developer deems it important and the corresponding anomaly fixable [15]

[4]We consider a tool popular based on its coverage in the literature [11], [15], [28], [33], [35], [42] and customer satisfaction (based on user reviews) and scale (based on market share, vendor size, and social impact) [16], [20]

or connectivity to existing secure implementation of certain cryptographic modules, our tool helps to remedy the problem of detecting misuse or deprecation of cryptographic packages. Additionally, our tool is not limited to web applications but to all types of projects implemented in Java. Our classification model can also be trained using data from other languages to provide more comprehensive support. This requires crawling data related to other languages and rebuilding the classification model, which will be done in a future update to our tool.

III. Approach

Analyzing code for vulnerabilities is challenging due to the size of projects, language constructs and the unique ways that modules can be implemented. While there has been much success in static analysis over the past two decades, current tools like FindBugs are not specifically geared toward program security. That is, these tools provide general static analysis in order to find common flaws in program code. Using machine learning techniques, we have built a tool that is geared specifically toward program security. We used supervised machine learning to train a classifier to recognize the absence/presence of secure modules in program code and make recommendations accordingly. It has been shown in the literature that programmers tend to use informative keywords to name variables, methods, and classes, in addition to comments that describe the purpose of their code [5], [12], [31]. Our classifier learns from the keywords in hundreds of open-source projects and documentation. Using the learned model, our tool runs in the background while the programmer is writing code to detect missing security modules or deprecated libraries and offers smart suggestions, so the programmer can take steps to implement more secure code. For the remainder of this section, we will describe the techniques used to create our hybrid system.

A. Term Frequency/Inverse Document Frequency (TF/IDF)

Term Frequency/Inverse Document Frequency is a statistical measure that is intended to reflect how important a word is to a document in a collection or corpus. Term frequency measures how frequently a term occurs in a document. Inverse document frequency measures the importance of a term. Using TF/IDF, we can discover important security terms that can help us create a recommender system. [10] reports that TF/IDF is used in 83% of text-based recommender systems in the digital library domain.

B. Topic Modeling

A topic model is a generative model for documents, which specifies a probabilistic procedure by which documents can be generated [26]. That is, one can create a document based on a random selection of words from a certain topic. Inversely, statistical techniques can be used to infer the set of topics that were responsible for generating a collection of documents [26]. Using this technique, we categorized source code into different groups in order to build a model to recommend

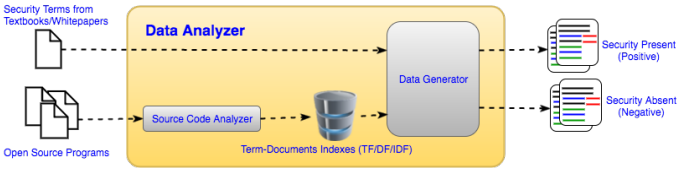

Fig. 1. Data Analyzer Model

security packages that may be missing from a project. We used MALLET[5] for topic modeling.

C. Source Code Analysis

Source code analysis is important in helping us understand what is implemented in a piece of code. In order to focus our efforts on more involved areas of our work, we utilized the code analysis features of FindBugs by merging portions of it with our classification model in order to create a versatile recommendation system. Section II provides more information on the FindBugs tool.

D. Parallel Data Crawling

Due to the volume of data required to build an efficient model for a recommender system, we used Hadoop[6] to crawl repositories. We secured access to the San Diego Supercomputer by way of XSEDE[7], thus enabling us to do speedy and efficient data crawling.

E. Data Analyzer

To create the Data Analyzer, we crawled program code, whitepapers, security textbooks, and PDFs to generate security terms that are used to determine the presence or absence of security terms. Figure 1 shows a flow diagram of the data analyzer.

Frequent itemset[8] mining was used to generate positive(security present) examples. The code/documents missing frequent security itemsets were used to generate the negative examples. For illustrative purposes, Table II shows example terms that indicate the presence of a secure module and the corresponding vulnerability that would result from the absence of these modules.

F. Recommender System

The recommender system incorporates the model learned during classification. It contains a *Repository Tracker* that runs in the background and verifies links to existing cryptographic/security implementations, a tool to provide suggestions in the form of a tooltip/callout with links to code that users can download (See Figure 4), and a mechanism to offer suggestions if code is missing common security modules (based on keywords gleaned from the programmer's project

[5]MAchine Learning for LanguagE Toolkit (http://mallet.cs.umass.edu/topics.php)

[6]http://hadoop.apache.org/

[7]The Extreme Science and Engineering Discovery Environment (http://xsede.org)

[8]Terms that appear together frequently such as "network" and "security"

TABLE I
COMPARISON OF POPULAR TOOLS IN THE AREA OF CODE ANALYSIS [11], [15], [16], [28], [33], [35], [42]

						Static										Dynamic	Hybrid		
						Source Code Analyzer													
	Supplier	Product Name	Languages/Application Supported	License	Design Technique (Method)	Attack modeling	Warning Flags	Source Code Quality Analyzer	Source Code Weakness Analyzer	Context-configured source code weakness analyzer	Knowledge Extractor	Binary/bytecode analysis	Secure Platform selection	Traditional Virus/Spyware Scanner		Test Coverage Analyzer	Hardening Tool	Execute & Compare with manifest
1	Microsoft	SDL Threat Modeling Tool	.	Free/Closed source	Data flow diagrams (DFDs)	●													
2	Led by SINTEF	Seamonster		Free/Closed source	Graphical Modeling	●													
3	GNU	gcc options -Wall, -Wextra, -ftrapv	C, C++, Objective-C, Java, Ada	Free/Closed source	Push Down Automata		●												
4	Microsoft	Visual C++ /W4	C (89 only), C++, and C++/CLI	Free/Closed source	Push Down Automata		●												
5	Perl	perl -w, use warnings "all"	Perl	Free/Closed source	Turing-complete grammar		●												
6	Coverity	Coverity Quality Advisor, Coverity Save	C, C++, Java; C# in progress for advisor	Commercial	Interprocedural dataflow analysis, Boolean satisfiability solvers, Design pattern intelligence, Additional patented analysis techniques			●	●										
7	Klocwork	Truepath® Refactoring®	Truepath: C/C++, Java and C#; Refactoring: C/C++	Commercial	Syntactic pattern matching, Data flow analysis			●							●				
8	Sonar	SonarSource	C, C++, Java, C#, VB.net, PL/1, COBOL, PHP, Python, VB6, Natural, Javascript, XML, etc; (caveat: some languages require Commercial license)	LGPL V3	Stored configuration, historical analysis, hybridization (PMD, FindBugs, etc)			●								●			
9	CAST	Application Intelligence Platform	Includes .NET, Java, COBOL	Free Trial/Commercial	Logic modeling combined with rules based on CWE, CISQ, etc			●											
10	Grammatech	CodeSonar	C, C++	Free Trial/Commercial	Proprietary methods				●			●							
11	HP Fortify	Static Code Analyzer (SCA)	C, C++, C# and other .NET languages, COBOL, Java, JavaScript/ AJAX, PHP, PL/SQL, Python, T-SQL	Free Trial/Commercial	Proprietary methods				●										
12	Checkmarx	Checkmarx	Java, C#/.NET, PHP, C, C++, Visual Basic 6.0, VB.NET, Flash, APEX, Ruby, JavaScript, ASP, Android, Objective C, Perl	Free Trial/Commercial	Proprietary methods				●										
13	IBM (formerly Ounce Labs)	AppScan Source	C, C++, Java, VB.NET, C#	Free Trial/Commercial	Proprietary methods				●										
14	IBM	Rational Asset Analyst	.	Free Trial/Commercial	Proprietary methods						●								
15	Micro Focus	Enterprise Analyzer	.	Free Trial/Commercial	Proprietary methods						●								
16	McAfee	McAfee AntiVirus Plus	.	Commercial	Proprietary methods									●					
17	Symantec	Norton Anti- virus	.	Commercial	Proprietary methods									●					
18	Trend Micro	Titanium Antivirus Plus	.	Free Trial/Commercial	Proprietary methods									●					
19	FindBugs project	FindBugs	Java, Groovy, Scala	LGPL	Syntactic pattern matching							●							
20	FindSecurityBugs project	FindSecurityBu gs (plug-in to FindBugs)	Java, Groovy, Scala	LGPL	Syntactic pattern matching							●							
21	Mono Project	Gendarme	.NET	Free	Rule-based							●							
22	PMD	PMD	Java, JavaScript, Salesforce.com Apex and Visualforce, PLSQL, Apache Velocity, XML, XSL	BSD-style	Syntactic pattern matching							●							
23	.	Checkstyle	.	GPL	Syntactic pattern matching, Data flow analysis			●											
24	.	JLint	Java	GPL	Syntactic pattern matching, Data flow analysis			●											
25	KindSoftware	ESC/Java2	Java	Free/Open source	Theorem proving							●							
26	.	Lint4j	Java	Free/Open source	Data flow analysis, lock graph							●							
27	VeraCode	VeraCode Static Analysis	Java, .NET	Commercial	Proprietary methods							●							
28	Galois	FUSE	Android	Public Release	Call graph, lint techniques							●							
29	.	Lintent	Android	Open Source/GitHub	Type-based analysis							●							
30	ViaForensics	ViaLab (portion)	.	Subscription	Proprietary methods							●							
31	Hex-Rays	IDA Pro	.	Free/Open Source	Disassembly algorithms							●			●				
32	Apache	Shiro	Java	Free/Open Source	Object Graph								●						
33	OWASP	OWASP Enterprise Security API	Java; many others in various stages of maturity	BSD license	Security control references								●						
34	Nmap.org	nmap	.	GPLv2.0	Port scanning techniques										●				
35	Gibson Research Corporation (GRC)	Shields up (online port scanning service)	.	Commercial	Application-level filtering and response										●				
36	Colasoft	Capsa Network Analyzer	.	Free Trial/Commercial	Packet sniffing and other proprietary methods										●				
37	Massimiliano Montoro	Cain and Abel (password recovery tool for Microsoft operating systems, includes sniffing)	.	Free/Closed Source	Packet sniffing, rainbow tables, and other proprietary methods										●				
38	Dug Song	dSniff	.	3-clause BSD	Third-party package references										●				
39	Telerik	Fiddler	.	Freeware	Event-based scripting, Third-party package references										●				
40	Microsoft	Microsoft Message Analyzer	.	Proprietary	Architecture modeling, Decorated syntax tree, compiled protocol descriptions, and other proprietary methods										●				
41	Sky software	SkyGrabber; LanGrabber	.	Shareware	Proprietary methods										●				
42	Oracle	snoop (in Solaris)	.	Commercial	Proprietary methods										●				
43	Tcpdump (project)	tcpdump	.	BSD	Promiscuous packet capturing mehods										●				
44	Wireshark (project)	Wireshark (formerly known as Ethereal).	.	GPL	Protocol tree, dissectors, display filters, library references										●				
45	Tenable Network Security	Nessus	.	Free Trial/Commercial	Port scanning techniques, other proprietary methods										●				
46	OpenVAS project	OpenVAS	.	GPL	Rule-based										●				
47	Rapid7	Nexpose	.	Free Trial/Commercial	Exploit modeling, other proprietary methods										●				
48	Rapid7	Metasploit	.	Open Source	Exploit modeling										●				
49	IBM	IBM Security AppScan Standard	.	Commercial	Proprietary methods										●				
50	.	Wapiti	.	GPLv2.0	Black-box scanning and fuzzing tehniques										●				
51	W3af project	W3af	.	GPLv2.0	Black-box web application scanning techniques, plugin references										●				
52	PortSwigger Ltd	Burp Suite	.	Free Trial/Commercial	Attack surface mapping, tool integration, proprietary techniques										●				
53	White Hat Security	SecurityCheck	.	Free Trial/Commercial	Proprietary methods										●				
54	Cisco	Intrusion Prevention System (IPS)	Network, host based intrusion prevention solutions â€" integrated into FW or separate	Commercial	Inline, deep-packet inspection techniques, and other proprietary techniques										●				
55	Cisco (formerly Sourcefire)	SNORT	free and open source network intrusion prevention system	GPLv2+Commercial	Rule-based, third party tool references										●				
56	FireEye	Virtual execution engine	Selected document formats on Windows	Commercial	Proprietary methods										●				
57	Appthority	App risk management service	Android, iOS	Commercial	Proprietary methods										●				
58	Veracode	Veracode App Intelligence (vAI) (partial)	Android, iOS	Free Trial/Commercial	Proprietary methods										●				
59	GNU	gcov	C, C++, Ada, Fortran	GPL	Syntactic pattern matching											●			
60	EclEmma team	JaCoCo	Java bytecode	EPL	Byte code instrumentation							●				●			
61	Atlassian	Clover	Java	Apache 2.0	Source code instrumentation											●			
62	McAfee	Vulnerability Manager	.	Commercial	Proprietary methods												●		
63	Syhunt	Syhunt Apache/PHP Hardener	.	Free Trial/Commercial	Proprietary methods												●	●	
64	Open Source	Permlyzer	.	Free/ Closed Source	Method profiling, search tree-based algorithm													●	
65	Kryptowire	Kryptowire (Android edition)	.	Commercial	Proprietary methods						●				●				
66	.	TaintDroid	Android	Free/Closed Source	Dynamic taint analysis via mix of variable-level, message-level and method-level tracking,							●			●				
67	Berkeley	TaintEraser	Android	Free/Closed Source	Dynamic taint analysis using dynamic binary translation coupled with application-level, object-level, and byte-level taint propagation							●			●				

TABLE II
EXAMPLES OF SECURITY TERMS THAT SUGGEST THE ABSENCE OF A SECURE MODULE

Security Vulnerability	Missing Security Terms
Unencrypted Socket Communication	ssl, keystore, tls, sunx509, x509certificate, sslcontext, keymanagerfactory
Insecure Authentication	textcallbackhandler, security, logincontext, krb5loginmodule, authpermission
Unvalidated SQL Input	executewithkey, queryfactory, sqlinsertclause, preparedstatement
Insecure Key Storage	ssh, credential, parsesecret, ttl

code such as "shopping cart", "socket", "login", etc.). These secure suggestions were created using FindBugs Regular Expression Builder. Figure 2 shows the innerworkings of the recommender system.

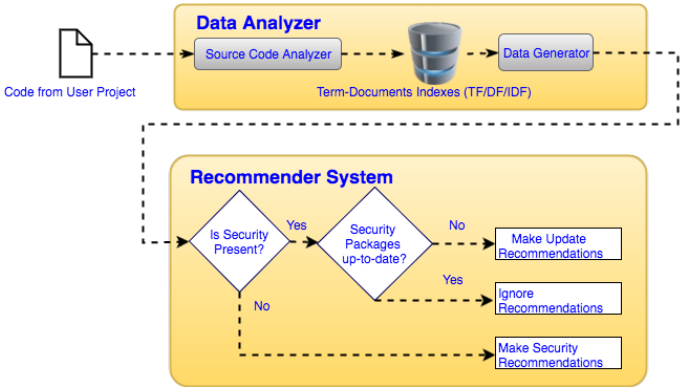

Fig. 2. Application of our hybrid system in Processing Source Code

IV. CLIENT/SERVER SOCKET COMMUNICATION CASE STUDY

The goal of this work is to build a classification model that, given an area of focus or topic, can detect when code is missing secure modules that appear in common open source projects and make recommendations to the programmer of where sample code can be located online. In this section, we present an example of the recommendation made by our tool when the user's project is missing known secure modules. In this case study, we applied our tool to a user's implementation of a client/server and analyzed the results obtained.

First, after crawling documentation and source code related to security, we used TF/IDF and topic modeling to classify projects into the four topics shown in Table II. We then manually labeled ten (10) examples that fall into the category of secure socket communication and five (5) examples that fall into insecure socket communication.

From the five example projects that were missing secure libraries and keywords, we observed that the websocket code followed a format that contained the following methods that process strings: `onOpen`, `onMessage`, `sendMessage`. Additionally, the insecure example projects imported generic java packages such as `javax.websocket`.

From the ten projects that contained secure implementations of web sockets, we performed TF/IDF to find key terms and, by extension, frequent itemsets that were common in security-present examples. After tokenizing the Java code, we used Lucene[9] to perform stemming and to remove stop-words. Figure 3 shows a sample of the stemmed words with frequency between 0.6 and 1.0. Further, these keywords were used to generate frequent itemsets. The top three (3) frequent itemsets for each stemmed keyword are shown in Table III. From the table, it can be seen that at least one term related to security appeared in each itemset. We observed that 70% of the secure server implementations imported a `crypto` or `keyfactory` library that was used to share secret keys between the server and client. Also, 70% of the secure client implementations imported `java.net.security` and the `javax.net.ssl` packages.

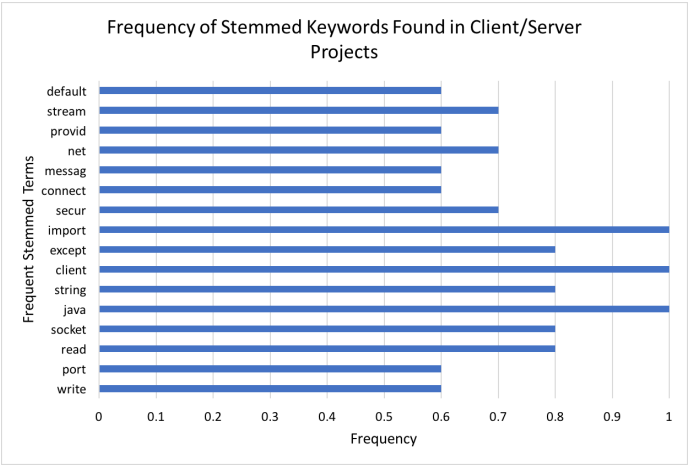

Fig. 3. Frequent stemmed words found in client/sever example projects

Our classification model was then trained with knowledge from the fifteen (15) client/server projects. Further, we extended the code analysis feature of FindBugs by adding predefined recommendation messages related to socket communication and improved the interface to offer customized tooltips. The recommender system was then tested on a user project.

Figure 4 shows the recommendation given when the user entered code with our tool activated. As can be seen in the figure, the recommender system was able to detect an insecure implementation of a web socket because it was missing terms and libraries related to security that were found in similar but more secure implementations of web sockets. This code was missing expected packages such as cipher-suites and java security packages. Thus, the recommender system was able to perform as expected. If the design choice is what the programmer intends, the option is given to ignore the recommendation or turn off the recommender system.

[9]https://lucene.apache.org/core/

TABLE III
FREQUENT ITEMSETS BASED ON FREQUENT TERMS FOUND IN TEN (10) CLIENT/SERVER EXAMPLE PROJECTS

Frequent Terms	Frequent Itemsets
write	CryptoSuite.writeSocket, OutputStreamWriter, PrintWriter
port	SslClient().run(host, port), serverPort, portNumber
read	PEMReader, BufferedReader, InputStreamReader
socket	ssl.socket.client, java.nio.channels.ServerSocketChannel, SSLSocketFactory
java	import java.security.*, import java.io., java.net.Socket
string	String sessionKey, String hostName, String CLIENT*
client	ssl.socket.client, public *client, *client.sendBytes
except	import java.io.IOException, throws *SecurityException, SSLException
import	import java.security.*, import java.io, import java.nio,
secur	import java.security.*, SecurityException, import java.security.KeyFactory
connect	Connect* to server, *Channel.connect(, client has disconnected
messag	messageDigest, decrypt* messa*, sendMessage
net	import javax.net.ssl., import java.net.Socket, import java.net.Socket
provid	socket.SslContextProvider, IdentityProvider, SOFTWARE IS PROVIDED
stream	import java.io.Input Stream, * Stream(), PEMReader(
default	KeyStore.getDefault, default Identity, default: throw new * Exception
server	server certificate, server socket, server reply

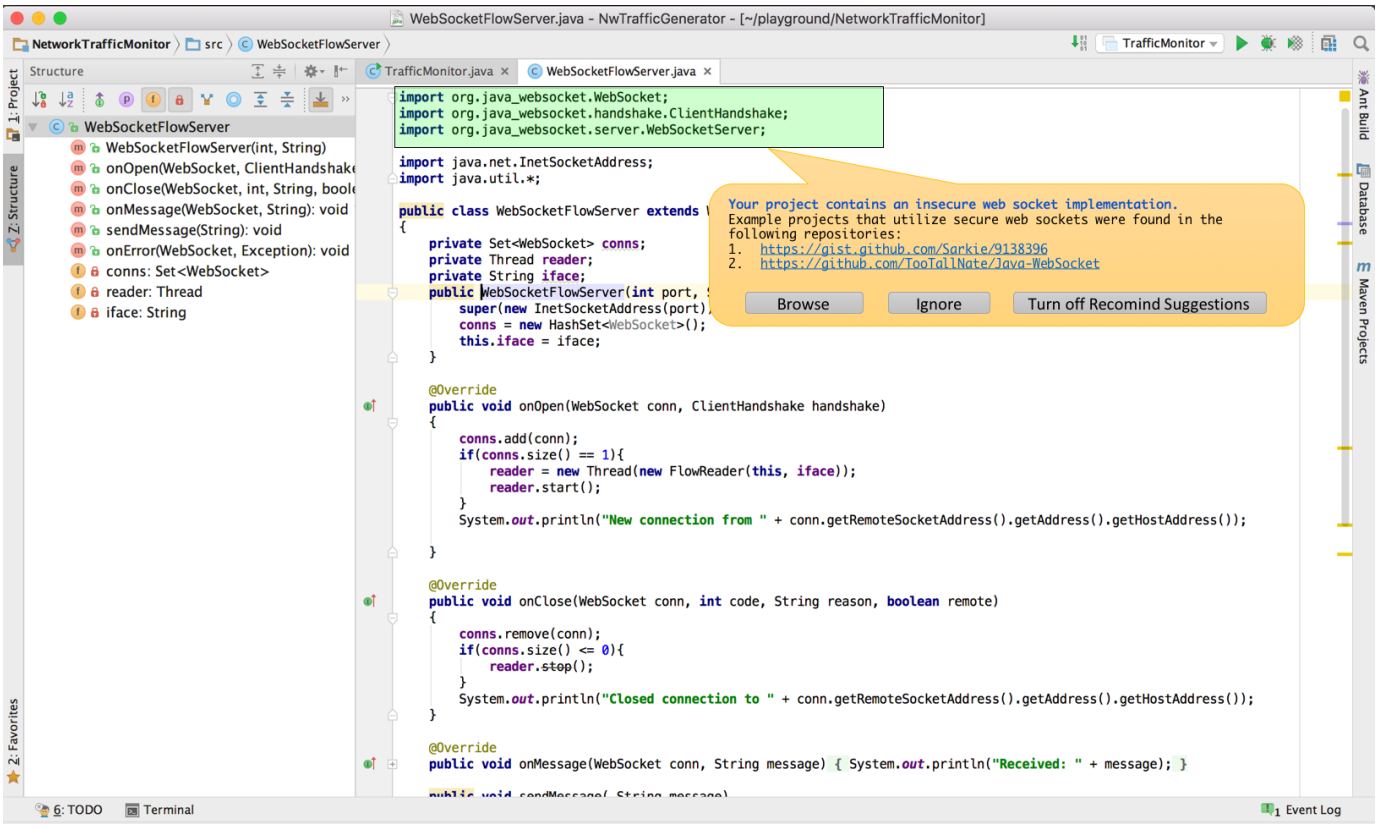

Fig. 4. A Recommendation offered by our tool when applied to a client/server web socket project

V. CONCLUSION

In this paper, we discussed our approach in building a hybrid recommender system that helps programmers write more secure code. By creating a crawler that crawls the web for documents and open-source projects related to security, we were able to use text mining and machine learning techniques to create a model that can detect when security is missing from a programming project. Our tool also makes recommendations to the programmer of common practices employed in similar projects, and provides links to code that the programmer can use as a guide for building secure systems. Using a case study, we were able to show that our system works by providing recommendations to the programmer when certain unsafe practices were followed.

REFERENCES

[1] Cas static analysis tool study - methodology. Technical report, Center for Assured Software, National Security Agency (NSA), Dec 2012.

[2] Stack Overflow: Developer Survey Results 2016. http://stackoverflow.com/insights/survey/2016, 2016. Accessed: 2017-04-15.

[3] Irene Abezgauz. Introduction to Interactive Application Security Testing (IAST). http://www.quotium.com/resources/interactive-application-security-testing/, Feb 2014. Accessed: 2017-07-08.

[4] Mamdouh Alenezi and Yasir Javed. Developer companion: A framework to produce secure web applications. *International Journal of Computer Science and Information Security*, 14(7):12, 2016.

[5] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, Oct 2002.

[6] Philippe Arteau. Find security bugs. https://find-sec-bugs.github.io/, 2017. Accessed: 2017-07-07.

[7] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, Sept 2008.

[8] Aniqua Z Baset and Tamara Denning. Ide plugins for detecting input-validation vulnerabilities. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 143–146, 2017.

[9] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 332–345. IEEE, 2010.

[10] Joeran Beel, Bela Gipp, Stefan Langer, and Corinna Breitinger. Research-paper recommender systems: A literature survey. *Int. J. Digit. Libr.*, 17(4):305–338, November 2016.

[11] Jakob Bleier, Erik Poll, Haiyun Xu, and Joost Visser. Improving the usefulness of alerts generated by automated static analysis tools. 2017.

[12] Jane Cleland-Huang, Brian Berenbach, Stephen Clark, Raffaella Settimi, and Eli Romanova. Best practices for automated traceability. *Computer*, 40(6):27–35, June 2007.

[13] M. Curphey and R. Arawo. Web application security assessment tools. *IEEE Security Privacy*, 4(4):32–41, July 2006.

[14] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, Jan 2002.

[15] Sarah Heckman and Laurie Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53(4):363 – 387, 2011. Special section: Software Engineering track of the 24th Annual Symposium on Applied Computing.

[16] Software Testing Help. Top 40 static code analysis tools. http://www.softwaretestinghelp.com/tools/top-40-static-code-analysis-tools/, 2017. Accessed: 2017-09-15.

[17] James Howison, Megan Conklin, and Kevin Crowston. Flossmole: A collaborative repository for floss research data and analyses. *International Journal of Information Technology and Web Engineering*, 1(3):17–26, 2006.

[18] HP. Hp fortify program trace analyzer (pta). https://softwaresupport.hpe.com/document/-/facetsearch/document/KM01024969, Jul 2014. Accessed: 2017-07-08.

[19] Yao-Wen Huang, Chung-Hung Tsai, Tsung-Po Lin, Shih-Kun Huang, D.T. Lee, and Sy-Yen Kuo. A testing framework for web application security assessment. *Computer Networks*, 48(5):739–761, 2005.

[20] G2 Crowd Inc. Best static code analysis software. https://www.g2crowd.com/categories/static-code-analysis, 2017. Accessed: 2017-09-15.

[21] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press.

[22] Job P Jonkergouw, Ming Zhu, and Magiel Bruntink. Effectiveness of automated security analysis using a uniface-like architecture. 2014.

[23] Ugur Koc, Parsa Saadatpanah, Jeffrey S Foster, and Adam A Porter. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 35–42. ACM, 2017.

[24] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *ACM SIGSOFT Software Engineering Notes*, SIGSOFT '04/FSE-12, pages 83–93, New York, NY, USA, 2004. ACM.

[25] James A. Kupsch, Elisa Heymann, Barton Miller, and Vamshi Basupalli. Bad and good news about using software assurance tools. *Software: Practice and Experience*, 47(1):143–156, 2017.

[26] T. Landauer, S. Dennis McNamara, and W. Kintsch, editors. Laurence Erlbaum, 2007.

[27] Gregory Larsen, EK Fong, David A Wheeler, and Rama S Moorthy. State-of-the-art resources (soar) for software vulnerability detection, test, and evaluation. Technical report, INSTITUTE FOR DEFENSE ANALYSES ALEXANDRIA VA, 2014.

[28] P. Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, July 2006.

[29] Kayla Matthew. Recent Study Shows Employees Believe Productivity Is More Important Than Security. http://productivitybytes.com/recent-study-shows-employees-believe-productivity-is-more-important-than-security/, 2014. Accessed: 2017-04-20.

[30] CC Michael et al. Black box security testing tools. Citeseer, 2005.

[31] Mehdi Mirakhorli and Jane Cleland-Huang. Detecting, tracing, and monitoring architectural tactics in code. *IEEE Trans. Software Eng.*, 42:205–220, 2016.

[32] Mehdi Mirakhorli, Ahmed Fakhry, Artem Grechko, Matteusz Wieloch, and Jane Cleland-Huang. Archie: A tool for detecting, monitoring, and preserving architecturally significant code. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 739–742, New York, NY, USA, 2014. ACM.

[33] National Institute of Standards and Technology (NIST). Source code security analyzers. https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html, 2017. Accessed: 2017-09-15.

[34] OWASP. Category:vulnerability scanning tools. https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools, 2017. Accessed: 2017-07-07.

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

[36] Andrey Petukhov and Dmitry Kozlov. Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing. *Computing Systems Lab, Department of Computer Science, Moscow State University*, 2008.

[37] PRNewswire. Interactive application security testing (iast) named by gartner analysts in top 10 technologies for information security in 2014. http://www.prnewswire.com/news-releases/interactive-application-security-testing-iast-named-by-gartner-analysts-in-top-10-technologies-for-information-security-in-2014-265390091.html, Jul 2014. Accessed: 2017-07-08.

[38] Matthias Rohr. Sustainable application security. https://blog.secodis.com/2015/11/26/the-emerge-of-iast/, Nov 2015. Accessed: 2017-07-08.

[39] Bojan Simic, Arshan Dabirsiaghi, and Jeff Williams. Library code security analysis. Technical report, ASPECT SECURITY COLUMBIA MD, 2013.

[40] M. Steyvers and T. Griffiths. Probabilistic topic models. In Landauer et al. [26].

[41] Techopedia. Dynamic application security testing (dast). https://www.techopedia.com/definition/30958/dynamic-application-security-testing-dast, 2017. Accessed: 2017-07-07.

[42] Mikko Vestola et al. Evaluating and enhancing findbugs to detect bugs from mature software; case study in valuatum. 2012.

[43] Jeff Williams and Arshan Dabirsiaghi. Interactive vulnerability analysis enhancement results. Technical report, ASPECT SECURITY COLUMBIA MD, 2012.

[44] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 97–106, New York, NY, USA, 2004. ACM.