Chapter

Conversational Code Analysis: The Future of Secure Coding

Fitzroy Nembhard and Marco M. Carvalho

Abstract

The area of software development and secure coding can benefit significantly from advancements in virtual assistants. Research has shown that many coders neglect security in favor of meeting deadlines. This shortcoming leaves systems vulnerable to attackers. While a plethora of tools are available for programmers to scan their code for vulnerabilities, finding the right tool can be challenging. It is therefore imperative to adopt measures to get programmers to utilize code analysis tools that will help them produce more secure code. This chapter looks at the limitations of existing approaches to secure coding and proposes a methodology that allows programmers to scan and fix vulnerabilities in program code by communicating with virtual assistants on their smart devices. With the ubiquitous move towards virtual assistants, it is important to design systems that are more reliant on voice than on standard point-and-click and keyboard-driven approaches. Consequently, we propose MyCodeAnalyzer, a Google Assistant app and code analysis framework, which was designed to interactively scan program code for vulnerabilities and flaws using voice commands during development. We describe the proposed methodology, implement a prototype, test it on a vulnerable project and present our results.

Keywords: secure coding, virtual assistant, code analysis, static analysis

1. Introduction

Computing systems face serious threats from attackers on a day-to-day basis. Devices within a network could be targeted or used as launching pads to spawn malware and other attacks to critical systems and infrastructure. A system is as secure as its weakest link [1]. Therefore, software engineers must be cognizant of the cyber-related challenges that plague modern computer systems and engineer software with credible defenses. One of the first defenses against potential threats to computer systems is careful analysis of program code during development and taking necessary steps to minimize/eliminate vulnerabilities.

Program analysis falls into three main categories: static application security testing (SAST) or static analysis, dynamic application security testing (DAST) or dynamic analysis, and interactive application security testing (IAST). Static analysis is a "technique in which code listings, test results, or other documentation are ... examined ... to identify errors, violations of development standards, or other problems" [2]. Dynamic analysis is the "process of evaluating a system or component based on its behavior during execution" [2]. IAST involves instrumenting a

program with sensors to monitor program code in memory during execution in order to find specific events that could cause vulnerabilities [3]. Two or more of these approaches may be combined to create hybrid tools and techniques for analyzing program code. These hybrid systems are designed to achieve more comprehensive coverage and to decrease the false positives and false negatives of existing approaches.

While researchers are interested in designing sound and complete code analysis tools, achieving soundness and completeness remains an intractable problem [4–6]. Consequently, a lot of research in code analysis is centered on improving the alerts of static analysis tools [4, 7]. More recently, several researchers have proposed models based on deep learning and other machine learning approaches to scan and fix vulnerabilities in program code [8]. Many of these tools are still at an infant stage and have not yet made it to market. Based on current trends, we believe that the future of code analysis will involve more refined tools based on artificial intelligence (AI), machine learning, and other hybrid approaches.

In this work, we propose a hybrid code analysis framework that employs the use of voice assistants (VAs) to allow a programmer to conversationally scan for and fix potential vulnerabilities in program code. The use of voice assistants have grown significantly in recent years. This work focuses primarily on the Google Assistant[1] as it is the most popular [9] among other virtual assistants.

The rest of the chapter is organized as follows: first, we discuss related work in the area of hybrid analysis in Section 2 followed by a discussion on challenges affecting adoption of existing approaches in Section 3. In Section 4, we theorize about the future of secure coding and propose a new code analysis approach in Section 5. We then use a case study to evaluate our proposed approach in Section 6 and present our conclusion in Section 7.

2. Related work

This work falls in the area of hybrid analysis. In this section, we summarize works in this area.

In 2006, Aggarwal and Jalote [10] combined static and dynamic analysis to detect buffer overflow in C programs. Both static and dynamic approaches have advantages and disadvantages. One of the disadvantages of dynamic analysis is the requirement of a large number of test cases, which present an overhead. Some dynamic analysis tools use a feature know as generate-and-patch or generate-and-validate in an effort to auto-fix vulnerabilities. In 2015, the authors of [11] analyzed reported patches for several DAST tools including GenProg, RSRepair, and AE, and found that the overwhelming majority of reported patches did not produce correct outputs. The authors 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 [11].

In 2012, [12] proposed a hybrid approach that uses source code program slicing to reduce the size of C programs while performing analysis and test generation. The authors used a minimal slicing-induced cover and alarm dependencies to diminish the costly calls of dynamic analysis [13].

[1] Google, Google Assistant, and Dialogflow are registered trademarks of Google, Inc. The use of these names or tools and their respective logos are for research purposes and does not connote endorsement of this research by Google, Inc. or any of its partners.

In 2014, [14] implemented a hybrid architecture as the JSA analysis tool, which is integrated into the IBM AppScan Standard Edition product. The authors augmented static analysis with (semi-)concrete information by applying partial evaluation to JavaScript functions according to dynamic data recorded by the Web crawler. The dynamic component rewrites the program per the enclosing HTML environment, and the static component then explores all possible behaviors of the partially evaluated program.

In 2015, [15] applied a program slicing technique, similar to [12], to create a tool called *Flinder-SCA*. The authors also implemented their program using the *Frama-C* platform. The main difference between [12, 15] is that [15] performs abstract interpretation and taint analysis via a fuzzing technique wheres [12] does not perform taint analysis or fuzzing.

Also, in 2015, [16] proposed a hybrid malicious code detection scheme that was designed using an AutoEncoder and Deep Belief Networks (DBN). The AutoEncoder deep learning method was used to reduce the dimensionality of data. The DBN was composed of a multilayer Restricted Boltzmann Machines (RBM) and a layer of BP neural network. The model was tested on the KDDCUP'99 dataset but not on actual program code.

In 2019, [17] proposed SapFix, a static and dynamic analysis tool which combines a mutation-based technique, augmented by patterns inferred from previous human fixes, with a reversion-as-last resort strategy for fixing high-firing crashes. This tool is built upon Infer [18] and a localization infrastructure that aids developers in reviewing and fixing errors rapidly. Currently, SapFix is targeted at null pointer exception (NPE) crashes, but has achieved much success at Facebook [18].

In a dissertation produced in 2021, [19] proposed a code generation technique for Synchronous Control Asynchronous Dataflow (SCAD) processors based on a hybrid control-flow dataflow execution paradigm. The model is inspired by classical queue machines that completely eliminates the use of registers. The author uses satisfiability (SAT) solvers to aid in the code generation process [19].

To the best of our knowledge, our work is the first to employ modern virtual assistants to conversationally scan and fix vulnerabilities in program code. In [20], the authors established a voice user interface (VUI) for controlling laboratory devices and reading out specific device data. The results of their experiments produced benchmarks of established infrastructure and showed a high mean accuracy (95% $\pm$ 3.62) of speech command recognition and reveals high potential for future applications of a VUI within laboratories. In like manner, we propose the integration of personal assistants with code analysis systems to encourage programmers to produce more secure code.

3. Challenges affecting adoption of existing approaches

Several code analysis and vulnerability detection surveys have categorized tools in the literature [7, 21–23]. While surveys are essential in advancing research, many of them do not focus on tools found on websites. It must be noted that the average programmer does not look for tools in research papers. To that end, we conducted a Google search and found several popular websites that present various tools that programmers may use to scan their code for vulnerabilities. **Figure 1** shows a bar chart highlighting the number of tools found on these websites. As shown in the figure, GitHub and Wikipedia list the most tools and are often the top websites returned in search results due to their popularity. We further grouped the most popular static analysis tools found on these websites by language as shown in

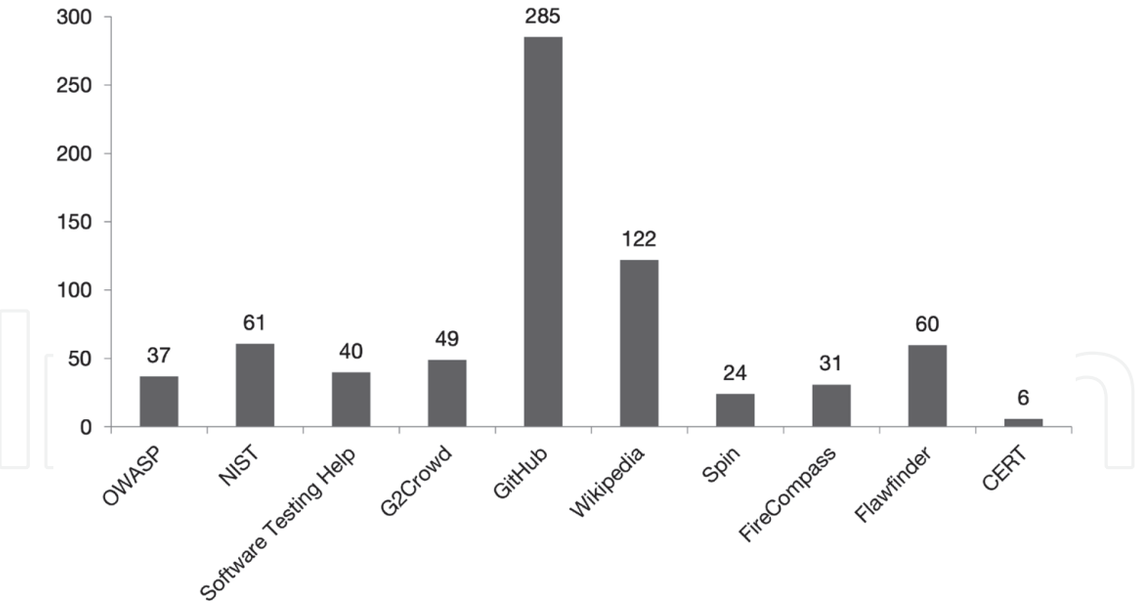

Figure 1.
The large number of code analysis tools found on popular websites.

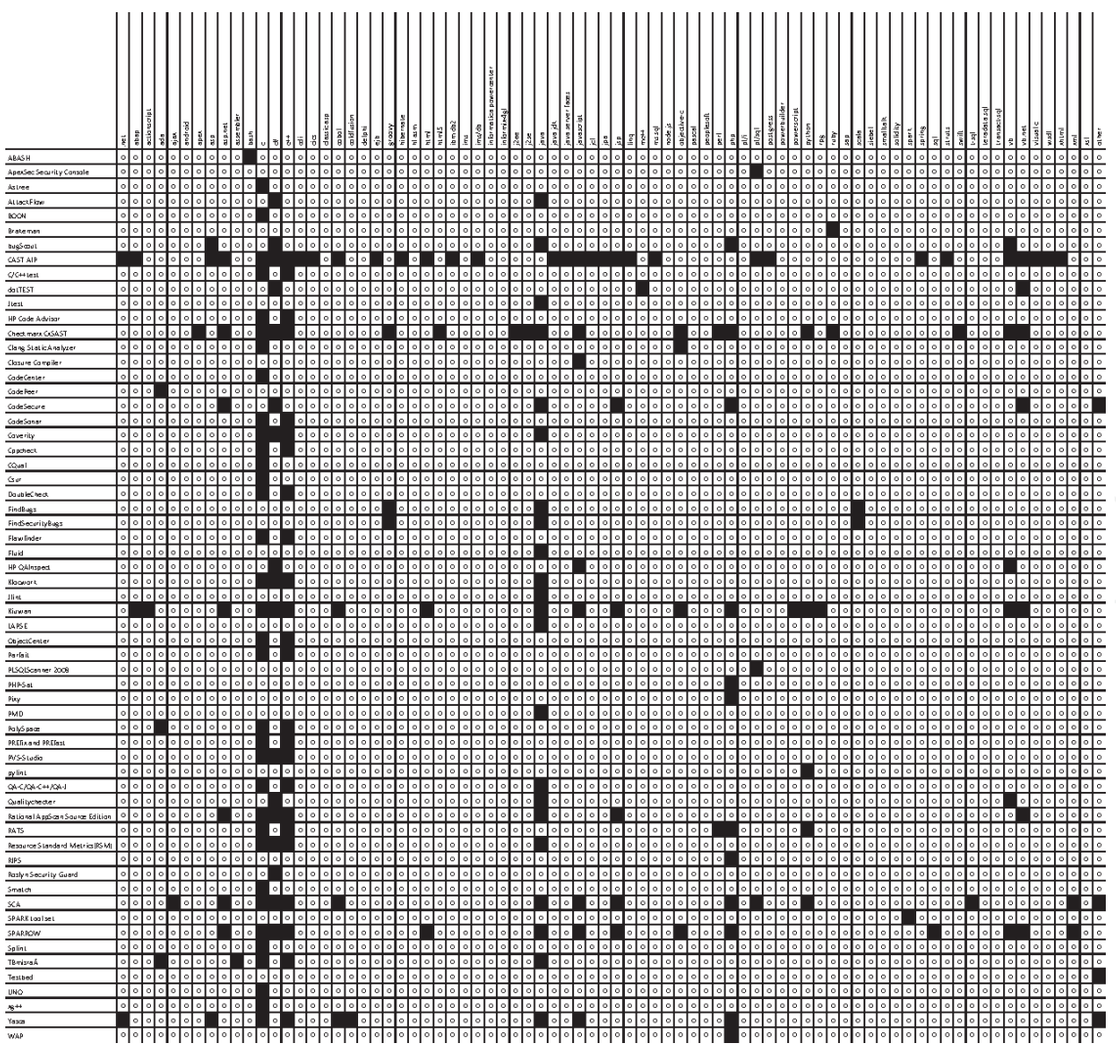

Figure 2.
Static analysis tools categorized by programming language.

Figure 2. As can be seen, this non-exhaustive list could overwhelm many programmers in determining the best tools for their projects.

In addition, the ability to combine code analysis approaches coupled with the number of programming languages that exist result in a large number of tools from which coders can choose to analyze their code. This makes it onerous for a programmer or organization to decide on a particular code analysis tool. Further, tools often require special configuration, which may take time to fine tune for best results. Many tools also suffer from usability issues, lengthy vulnerability reports, and false positives, making programmers avoid them altogether [24–26].

Another challenge affecting adoption of code analysis tools is monopolization of the market by certain companies. For-profit companies usually have the resources to improve tools by adding more state-of-the-art approaches such as cloud-based scanning, IAST support, and report generation. While these developments often advance the field of code analysis, they sometimes discourage small organizations and individuals from investing the effort and resources required to procure state-of-the-art tools. Thus, a streamlined, modern, cost-effective approach is needed to help encourage programmers to produce more secure code.

4. The future of code analysis

We believe that the future of code analysis lies in hybrid systems that combine several approaches to achieve useful analyses and actionable reports that will encourage programmers to produce more secure software. Based on current trends in machine learning, especially in deep learning, and natural language processing (NLP) (e.g., virtual assistants), it is safe to say that future code analysis will rely heavily on AI, ontologies, NLP, and machine learning. For example, when discussing the trends and challenges of machine learning, the authors in [27] "envision a fruitful marriage between classic logical approaches (ontologies) with statistical approaches which may lead to context-adaptive systems (stochastic ontologies) that might work similar to the human brain" [27].

Our projection is that code analysis frameworks will facilitate plug-and-play (PnP) models. **Figure 3** illustrates a generalized PnP model that uses virtual assistants to manage the analysis process. Using this plug-and-play model, programmers may select the code analyzer that best fits their project based on factors such as project type, project size, speed, efficiency, security, etc. This is similar to the

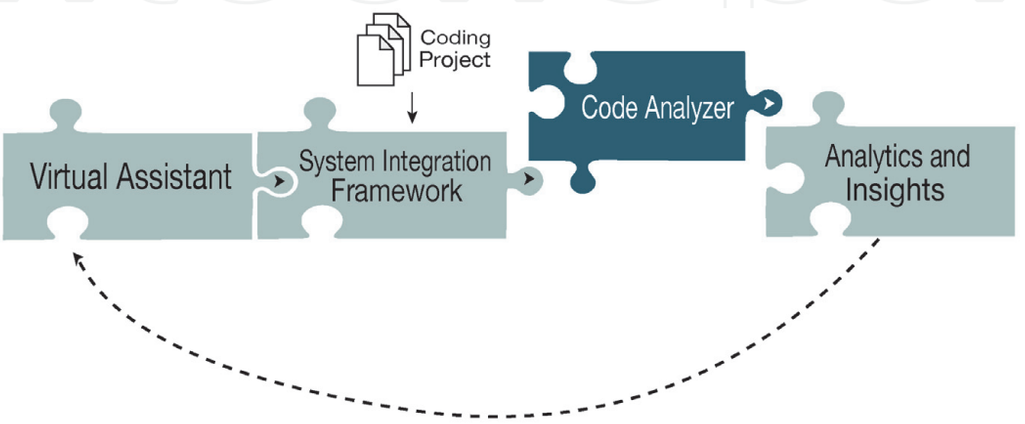

Figure 3.
A suggested model showing code analysis as part of a plug-and-play paradigm that facilitates the inclusion of any analysis tool and the use of a virtual assistant to manage the analysis process.

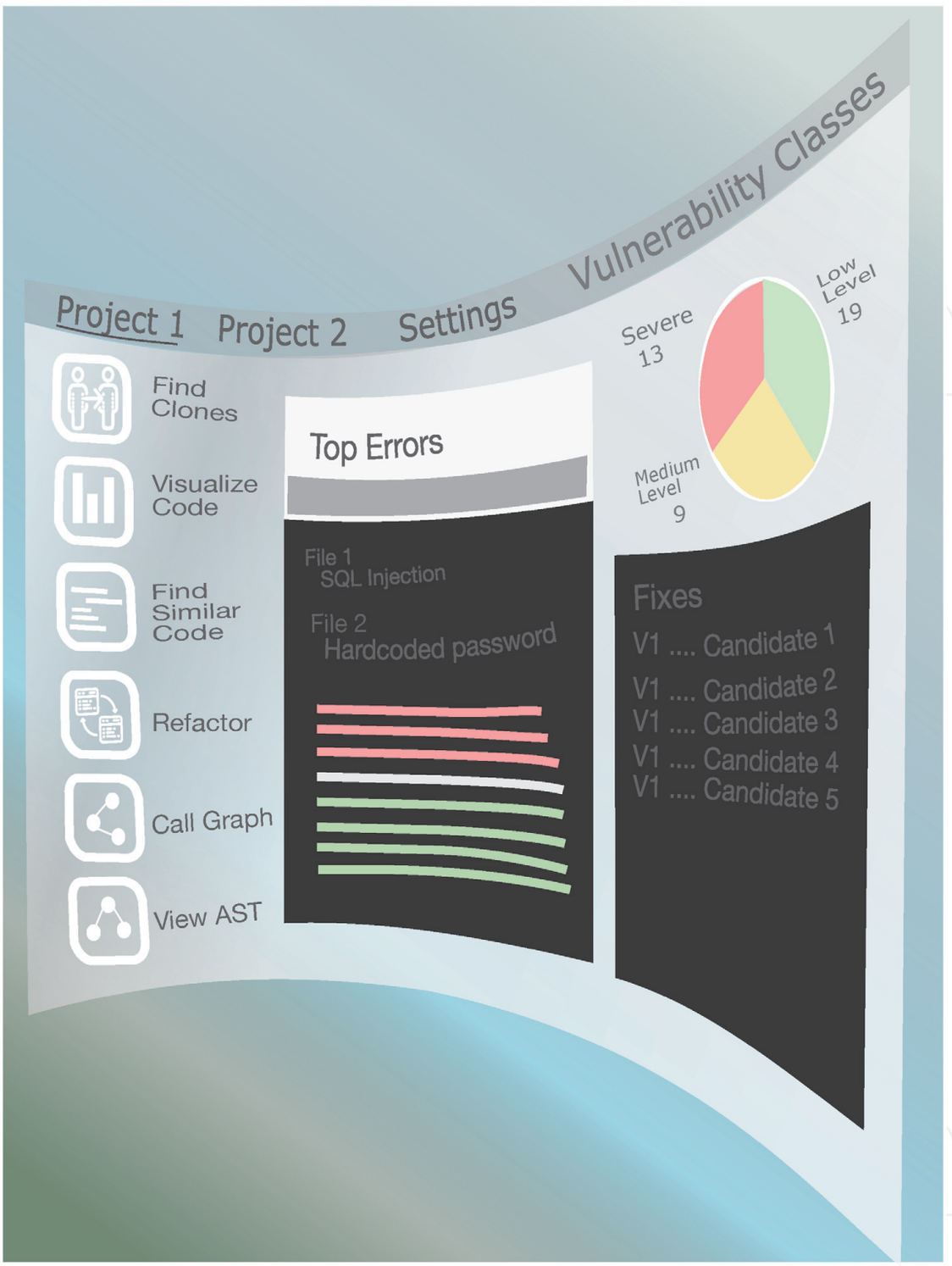

Figure 4.
A mockup of an analytical dashboard for code analysis on a curved display.

current landscape with virtual assistants and recommender systems. Currently, a person may use a virtual assistant like the Google Assistant to navigate a list of restaurants based on price, location, menu, reviews, etc. The virtual assistant may update the users preferences based on selections over time. This concept can also apply in code analysis where the chosen scanner used in the PnP model could be based on past scans or popularity.

The code analyzer featured in the model in **Figure 3** may use any combination of approaches including SAST, DAST, and IAST, which could be cloud-based or localized to the user's computer. These approaches could be backed by any algorithm

that results in significant performance gains. It has been shown in the literature that deep learning and other ensemble methods perform very well in a large number of contexts including infected host detection [28], intrusion detection systems [29, 30], and malware analysis [31, 32], to name a few. Interestingly, many of these approaches can be used to create or improve code analyzers in an effort to help programmers produce more secure software.

Another feature of code analyzers of the future is a deep reliance on data analytics, visualizations and state-of-the-art interfaces. As discussed in the literature [8, 33], the interface of a code analyzer can have a negative or positive impact on its use and adoption. Therefore, for a system to be adopted in any project or organization, users must be able to gain insights from the way it presents its results. **Figure 4** shows a mockup of what we believe the interface of future code analyzers will look like. These interfaces will be in the form of dashboards instead of the customary lengthy bug reports displayed in a console.

5. Proposed approach

The proposed approach is to integrate a virtual assistant with a code analysis framework that allows users to scan, analyze, refactor and fix their code of inconsistencies and vulnerabilities. In this section, we describe the proposed approach using the system architecture.

5.1 System architecture

The system architecture for MyCodeAnalyzer is shown in **Figure 5**. The system consists of three main components: the virtual assistant, the webhook API and the

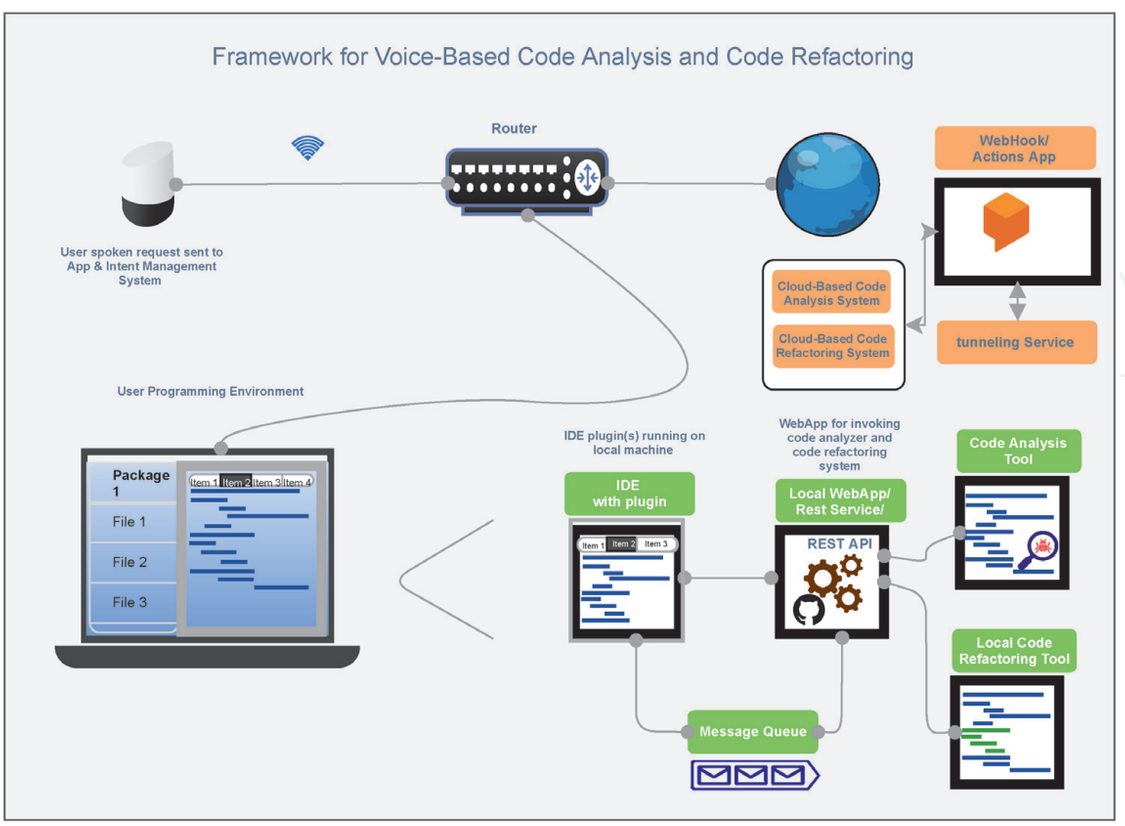

Figure 5.
MyCodeAnalyzer system architecture.

code scanning environment. The code scanning environment consists of a web app, an integrated development environment (IDE) plugin, code analyzers and refactoring tools. Google Assistant was chosen as the virtual assistant because of its popularity and easy-to-use App Engine and Dialogflow frameworks. The process flow is as follows: a user invokes a Google Assistant app (aka, Google Actions app) using a set of phrases understood by the system. This app is specially designed to understand trigger phrases associated with code analysis. Trigger phrases are training phrases that are entered into Dialogflow using an intent management system. Dialogflow is a natural language understanding platform that allows users to design and integrate a conversational user interface into a mobile app, web application, device, bot, interactive voice response system, etc. [34]. **Figure 6** captures the current intents incorporated into MyCodeAnalyzer. Each intent is backed by machine learning and NLP technology that uses named entity recognition (NER) and other approaches to extract entities from speech, determine context, and carry out tasks.

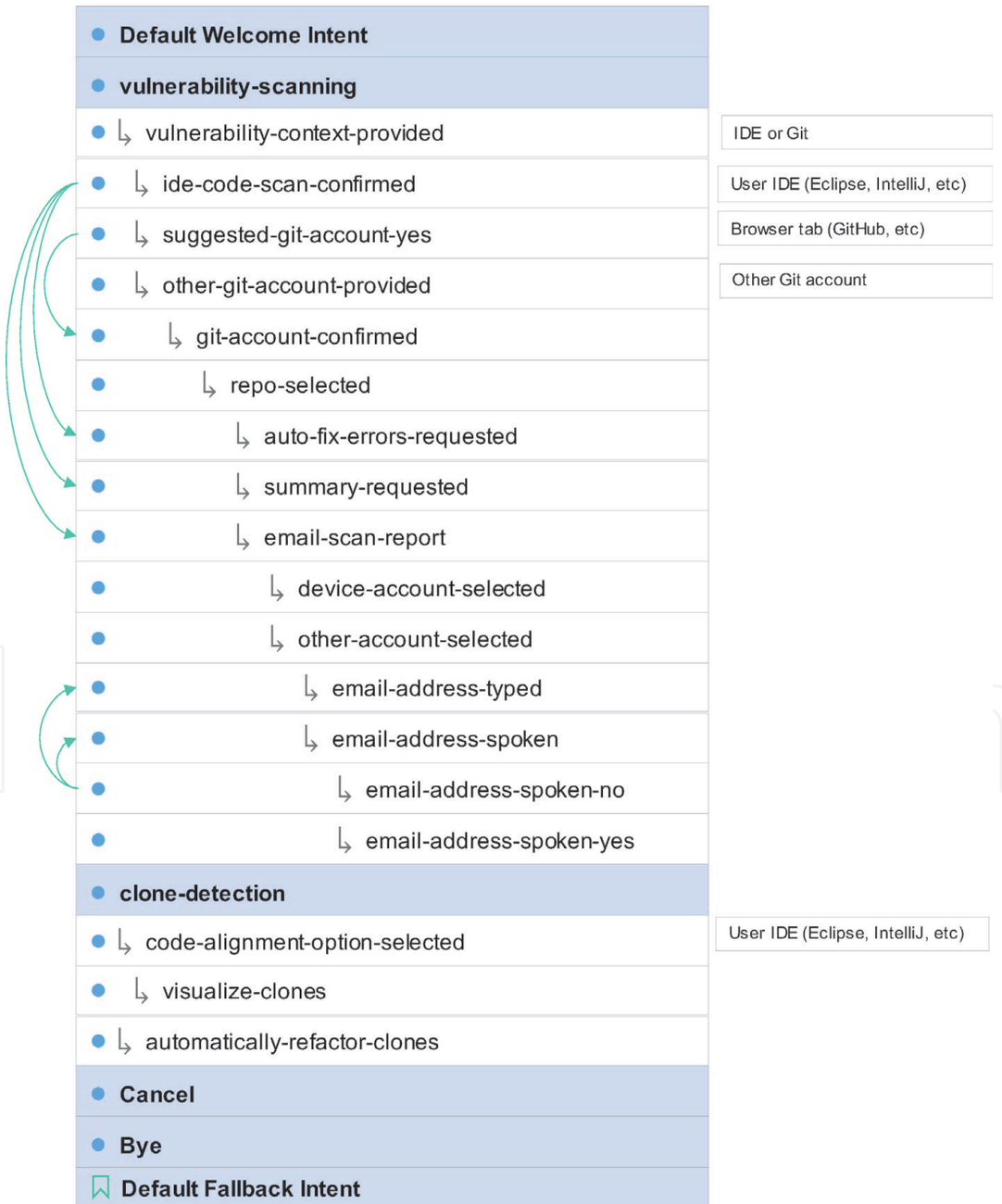

Figure 6.
Current Dialogflow intents used by MyCodeAnalyzer.

The intents in MyCodeAnalyzer are organized into 6 main categories: *Default Welcome Intent, vulnerability-scanning, clone-detection, Cancel, Bye, and Default Fallback Intent*. The *Default Welcome Intent* is used to welcome the user to the system and provide a description of potential requests that the application can fulfill. The *vulnerability-scanning* intent is the most complex of the intents and uses a tree-like structure to allow the user to conversationally scan a project for vulnerabilities, email a scan report or auto-fix errors based on the capabilities of the code analyzer. The *clone-detection* intent is used to scan a project for duplicated code and to provide a visualization showing a side-by-side comparison of similar code. While clones may not be vulnerable, they could become bloat in a project and could potentially lead to vulnerabilities. The *Cancel* intent is used to exit a task currently underway. *Bye* is used to exit the system and the *Default Fallback Intent*, as the name suggests, is used to ask the user to repeat a phrase for clarification or serve as a graceful fail mechanism.

Once invoked, the Google Assistant app communicates with the Google Conversation API to determine the user's intent. After intent has been determined, the Google Actions app then uses webhooks to communicate with a web service running on the user's computer. Using a tunneling service, the web service interacts with the user's IDE by way of a plugin. This plugin invokes a code analyzer or refactoring tool, takes actions based on the user's request, and places a message in a message queue. The web service then reads the queue and returns the message to the Google Assistant app, which then reads the message back to the user. The webhooks were set up in Dialogflow and run as servlets on Google App Engine. A servlet accepts valid Dialogflow POST requests and responds with data that is processed by the Google Assistant app and returned as output messages to the user. **Figure 7** further shows the internals of the system during a conversation between the user and the assistant. While only the static analysis portion of the system is demonstrated in this work, the system is modular enough for dynamic and hybrid analysis tools to be incorporated

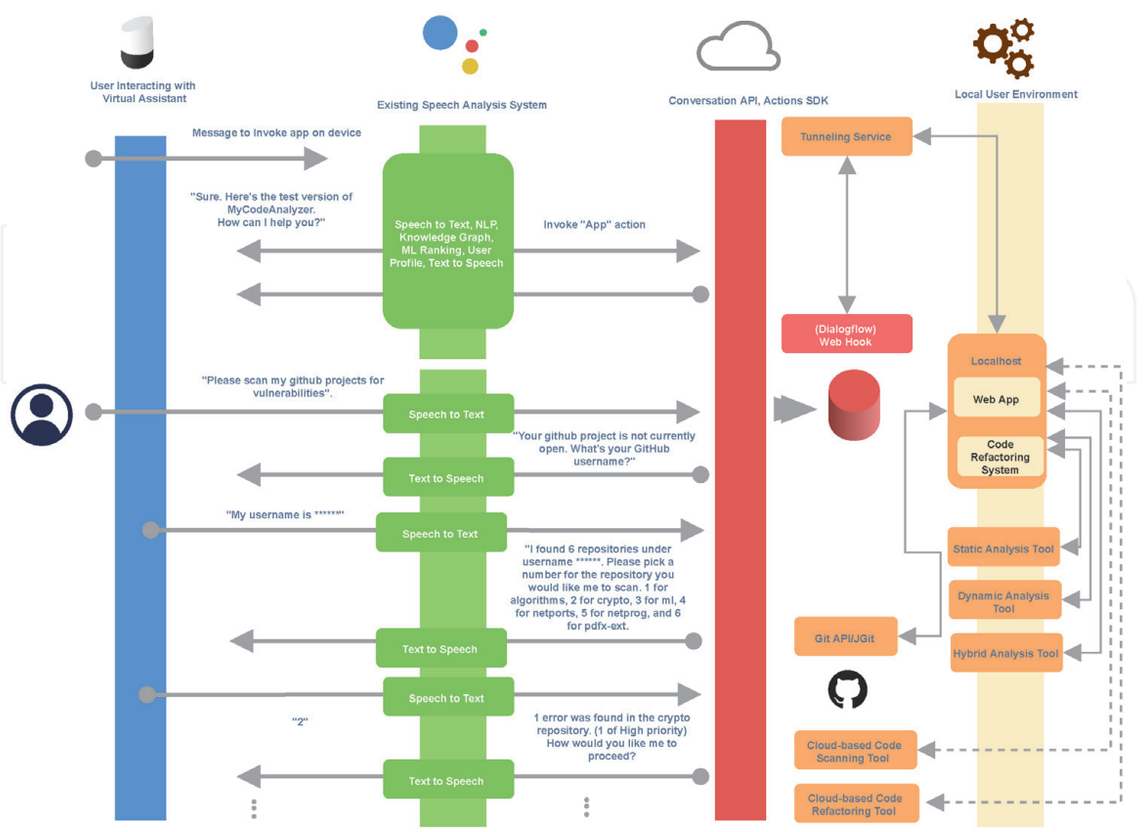

Figure 7.
Internals of MyCodeAnalyzer showing the flow of information throughout the system.

using the PnP approach discussed in Section 4. This approach provides a more complete code analysis depending on the user's preferences.

5.2 Accessing information about the coding environment

Two types of code-related information are accessed on the user's computer: code within the IDE and code from a Git repository (e.g., GitHub) currently opened in a web browser. The first type of information is important because it helps us to scan code being actively developed, while the second type is used in the case where the user would like to ensure that a repository is safe before forking it. MyCodeAnalyzer can detect GitHub pages that are open in a browser. On systems running MacOS, Applescript is used to communicate with the web browser. Other approaches will be employed in the future to reproduce this functionality on machines running other operating systems.

In order to access the user's computer to scan the code being worked on in the IDE or referenced in the browser, a methodology must be established to access this information in a minimally invasive manner. To do so, we created a plugin for a given IDE. Currently, we have plugins for IntelliJ IDEA and Eclipse. The plugin becomes a part of the IDE, monitors the code being developed, and updates a message queue (data file) with information about the code files and projects manipulated by the programmer. Also, special system calls are used to access any browser tabs that point to GitHub projects. A local web app in the form of a Spring MVC REST API [35] runs on the user's computer. The job of the local web app is to communicate with MyCodeAnalyzer by way of a tunnel in order to scan local code or GitHub projects displayed in the user's web browser.

5.2.1 Accessing code within the IDE

Listing 1 shows the Applescript code that is used to check for gui-based applications that are currently open on the user's computer. Following this is a snapshot of the corresponding output, which includes the Intellij IDEA IDE in the list. This Applescript code is added to the REST app where it is run on localhost and invoked by MyCodeAnalyzer to determine if the user is actively using an IDE. To further contextualize the process of determining which code the user would like to scan, it is also of interest to find out the *frontmost* or most active application on the user's computer. To do so, the code shown in Listing 2 was used. This code is expected to return a single application, which in turn allows MyCodeAnalyzer to return a more direct response to the user. For example, a response might be, "*Say IDE, if you would like me to scan the code that you are currently working on in IntelliJ*" instead of using indirect phrases such as "*... may be working on.*"

```
set text item delimeters to ", "
tell application "System Events" to
(name of every process where background only is false) as text end tell
```

Listing 1. Applescript code used to list all gui-based applications that are currently running on the user's computer.

The following is a sample output generated using the code in Listing 1:

```
"Google Chrome, Sublime Text, Terminal, idea, pycharm,Teams, Mail, teXShop,
Notes, Spotify, Finder, Microsoft PowerPoint, X11.bin, AdobeReader, iTunes,
Microsoft Excel,Script Editor, Activity Monitor, System Preferences, Safari,
Preview"
```

Since most IDEs are standalone applications, we believe the best way to have access to the user's code in a minimally invasive manner is to be an "insider" (That is, to use a plugin that becomes part of the IDE). Consequently, the goal of the plugins was to monitor the code being developed by taking note of the coding project and the coding files being manipulated by the user. To accomplish this, listeners were added to the IDE to detect when the text editor portion of the IDE is active, when tabs are activated or switched, and when code files are edited. The message queue is updated with the following pieces of information when the aforementioned actions are performed: *ProjectName, ProjectLocation, CurrentFile, DateAdded, CurrentlyActive*. This queue is then queried for active files and projects when POST requests are made by the Google Assistant app to the local REST service running on the user's computer.

```
tell application "System Events"
name of application processes whose frontmost is true end tell
```

Listing 2. Applescript code used to determine the most active application on a computer.

5.2.2 Accessing code referenced by tabs opened in the web browser

Like IDEs, web browsers provide little to no way for outside tools to access their core areas. However, the Applescript-based techniques used previously for accessing the System Events utility can be used to access the tabs that are currently open in the web browser on the user's device. Listing 3 is used to retrieve tabs currently open in Google Chrome. This script can be modified to get tabs in other browsers such as FireFox or Safari. MyCodeAnalyzer then checks if any of the URLs point to valid public GitHub accounts, which are then searched for coding projects if the user requests that a scan of a Git project be performed.

```
set text item delimeters to ","
tell application "Google Chrome" to URL of tabs of every window
as text
end tell
```

Listing 3. Applescript code used to retrieve tabs currently open in Google Chrome.

6. Case study

In this section, we present a case study that demonstrates an implementation of our proposed methodology. The main goal of this case study is to demonstrate the applicability of integrating a virtual assistant into a code analysis framework to allow the user to conversationally scan their code for vulnerabilities. The system is currently in a prototypical stage. Here we perform a scan of a coding project using the Google Assistant app via an Apple iPhone.

The following was done based on the proposed approach discussed in Section 5:

1. Create a Google Assistant app

   A Google Assistant app was created based on the intents depicted in **Figure 7**. Dialogflow, Google App Engine, and Google Actions Console are key

components in the design of the app. Once designed, the app was tested using the Google Actions API Simulator as well as released in alpha mode and tested on a smart phone running the Google Assistant.

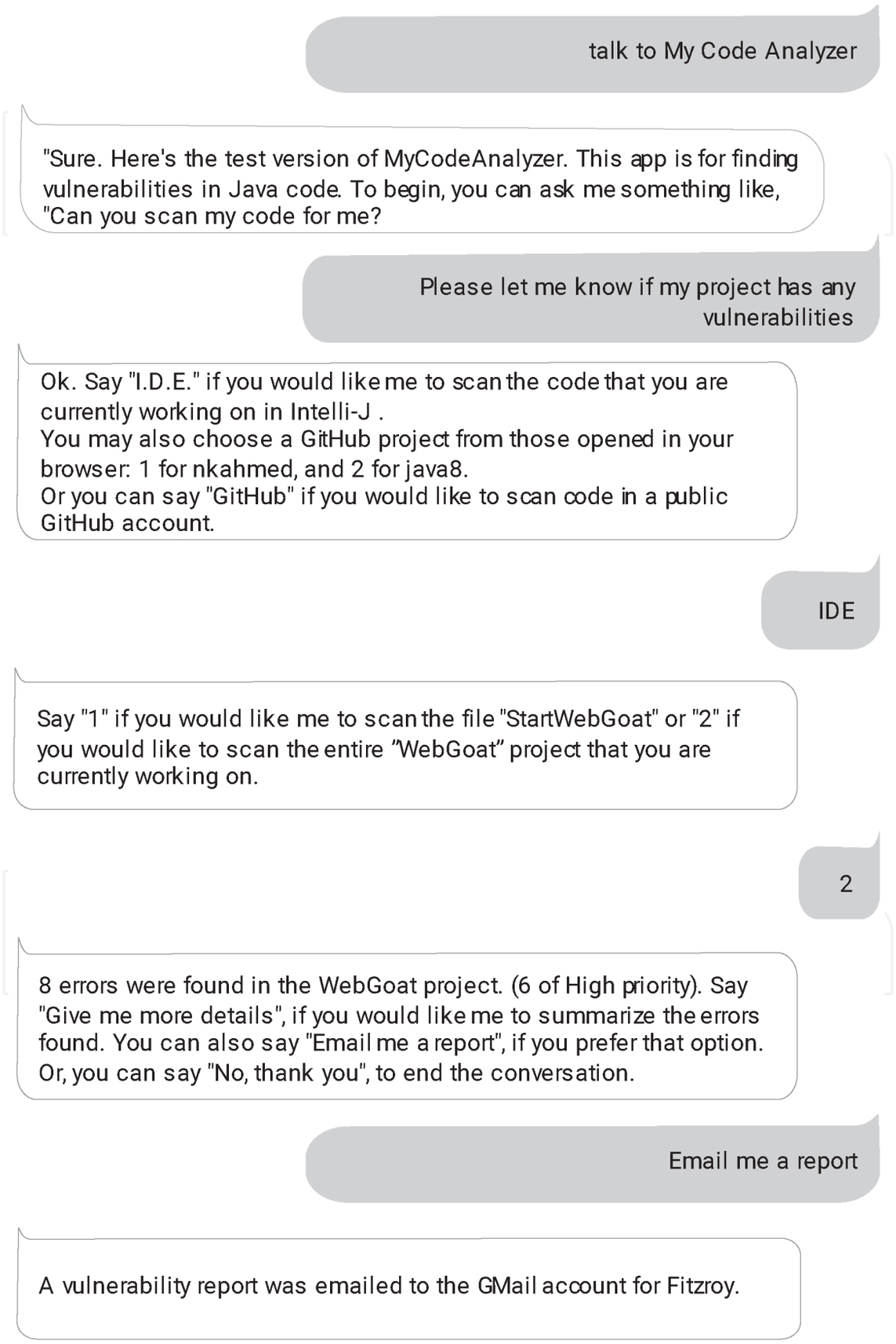

Figure 8.
A conversation between MyCodeAnalyzer and a human tester while scanning the OWASP WebGoat project.

2. Create a local web app to interface with the Google Assistant app and the coding environment

   The local web app was created using Spring Boot [35] and was launched on the computer via Apache Tomcat [36].

3. Create an IDE plugin for IntelliJ IDEA

   Our IntelliJ IDEA plugin was created and installed in IntelliJ version 2020.3.2. The plugin was installed using the IntelliJ plugin installer, which installs a local plugin from a JAR (Java ARchive) file.

4. Choose and integrate a code analyzer

   PMD [37] static code analyzer (version 6.31.0) was chosen for this study. PMD uses a rule-based system to find common programming flaws in code written in 8 programming languages, offering the most support for Java and Apex. The rules used by PMD are divided into categories such as best practices, error prone, and security. For this case study, a set of rules was selected from the error prone and security categories.

5. Chose a vulnerable project

   The OWASP WebGoat [38] project was used to evaluate the system. WebGoat is an insecure application that allows researchers and developers to test vulnerabilities commonly found in Java-based applications that use common and popular open source components [38].

6. Test the system and report results

   To integrate the Google Actions app with the local web app, Ngrok [39] was chosen as the tunneling tool. Ngrok is a tool that exposes local servers behind NATs and firewalls to the public Internet over secure tunnels [39].

6.1 Results and discussion

In this section, we capture a conversation between the Google Assistant app during the analysis of the WebGoat Project, present the report generated by the assistant, and discuss the results. It must be noted that the errors found by the Assistant during the code analysis are the same as those that would be produced by the standalone PMD project.

At this early stage of the project, the main benefit of the system is the ability to use a virtual assistant to perform code analysis while multitasking, thus improving productivity. After the system is setup, the programmer can configure and engage with the VA by voice without having to manually configure the code analyzer or browse and try to understand lengthy bug reports. The assistant can be used to perform actions based on the severity of the vulnerabilities found in the project. In the current version of MyCodeAnalyzer, Google Assistant can email the user a well-formatted report or read out the most important action items after analyzing the code. **Figure 8** captures a conversation between a human tester and the Google Assistant. **Figure 9** shows a formatted vulnerability report generated by the assistant and emailed to the user after scanning the WebGoat project. The WebGoat project has more severe vulnerabilities, but only those in the figure were captured by PMD based on the rulesets used by the analyzer. As can be seen from the report, MyCodeAnalyzer was able to process the lengthy XML reported returned by PMD

Vulnerability Scan Report

Below is the vulnerability scan report for your WebGoat Project

Problem	Class	Method	Priority	Line	Desc
Constructor calls overridable method	UserSessionData	UserSessionData	High	16	Overridable method 'setValue' called during object construction
Constructor calls overridable method	WebGoatUser	WebGoatUser	High	39	Overridable method 'createUser' called during object construction
Avoid branching statement as last in loop	JWTLessonTest	getSecretToken	Medium High	72	Avoid using a branching statement as the last in a loop.
Constructor calls overridable method	MD5	MD5	High	39	Overridable method 'reset' called during object construction
Return empty array rather than null	JWTFinalEndpoint	resolveSigningKeyBytes	High	164 - 175	Return an empty array rather than null.
Avoid branching statement as last in loop	JWTFinalEndpoint	resolveSigningKeyBytes	Medium High	169	Avoid using a branching statement as the last in a loop.
Constructor calls overridable method	DisplayUser	DisplayUser	High	54	Overridable method 'genUserHash' called during object construction
Constructor calls overridable method	WebGoatUser	WebGoatUser	High	61	Overridable method 'createUser' called during object construction

This report was generated by MyCodeAnalyzer. Vulnerability scan was conducted on 2021-02-20T20:57Feb 20, 2021 8:57:15 PM net.sourceforge.pmd.PMD encourageToUseIncrementalAnalysis WARNING: This analysis could be faster, please consider using Incremental Analysis: https://pmd.github.io/pmd-6.31.0 /pmd_userdocs_incremental_analysis.html :15.921

Figure 9.
The report generated by MyCodeAnalyzer and emailed to the user after scanning the OWASP WebGoat project.

into a more easily understood report that captures only pertinent information. These results demonstrate the applicability of using a framework backed by virtual assistants to scan code for vulnerabilities and generate meaningful reports.

6.2 Challenges

It is important to outline some challenges with the use of VAs for code analysis and mitigation of vulnerabilities. The main challenge with this new approach to code analysis is adoption. A recent study involving a small sample of participants shows that currently the primary use of VAs are for music procurement (40% of users), for information (17%), and automation (9%) [40]. Since this is a new avenue of research, there may be initial challenges with adoption in the code analysis arena. However, we believe that as the market grows and coders get exposed to this technology, the adoption rates will increase. Researchers predict a growing use for digital voice assistants over the next few years [41, 42].

Another challenge with using the PnP model discussed in this research is handling the differences between output reports from different code analyzers. To mitigate this issue, the code analysis community may require standardization of vulnerability reports in popular formats such as XML, JSON, and HTML. Currently, most tools include information such as files, classes, and line numbers where errors are found. While the output formats may be different, NLP techniques such as NER can also be used to mine these reports for key pieces of information to achieve a

standard format that can be handled by the virtual assistant and the proposed analysis framework.

7. Conclusion

Getting programmers to write secure code remains a challenge. Security is often sacrificed in an effort to add a feature to a software product or to meet a deadline. When security is sacrificed for other gains, the end result is a product riddled with bugs or vulnerabilities. Steps must be taken to encourage programmers to produce more secure software. In this research, we discussed the limitations of existing code analysis approaches and propose a framework that allows programmers to use virtual assistants to conversationally scan and fix potential vulnerabilities in their code. Virtual assistants are becoming popular in everyday activities such as procuring and listening to music, finding places of interest, managing a smart home, shopping, etc. We posit that as they become more mainstream, they can be used to manage code analysis while keeping programmers productive. We implement our proposed methodology using the Google Assistant and demonstrate its utility in an effort to find new, creative ways to help programmers produce more secure software. Future work will involve extending the model to use any applicable code analyzer based on a plug-and-play paradigm, adding data analytics and visualizations to help programmers draw insights from their code, implementing the refactoring and auto-fixing modules, and conducting a user study to evaluate the framework.

Abbreviations

DAST	Dynamic application security testing
IAST	Interactive application security testing
NLP	Natural language processing
PnP	Plug-and-play
SAST	Static application security testing
SCAD	Synchronous control asynchronous dataflow

Author details

Fitzroy Nembhard* and Marco M. Carvalho
Florida Institute of Technology, Melbourne, Florida, USA

*Address all correspondence to: fitzroy@ieee.org

References

[1] Bruce Schneier. Secrets & lies: Digital security in a networked world new york: Wiley computer publishing. 2000. *Ch*, 16:245–246.

[2] Iso/iec/ieee international standard - systems and software engineering–vocabulary. *ISO/IEC/IEEE 24765:2017 (E)*, pages 1–541, 2017. doi: 10.1109/IEEESTD.2017.8016712.

[3] F. Nembhard, M. Carvalho, and T. Eskridge. A hybrid approach to improving program security. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, 2017. doi: 10.1109/SSCI.2017.8285247.

[4] 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.

[5] B. Chess and G. McGraw. Static analysis for security. *IEEE Security Privacy*, 2(6):76–79, Nov 2004.

[6] Brian Chess and Jacob West. *Secure programming with static analysis*. Pearson Education, 2007.

[7] T. Muske and A. Serebrenik. Survey of approaches for handling static analysis alarms. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 157–166, Oct 2016.

[8] Fitzroy D. Nembhard, Marco M. Carvalho, and Thomas C. Eskridge. Towards the application of recommender systems to secure coding. *EURASIP Journal on Information Security*, 2019(1):9, 2019. doi: 10.1186/s13635-019-0092-4. URL https://doi.org/10.1186/s13635-019-0092-4.

[9] Andreas M Klein, Andreas Hinderks, Maria Rauschenberger, and Jörg Thomaschewski. Exploring voice assistant risks and potential with technology-based users. In *Proceedings of 16th International Conference on Web Information Systems and technology (WEBIST)*, pages 1–8, 2020.

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

[11] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36. ACM, 2015.

[12] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1284–1291, New York, NY, USA, 2012. ACM.

[13] Fitzroy Nembhard. *A Recommender System for Improving Program Security Through Source Code Mining and Knowledge Extraction*. PhD thesis, Florida Institute of Technology, 2018.

[14] Omer Tripp, Pietro Ferrara, and Marco Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 49–59, 2014.

[15] Nir Piterman, editor. *Hardware and Software: Verification and Testing: 11th*

International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings. Springer International Publishing, Cham, 2015.

[16] Yuancheng Li, Rong Ma, and Runhai Jiao. A hybrid malicious code detection method based on deep learning. *International Journal of Security and Its Applications*, 9(5):205–216, 2015.

[17] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. Sapfix: Automated end-to-end repair at scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 269–278, 2019. doi: 10.1109/ICSE-SEIP.2019.00039.

[18] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 3–11, Cham, 2015. Springer International Publishing.

[19] Anoop Bhagyanath. *Code Generation for Synchronous Control Asynchronous Dataflow Architectures*. PhD thesis, Technical University of Kaiserslautern, 2021.

[20] Jonas Austerjost, Marc Porr, Noah Riedel, Dominik Geier, Thomas Becker, Thomas Scheper, Daniel Marquard, Patrick Lindner, and Sascha Beutel. Introducing a virtual assistant to the lab: A voice user interface for the intuitive control of laboratory instruments. *SLAS TECHNOLOGY: Translating Life Sciences Innovation*, 23(5):476–482, 2018.

[21] T. Muske and A. Serebrenik. Survey of approaches for handling static analysis alarms. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 157–166, 2016. doi: 10.1109/SCAM.2016.25.

[22] Anjana Gosain and Ganga Sharma. Static analysis: A survey of techniques and tools. In Durbadal Mandal, Rajib Kar, Swagatam Das, and Bijaya Ketan Panigrahi, editors, *Intelligent Computing and Applications*, pages 581–591, New Delhi, 2015. Springer India.

[23] G. Lin, S. Wen, Q. L. Han, J. Zhang, and Y. Xiang. Software vulnerability detection using deep neural networks: A survey. *Proceedings of the IEEE*, 108(10): 1825–1848, 2020. doi: 10.1109/JPROC.2020.2993293.

[24] F. Nembhard and M. Carvalho. The impact of interface design on the usability of code analyzers. In *2019 SoutheastCon*, pages 1–6, 2019. doi: 10.1109/SoutheastCon42311.2019.9020339.

[25] 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. ISBN 978-1-4673-3076-3.

[26] 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. doi: 10.1145/1029894.1029909.

[27] Andreas Holzinger, Peter Kieseberg, Edgar Weippl, and A. Min Tjoa. Current advances, trends and challenges of machine learning and knowledge extraction: From machine learning to explainable ai. In Andreas Holzinger, Peter Kieseberg, A Min Tjoa, and Edgar Weippl, editors, *Machine Learning and*

Knowledge Extraction, pages 1–8, Cham, 2018. Springer International Publishing.

[28] Paula Venosa, Sebastian Garcia, and Francisco Javier Diaz. A better infected hosts detection combining ensemble learning and threat intelligence. In Patricia Pesado and Marcelo Arroyo, editors, *Computer Science – CACIC 2019*, pages 354–365, Cham, 2020. Springer International Publishing.

[29] Ngoc Tu Pham, Ernest Foo, Suriadi Suriadi, Helen Jeffrey, and Hassan Fareed M Lahza. Improving performance of intrusion detection system using ensemble methods and feature selection. In *Proceedings of the Australasian Computer Science Week Multiconference*, ACSW '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450354363. doi: 10.1145/3167918.3167951. URL https://doi.org/10.1145/3167918.3167951.

[30] S. A. Ludwig. Intrusion detection of multiple attack classes using a deep neural net ensemble. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–7, 2017. doi: 10.1109/SSCI.2017.8280825.

[31] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*, pages 137–149. Springer, 2016.

[32] Daniel Gibert, Carles Mateu, and Jordi Planes. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications*, 153:102526, 2020. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2019.102526. URL https://www.sciencedirect.com/science/article/pii/S1084804519303868.

[33] F. Nembhard and M. Carvalho. The impact of interface design on the usability of code analyzers. In *2019 SoutheastCon*, pages 1–6, 2019. doi: 10.1109/SoutheastCon42311.2019.9020339.

[34] Dialogflow, 2021. URL https://cloud.google.com/dialogflow/docs. Accessed: 2021-02-19.

[35] Spring boot, 2021. URL https://spring.io/projects/spring-boot. Accessed: 2021-02-19.

[36] Apache tomcat, 2021. URL http://tomcat.apache.org/. Accessed: 2021-02-19.

[37] PMD. Pmd source code analyzer project, 2021. URL https://pmd.github.io/. Accessed: 2021-02-18.

[38] Owasp webgoat, 2021. URL https://owasp.org/www-project-webgoat/. Accessed: 2021-02-19.

[39] ngrok. Ngrok, 2021. URL https://ngrok.com/. Accessed: 2021-02-19.

[40] Frank Bentley, Chris Luvogt, Max Silverman, Rushani Wirasinghe, Brooke White, and Danielle Lottridge. Understanding the long-term use of smart speaker assistants. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2 (3):1–24, 2018.

[41] Sven Tuzovic and Stefanie Paluch. Conversational commerce–a new era for service business development? In *Service business development*, pages 81–100. Springer, 2018.

[42] Andreas M Klein, Andreas Hinderks, Maria Rauschenberger, and Jörg Thomaschewski. Exploring voice assistant risks and potential with technology-based users. In *Proceedings of 16th International Conference on Web Information Systems and technology (WEBIST)*, pages 1–8, 2020.