# Global Information Assurance Certification Paper

# Analyzed Java Code Snippets: The Corpus

*GIAC GWAPT Gold Certification*

Author: Hitarth Patel, contact@hitarthpatel.com
Advisor: *Jonathan Risto*

Accepted: August 19, 2021

Abstract

Static Code Analysis is a way to find vulnerabilities in source code. However, this process is flawed due to the significant amount of false-positive findings that take additional time and resources to address, taking away from remediating actual vulnerabilities. For this research, a corpus was created to begin the process of developing a machine learning model that could potentially weed out false positives. Due to the lack of datasets available for this project, it is focused on the process of developing the datasets that could feed the machine learning model. This paper is a blueprint for future research and improvement of processes to find vulnerabilities without false positives.

# 1. Introduction

Static Code Analysis or Source Code Analysis (SCA) refers to the ability to discover vulnerabilities via Taint Analysis and Data Flow Analysis in static (non-running) source code ("Static Code Analysis," n.d.). Software developers and security engineers use SCA tools to scan source code, identify vulnerabilities, analyze the results to eliminate false-positive findings and address potential vulnerabilities. Many SCA tools can be integrated into an organization's Software Development Life Cycle (SDLC), enabling developers to take early action and produce secure applications. It is important to note that taking early action on these vulnerabilities proves to be the cheapest solution for developers long term, in addition to allowing them to address and manage risks effectively (Koc, 2019).

Despite SCAs ability to run independently, a significant issue often encountered is the amount of "false-positive" findings that developers and engineers must deal with (Ayewah, 2010). A false-positive finding occurs when SCAs incorrectly identify a vulnerability that is not present. A willing trade-off that these tools have is to make the analyses faster, increasing "false-positive" findings (Koc, 2019). Therefore, fast scan requisites arise, perpetuating the need for development teams to reduce false-positive friction (Wisseman, n.d.). Security and development teams must exhaust valuable resources to analyze findings in order to eliminate potential false-positives and to facilitate the count of false findings. The demand for such investments ultimately results in teams being disinclined to utilize SCA.

The motivation to create the proposed dataset arose as a need for machine learning research aiming to support the prediction of false-positive findings using source code. A binary classification model (AutoAudit) or a neutral network conceptualized to identify false-positive findings will work with the Java corpus created in this research to identify false-positive vulnerabilities or vulnerabilities. Additionally, the corpus could be used in other aspects such as generating secure code based on the category, automatically

providing remediated solutions to known vulnerabilities, and identifying potential secure code corrections based on previously learned patterns.

## 2. Fortify Project Reports

Fortify Static Code Analyzer is a tool provided by MicroFocus CyberRes. The SCA tool is utilized to detect potential vulnerabilities within source code which helps development teams resolve vulnerabilities before pushing code into production. The project produces a Fortify Project Report (FPR) once scanned. The FPR is usually located at "path/to/project/target/fortify/<project-name>.fpr." FPR files archive various XML files containing information about each finding within the FPR and directories containing every Java file with potential vulnerabilities. For future reference, it is important to note that in addition to Java, Fortify SCA also supports many other languages such as ASP.NET, C/C++, C#, ColdFusion, JSP, PL/SQL, T-SQL, XML, VB.NET, and other .NET languages. Due to the extensibility to other programming languages, it is possible to create datasets using similar methods in any of the languages supported by Fortify SCA.

Fortify SCA is used to discover and fix the following programming weaknesses at the root cause: Buffer Overflow, Command Injection, Cross-Site Scripting, Denial of Service, Format String, Integer Overflow, Log Forging, Password Management, Path Manipulation, Privacy Violation, Race Conditions, Session Fixation, SQL Injection, System Information Leak, and Unreleased Resource. It sorts, filters, and categorizes these issues found in different structures, making it simple to survey and examine through the Audit Workbench. The outcomes can likewise be sent out as a report in configurations like HTML and XML (D' Souza, 2007). This facilitates collaboration and an easy understanding of the weaknesses that need to be addressed.

### 2.1. False Positive Rates

SCAs must deal with the rate of false-positive findings within their reports. Usually, many SCAs will try to combat this by improving upon whichever method they

prefer to identify possible exploits. For instance, Fortify SCA uses a collective form of analysis containing five different analyzers: Data Flow, Semantic, Control Flow, Configuration, and Structural (D'souza, 2007). Additionally, Fortify SCA also checks for secure coding practices using rulepacks. There are continuous updates that Fortify maintains for rulepacks as well the Fortify Global Analyzer to combat the rate at which false-positive findings are identified. Moreover, the rate of false-positive findings also reflects on the size of the codebase and development practices used by the developers.

## 2.2. Audit Workbench

The Fortify Project Report file can only be opened by Audit Workbench for analysis. The software provides an interface to perform analysis as well as preserve historical data for the project. In Figure 1, the "Details" tab will provide information on why the finding was alerted and information displaying what the category identified is. Additionally, the "Recommendations" tab allows the analyst to get exposed to possible remediations for the specific category. The "Analysis Trace" allows the analyst to follow the stack trace on where the vulnerability is and the possible path taken by the vulnerability. This helps the analyst provide an educated analysis report of any finding. Moreover, the analyst can create a historical record of each finding along with the remediation comment in the "Audit" tab, as shown in Figure 2.

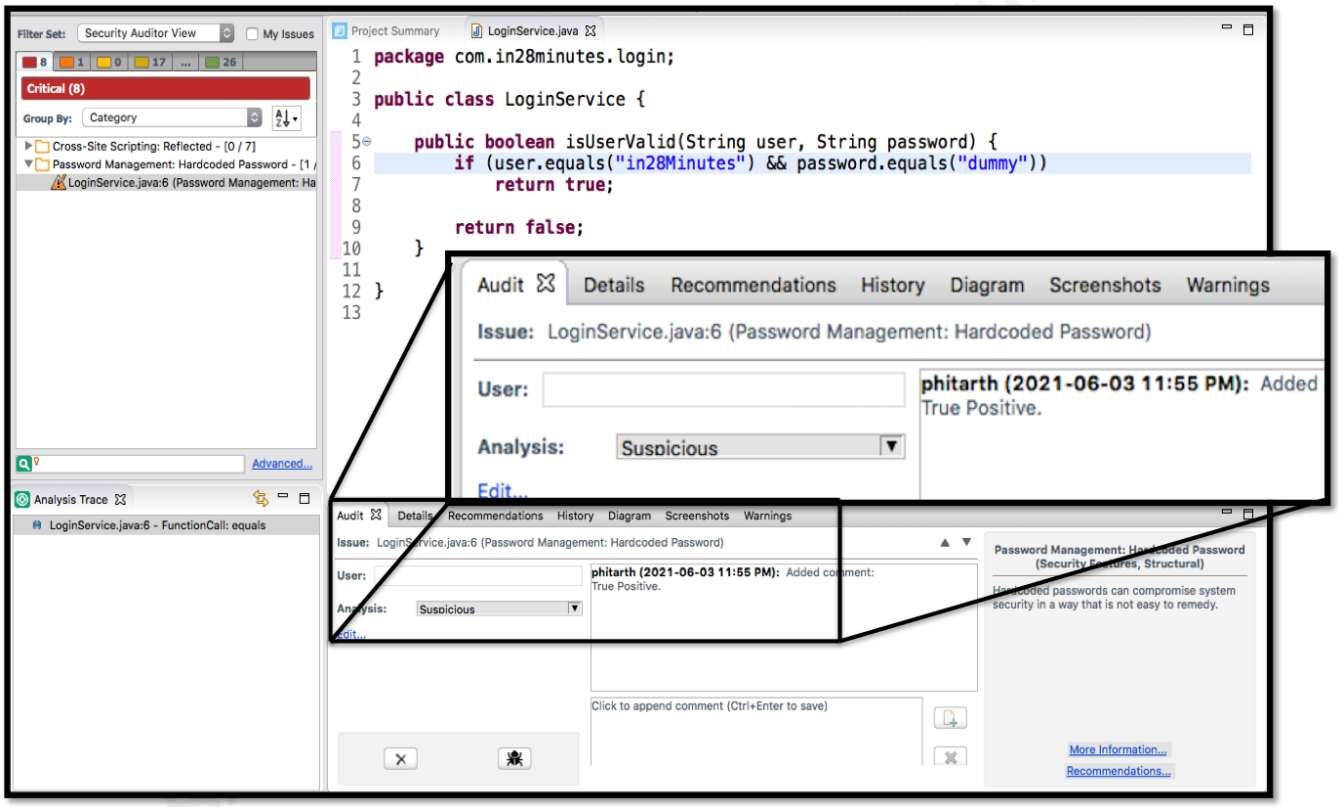


**Figure 1. Example of the "Audit Workbench" software from Fortify**

*Notes.* Audit Workbench is a software tool provided by Fortify that allows security analysts to perform analysis to identify false-positive findings and confirm legitimate security vulnerabilities.

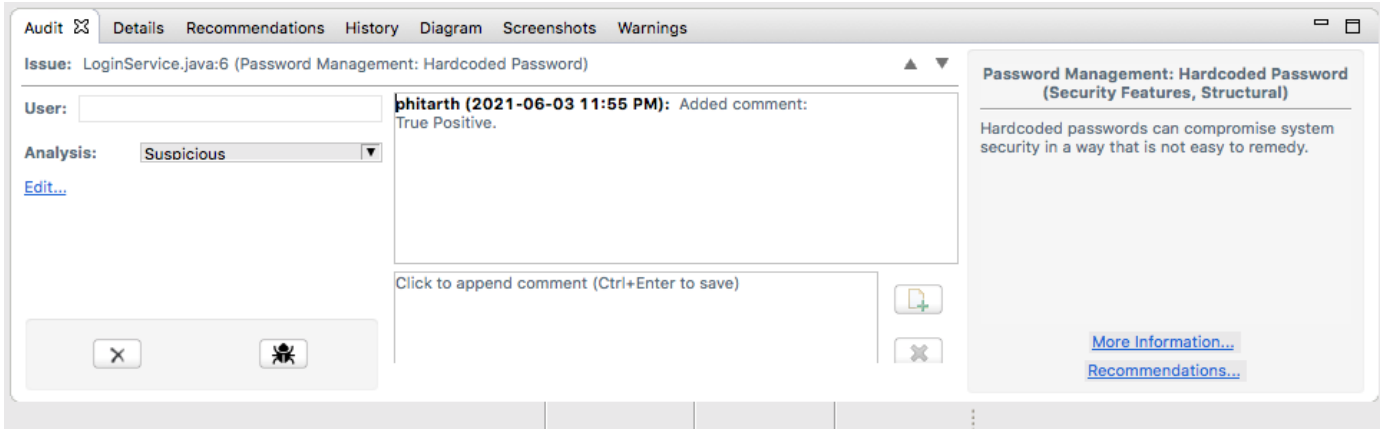


**Figure 2. Audit workbench audit view**

Hitarth Patel, contact@hitarthpatel.com

*Notes.* The "Analysis" dropdown menu can be changed per analysis and is one of the mined data points when parsing an FPR.

## 3. Dataset Creation

To predict whether a finding is true or false based on source code, it would require a very fine-tuned dataset that highlights where the taint is within the source code. For a problem identifying whether a piece of code is exploitable or not, a labeled dataset needs to be created to be ingested by an agnostic model. The code should be labeled correctly (e.g., "Not a Finding" or "Suspicious") since incorrect labeling could lead to unexpected predictions by the model. Various open-source Java projects were used to create a labeled database to approach this regardless of the number of stars or forks; furthermore, the first iteration of projects scrapped from GitHub had to be built using maven, and this is because the only available tool to produce the data is the maven plugin from Fortify. The crawler script was revised to download all Java web application projects focused on most stars to least. This revision resulted in a total of 14,092 projects that could be scanned to increase the size of the dataset. Additionally, the mavenFortifyScan.py script was revised to allow for scanning projects that do not utilize maven resulting in an increase in the number of projects downloaded.

Due to the lack of publicly available datasets that contain analyzed code snippets for Java Projects, the paper presents a newly curated corpus containing 221 projects scraped from GitHub, composed of 96,928 java source code files. Using Fortify Static Code Analyzer, every application was scanned to generate a Fortify Project Report. These FPRs contain categorized findings that are analyzed to determine false-positive or suspicious findings. Due to the complexity and time constraints, the analysis was performed on only a few selected code quality categories:

- Password Management: Hardcoded Password
- Hardcoded Password
- Weak Cryptography

- Empty Password
- Password in Comment
- Missing Check for Null Parameter
- Access Control Database

Furthermore, the specific categories in this research resulted in a smaller than significant dataset than expected. Even in open-source software, the importance of password management is prioritized, which results in very few projects having this vulnerability flagged. The dataset aggregation process was integrated into a core tool called "AutoAudit." AutoAudit is a classifier that will identify false-positive findings based on the produced FPR for a project as well as perform multiple utility functions such as dataset preparation and FPR parsing. After all the datasets that met the requirement for AutoAudit were analyzed and labeled, the results were compiled in a file and set in the Attribute-Relation File Format (ARFF), which is a numerical form that can be used in the Waikato Environment for Knowledge Analysis (WEKA) classifiers for future direct use by security researchers.

## 3.1. Mining Repositories - The Process

Figure 3 helps visualize the process explained step-by-step in Figure 4, developed for easy reproducibility. Section 3.2 and 3.3 will provide a brief and condensed overview of each step listed below.

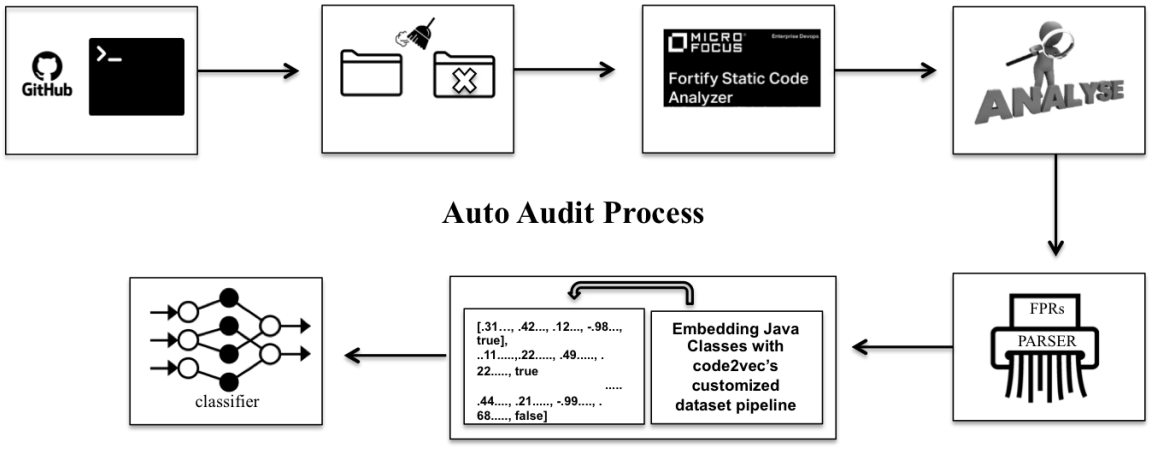

**Figure 3. Visualization of the dataset collection and preparation.**

| Step | Task |
|---|---|
| 1 | Use the "Crawl" script to download repositories using GitHub API.<br>Script: Crawler/Crawl.py |
| 2 | Clean up invalid projects/ projects with corrupted zips.<br>Script: Crawler/Crawl.py |
| 3 | Use Fortify SCA maven plugin and a custom script to scan all projects and move all ".fpr" files produced to a common directory "FPRs."<br>Script: mavenFortifyScan.py |
| 4 | Manually analyze all FPRs for the dataset.<br>Script: Human. |
| 5 | Use the FPRParser script to parse all analyzed FPRs for all issues and extract all source code files associated with any actual or false findings for dataset creation.<br>Script: FPRParser/FPRParser.py |
| 6 | Use the extracted source files in the obfuscated-code2vec dataset pipeline (Compton, 2020) and convert each source code file in .arff<br>Script: obfuscated-code2vec/pipeline/create_datasets.sh<br>GitHub: https://github.com/basedrhys/obfuscated-code2vec |

**Figure 4. Dataset creation process**

The first task in the application was to crawl and identify potential projects to download from GitHub API. The script specifically looked for Java applications utilizing the Maven project management tool. The script found a total of 221 total potential projects.

The next task was to automatically identify all scannable project locations to feed into the "mavenFortifyScan.py" script. This involved identifying the parent pom.xml file and moving the directory containing the file to another folder. The script also removed any duplicate projects.

The directory containing the scannable projects was then iterated by the "mavenFortifyScan.py" script. It used the Maven Fortify SCA plugin to scan all projects and separate projects that did not scan successfully. This resulted in two more directories containing scanned projects and projects that produced errors during scanning. Furthermore, another directory was generated with all the FPRs generated from successful scans.

Next, an intermediate step requiring human intervention was implemented. A human analyzed the generated FPRs to produce accurate labels for each finding. Doing this allowed for easy extraction of source code and its corresponding analysis.

Once the FPRs were analyzed, the directory containing all the FPRs was iterated, and each FPR was parsed using the "FPRParser.py" script. The parser gathered all necessary information from the FPR and created a finding object to keep track of each finding, its analysis, and the related source code snippet file location. The parser was also used to fetch any issue, whether analyzed or not, adding versatility to the tool. Moreover, the script extracted all source code files to their corresponding analysis, aggregating source code for all false-positive detections in one directory.

Once all the FPRs were parsed, the final step produced the ARFF format for our source code to prepare it for any machine learning model. To produce the ARFF files for

the dataset, the obfuscated-code2vec dataset pipeline was used, as shown in Figure 5.

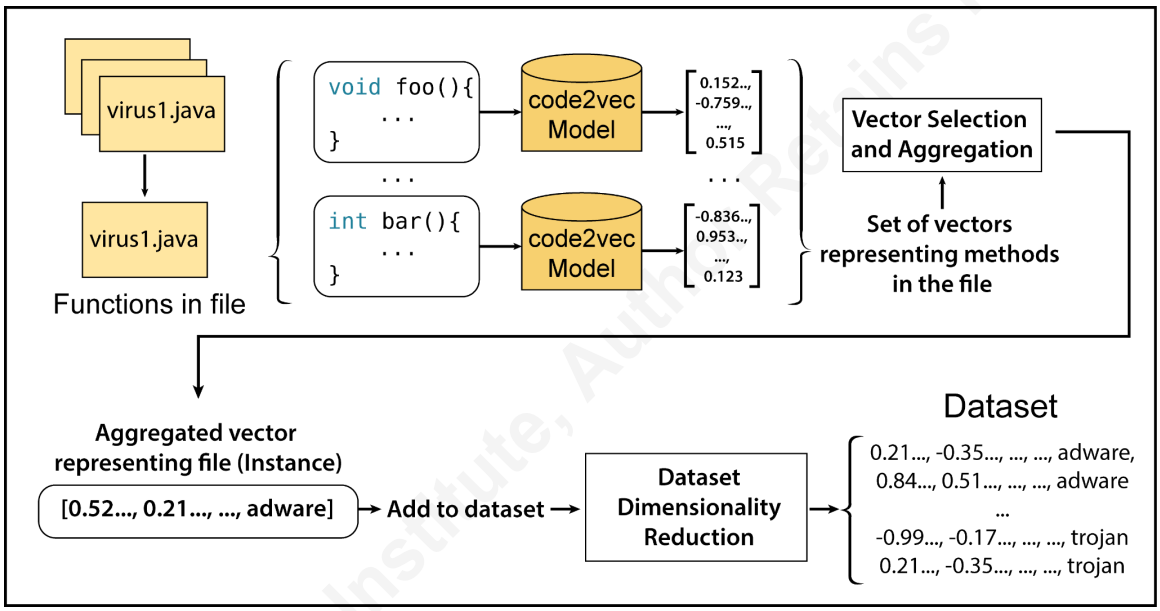


**Figure 5. Dataset Pipeline from Obfuscated-Code2Vec**

*Notes.* This figure was produced by Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay for their research paper Obfuscated-Code2Vec. The repository uses MIT licensing. From Compton, R., Frank, E., Patros, P., & Koay, A. (2020, April 6). *Embedding Java Classes with code2vec: Improvements from Variable Obfuscation*. arXiv.org e-Print archive. https://arxiv.org/pdf/2004.02942.pdf

The source code files, divided by classes, were fed into the dataset pipeline. It utilized the code2vec model to create vectors for each source code file. The pipeline ingested the source code file, parsed it, separated each method within the class, and produced vectors. The vectors were then aggregated into a vector representing the entire code snippet file. After all classes were converted into vectors, the vectors were aggregated into a dataset that could be utilized in any WEKA classifier or machine learning model supporting the input, as seen in Figure 3.

## 3.2. Identify Valid Projects

To ensure that the new dataset contains pertinent information, the GitHub Crawler must define the requirements before downloading projects needed to scan. The GitHub

Crawler uses GitHub's REST API to search, filter, and download Java projects using maven. The crawler looks for any project containing the "pom.xml" file needed to build any maven project. The script will then download the projects to a directory called "extractedApps" under "../data/extractedApps". For this research, the crawler found a total of 221 valid maven projects to scan, of which 15 projects were not successfully scanned due to corrupt zip archives downloaded from GitHub. Upon further research, it was found that some scenarios, such as a large zip archive over the size of 1 GB, result in Github corrupting the zip archive.

## 3.3. Scanning and Analysis

To scan any Java project, the SCA follows four distinct phases as described in the latest Fortify SCA documentation (Micro Focus, n.d.). The initial phase is build integration. The Fortify Maven plugin allows the SCA to be integrated within the maven clean, install, translate, build phases (Microfocus, n.d.). This allows for SCA to scan the project in any of the lifecycle phases. Following the build, the integration phase is the translation phase. The source code is translated into an intermediate format associated with the "buildId," usually the project name. Following the translation phase is the scanning phase, where the translated source code is scanned and the phase where the FPR file is generated. The last phase verifies that the source code was scanned using the correct rulepacks and ensures no errors were returned. The pipeline is fully automated, allowing for projects to be scanned automatically once cloned from GitHub; moreover, the scans usually run between 15 minutes to 30 minutes, depending on the size of the project. As a result of a successful scan, the scanner will create a Fortify Project Report containing all identified vulnerabilities that an analyst will analyze for false-positive findings.

Due to how findings are identified within an FPR, there is an excellent chance that false-positive findings will be present. In order to reduce the rate of false-positive findings, each finding has to be analyzed manually. Furthermore, if continuous integration is implemented, then analyzing findings becomes tedious and time-

consuming. Depending on the size of the projects, an analyst would have to go over thousands of findings to weed out false-positives automatically identified by the scanner.

## 4. The GitHub Java Corpus

The dataset contains all projects downloaded from GitHub using GitHub API. GitHub API was chosen as a method of mining due to its availability and ease of use. It should be noted that there was a rate limit of 5,000 requests per hour that would be inconvenient for a larger dataset and project; however, it did not cause any disruption on the query used by the crawler script. Therefore, it may be something to keep in mind when mining for more projects. Another reason why GitHub was the only source for code is that it has the most extensive software projects. The number of projects used in the dataset creation process was limited by the sole use of GitHub, a source for projects that are usually tools or frameworks.

Additionally, the quality of the projects was not considered, as noted in the Blincoe paper regarding the promises and perils of mining GitHub (Blincoe, 2014). In total, 221 projects were collected. Out of the 221 projects, only 206 projects were scanned without producing any errors and contained 96,928 files and 686,858 lines of code. A total of 107 FPRs were analyzed to produce a total of 51 findings that were of some variety of the aforementioned categories in section 3. In total, Fortify SCA found a total of 2,466 findings across all scanned projects.

Furthermore, the dataset is in the .arff (Attribute-Relation File Format) as shown in Figure 6, so it can easily be used in a WEKA classifier without much data preparation.

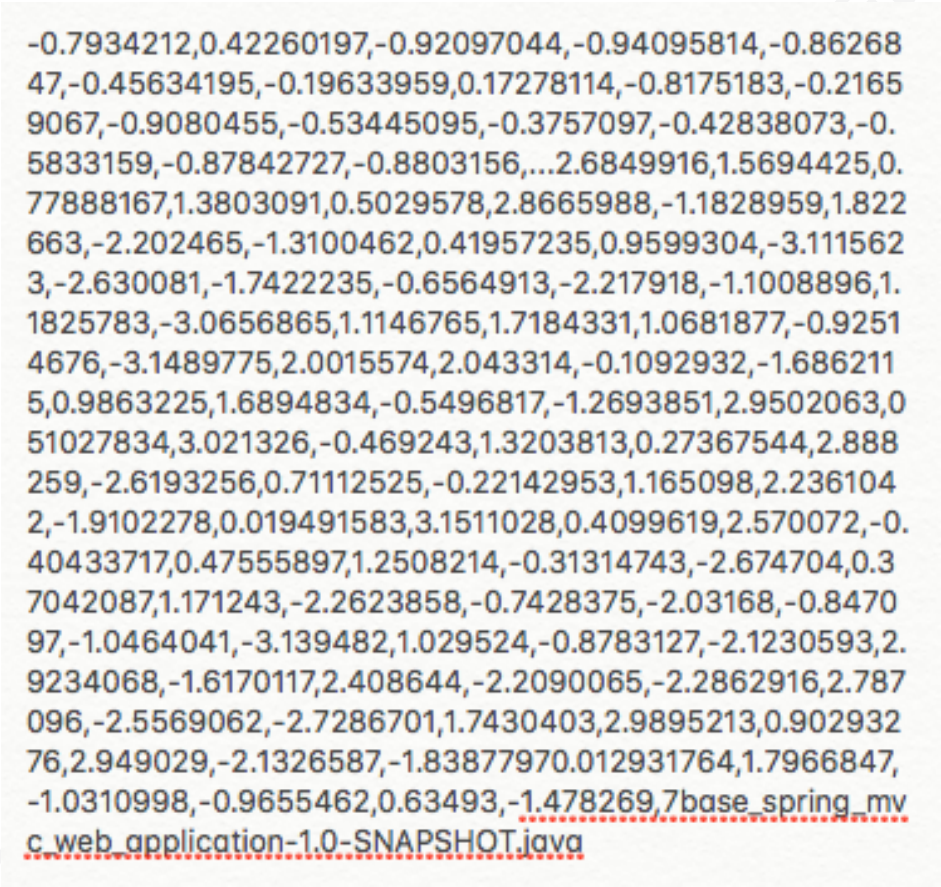


**Figure 6. A sample condensed aggregated class vector can be found within a .arff file.**

*Notes.* The figure represents an aggregated class "7base_spring_mvc_web_application-1.0-SNAPSHOT.java" vector that can be used as an input in machine learning models. The dataset .arff file contains all aggregated class vectors.

## 5. Recommendations and Implications

Since development is usually a human-based process, there will always be insecure code written, which will require the security community to work together to reduce the number of false-positive findings. Due to the lack of public datasets that characterize categorical findings with static source code analysis, the aim was to create a dataset containing analyzed findings or bugs labeled per category as true-positive or

false-positive. There will be an imbalance of data regarding the ratio of false-positive findings to true-positive findings in any natural Java project. Considering the imbalanced data within the corpus, the dataset attempted to balance the findings evenly between the two classes, “Not An Issue” and “Suspicious,” as labeled in Fortify Audit Workbench Figure 2. This corpus can provide researchers real value by offering a balanced analyzed dataset per category for their new defect prediction experiments.

## 5.1. Implications for Future Research

The dataset only consists of one broad category, Password Management, which enables speedy analysis. The complexity of findings and categories such as, among others, SQL Injection, Cross-Site Scripting, and Access Control are proportional. The time would scale according to the complexity of the analysis. Ideally, time and team effort would allow more datasets to be collected to find a significant amount of data to feed to a machine learning model. The findings may have input taints from different classes such as controllers or microservices that an analyst would need to consider when evaluating findings.

In the future, there would be a need to change the “JavaExtractor" used by the Obfuscated-code2vec model (Compton, 2020) to capture the complex semantics of the code. Due to the simplicity of the Password Management category, the needed information can usually be captured within the same class where the taint is identified. Additionally, a couple more categories, such as Access Control: Database, and Missing Check for Null Parameter, were analyzed to boost the size of the dataset. Moreover, the categories analyzed attempt to ensure that the vulnerability is captured from the originating source file allowing for capturing the complete semantics of the code. The Access Control: Database was a complex finding category. The analyst was required to ensure that the controller, or wherever the input was provided, took into account which user policy or permission is required in order to execute the method. The semantics for this category involves multiple code snippets that need to have a relationship defined in order to help the classifier predict more accurately. Therefore, Compton's JavaExtractor

used in the Obfuscated-code2vec model was not modified and used as a transfer learning model.

Considering that the dataset is minimal, it would make sense to expand into other code quality categories to increase the size of the dataset. Some possible categories that can be tackled are the "Dead Code" category, which consists of identifying findings that have unused variables, fields, or methods; "Poor Style" category, which consists of identifying findings where the values of variables are never read; "Confusing Naming," where a variable and a method will both have the same name; and "Poor Error Handling" category, which consists of empty catch blocks or overly broad catch or throws when it comes to exceptions. These few categories can help increase the size of the dataset; however, the problem with imbalanced data would still have to be tackled. There is no guarantee that each category will have an even number of false-positive to true-positive ratios.

Lastly, the process calls for maintenance for the dataset, as well as growing the dataset, since small datasets often result in overfitting, meaning that the machine learning model will learn the training data too well and thus perform poorly when new data is presented. In order to prevent the model from overfitting, the size of the dataset must build over time.

## 6. Conclusion

The lack of publicly available datasets containing analyzed Java code snippets that correspond to a specific secure code category made it imperative to create a corpus that tackled this gap since the need for this dataset arose when trying to develop a machine learning model that could identify false-positive findings from code snippets. In order to produce the dataset, a custom tool, AutoAudit, was developed to crawl, clean, and prepare the data in the ARFF file format, which allowed for direct dataset use with WEKA classifiers. The method presented attempted to provide balanced data as an easy-to-use, plug-and-play dataset. To develop a significant corpus, time and resources would

play a significant role. Moreover, the dataset can aid in developing a machine learning model to identify false-positive findings. The successful development of a fully functioning model would allow experts to redirect their resources to address actual vulnerabilities. Furthermore, the dataset can be used for future research regarding false-positive finding prediction and automatically generated secure code.

# 1. References

Ayewah, N. (2010). Static Analysis in Practice. drum.lib.umd.edu. https://drum.lib.umd.edu/bitstream/handle/1903/10934/Ayewah_umd_0117E_11597.pdf?sequence=1

Blincoe, K., Kalliamvakou, E., Singer, L., Gousios, G., German, D., & Damian, D. (2014, May 31). *The Promises and Perils of Mining GitHub*. https://kblincoe.github.io/publications/2014_MSR_Promises_Perils.pdf

Compton, R., Frank, E., Patros, P., & Koay, A. (2020, April 6). *Embedding Java Classes with code2vec: Improvements from Variable Obfuscation*. arXiv.org e-Print archive. https://arxiv.org/pdf/2004.02942.pdf

D'souza, D. (2007, April 24). *TOOL EVALUATION REPORT: FORTIFY*. Carnegie Mellon School of Computer Science |. https://cs.cmu.edu/~aldrich/courses/654-sp07/tools/dsouza-fortify-07.pdf

Koc, U. (2019). *Improving the Usability of Static Analysis Tools Using Machine Learning* [Doctoral dissertation]. https://drum.lib.umd.edu/handle/1903/25464

Micro Focus. (n.d.). Using Micro Focus Fortify Static Code Analyzer. Digital Transformation and Enterprise Software Modernization | Micro Focus. Retrieved August 6, 2021, from https://www.microfocus.com/documentation/fortify-static-code-analyzer-and-tools/2110/SCA_Help_21.1.0/index.htm#Resources/HTMLelements/Title_Page.htm?TocPath=_____1

Microfocus. (n.d.). *Fortify integration ecosystem - MAVEN | Micro focus*. Digital Transformation and Enterprise Software Modernization | Micro Focus. Retrieved July 2021, from https://www.microfocus.com/en-us/fortify-integrations/maven

*Static code analysis*. (n.d.). OWASP Foundation | Open Source Foundation for Application Security. Retrieved June 7, 2021, from https://owasp.org/www-community/controls/Static_Code_Analysis

Wisseman, S. (n.d.). *In-depth analysis comes at a cost – so does a breach!* Micro Focus Community. Retrieved June 7, 2021, from

https://community.microfocus.com/cyberres/b/sws-22/posts/in-depth-analysis-comes-at-a-cost-so-does-a-breach

# Appendix

## Project Source Code and Help Docs

To follow the progress of the AutoAudit tool, please refer to the GitHub repository below.

https://github.com/Gitarth/autoaudit

## The Attribute Relation File Format (.arff) dataset

For access to the created dataset, please refer to the Google Cloud Storage link below.

https://storage.googleapis.com/autoauditdataset/autoauditV0.1.zip