Research Article **IJNSR (2021) 6:16**

International Journal of Natural Science and Reviews
(ISSN:2576-5086)

A SonarQube Static Analysis of the Spectral Workbench

Ifeanyi Rowland Onyenweaku[1*], Michael Scott Brown[2], Michael Pelosi[3], M. H. Shahine[1]

[1]Harrisburg University of Science and Technology, Harrisburg, Pennsylvania, United States.
[2]University of Maryland Global Campus, Adelphi, Maryland, United States.
[3]Texas A&M Texarkana, Texarkana, Texas, United States.

ABSTRACT

The Spectral Workbench is an open-source, community driven software suite to obtain and disseminate spectral data. It consists of a client application that collects spectral readings and a server application that is an online database of spectral data. It is difficult to detect software defects in the Spectral Workbench application. A static analysis tool, SonarQube, was selected to find these defects. Numerous defects were detected and documented. SonarQube will increase the reliability of the Spectral Workbench, which provides numerous benefits including increased confidence in its data and effectiveness which will drive additional number of users for spectral repository data collection.

Keywords: SonarQube; Static analysis; Defects; Security; Spectral Workbench

***Correspondence to Author:**
Ifeanyi Rowland Onyenweaku
Harrisburg University of Science and Technology, Harrisburg, Pennsylvania, United States.

How to cite this article:
Ifeanyi Rowland Onyenweaku, Michael Scott Brown, Michael Pelosi, M. H. Shahine. A SonarQube Static Analysis of the Spectral Workbench. International Journal of Natural Science and Reviews, 2021; 6:16.

eSciPub LLC, Houston, TX USA.
Website: https://escipub.com/

1. Introduction

The Spectral Workbench [1] is an open-source software suite. Anyone can use a spectrometer or simple cell phone to collect spectral (light Wavelength spectrum) data using an application. Then it can be uploaded it to the second application in the suite - the web-based repository. The publicly available repository is used for research and education. Currently there are over 100,000 recordings in the repository that range from argon to a leaf.

It is difficult to detect defects in software [2]. The Spectral Workbench is not immune to this problem. Software is very complex and introducing defects is common. Normally software development teams use a variety of techniques to locate defects and resolve them, but defects still make it into production software.

Over the past decades numerous methods to detect defects have been developed. All methods can be divided into Static and Dynamic Analysis [3]. Static analysis only looks at the source code while Dynamic analysis executes the program. Static analysis is a popular method of detecting software defects [3].

The approach to this research was to conduct static analysis on the JavaScript Project of Spectral Workbench. This JavaScript program is used to record, manipulate, and analyze spectral data. This program is built on JavaScript which uses the node.js package. The code base as of version 0.1.6, Fall 2020 was used in this research.

To detect defects in the JavaScript code of Spectral Workbench, this research approach is to conduct a static analysis on the program. This enables the developers to find defects, understand the code structure, and know the test coverage percentage in a program. This helps the developers detect the defects in the program, and hence, resolve them. Presently, there are Jasmine tests that can be run to find defects in the application, however, this dynamic analysis will enhance the testing of Spectral Workbench and increase its quality.

This research uses the following static analysis tool. SonarQube [4] static analysis tool is used to run analysis on the program. SonarQube is a tool that supports the framework, programming language and libraries being used in Spectral Workbench. Once the analysis is run on the JavaScript core of Spectral Workbench, thus results are generated and grouped into different categories according to the severity of the issues or non-issue results. Here, the fix suggestions are taken and implemented by the engineer or by the tool as set in the selected analytic tool, therefore, eliminating the defects. A Pull Request with those changes are done to the main master branch of the program repository. This creates a basis to suggest to the Public Lab team to integrate a static analysis tool to their new code review process. This enables the JavaScript core of Spectral Workbench to have a definite method to detect defects for every new code push.

In order for the Spectral Workbench to be successful, users need to have faith that the applications are reliable. The Spectral Workbench community depends upon a large number of people donating time to scan and upload subjects. It also depends upon researchers and educators to be able to quickly and reliably locate scans and download them. This makes the reliability of Spectral Workbench critical to its success.

2. Literature Review

This Literature Review is organized into three sections. There is a detailed review of the Spectral Workbench, followed by a similar review of SonarQube. Then there is a review of similar research that addresses the same problem using static analysis as this research.

2.1 Spectral Workbench Background

Spectral Workbench is an open-source program that performs low-cost [5] dynamic analysis [1]. These results are shared online. The results from this analysis are used for various purposes not excluding the measurement of gases in the atmosphere, measurement of the artificial light sources, and the verification of Beer’s Law. Beer's Law states that the concentration of a chemical solution is directly proportional to its absorption of light [6]. This is the attenuation of

light to the properties of the material through which the light travels. The Spectral Workbench program was originally written on Ruby on rails, but its second version is written on JavaScript. There is also a client program that captures scans and a server website to upload, store and search the database records for them. This research is focused on the client program. The Spectral Workbench is not the only product that does this [7].

2.2 Importance of Spectral Workbench

The Spectral workbench program is an important program as it gives its users the ability to perform experiments that will help solve problems. These experiments help determine the composition of gases, artificial light measurements, comparisons of oil samples and residues in UV lights and much more.

One of the main activities that Spectral workbench helps with is the measurement of gases in the atmosphere. In this activity, one can easily step outside and point the spectrometer at the sky or a cloud. When this is done, molecules between the spectrometer and the sun (the light source) can be seen or noticed. These are seen in the form of lines, which are also known as Frauenhofer lines. These molecules include carbon dioxide, oxygen, ozone and more. With this activity, one can tell if his environment is polluted, high in oxygen or low in oxygen.

Just as one can point the spectrometer to a natural source of light to recognize the atmospheric molecules present in their environ- ment, one can also point to an artificial source of light for example a light bulb. This is also known as Flame spectroscopy. Here the spectrometer is used to determine emissions seen when gases or liquids are exposed to UV lights, lasers, or fluorescent bulbs.

Another example activity that shows the use of spectral workbench is the Beer's law experiment. Beers Law relates that the concentration of a compound is proportional to its absorbance of light [6]. This means that the amount of light absorbed when it passes through a solution or compound is proportional to the amount of solution that makes up the compound. If the concentration of a solution that makes up a compound is thick, the light absorbed is high, and if the concentration is diluted, the light absorbed is lower. Therefore, spectrometer in this case will use a sample of a known concentration to discover the concen- tration of an unknown sample.

Furthermore, spectral workbench enables the comparison of different oil samples and residues in UV light. This will help determine if there are harmful substances to our everyday oil use by differentiating the different oil substances via spectrometer. In this activity, some of the expected results are to find what a suspected motor oil residue, baby oil, BP oil spill and more look like. This way you can compare the components of our everyday home oil liquids. The Spectral Workbench is important to the field of Natural Sciences.

2.3 Structure of Spectral Workbench

Spectral workbench has a unique structure which has been developed over many years. This program is made up of three major classes. These classes are core classes, API classes and the UI classes. These classes are structured around a series of JavaScript classes shown in Figure 1.

The Core classes of spectral workbench are the commonly used and useful API methods [1]. One of the core classes is the index file that is the root of all the other classes of the Core classes. Another core class is the *Graph.js* class. This is the base class for the display for spectral workbench. It is the class that generates the graph and chart that the client sees after performing a spectral activity. Another core class is the *Image.js* class. This class handles the display of the image above the graph. The *Importer.js* class is another class from the Core class group that handles the importation of parsing of the server-side data requests to Sets and Spectra. Furthermore, the *Datum.js* class is a parent class that manages all data from the server thus, tagging these data to either categorize or describe the spectra. This activity makes it easier to search for a spectrum. Nevertheless, we have the *Tag.js* class, which is

the basic class for tagging. It extends the *PowerTag.js* class which is much more powerful as it is used for the manipulation of spectral data in a reversible way. *PowerTags* is also known as *Operations*.

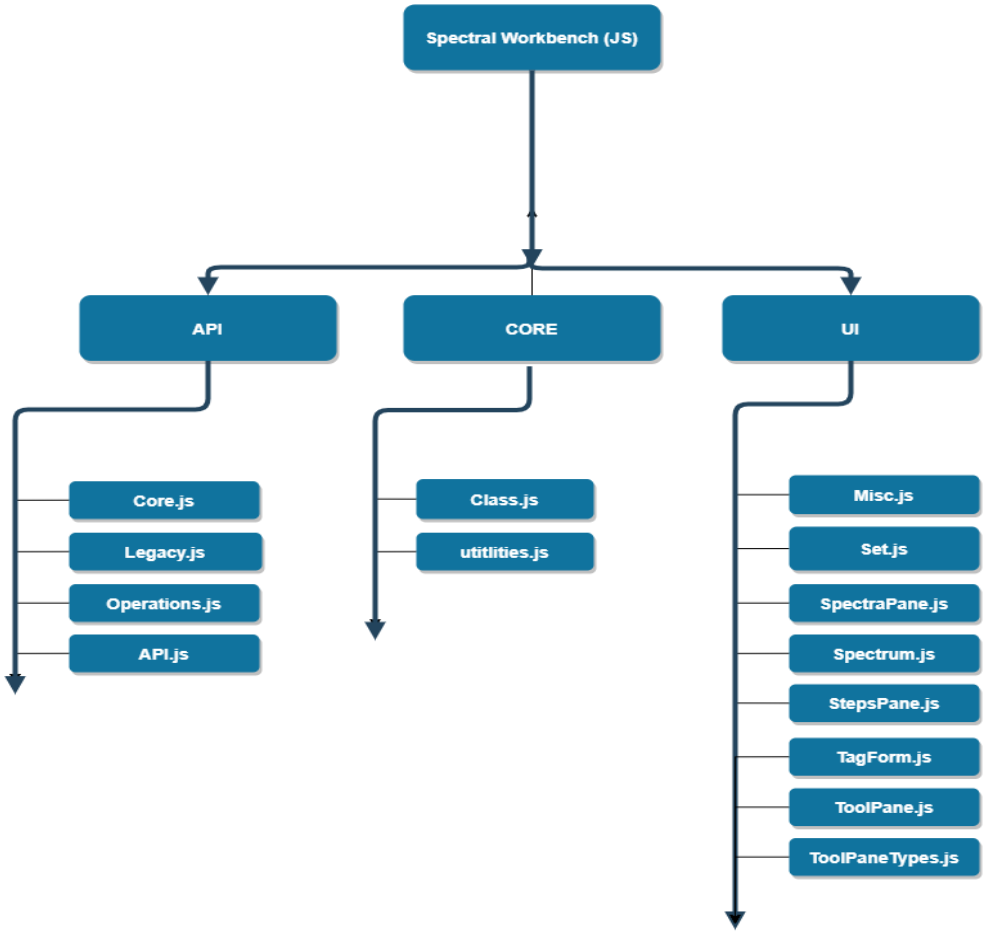

Figure 1: Spectral Workbench Class Structure

The first major group of classes to be discussed is the API classes. The API classes are split into three classes. These are the Core class, the Operations class, and Legacy class. The core class houses the most used and useful API methods. The Operations class contains individual operations that applies to spectra. It also defines the common actions for all operations in spectral workbench. Some of these operations are flip, error, transform, linear calibration, blend, subtract and more. To give more insight, the flip operation indicates that the image being generat- ed is to be flipped, the error operation handles error during the spectral activities and is more of a passive operation as it does not affect data. The transform operation filters the spectrum with a math expression. The linear calibration operation calibrates the spectrum suing two reference points. The blend operation is like the transform operation as it filters a spectrum with a math expression but combines it with data already gotten from the spectrum. There are many more operations in the operations class and as we go further into this research, we will discuss them as needed.

The second group of classes of the Core classes is the Legacy class. This class handles the API methods and backwards compatibility for the Spectral Workbench previous version. This version is written in Ruby on Rails as earlier mentioned.

The third major group of classes is the UI class. As its name signifies this class group houses the main code that handles the User Interface aspect of spectral workbench. The classes that make up this class group are the *Sepctrum.js* class. This houses the spectrum specific UI code. Another class is the *Set.js* class that

handles the setting of the data specifics class. The *ToolPane.js* class is the class that houses the pane interface itself. This pane interface displays the selection of spectra used in the manipulation of the current data point. Another class is the *TagForm.js* class that contains the code that generates the tagging form on the spectrum pages. The last but not the least class is the *Misc.js* class. This class is responsible for the "like" button presently. This class is to house any miscellaneous operations for spectral workbench in the future.

2.4 SonarQube Background

SonarQube is an open source software tool that is used to conduct static analysis on software programs. This tool is used to conduct code checks, hence, reviewing the code-quality. SonarQube is used to detect defects, vulnerabilities, code smells in more than twenty programming languages.

Code smells are code segments that most likely work correctly, but is difficult to maintain. Examples of code smells include redundant or duplicate code, complex code and codes that are not covered in unit tests.

Languages supported by SonarQube include java,

JavaScript, typescript, C++, C, C-Sharp, GO, python, COBOL, apex, php, and swift. For our use in this paper attention is paid to just one language which is JavaScript. Developers add the SonarQube software to the Continuous Integration Continuous Delivery (CICD) pipeline, this is as SonarQube has the capability to efficiently support the automatic static code analysis inspection during every major code change, as wanted designed by the developers.

SonarQube does not have the ability to detect vulnerabilities on JavaScript applications. The Spectral Workbench server-side application is written in JavaScript. For this reason, vulnerability detection is not part of this research.

2.5 Importance of SonarQube

SonarQube is a tool that helps with code reliability, application security and reduction of technical debt. As earlier said, it is used to continuously review your code across a program branches and pull requests. The issues SonarQube detects can be broken down into three types of results. These are code smells and bug detection and security hotspots.

Defect detection is another type of SonarQube issue whereby codes that do not perform as intended by the developer are detected or just the detection of wrong syntax. This is pointed in the different locations of the code base.

2.6 Structure of SonarQube

SonarQube is divided into four components. These components are SonarQube servers, database, plugins and sonar scanners.

The servers perform three main tasks. Firstly, the web server of SonarQube browses through the snapshots as well as configure the SonarQube instance. Secondly, searches from the UI are backed by the SonarQube elastic search servers. Finally, there is the compute engine server. This server processes the code analysis report and saves its' findings in the SonarQube database.

Now that we have gotten to know the four components of SonarQube, we will go ahead to discuss how it is integrated with the code workflow. After developers are done making code changes, they can choose to run analysis on the local code base using the tool sonar-lint. Sonarlint is a plugin for SonarQube. Nevertheless, the code changes are pushed to a source code management (SCM) of their choosing. This could be either git, TFVC, SVN and more. The continuous integration server in the SCM of choice starts up a build which triggers the sonar scanner needed to run the analysis. Once this analysis is done, the report generated is sent to the SonarQube servers for further processing. In these servers, the report is processed, saved in the SonarQube database and displayed on the SonarQube UI for the developers to review and make necessary changes.

Furthermore, SonarQube uses a term called *Quality Gate*. Quality Gates are the ways SonarQube enforces quality policies for an organization. Once a report analysis is done, part of the results being displayed is the Quality Gate, where its' value is either *passed* or *failed*. This

value is displayed at the top of the report page. Quality gate is generated by the percentage of code smells and bugs are found in your newly written code. If the Quality Gate is a failure, the developer is notified of this. Thus, the attention and immediate review of the SonarQube findings is done. This percentage of the issues found in the report analysis represents the grades of your new code changes. These grades are explained below:

A: greater than, equal to 80%

B: greater than, equal to 70% and less than 80%

C: greater than, equal to 50% and less than 70%

D: greater than, equal to 30% and less than 50%

E: less than 30%

Earlier in this paper, we had mentioned the three types of issues in SonarQube. These issues are defect (bugs), code smells and security hotspots. These issues when reported to the developer are broken down into different severities. Their severity determines how important the fix of this issue is, hence, there are five types of severities; they are *blocker*, *critical*, *major*, *minor* and *info*.

The blocker severity is the type of severity that tell-

s the developer that this issue must be fixed as soon as possible. This is a red alert, and thus, a good example is an unclosed database connection. Know that in this severity the functionality of your software project is impacted, thus at risk.

The critical severity is a severity that has a low probability to impact the functioning of the project. This could either be a bug or a security flaw. An example is an empty catch block.

The major severity is another type of severity that affects the quality of the project. This is mainly seen in issues that call out redundant codes, uncovered lines of codes for unit testing and unused parameters. This severity tends to tell the developer that his or her productivity is not maximized.

Finally, severity is info. This severity is like a warning, it is not a bug or defect but a finding.

In addition to issues being found, it is important to note that all issues have a lifecycle. This is a way the issues are being tracked from when they are found. There are five different lifecycles. These are *open*, *confirmed*, *resolved*, *reopened*, and *closed*. When an issue is found it is automatically set to the open status by SonarQube. Once the issue is reviewed by the developer, he or she has the permission to set the issue to confirmed or resolved. Note that confirmed and resolved are the only two statuses that are set manually by the developer. When an issue is set to confirmed, this means that the developer acknowledges that the issue is important and will be looked into. However, if the issue is set to a status of resolved, this could mean only two things: that the issue is a *false positive* or a *won't fix* issue. The reopened status of the issue signifies that a resolved issue has not actually been resolved while the closed issue is automatically set by SonarQube if the issue has been *fixed* or *removed*. A *fixed* issue means that after subsequent analysis, SonarQube has found out that the issue is no longer seen in the code while a *removed* issue means that the file, location or path has either been deleted or removed from being reviewed by SonarQube.

2.7 Evaluating Static Analysis Defect Warnings on Production Software

The research problem in [8] addresses the problem that there is little to no information about experimental evaluations performed on different production software, and how accurate the reports are [8]. Therefore, it can be said that this paper deals with the evaluation of static analysis tools with regards to the accuracy and seriousness of the warnings they report. Dealings with false positives and findings of inconsistent or deviant code instead of real defects. Although this paper uses the FindBugs tool during its methodology, this paper gives a new light on how the developers should review and triage on the reports generated from analytic tools like SonarQube.

The authors of this paper used Findbugs to perform static analysis on three major code bases: Sun's JDK, Sun's Glassfish J2EE server

and finally a portion of Google's Java code base [8]. After the reports were generated, they were triaged by the developers in the different organizations. The authors of this paper observed the routine to which these developers responded to the reports (whether routinely or random). They also documented to what extent that the issues found were fixed or listed as false positives.

2.8 Static Analysis of Programs with Graphical User Interfaces

Staiger [9] tried to address the problem of static analysis on programs with Graphical User Interface (GUI). Little research addresses this domain. Staiger created a new approach for analyzing Graphical User Interface applications whereby Staiger's goals were to support the program understanding, support maintenance at the same time handle architecture recovery while performing static analysis on GUI applications.

Staiger was able to break down his method; the first thing that needed to be done was to determine the entities that belongs to the GUI [9]. After these findings, the widgets are to be located next. Finally, the functions that reacted to events that caused the generation of the GUI were to be recognized. After all these are completed, then can the reports be generated. The tool that was used by Staiger is the Bauhaus static tool.

Staiger conducted this theory and concluded that this algorithm generated for the Bauhaus tool was indeed efficient. Staiger's approach is of importance to this paper as it relates to the approach and investigation on how SonarQube works with JavaScript.

2.9 A Template-based Approach to Automatic Program Repair of SonarQube Static Warnings

Haris Adzemovic [10] looked at the problem that developers spend a lot of time discovering, identifying and resolving bugs instead of advancing their software [10]. Adzemovic wrote about how half of developer's time is used in debugging and reviewing alerts, warnings and issues raised by static analysis tools, thereby reducing developer's productivity. Therefore, Adzemovic proposed that an automatic program repair should be designed to automatically resolve repetitive issues whether false positives or not. This will reduce the number of alerts sent to the developer, hence, increasing productivity time [10].

To achieve this, Adzemovic [10] used the SonarQube-repair. This is a tool that can automatically find, repair, and submit fixes of issues found in projects based on the configured rules. The idea is to review all the issues seen in the open source projects Adzemovic chooses to contribute to, compile those issues and generate the right rules that will enable or fit to build a template-based repair approach. And, finally integrating these rules to the continuous integration, continuous delivery pipeline. Adzemovic, went further to compare SonarQube-repair to SpongeBugs, to see if building a template is applicable to other automatic program repair tools. Adzemovic [10] was able to conclude that using repair tools was an efficient way to maximize productivity, and this is applicable to other automatic program repair tools. However, using pull requests was not an option during Adzemovic's experiment, hence, this will not fit in to most developers' usages today.

2.10 Improved Metrics Handling in SonarQube for Software Quality Monitoring

García-Munoz, García-Valls and Escribano-Barreno's research [11] addresses the limitation of SonarQube has with only supporting Java-based languages. This research proposed a new way to support other languages. This paper describes how to add additional functions to the reporting tool by combining the analysis results from external rules tool to its' own analysis results to give rise to more prioritized issues on new code changes.

3. Methodology

This section describes the methodology that was used in this research. The type of methodology that will be used is the empirical research methodology. The next parts of this section would go into the details on how this experiment will be conducted on the static analysis of Spectral Workbench.

The empirical methodology is used in this research.

3.1 Gathering of All Prerequisites:

There were certain prerequisites needed for this experiment to be conducted. This research was conducted on a computer that met the minimum configuration for SonarQube. SonarQube requires a machine having at least 2GB of RAM to run efficiently and 1GB of free RAM for the OS. The computer had a 64-bit processor.

In addition to meeting the hardware requirements, there are a few software requirements that needed to be met. These are:

1. Java JRE version 11.
2. Database: For this experiment we used Microsoft SQL server. This as explained earlier will house the reports.
3. Web browser: This could be Microsoft Edge, Google Chrome, Safari, Mozilla Firefox or Opera. In our experiment, we used Google Chrome. This enabled us to view reports and visualization or analysis of the completed scanning.
4. SonarQube: This is the main program that was installed on our computer. The version installed was the community edition.

3.2 Running and Report Analysis:

In this step, SonarQube was run and the reports were analyzed. When run, the reports are generated, logged, and can be viewed on the default SonarQube localhost site for your project. The reports are to be verified. Thus, the reports are to be reviewed by the developer to make sure that it is accurate and consistent. This report will look like the diagram in Figure 2.

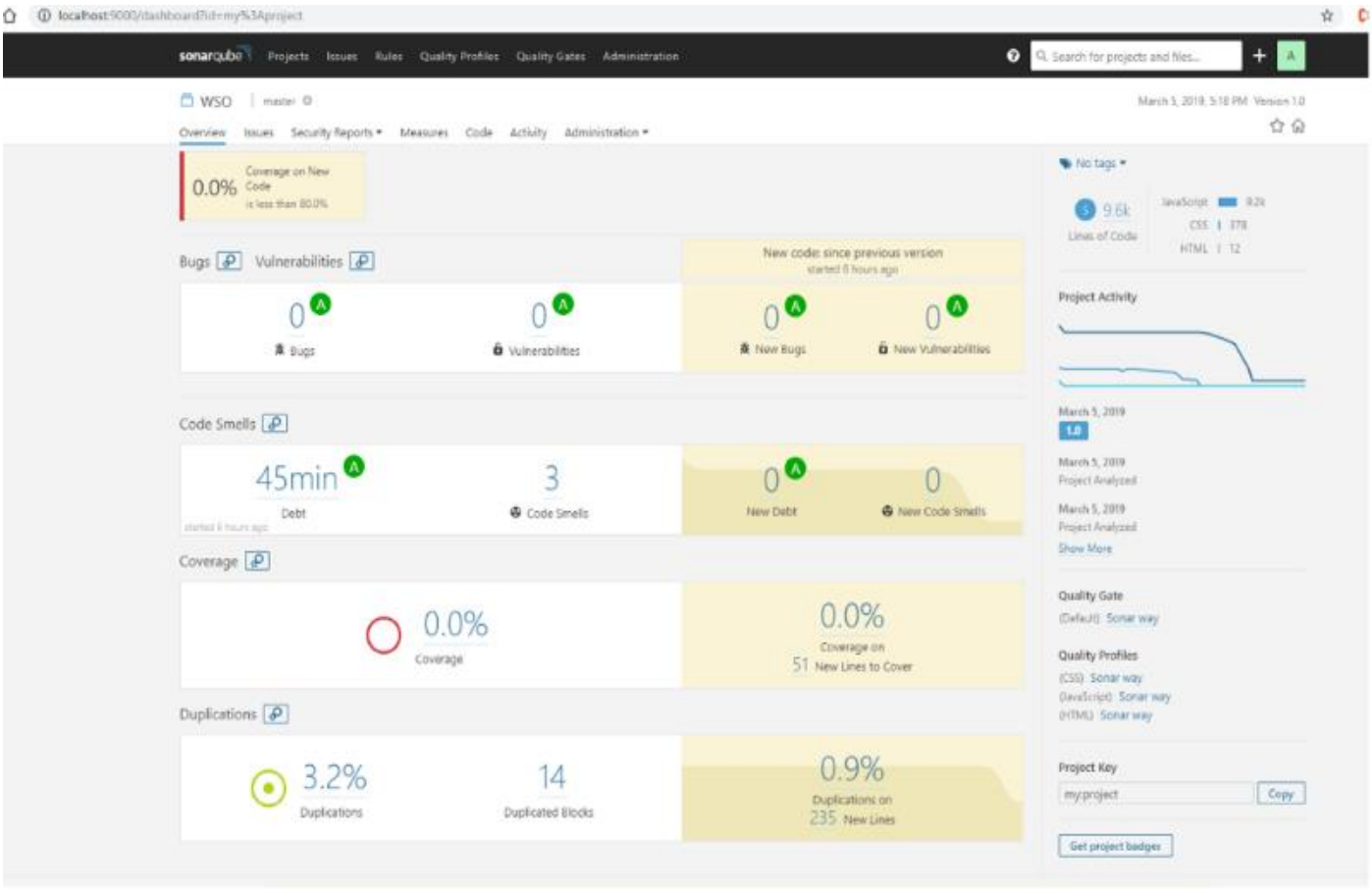

Figure 2: Sample overview analysis

The percentage coverage was reviewed, followed by the percentage code duplication and code smells. We were able to review in detail the exact classes, functions and exact line number that have the duplicated lines of codes and code smells.

4. Results

In this section, we present reports generated by SonarQube because of the analysis run on Spectral Workbench. SonarQube scanner was run on Spectral Workbench JavaScript project, of which it is results were analyzed and uploaded to the SonarQube server. Table 1 shows the overall finding of the research.

Table 1. Summary of Findings.

Type of Report	Number of Reports
Bugs (Defects)	63
Security Hotspots	15
Code Smells	232

According to the table above, the overall gate status is *Passed*. This is because of the grades seen on the different categories of the analysis done by sonar scanner. There were 63 bugs found in the code base, thus getting the status E for reliability. There were 15 Security Hotspots found which resulted to another status E for Security Review and finally, 232 code smells which resulted to an A status for maintainability.

4.1 Spectral Workbench Defects

SonarQube analysis shows that there are 63 bugs in the Spectral Workbench project. These 63 bugs are categorized to Blocker, Major and Minor. There are 2 Blocker bugs, 17 Major bugs and 44 Minor bugs. The blocker bugs refer to the callback functions that have a missing return statement in them. This will result in failures if these functions are called. The major bugs refer to missing assignment or function calls and the likes. These bugs may or may not cause an error or failure when the respective functionality is called, however, the functionality will not work as expected or be accurate. The minor bugs are mostly HTML bugs of which will cause little or no functional errors, but cosmetic errors. Figure 3 shows a screen shot of the report and Table 2 shows all of the defects located.

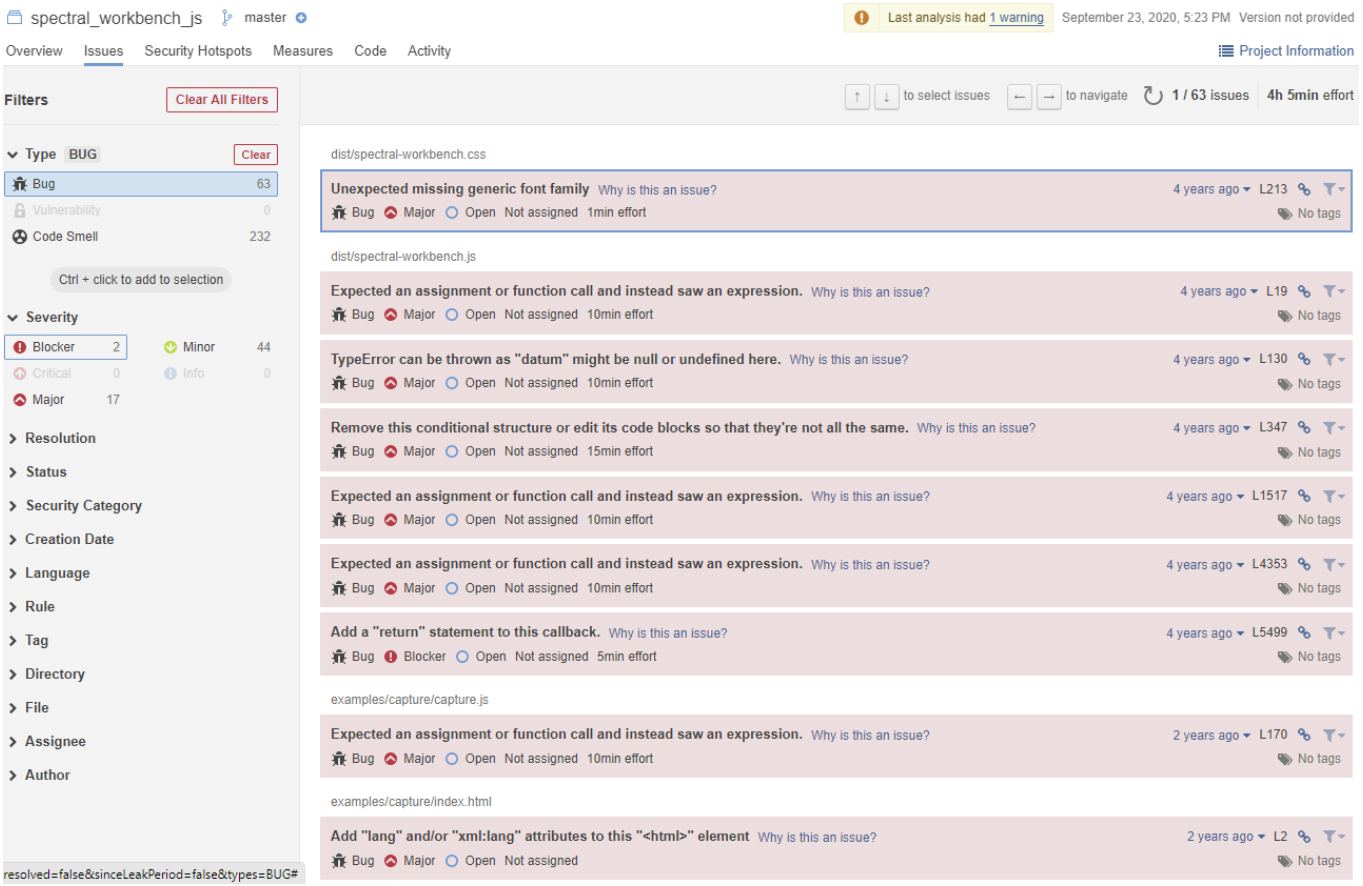

Figure 3: Defect Report Screen

4.2 Spectral Workbench Code Smells

A total of 232 code smells were discovered in this application. There are 62 blockers, 142 majors, 27 minors, and 1 of the info categories in code smells.

Code smells deals with the quality of the code base. The code smells found under the blocker category can be when variables are mis-declared, for example, making a supposedly private variable global. The major code smells can be seen when a variable is declared twice in Spectral workbench. Table 3 shows a sample of 50 of the 232 Code Smells discovered. Figure 4 shows a screen shot of the report.

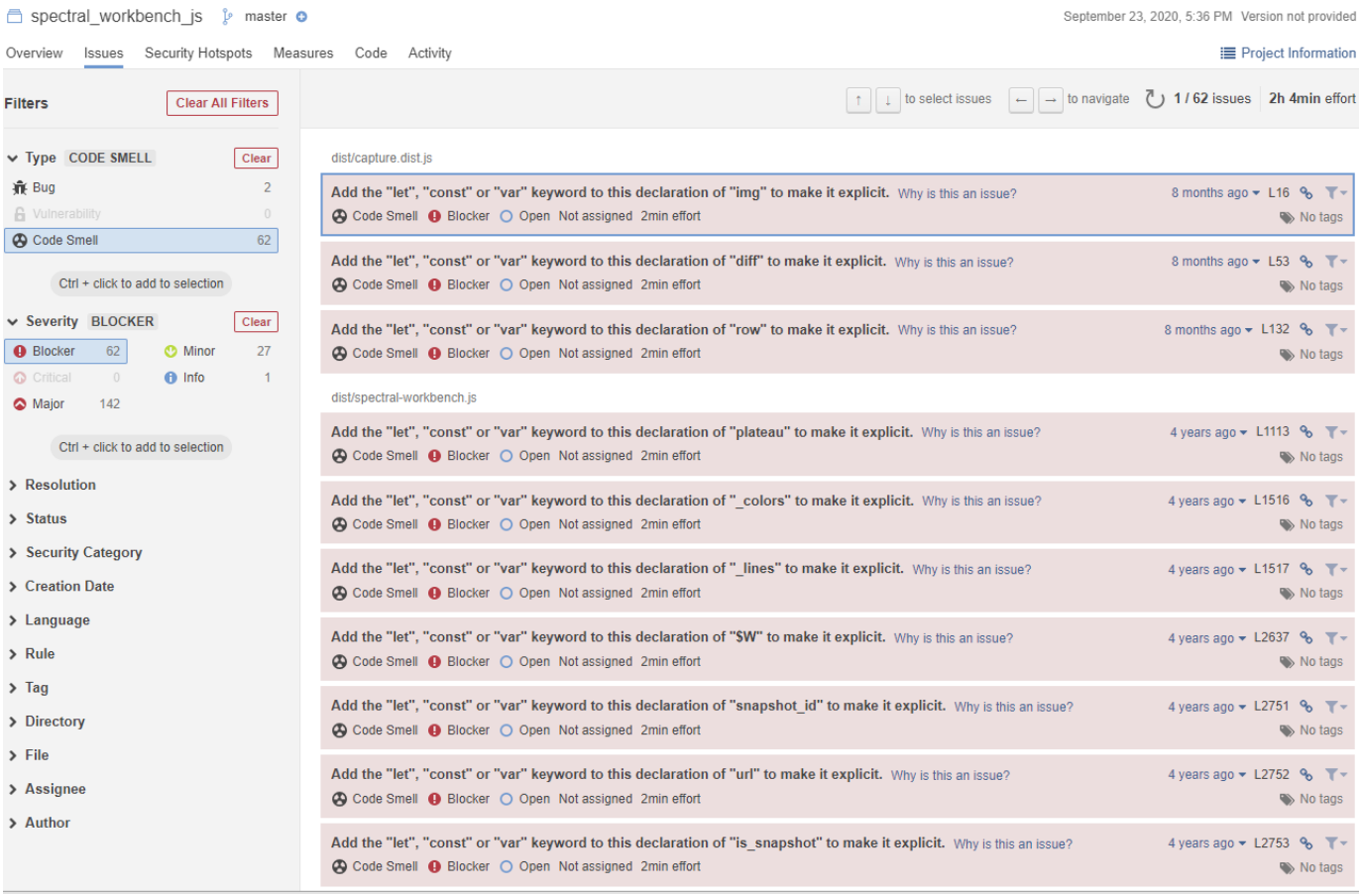

Figure 4: Code Smells Report Screen

4.3 Spectral Workbench Security Hotspots

Finally, there is the Security hotspot section of the SonarQube analysis. Security Hotspots are pieces of codes that needs to be reviewed to see if they of threat to the project or not. There are two High review priority. These two are command injection where command line arguments are used in the code base. The other sections are of medium review priority. These sections are Denial of Service, Code Injection and Weak Cryptography. Table 4 shows all of the Security Hotspots located. Figure 5 shows a screen shot of the Security Hotspots report.

5. Conclusion

This research applied static analysis to the server-side core of the Spectral Workbench. Defects, Security Hotspots and Code Smells were located and documented in this research for resolution. Since Spectral Workbench is an open source application, anyone can resolve the coding issues. Perhaps the Spectral Workbench core team at PublicLab will use SonarQube in the future.

This research has numerous benefits to both Computer Science and the Natural Sciences. As Empirical Research it extends our knowledge of Applied Static Analysis and the benefits of SonarQube. Ultimately it will improve the quality of the Spectral Workbench, which has enormous benefits to the Natural Sciences. The data that researchers use from the Spectral Workbench will increasingly be more reliable.

This research demonstrates that the Spectral Workbench is a reliable application. Some issues have been located but when compared to other benchmarks this application has an acceptable level of quality. This research is evidence that data in Spectral Workbench is reliable.

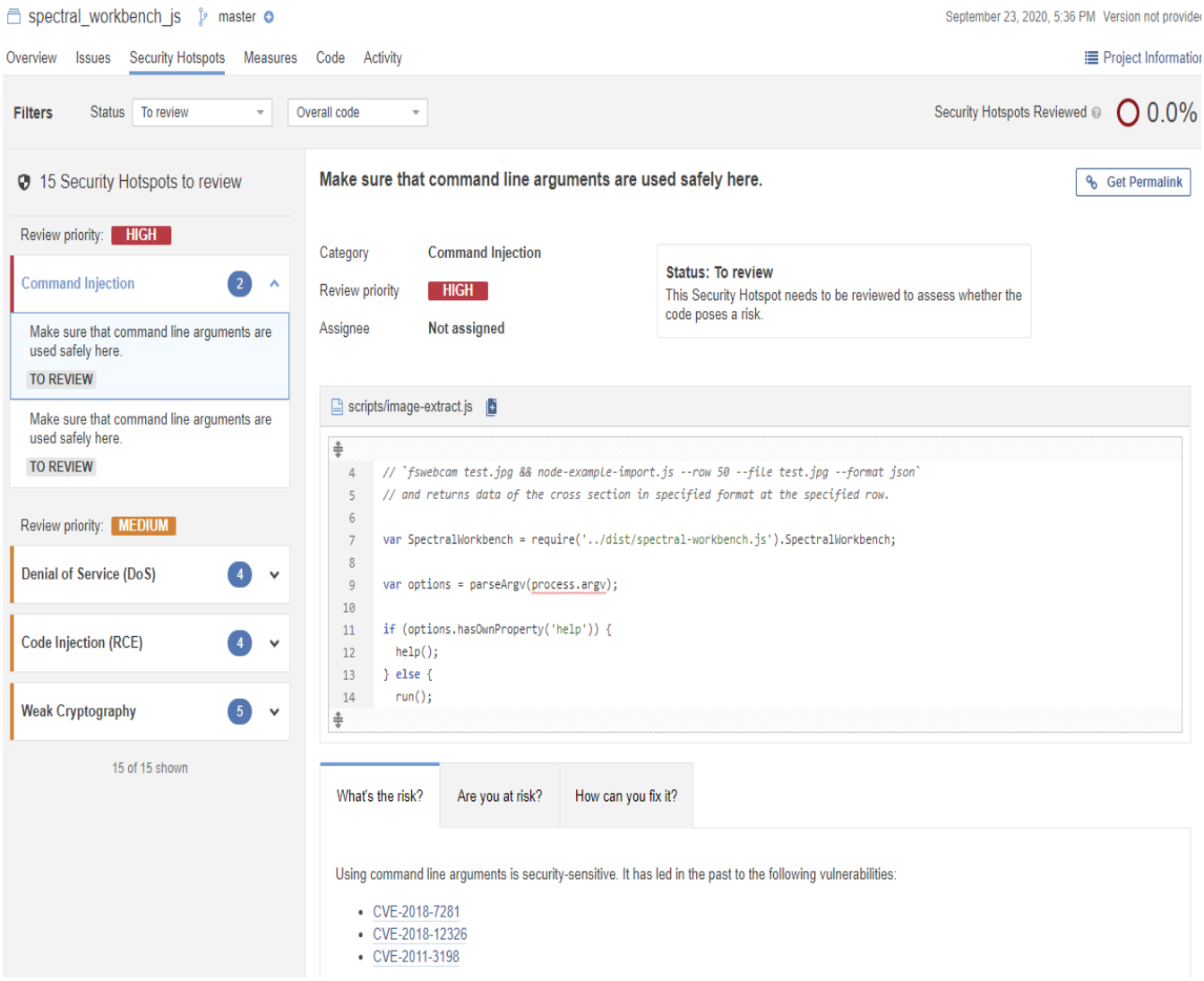

Figure 5: Security Hotspots Report Screen

Table 2. Spectral Workbench Defects (Excluding defects in the *examples* folder).

Description	Line Number	Type
Source File dist/spectral-workbench.css		
Unexpected missing generic font family	L213	Major
Source File dist/spectral-workbench.js		
Expected an assignment or function call and instead saw an expression.	L19	Major
TypeError can be thrown as "datum" might be null or undefined here.	L130	Major
Remove this conditional structure or edit its code blocks so that they're not all the same	L347	Major
Expected an assignment or function call and instead saw an expression.	L1517	Major
Expected an assignment or function call and instead saw an expression.	L4353	Major
Add a "return" statement to this callback.	L5499	Blocker
Source File scripts/image-extract.js		

options.file' is assigned to itself.	L24	Major
spec/javascripts/fixtures/graph.html		
Add an "alt" attribute to this image.	L3	Minor
Replace this <i> tag by .	L9	Minor
Replace this <i> tag by .	L10	Minor
Add a description to this table.	L26	Minor
Add either an 'id' or a 'scope' attribute to this <th> tag.	L28	Major
Replace this <i> tag by .	L39	Minor
Replace this <i> tag by .	L46	Minor
Replace this<b> tag by	L68	Minor
Replace this <i> tag by .	L78	Minor
Replace this <i> tag by .	L107	Minor
Add "<th>" headers to this "<table>".	L137	Major
Add a description to this table.	L137	Minor
Source File src/SpectralWorkbench.Graph.js		
Add a "return" statement to this callback.	L359	Blocker
Source File src/SpectralWorkbench.Image.js		
Remove this conditional structure or edit its code blocks so that they're not all the same.	L208	Major
Source File src/SpectralWorkbench.Importer.js		
TypeError can be thrown as "datum" might be null or undefined here.	L26	Major
Source File src/SpectralWorkbench.Spectrum.js		
Expected an assignment or function call and instead saw an expression.	L723	Major
Source File src/core/Class.js		
Expected an assignment or function call and instead saw an expression.	L19	Major
Source File src/ui/SpectralWorkbench.UI.ToolPaneTypes.js		
Expected an assignment or function call and instead saw an expression.	L338	Major

Table 3. 50 of 232 Spectral Workbench Code Smells.

Description	**Line Number**	**Type**
Source File dist/capture.dist.js		
Add the "let", "const" or "var" keyword to this declaration of "img" to make it explicit.	L16	Blocker
Either use this collection's contents or remove the collection.	L18	Major
Add the "let", "const" or "var" keyword to this declaration of "diff" to make it explicit.	L53	Blocker
This line will not be executed conditionally; only the first statement will be. The rest will execute unconditionally.	L107	Major
Add the "let", "const" or "var" keyword to this declaration of "row" to make it explicit	L132	Blocker
wavelength' is already defined.	L179	Major
‘w' is already defined.	L182	Major
Source File dist/capture.dist.js		
Unexpected empty block Why is this an issue?	L56	Major
Unexpected empty block Why is this an issue?	L92	Major
Unexpected duplicate selector ".swb-graphing .tools a", first used at line 131	L135	Major
Source File dist/spectral-workbench.js		
Change this argument to the documented type: String. Why is this an issue?	L19	Major
name' is already declared in the upper scope. Why is this an issue?	L39	Major
Remove this useless assignment to variable "importer".	L109	Major

Remove the declaration of the unused 'importer' variable. Why is this an issue?	L109	Minor
‘datum' is already defined.	L123	Major
Canvas' is already declared in the upper scope.	L178	Major
Remove this useless assignment to variable "Image".	L179	Major
Remove the declaration of the unused 'Image' variable. Why is this an issue?	L179	Minor
‘tag' is already declared in the upper scope.	L470	Major
‘tag' is already declared in the upper scope.	L498	Major
‘tag' is already declared in the upper scope.	L500	Major
‘tag' is already declared in the upper scope.	L542	Major
‘tag' is already declared in the upper scope.	L544	Major
‘end' is already defined.	L1040	Major
Remove this useless assignment to variable "channel".	L1086	Major
Add the "let", "const" or "var" keyword to this declaration of "plateau" to make it explicit.	L1113	Blocker
Remove this useless assignment to variable "channel".	L1117	Major
‘i' is already declared in the upper scope.	L1384	Major
callback' is already declared in the upper scope.	L1405	Major
‘data' is already declared in the upper scope.	L1408	Major
‘data' is already declared in the upper scope. Why is this an issue?	L1502	Major
‘threshold' is already declared in the upper scope.	L1502	Major
consecutive' is already declared in the upper scope.	L1502	Major
Add the "let", "const" or "var" keyword to this declaration of "_colors" to make it explicit.	L1516	Blocker
Unexpected use of comma operator.	L1516	Major
Add the "let", "const" or "var" keyword to this declaration of "_lines" to make it explicity.	L1517	Blocker
Remove this useless assignment to variable "line". Why is this an issue?	L1529	Major
Remove the declaration of the unused 'line' variable.	L1529	Minor
data' is already declared in the upper scope.	L1637	Major
‘callback' is already declared in the upper scope.	L1787	Major
‘name' is already declared in the upper scope.	L1795	Major
‘callback' is already declared in the upper scope.	L1848	Major
‘tag_response' is already defined.	L1862	Major
‘callback' is already declared in the upper scope.	L1905	Major
‘callback' is already declared in the upper scope.	L1926	Major
‘json' is already declared in the upper scope.	L2066	Major
‘name' is already declared in the upper scope.	L2089	Major
‘callback' is already declared in the upper scope.	L2128	Major
‘string' is already defined.	L2288	Major
‘callback' is already declared in the upper scope.	L2450	Major

Table 4. Spectral Workbench Security Hotspots.

Type	Sample Location	Sample Code	Priority
Command Injection	scripts/image-extract.js	var options = parseArgv(process.argv);	High
Command Injection	scripts/upload.js	var options = parseArgv(process.argv);	High
Denial of Service	dist/spectral-workbench.js	if (name.match(/[\w\.]+:[\w0-9\-#*\+\[\]\(\)]+/))	Medium

Denial of Service	dist/spectral-workbench.js	if (key.match(/[a-zA-Z-]+:[a-zA-Z0-9-]+/)) color = " purple";	Medium
Denial of Service	src/SpectralWorkbench.Datum.js	if (name.match(/[\w\.]+:[\w0-9\-#*\+\[\]\(\)]+/))	Medium
Denial of Service	src/ui/SpectralWorkbench.UI.Tag-Form.js	if (key.match(/[a-zA-Z-]+:[a-zA-Z0-9-]+/)) color = " purple";	Medium
Code Injection	dist/spectral-workbench.js	eval('var transform = function(R,G,B,A,X,Y,I,P,a,r,g,b)	Medium
Code Injection	dist/spectral-workbench.js	eval('var blend = function(R1,G1,B1,A1,R2,G2,B2,A2,X,Y,P)	Medium
Code Injection	src/api/SpectralWorkbench.API.Core.js	eval('var transform = function(R,G,B,A,X,Y,I,P,a,r,g,b)	Medium
Code Injection	src/api/SpectralWorkbench.API.Core.js	eval('var blend = function(R1,G1,B1,A1,R2,G2,B2,A2,X,Y,P)	Medium
Weak Cryptography	dist/spectral-workbench.js	var id = parseInt(Math.random()*100000)	Medium
Weak Cryptography	dist/spectral-workbench.js	+ ".json?t=" + parseInt(Math.random()*10000),	Medium
Weak Cryptography	examples/capture/capture.js	var id = parseInt(Math.random()*100000)	Medium
Weak Cryptography	src/SpectralWorkbench.Graph.js	+ ".json?t=" + parseInt(Math.random()*10000),	Medium
Weak Cryptography	src/ui/SpectralWorkbench.UI.Spectrum.js	var id = parseInt(Math.random()*100000)	Medium

5.1 Future Work

There are numerous areas of future research. Here are the main future research areas that we have identified.

The research applied SonarQube to the server-side application of Spectral Workbench. It should also be applied to the client-side applications that records the spectral data. The client-side application is used to collect the spectral data before it is uploaded to the repository.

This research used the version of Spectral Workbench from Fall 2020, version 0.1.6. But this code base will change over time. The experiment should be performed again in the future.

This research should be applied to other Natural Science software to also improve quality. Quality improvements in Natural Science software will benefit Natural Science research.

This research provides numerous benefits for the Spectral Workbench community.

Disclosure statement

There are no potentially conceivable conflicts of interest reported by the authors.

Reference List

[1] https://spectralworkbench.org/

[2] Ammann, P., & Offutt, J. (2016). *Introduction to software testing.* Cambridge University Press.

[3] Desai, S., & Srivastava, A. (2016). *Software testing: a practical approach*. PHI Learning Pvt. Ltd.

[4] https://www.sonarqube.org/

[5] Balado Sánchez, C., Díaz Redondo, R. P., Fernández Vilas, A., & Sánchez Bermúdez, A. M. (2019). Spectrophotometers for labs: A cost-efficient solution based on smartphones. *Computer Applications in Engineering Education*, 27(2), 371-379.

[6] Mayerhöfer, T. G., & Popp, J. (2019). Beer's Law–Why Absorbance Depends (Almost) Linearly on Concentration. *ChemPhysChem*, *20*(4), 511-515.

[7] Padalia, H., Pandey, K., & Arumugam, R. A. (2017). Development of a web–enabled spectral data archival, visualisation and analysis architecture for tropical phytodiversity inventory. *Tropical Ecology*, 58(2), pp. 307-314.

[8] Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J., & Zhou, Y. (2007, June). Evaluating static

analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 1-8.

[9] Staiger, S. (2007, March). Static analysis of programs with graphical user interface. In *11th European Conference on Software Maintenance and Reengineering,* pp. 252-264. IEEE.

[10] Adzemovic, H. (2020). *A template-based approach to automatic program repair of Sonarqube static warnings*. (Master's thesis).

[11] García-Munoz, J., García-Valls, M., & Escribano-Barreno, J. (2016). Improved metrics handling in SonarQube for software quality monitoring. In *Distributed Computing and Artificial Intelligence, 13th International Conference,* pp. 463-470. Springer, Cham.

[12] Guaman, D., Sarmiento, P.A., Barba-Guaman, L., Cabrera, P. and Enciso, L. (2017). SonarQube as a Tool to Identify Software Metrics and Technical Debt in the Source Code through Static Analysis, *Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering*, Beijing, pp 171-175.