An Evaluation of FOSS Static Code Analysis Tools for C

Daniel O'Neill
Iowa State University
dponeill@iastate.edu

Kole Swesey
Iowa State University
kwswesey@iastate.edu

***Abstract*— The nature of cyber security's growth and amount of code being written daily calls for the need of automated code analysis tools. Publishing a brand new code to an updated version of a software without investigating it for vulnerabilities and weaknesses can lead to major issues in the software or the company. If a new product or update is sent out using code that has a vulnerability, like a buffer overflow, a customer's data or their own data could be at major risk. There are multiple tools in which a company or an individual can use and choose the correct solution. However, these tools can be expensive. A majority of other solutions make it so that you are required to do a bunch of leg work in creating functions and specific function calls. In this research project, we decided we would pick free open source solutions and ones that will not cause a bunch of overhead. We did our testing on Juliet test suites using Flawfinder, Splint, and VisualCodeGrepper in order to analyze them. From our research we determined that each tool has their own unique rate and use cases. Ranking the tools in how well they catch vulnerabilities and weaknesses, we will rank them as Splint as the best of the tools, to VisualCodeGrepper, and finally Flawfinder as the worst of the three tool sets that were used. However, the false positive rate generated by Splint might make a user think twice about using it as their code analysis tool.**

Keywords—code analysis tool, c repositories, Virtual Machine, CWE

I. Introduction

A. Purpose

The purpose of this document is to satisfy the requirement for the CprE 562X class's research project group assignment. The project was to do an independent research project activity. One of the options that we decided to go with was the "Tool evaluation" activity. We decided to take it further by doing an evaluation of several different tools in order to try and show options for open source automated code analysis tools. In the research we will take a look at three different open source tools and compare them to a test set that is provided by Juliet, as recommended by our professor. We will test against a subset of the test cases (five "Flaws" and five "Potential Flaws"), for all 109 different CWEs provided in the data dump. After running the tools, we will check to see the differences between the tool sets and provide analysis into each of them to determine the best tool to use depending on the personnel and amount of work required to comb through the information given by the tools.

B. Problem

Performing effective source code analysis is integral to developing secure software. While manual code audits could be all a developer needs on small projects, ensuring security in programs with large teams of developers and hundreds of thousands of lines of code is difficult to do. The solution to this problem comes in the form of Source Code analysis tools, which are designed to quickly audit the source code of a program to discover potential vulnerabilities. The OWASP foundation maintains a list of over 90 source code analysis tools, however many of these tools require a commercial license in order to be used by developers [1]. Many developers may not be willing to spend additional money on security and will instead opt to use the free tools that are available.

C. Background

With the advancement of technology and the birth of the internet, there was a major expansion in both information and new technologies that we all use. Of course, with this new technology, there will always be weaknesses and vulnerabilities that are created with each new advancement. With that being said, a new field came into play in this environment called Cyber Security. There are multiple different kinds of Cyber Security fields, and one of them involves auditing and testing of code. In this, these experts need to check code before and during the writing process for any vulnerabilities or weaknesses. One such way to be able to accomplish this daunting task is using automated code analysis tools, in order to find these quickly. These tools will scan thousands of lines of code and will try and pinpoint as many vulnerabilities and weaknesses as possible in a short amount of time. Out of these tools, most of them are very expensive and some require a paid subscription just to be able to use it. However, with the creation of open source software, there have been tools created to be able to accomplish the same amount of precision and accuracy of the paid tools. In this project we tested some of these open source tools to see how accurate these tools are, and to compare and contrast them. To do this we needed to get a large amount of C files to be able to test these tools against. There are different locations out there who provide this

service by providing large dumps of code samples that have different weaknesses. The example that was used during this research paper was Juliet [7], which includes a multitude of test cases and includes which line numbers are flaws and potential flaws. We wanted to make this paper for the purpose of helping someone decide on if using a certain open source automated code analysis tool would benefit themself or not.

D. Scope

After the project proposal, we had some discussion about our project and determined that our scope at the time was too broad and that we needed to make it focused on a single data set and narrow the amount of tools that we wanted. We originally wanted to try and run these tools against really large repositories and other repositories of varying sizes. From the meeting we went from wanting to test large repositories like linux's kernel and projects. We also planned on potentially testing up to five different tool sets to see how they were all comparing against each other. After running the experiment on the data set given by Juliet, it allowed us to give a more detailed experiment and were able to get a much better data set to compare and contrast.

II. Discussion

A. Approach

The approach of this project is detailed in steps in the statement of work below this section. Our approach to this project is to research different automated code analysis tools that are intended on auditing C files and look for weaknesses and vulnerabilities. After choosing these free and open source tools, we downloaded them into virtual machines, this will be how we will run the scans. We split up the 118 different CWEs, ran these tools, and recorded the output. From there we compared and contrasted the results to figure out the best tools to use given certain circumstances. After that, we will compare all the outputs from the systems and make graphs and charts to base our decisions on and include them in the final report to help our final result.

B. Result

From the results of this activity, we hope to learn more about how code analysis tools work and how they compare against each other. Doing this research activity will help us in the future when we look at potential automated code analysis tools in our jobs and have a good understanding of how they work and how they compare. This report also is helpful in determining how well they can compare to many industries paid for subscriptions. This paper will also help the reader to understand the differences between open source tools that exist and how they can help to determine weaknesses and vulnerabilities in their code.We can then report to the class our findings in hopes to help students see the differences and similarities before using these tools in their work.

C. Impacts

There are several different impacts that can be taken from reading this report. One of the major impacts is to be able to see if open source tools can measure up to the paid subscription toolsets. This impact could potentially save someone hundreds to thousands of dollars when you just need to audit some of the code in which a company would put out. Another impact that this paper can make is a decision between a few open source tool sets. This research paper will go into the efficiency and accuracy of three different tools to help make that decision clearer. We will also go into the amount of false positives and true positives, in which these tools generate. The research will also include the amount of false negative rates, which is one of the worst that you can have when looking for these weaknesses. This will help in decisions based on personnel and amount of focus into the auditing team.

The tools all have different levels of noise (finding issues that do not exist) and accuracy, hopefully this paper will help impact their decision for the better.

D. Limitations

There were few limitations that we had going into this project. One major limitation was that we wanted to use open-source tools that were not paid subscriptions or trial code analysis tools. The three tools that were going to be used to test the code samples are all free and open-source tools. Another limitation was that the tools that were chosen must be able to conduct the code samples without needing to write new code or need to be run by creating main codes like in our other assignment. These automated code analysis tools are able to read the code and conduct the scan without needing to do a lot of set up and can just be run. This is a good thing to have so that a developer can test their new code without breaking it apart completely to test every single function. Another limitation that was needed to be set was the dataset in which the tests were being done on. The Juliet data suite that we used to test included both C and CPP files to test with. Since we did not want more control variables in our experiment, we decided to do comparing and contrasting only using the C files in each folder. This way if certain tools are better at detecting C++ file vulnerabilities, it would not affect the rest of our testing pool.

E. Statement of work

In order to reach the goals of this project in a successful manner, we will need to follow the following steps. We will lay it out in a way for others to complete the same research that we did.

Task 1: Research different code analysis tools to test, then select a few of them to use against the code sets. Then download them into a Virtual Machine to do those tests.

Task 2: Find a large dump of code samples to download into the same virtual machine to run the tests against. In this specific case, we chose a dump of C files to run our tests against.

Task 3: Run these three automated code analysis tools against the dump of C files and obverse the differences and similarities between the results of the tools.

Task 4: Make sure that you get code to test and know where the vulnerabilities and weaknesses are in them. This will make it easier to see if the result was a true positive or if it was a false positive. This will help to show if the tools are getting false negatives, which means that they missed serious weaknesses and vulnerabilities in the code.

Task 5: Compare the findings between the tools in which you are testing and do a compare and contract of the files. From this someone would be successful in determining if a code analysis tool would be helpful to use or not.

III. Resources

A. Personnel

Both of the researchers in the article have graduated with a bachelor's degree in Computer Engineering and Computer Science. Both have been working towards a Master's Degree in Cyber Security and will graduate November 25, 2020. Both have participated in several different competitions that work with code analysis and penetration testing of various codes and softwares. Both have experiences in identifying vulnerabilities and remediating these threats.

B. Equipment

The three tools that we chose to evaluate are Flawfinder, Splint, and VisualCodeGrepper. Flawfinder is a tool written in python that has been primarily developed and maintained by David A Wheeler of the Linux foundation. It was first released in 2007 and is still under active development today. Flawfinder does not compile the source code when looking for vulnerabilities, it instead uses lexical scanning to find keywords in the code indicative of vulnerabilities. This approach means that it is incapable of detecting vulnerabilities that arise from control flow, but can be used on code that does not compile. Flawfinder is designed to work exclusively on C and C++ source code. Vulnerabilities that are reported are given an estimated risk level [2].

Splint is a tool written in C that was first released in 1994 under the name LCLint after being initially developed by professors David Evans and John Guttag of University of Virginia and Massachusetts Institute of Technology, respectively [3] [4]. It received regular updates up until 2010 when development slowed down, with the most recent version being released in May 2020. Splint only works on C source code and requires the code to be compiled in order for it to function [5].

VisualCodeGrepper is a tool written in Visual Basic .NET that was originally released in 2015 and has been developed and maintained by Nick Dunn since then, with the most recent update being in 2019. VisualCodeGrepper is capable of finding issues in comments, code, and the use of dangerous functions. It also ranks the vulnerabilities it detects from low to high and is capable of creating pie charts from the data it generates. VisualCodeGrepper is capable of auditing C, C++, Java, C#, VisualBasic, and PL/SQL source code [6].

For our testing environment, we had to use multiple operating systems since Flawfinder and Splint are made for UNIX based systems, whilc VisualCodeGrepper can only be run on Windows systems since it is written in Visual Basic .NET. For testing Flawfinder and Splint we used Manjaro Linux version 20.1. For testing VisualCodeGrepper, we used Windows 10 build 10.0.19042. We made sure to use the most up to date version of each program, which at the time was Flawfinder 2.0.11, Splint 3.1.2a, and VisualCodeGrepper 2.2.0.

For our test cases, we used the Juliet Test Suite v1.3 for C/C++ that was released by NIST in October 2017. This test suite contains code that is vulnerable to 118 different CWEs, with hundreds of different source code files for each of the CWEs. These source code files consist of simple programs that will have vulnerable lines of code, which are either marked as a flaw or a potential flaw by a comment [7].

While our intention was to test all of the CWEs, we realized that not every CWE test suite would be applicable to every program we were testing. Many test suites made use of windows specific libraries that were used for the vulnerable line of code. Since Splint requires the test cases to be compiled in order to test them, we could not use Splint on these test cases and had to skip over them. We also noticed that some CWEs only contained example C++ code and not C code. Since we are only interested in C

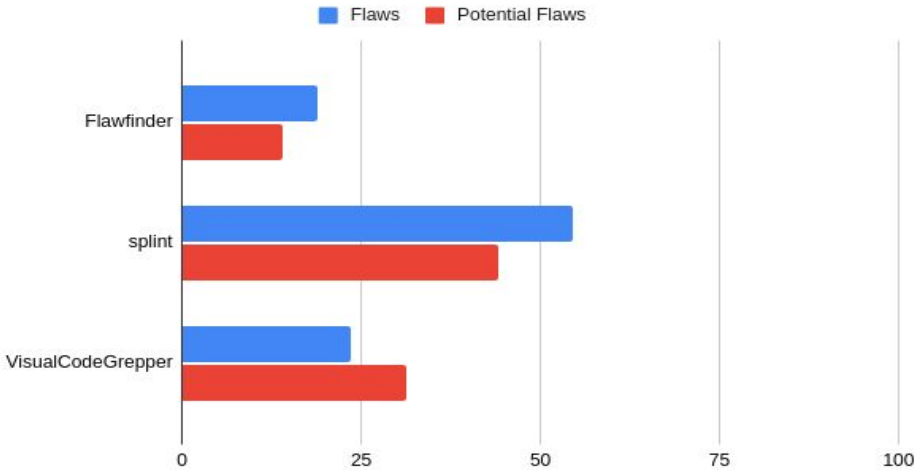

Figure 1: The percentage of flaws and potential flaws detected by each tool

vulnerabilities for the purpose of this experiment, we skipped over those as well. Out of the 118 CWEs included in the Juliet test suite, we were able to run these tools on 96 of them.

IV. Findings

A. Data

The first thing we did after completing all the tests was to calculate the overall success rate of each of the tools. To do this, we divided the number of true positives on the flaws the tools were tested on by the number of flaws, to get a percentage. We then did the same thing with the potential flaws in the code. The results of this can be seen in Figure 1. Splint outperformed both Flawfinder and VisualCodeGrepper by a significant margin, detecting 54.48% of the flaws and 40.09% of the potential flaws that it was ran on.VisualCodeGrepper was the second most successful, detecting 23.36% of the flaws and 31.36% of the potential flaws it was ran on. Flawfinder performed the worst of the three, only detecting 18.09% of the flaws and 14.10% of the potential flaws that it ran on.

We also wanted to look at what CWEs were being missed by severity. Now all CWEs are equally dangerous, and if a user knew that their tool is only going to detect certain CWEs, it would be ideal for the CWEs that are detected to be the ones that are considered more severe. In order to do this, we looked at a couple metrics that are provided by MITRE. The first is their list of the top 40 most dangerous CWEs from 2019 and 2020 [8] [9]. 9 of the CWEs on these lists were present in our test suite (many of the other CWEs on those lists are web based and as a result were not included in the test suite). We also acted as if CWE 534 was on this list, since MITRE marked that CWE as being deprecated for not being different enough from CWE 532, which was in fact on the top 40 list. We compared how the three tools performed on those 10 CWEs when compared to the CWEs that are not on either list to see if it is better at catching these.

The results of this can be seen in figure 2. We noticed slight changes in the success rates of each of the tools. Flawfinder's positive rate on both flaws and potential flaws changed slightly from 18.09% and 14.10% to 14.29% and 17.5%. Splint's performance went down a bit, as its positive rates changed from 54.48% and 40.09% to 34.29% and 42.5%. Finally, VisualCodeGrepper's performance improved, as its positive rates increased from 23.36% and 31.36% to 28.57% and 40%. While these changes are measurable, they are the product of a small sample size of only 10 CWEs, so we don't want to make any statements from this data alone.

Another way of comparing the CWEs would be by using the 'likelihood of exploitation' rating that is provided by MITRE for each CWE. The CWEs are given a score of either high, medium or low; or could have no rating at all if the CWE content team has the listing marked as 'incomplete'. The score is based on the probability that an attacker who has discovered the weakness would be able to successfully exploit it [10]. Out of the 96 CWEs that we tested we had 10 CWEs that had a score of low, 19 CWEs that had a score of medium, 14 that had a score of high, and 53 that were not scored. We looked at how these tools did on the CWEs when grouped by their likelihood of exploitation rating,which can be seen in figure 3.

When it comes to the CWEs that aren't ranked, we observed slight movement amongst all three tools, but nothing major. Flawfinder performed slightly better, jumping up from 18.09% and 14.10% to 22.71% and 19.05%. Splint performed slightly better on flaws and moderately worse on potential flaws, going from 54.48% and 40.09% to 58.94% and 28.57%. VisualCodeGrepper also improved slightly, going from 23.36% and 31.36% to 28.5% and 42.86%. This category did not show anything significant enough that we did not draw any conclusions about ranked vs unranked CWEs.

For the CWEs that are ranked as high, we observed that both Splint and VisualCodeGrepper did not perform as well as they usually do, while Flawfinder performed slightly

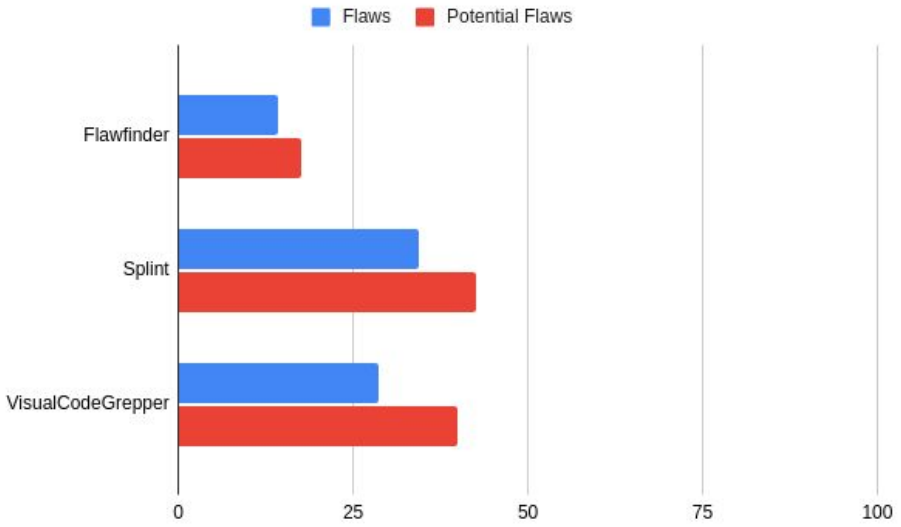

Figure 2: Percentage of Flaws Detected in the Top 10 CWEs by Each Tool

better than normal. Flawfinder showed modest improvements for flaws, increasing from 18.09% up to 22.35%, while potential flaws decreased slightly from 14.10% to 12.94%. Splint performed measurably worse, going from 54.58% and 40.09% to 35.29% for both. VisualCodeGrepper performed slightly worse as well, going from 23.36% and 31.36% to 12.94% and 17.65%.

For the CWEs that are ranked as medium, we observed a more slight decrease in performance amongst all three tools overall. Flawfinder went from 18.09% and 14.10% to 11.76% and 14.28%. Splint from 54.48% and 40.09% to 50.59% and 48.57%. VisualCodeGrepper went from 23.36% and 31.36% to 23.53% and 25.71%.

For the CWEs that are marked as low, we observed the biggest changes in the four groups for all three tools. Flawfinder did not find a single flaw or potential flaw amongst all 10 low CWEs. Splint showed strong increases from 54.58% and 40.09% to 64.44% and 66.67%. VisualCodeGrepper moved in both directions but primarily saw a decline in flaw detection, going from 23.36% and 31.36% to 11.11% and 33.33%.

When we ignored the None category, we were able to notice a few trends between the rankings. For Splint, it was clear that as the severity of the CWE went up, the chances of the flaw or potential flaw being detected went down. With Flawfinder, we observed the opposite trend where as the severity of the CWE went up, the chances of the flaw or potential flaw being detected also went up. There was not as much as an obvious trend with VisualCodeGrepper. As the severity of the CWE went up, VisualCodeGrepper would have a smaller chance of detecting potential flaws, but there is no trend when it comes to regular flaws. Since one would want the more severe vulnerabilities to be discovered if they couldn't find them all, this trend is good for Flawfinder and bad for Splint.

B. Interpretation

We can see above that there are significant differences between the different automated code analysis tools and how well they performed against each other.

For Flawfinder's results we can see that it really under performed in almost all of the categories to everyone other than the highs when compared to VisualCodeGrepper. However, when looking at all the other values it performs as one of the worst tools that we used. In terms of false positives, there was a constant rate of false positives that were caused on every CWE that we scanned. In comparison to Splint's huge amount of generated false positives.

For Splint we can see that it consistently found more flaws than either Flawfinder or VisualCodeGrepper no matter how we broke up the data. This was likely due to the fact that Splint is the only tool that actually compiles the source code, allowing it to detect vulnerabilities that exist in the control flow / data flow better. For false positives, Splint

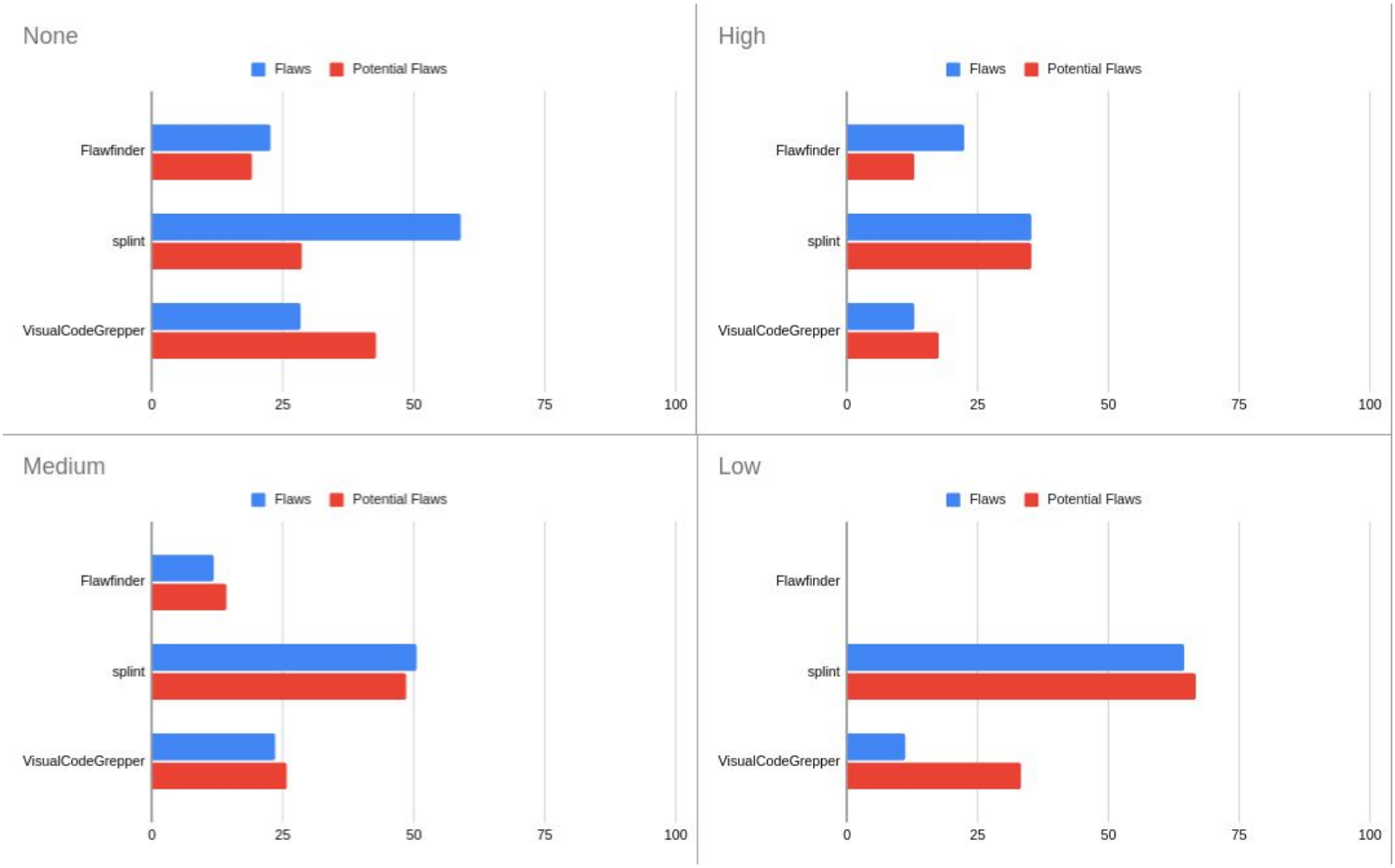

Figure 3: Percentage of flaws detected in each CWE organized by likelihood of exploitation

would flood output with a significant amount of false positives, steaming from header files to libraries themselves. A majority of this was due to how strict that we made the checking since we wanted to see them all. However, from the output for just the CWE's code that was run, we are able to see that there was a vulnerability that was found in almost every other few lines. This was the largest amount of flaws that were found, but almost masked by the flood of false positives. This leads us to believe that if an audit team were to have the personnel and enough time, that this tool would be most successful for them. If you wanted to make sure that you have most of the flaws found automatically, this is the best tool for that. In Figure 3 we can see that out of the tools that were tested for highs, mediums, and lows it excels above the rest.

For VisualCodeGrepper it performed rather well in several different potential flaws, which was a surprise to us. In fact, this tool found more potential weaknesses than it found flaws in every instance of highs, mediums, and lows. Looking at some of the findings, it was interesting to note that this tool, like it's name, looked for strings on every line including the comments. Some of the vulnerabilities and weaknesses included having debug statements in comments and other sections with comments. This tool was the only tool out of the ones that we used that was able to actually find these weaknesses and successfully report on them. For the comparisons of false positives, the tool rarely had any false positives that it generated. So between all of the tools it found a good amount of vulnerabilities, and without generating a whole bunch of false positives to comb through. This setup is especially helpful for smaller offices that don't have the personnel or time to look through all of the noise to find where weaknesses are. This is a good middle ground tool that balances the amount of found vulnerabilities and weaknesses, without having every line being suspicious. However, you regularly want to make sure that you are getting the least amount of false negatives that are missed by the tools. In this case the Splint tool will outweigh all other tools to use.

V. Conclusion

A. Assessment

In this paper, we set out to test how effective the static code analysis tools Flawfinder, Splint, and VisualCodeGrepper were at detecting vulnerable C code. To accomplish this, we ran them all on the Juliet test suite, containing example code for all of the CWEs that apply to C. Splint did the best job overall at detecting vulnerabilities, but also created a lot of false positives and performed worse when the vulnerability's risk was higher. VisualCodeGrepper had the second best rate of detecting vulnerabilities, and generated the least amount of false positives while providing the user with helpful graphs to summarize the results. Flawfinder had the lowest rate at detecting vulnerabilities and also generated a significant amount of false positives, but not as many as Splint.

B. Recommendations

Splint is a good fit for anyone that is able to invest time into using optimal flags and spending time weeding out the false positives. VisualCodeGrepper is the best option for anyone that wants something that is easy to use and effective. We would not recommend that Flawfinder be used in its current state, since it does not have any advantages over Splint or VisualCodeGrepper. We would also not recommend these tools be used by any large scale projects where security is critical, and developers should instead pursue getting a proprietary tool, since all three of these tools missed a lot of flaws. These tools should be used alongside manual code review for companies with a limited security budget based on our findings.

VI. References

[1] Source Code Analysis Tools. [Online]. Available: https://owasp.org/www-community/Source_Code_Analysis_Tools. [Accessed: 14-Nov-2020].

[2] D. A. Wheeler, GitHub. [Online]. Available: https://github.com/david-a-wheeler/Flawfinder. [Accessed: 14-Nov-2020].

[3] D. Evans, LCLint: A Tool for Using Specifications to Check Code. [Online]. Available: https://www.cs.virginia.edu/~evans/sigsoft94.html. [Accessed: 14-Nov-2020].

[4] Splint Home Page. [Online]. Available: http://Splint.org/. [Accessed: 14-Nov-2020].

[5] GitHub. [Online]. Available: https://github.com/Splintchecker/Splint. [Accessed: 14-Nov-2020].

[6] GitHub. [Online]. Available: https://github.com/nccgroup/VCG [Accessed: 14-Nov-2020].

[7] "Test Suites," Software Assurance Reference Dataset. [Online]. Available: https://samate.nist.gov/SRD/testsuite.php. [Accessed: 14-Nov-2020].

[8] "Common Weakness Enumeration," CWE. [Online]. Available: https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html. [Accessed: 14-Nov-2020].

[9] "Common Weakness Enumeration," CWE. [Online]. Available: https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html. [Accessed: 14-Nov-2020].

[10] "Common Weakness Enumeration," CWE. [Online]. Available: https://cwe.mitre.org/cwss/cwss_v1.0.1.html#2.5.3. [Accessed: 14-Nov-2020].