2018 1st International Conference on Data Intelligence and Security

Using Software Engineering Metrics to Evaluate the Quality of Static Code Analysis Tools

Enas A. Alikhashashneh
Department of Computer and Information Science
Indiana University-Purdue University Indianapolis
Indianapolis, IN, USA
ealikhas@iupui.edu

Rajeev R. Raje
Department of Computer and Information Science
Indiana University-Purdue University Indianapolis
Indianapolis, IN, USA
rraje@iupui.edu

James H. Hill
Department of Computer and Information Science
Indiana University-Purdue University Indianapolis
Indianapolis, IN, USA
hilljh@iupui.edu

***Abstract*—This paper presents a framework for evaluating the quality of static code analysis (SCA) tools in the context of different software engineering metrics. The framework supports up to 38 software engineering metrics. We applied the framework against both open-source and commercially available SCA tools. The results of our experiments show that software engineering metrics, such as cyclomatic complexity, fan-out, knots, and essential complexity can impact the ability of a static code analysis tool to identify potential vulnerabilities in source code.**

***Index Terms*—static code analysis tools, evaluation, software engineering metrics**

I. Introduction

Static Code Analysis (SCA) is an automated process of reviewing the source code for potential vulnerabilities that could eventually compromise software security. [1] SCA is traditionally carried out by running many different SCA tools against the code base. [1] For example, developers may choose from a mix of open-source and commercial SCA tools because different SCA tools may have some overlap and produce different results. [2] Likewise, some of the SCA tools may focus on identifying a specific vulnerability from the list of Common Weakness Enumerations (CWEs) [3]. Regardless of the SCA tool used to locate potential vulnerabilities, the end goal of this exercise is to improve software quality.

Because software developers and testers have many SCA tools to choose from, a challenge they face is identifying what tool to use against their code base. As mentioned above, different SCA tools have their strengths, weaknesses, and performance characteristics, which we call its *quality*, in terms of being able to correctly identify potential vulnerabilities. The problem is exacerbated when multiple SCA tools claim to check the same vulnerabilities, but generate different results. In this scenario, at least one of the SCA tools is generating both false positives (FPs), which are locations in source code that are incorrectly labeled to have a flaw, and false negatives (FNs), which are locations in source code that actually have a flaw and are not labeled at all.

In the past, there have been several attempts to evaluate the quality of SCA tools. For example, Knudsen [4] evaluated the ability of Visual Code Grepper, FindBugs and SonarQube to detect SQL, OS command and LDAP injection vulnerabilities against the Java Juliet Test Suite [5]. Their findings show that FindBugs outperforms the other tools in finding the most security flaws in the test cases. Likewise, McLean [6] evaluated several SCA tools against widely used open-source applications, such as: Apache OpenOffice (AOO) [7], PuTTY [8], NMAP [9], and Wireshark [10]. Lastly, Velicheti et al. [11] developed a framework to evaluate different static SCA tools agains the Juliet Test Suite for C++ and Java.

Although there have been several attempts in the past to evaluate the quality of SCA tools, none of the existing studies perform an in-depth analysis of SCA tools based on well-known software engineering metrics. For example, it is unknown how software engineering metrics like Knots [12], Essential Complexity [12], Cyclomatic Complexity [12], Fan-Out (CountOutput) [3], and Fan-In (*i.e.*, CountInput) [13] impact an SCA tool's true positive (TP) rate, which is an SCA tool's ability to correctly label a flaw, FP, and FN rate. Likewise, it is unknown which software engineering metrics have the most impact on the TP, FP, and FN rate for an SCA tool.

These are important questions that need to be answered because if we can understand how different software engineering metrics impact an SCA tool, then there is potential to assist tool developers understand the weak spots in their analytical capabilities. More importantly, we can provide guidelines to software engineers on how to write better code so that SCA tools will generate fewer FPs and FNs, and potentially more TPs.

The main contributions of this paper therefore are as follows:

- An extensible framework for evaluating SCA tools;
- Evaluating two commercial SCA tools and three open-source SCA tools in the context of software engineering

978-1-5386-5762-1/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDIS.2018.00017

metrics against the Juliet Test Suite [17]; and
- It discusses how software engineering metrics impact the TP, FP, and FN rate of the SCA tools;

Our experimental results show that overall commercial SCA tools may uncover more vulnerabilities when compared against open-source SCA tools. Likewise, our results also show that some SCA tools find less than 10% of known flaws when the source code has high complexity and high coupling.

Paper organization. The remainder of this paper is organized as follows: Section II discusses the design and implementation of our framework; Section III presents our experimental results; Section IV compares our work to the other related works; and Section V provides concluding remarks and lessons learned.

II. The Design and Implementation of Our Framework

This section discusses the design of the framework we created to evaluate the quality of SCA tools, which is illustrated in Figure 1. The framework is written in Python, and is designed to be extensible to any SCA tool we want to evaluate either locally or remotely. The framework is also designed to be extensible to different code bases used to evaluate an SCA tool. The key entities in the framework are as follows:

- **Command.** The Command is an interface that defines the different tasks/operations supported by the framework. Such commands include, but are not limited to: parsing the source code; building the knowledge base from a source code base acting as the test suite; analyzing an SCA tool's report; analyzing source code metrics like code complexity, dependency between the functions, and source lines of code (SLOC).
- **Tool.** The Tool is an interface that defines how an SCA tool integrates with the framework. The Tool interface allows the framework to perform several key operations of an SCA tool, such as checking if the SCA tool supports a specific weakness, and checking if the reported bug has the target type (*i.e.*, the type of flaw that the test case under testing targets). For example, the test cases for "CWE476" focus on *NULL Pointer Dereference* flaw [17].
- **DataManager.** The DataManager is used to define the output file format for each command in this framework. For example, the import command will use the DataManager to generate the knowledge base from the test cases. Likewise, the build command will use the DataManager to read the SCA tool report and convert it into a hierarchical abstraction (see Figure 2) that normalizes an SCA tool report.
- **Preprocessor**. The Preprocessor is an interface that allows the framework to preprocess the source code and complete any information that will be missing from an SCA tool report, like the name of the function that contains the known flaws.
- **TestSuite.** The TestSuite is an interface for integrating different test suites into the framework. The test suite is then used by the framework to construct a knowledge base (or test oracle) from the code base. The knowledge base is then used to determine the TP, FP, and FN rate of an SCA tool for the corresponding TestSuite.

A. Commands Supported in the Framework

Each command in the framework is responsible for a given task in evaluating the static code analysis tools. Currently, we have implemented the following set of commands:

- **Import.** The import command is used to create a knowledge base from the source code in a TestSuite. The source code must contain the tags identified in Table I, which originate from the Juliet Test Suite, to generate the correct knowledge base that is comprised of the entities listed in Figure 2.

TABLE I
The different tags used by the framework to correctly label locations of interest.

Tag name	Description
POTENTIAL FLAW	Indicates a flaw that has the target type and it appears based on specific conditions.
INCIDENTAL FLAW	Indicates a flaw that may be detected, but is not the main focus of the test case.
FIXED	Indicates a place in the source code that originally had a flaw and is no longer present.

- **Build.** The build command executes operations of the SCA tool. This includes executing the SCA tool against the TestSuite either locally or remotely; extracting results; and building an actionable knowledge base from the reported bugs.
- **Export.** The export command computes the TP, FP, and FN rate for the SCA tool by comparing its generated output against the constructed knowledge base. An output is identified as a TP when the SCA tool correctly labels the flaw. The output is identified as a FP in three situations: (1) when the SCA tool reports there is a flaw in the source code, but it really does not; (2) when the SCA tool reports the fix tag in the good function or in the good class as flaw; and (3) when the SCA tool reports the flaw tag in the bad function or in the bad class with wrong type. Lastly, an output is identified as a FN when the SCA tool does report a known flaw in a bad function or bad class.
- **Report.** The report command converts the output from the export command into a human readable report.
- **Metric.** The metric command is used to compute the different software engineering metrics for each source file in a test suite. The software engineering metrics are then integrated back into the knowledge base. Currently, we are using the Understand (https://scitools.com/) tool to generate our software engineering metrics. Our framework, however, is not limited to only using Understand.

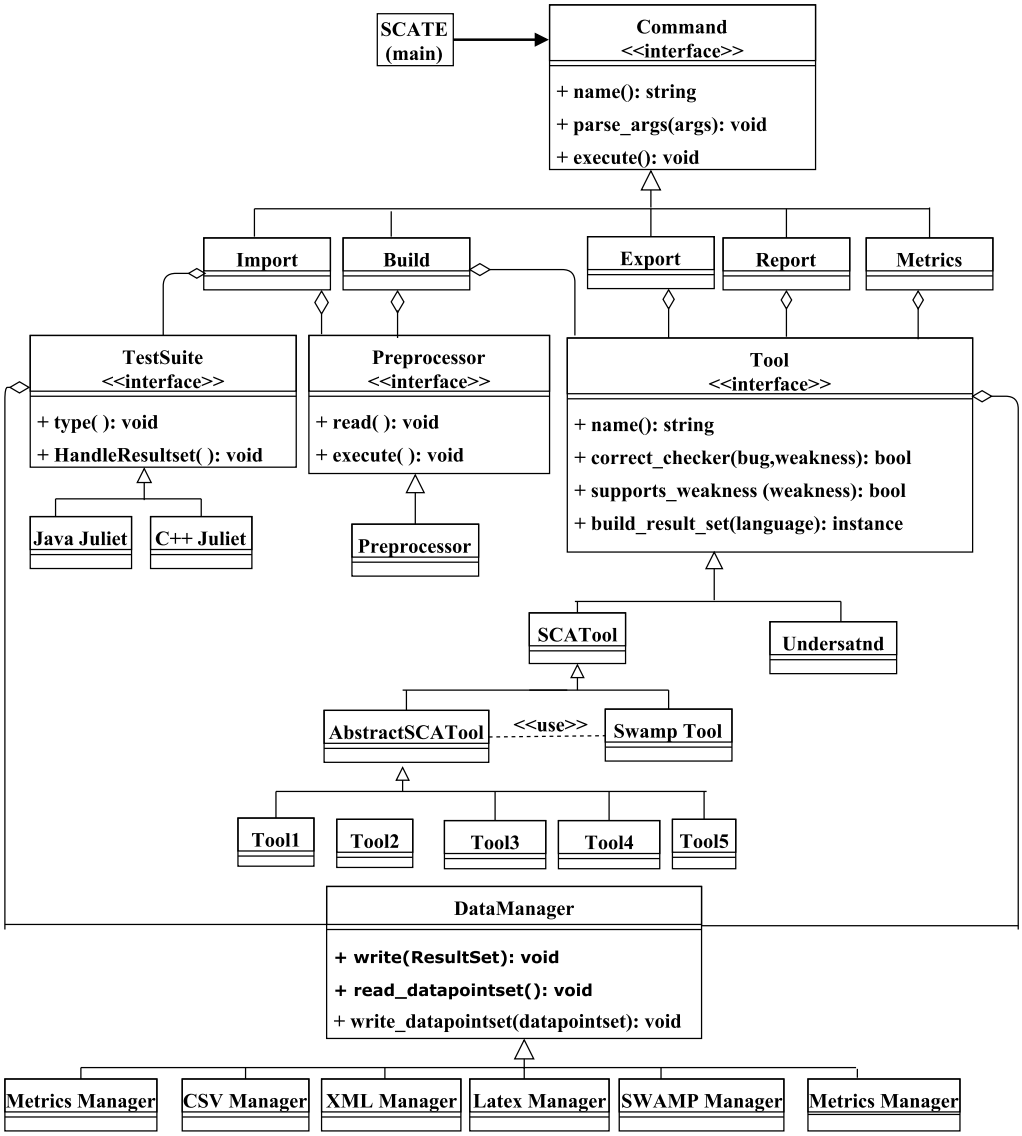

Fig. 1. The general design of our extensible framework for evaluating static code analysis tools.

B. Integrating With the SWAMP

The SoftWare Assurance MarketPlace (SWAMP) is a cloud environment for running source code against different static code analysis tools. The SWAMP provides 19 open-source SCA tool and 4 commercial SCA tools. Its SCA tools support five programming languages: C/C++, Java, Python, and Ruby. There are two ways to use the SWAMP. Either you use it via their hosted cloud computing platform (mir-swamp.org), or you can use the SWAMP-in-a-Box(SiB) open-source distribution [16].

Integrating our framework with SWAMP allows the developer to evaluate a wide variety of SCA tools in the context of software engineering metrics. To perform this integration, we implemented the following two components:

- **SWAMPManager.** The SWAMPManager reads the SWAMP Common Assessment Result Format (SCARF) files, which is the common format the SWAMP uses to for reporting the results of an SCA tool, and builds the abstract model hierarchy (see Figure 2) for our framework.
- **SWAMPTool.** The SWAMPTool acts as a proxy for running SCA tools run remotely in the SWAMP.

C. Classifying SCA Tools Output

We faced several challenges when evaluating the quality of SCA tools. For example, many of the open-source SCA tools do not document the Common Weakness Enumeration (CWE) [3] their checkers identify, which is the single entity in an SCA tool responsible for identifying a single problem. This is important because it allows us to correctly identify when an

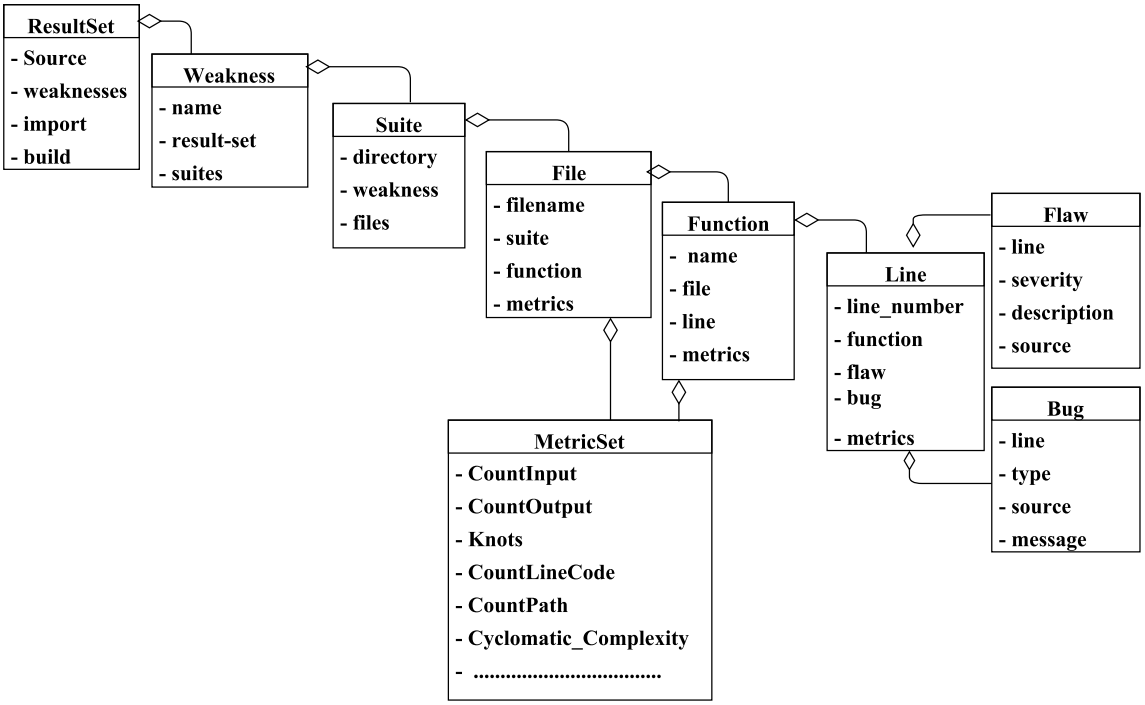

Fig. 2. Entities that make the knowledge base for a test suite in the framework.

SCA tool is generating a TP, FP, or a FN. In our work, we use the following approach to classify an SCA tool error message as a TP, FP, or FN:

- **True Positive (TP).** We consider an error message to be a TP if the SCA tool highlights the predefined flaw of the target type in the correct location. For example, in the Juliet Test Suite the reported flaw should be either in the function or class with the word `bad` in its name.
- **False Positive (FP).** We consider an error message to be a FP if the SCA tool highlights a flaw with incorrect type, the fix tag as flaw, or the flaw of the incorrect type in the bad function.
- **False Negative (FN).** We consider an error message to be a FN if the SCA tool does not identify the flaw for the corresponding weakness in a function or class that contains the word `bad`.

III. Experimental Results

This section discusses our experimental results for evaluating SCA tools in the context of different software engineering metrics. Due to the large number of software engineering metrics we support, we cannot include results for each one. We therefore only focus on the ones that yield results that highlight software engineering metrics that can impact an SCA tools ability to correctly identify potential flaws[1].

[1]All the results are available in https://github.com/ealikhas/Results

A. Experimental Setup

We used the Juliet Test suites to perform our experiments against the SCA tools. To setup and execute our experiment, we executed the following steps:

- We used the import command to parse the source files in the Juliet Test suites and build a knowledge base with the ground truth. The ground truth contains information about the flaws, such as function name and line number, labeled in the test cases. In this step, the framework parses approximately 61,000 C++ or Java files depending on what SCA tool we are targeting for evaluation.
- We ran the SCA tools either locally or remotely in the SWAMP against the source code in the Juliet Test Suite. We then capture the generated output of the SCA tool because we need this to evaluate if the SCA tool is correctly labeling the flaws in the source code.
- Next, we assessed the value of the software engineering metrics for each CWE by executing Understand against the source code for the corresponding test cases in Juliet.
- Last, we compared the ground truth with the output generated by the SCA tool, and assessed the performance of the SCA tool with respect to the different software engineering metrics.

We applied the steps above against the following SCA tools[2]: an open-source SCA tool that supports C++ program-

[2]At the time of writing this paper we keep the name of the SCA tools confidential, because we do not have time to discuss the results with the SCA tools' vendors

ming language (aka Tool1), two open-source SCA tools that support Java programming language(aka Tool4 and Tool5), and two commercial SCA tools that support C++ programming language(aka Tool2 and Tool3). Our framework, however, is not limited to only using these tools.

1) Selected software engineering metrics.: As mentioned above, we use Understand to compute the values of different software engineering metrics. Understand supports over 38 software engineering metrics. We, however, used some of the following software engineering metrics from Understand:

- **Basic Count Line Metrics.** These metrics retrieve information about each line in the source code in the scope of a function, a class, or a file. Within this family of metrics, we used CountLineBlank, CountLineCode, CountLineComment, CountLineCodeDecl, CountLineCodeExe, CountLineInactive, and CountLinePreprocessor.
- **Basic Count Metrics.** These metrics are divided into two main sets: CountDeclClass, which retrieve the number of classes in the file; and, CountDeclFunction, which retrieve the number of functions declared in the file.
- **Basic Token Metrics.** These metrics retrieve information about the source code complexity. Within this family of metrics, we used Cyclomatic, CyclomaticModified, CyclomaticStrict, Countsemicolon, and MaxNesting.
- **Control Flow Metrics.** These metrics are computed from the control flow graph of the function. Within this family of metrics, we used Knots, Essential, MinEssentialKnots, MaxEssentialKnots, and CountPaths.
- **Miscellaneous.** We used CountInput and CountOutput metric. These metrics are generally estimated at three levels: project-level, file-level, and function-level. In our experiments, we use the file- and function-level metrics only.

B. Experimental Results

Although the five SCA tools have been evaluated against 91 CWEs in the test cases. The results of the following CWEs have been discussed in detail:

- **CWE-369: Divide by Zero.** This weakness occurs when an unexpected value is provided to the product/calculation, or if an error occurs that is not properly detected [3].
- **CWE-415: Double Free.** This weakness occurs when the product calls free() twice on the same memory address. This weakness may lead to modification of unexpected memory locations [3].
- **CWE-457: Use of Uninitialized Variable.** This weakness occurs when the source code uses a variable that has not been initialized. This weakness may lead to unpredictable or unintended results [3].
- **CWE-762: Mismatched Memory Management Routines.** This weakness occurs when the application attempts to return a memory resource to the system, but it calls a release function [3].

Following the behavior of SCA tools based on a set of software engineering metrics have been discussed in detail.

1) Essential Complexity: Essential complexity is a measure of the source codes complexity after iteratively replacing all the well-structured control structures, such as if-then-else and while loops with a single statement. Figure 3 illustrates the behavior of SCA tools for CWE-762 in the context of Essential Complexity. As shown in this figure, Tool2 and Tool3 find more flaws than the other tools. We can infer this finding from the low value of the FN rate for Tool2 and Tool3 when the value of the Essential Complexity metric increases. On the other hand, Tool4 reports the highest number of FN warnings when the source code has a high value of Essential Complexity (e.g. value=5). Tool5 and Tool1 take second and third places after Tool4. Thus, Tool2 can handle the source code that has a high degree of structuredness and low degree of quality better than the other tools.

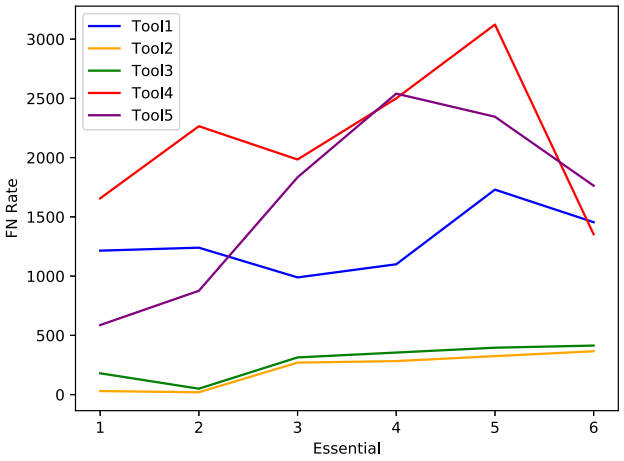

Fig. 3. The behavior of SCA tools based on Essential Complexity for CWE-762.

2) CountOutput: This metric belongs to object-oriented metrics, and computes the number of outputs of the function in the source code. The outputs may be classified into functions calls, parameters set/modify, and global variables set/modify. We use Understand to compute the value of this software engineering metric at function or at method level, which follows the information approach of the FANOUT.

To measure the coupling between the functions or the methods in the same test case we used Fan-In and Fan-Out metrics where a function f Fan-out coupling is the number of the functions that depend on function f. Likewise, a function f Fan-in coupling is the number of the functions that function f pends upon. This metric is important to developers because it helps developers understand how likely fixing a flaw in one function negatively impacts other functions.

Figure 4 shows the results of the SCA tools for CWE-369 based on the values of the CountOutput (Fan-Out) metric. As shown in this figure, Tool3 initially finds more flaws than the other tools when the value of the CountOutput metric

is low. On the other hand, as the value of the CountOutput metric increases, which means the functions in the source code have high degree of coupling, Tool3 finds fewer flaws than Tool2 and the other tools. Likewise, Tool2 finds more flaws than the other tools when the value of the metric increases. Unfortunately, none of the tools can find any flaws in the source code when the value of the metric becomes more than six.

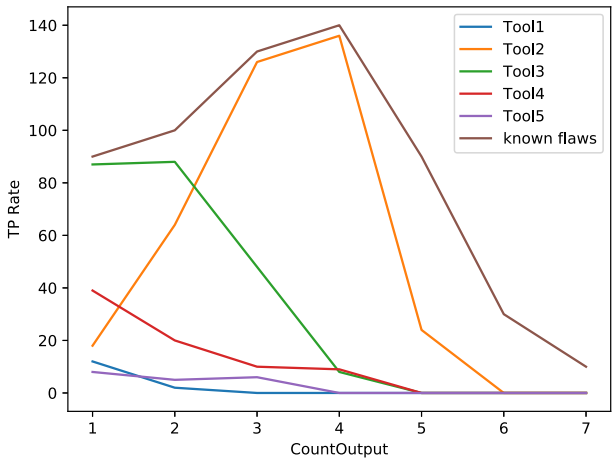

Fig. 4. The behavior of SCA tools based on CountOutput for CWE-369.

3) Cyclomatic Complexity: This metric computes the source code complexity by using the McCabe Cyclomatic complexity where the complexity of any structured source code with only one entrance and one exit point is equal to the number of decision points contained in that source code plus one [15]. Figure 5 highlights the SCA tool results for CWE-369 based on the Cyclomatic Complexity metric. As shown in this figure, Tool3 initially finds the highest number of flaws in the test cases when the complexity is low. As the metric increases, Tool2 finds more flaws than Tool1 and Tool3. This implies that Tool2 can handle source code with a high level of the complexity by ignoring the occurrence of the logical operators perfectly well.

4) Knots: This metric belongs to the Complexity metrics, and is a measure of overlapping jumps (i.e., for the corresponding source code, Knots equals to the number of crossing of the lines that determine where every jump in the flow of control occurs) [15]. Figure 6 illustrates the behavior of the SCA tools for CWE-369 based on the Knots. As shown in the figure, Tool3 finds the most flaws in the test cases as the number of the overlapping jumps in the source code increases. On the other hand, Tool1 finds the lowest number of flaws in the test cases. This implies that Tool3 handles the source code with a high value of Knots better than Tool2. Likewise, Tool2 handles the source code with medium value of knots better than Tool1.

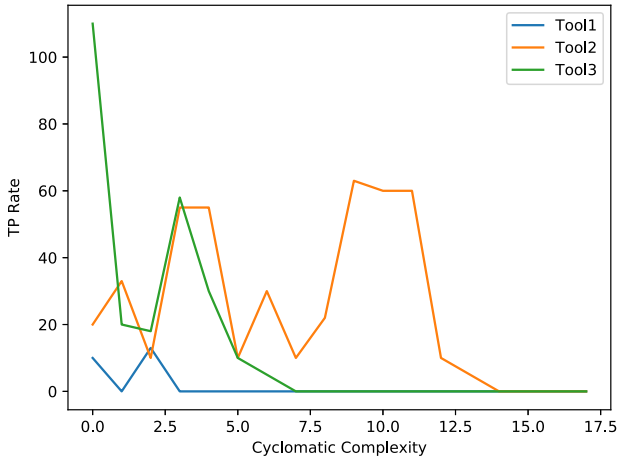

Fig. 5. The behavior of SCA tools based on Cyclomatic Complexity for CWE-369.

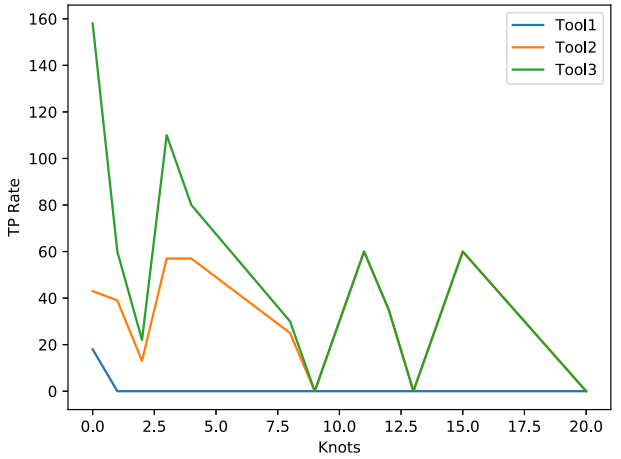

Fig. 6. The behavior of SCA tools based on Knots for CWE-369.

5) CountInput: This metric belongs to object-oriented metrics, and computes the number of inputs that the function uses plus the number of the other functions calling the function. The inputs may be classified into global variables that are used in the function, and the in parameters used in the function [15]. For our experiments, Understand computed the value of this software engineering metric at the function or method level. Figure 7 illustrates the behavior of the SCA tools for CWE-457 based on the CountInput. As shown in the figure, the number of uncovered flaws by both Tool2 and Tool3 decrease as the value of the CountInput increases. Tool2 finds more flaws than Tool3 when the source code has a high degree of Fan-In. In other words, when the functions in the given source code have a high number of calling functions and global variables read,

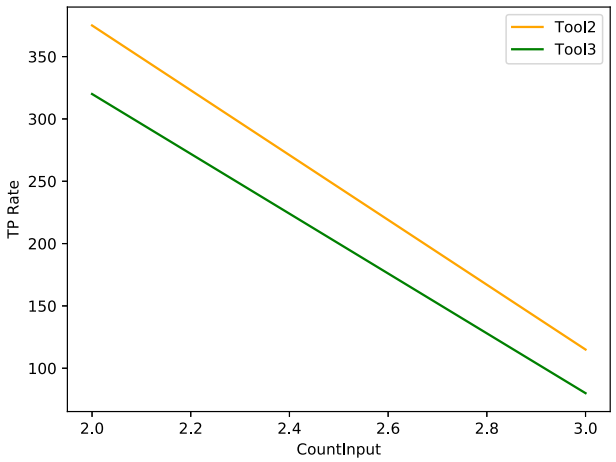

Fig. 7. The behavior of SCA tools based on CountInput for CWE-457.

Tool2 performs better than Tool3.

6) CountPathLog: This metric belongs to Complexity Metrics, and computes the logarithm of the total number of unique paths in the function— excluding the abnormal exits and the goto statements. Figure 8 showcases the results for CWE-369 based on the CountLogPath metric for SCA tools. As shown in the figure, Tool3 finds more flaws than Tool1 and Tool1 finds more flaws than Tool2 when the source code either does not include any control constructs and decision structures or includes a low number of unique paths. As the number of paths in the source code increases, the number of flaws that were uncovered by Tool3 and Tool1 declines to zero, while Tool2 finds more flaws.

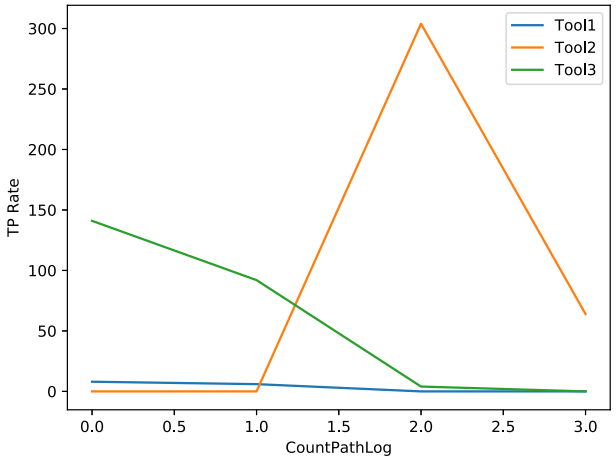

Fig. 8. The behavior of SCA tools based on CountPathLog for CWE-369.

C. Threats to Validity

For this research, the main threat to external validity is most SCA tools do not have an easily accessible mapping of CWEs to checkers, which we use to classify the tools' reported bugs as TP, FP, or FN. This threat causes many of the reported bugs considered as an FN not as TP and in sometimes increases the number of FP.

IV. Related Works

Knudsen et al. [4] test the ability of three of the open-source SCA tools: Visual Code Grepper,FindBugs and SonarQube to detect SQL, OS command and LDAP injection vulnerabilities against the Juliet test suite v1.2 for Java. The performance of these tools are evaluated using the OWASP Benchmark Project, which provides a system to evaluate the performance of the SCA tools using the Youden index metric [14]. Knudsen concludes that FindBugs is to be considered the best tool in detecting the LDAP injections. This work differs from our work in two main ways. First, our work evaluates the SCA tools in the context of different software engineering metrics. Second, we evaluate the SCA tools against 91 CWEs, while in the Knudsen et al. evaluates the SCA tools against only three CWEs.

McLean et al. [6] compared two SCA tools, RATS and Flawfinder, and their ability to find vulnerabilities in three open-source applications. The results of their study concluded that the Flawfinder uncovered 3,189 flaws and while RATS found only 1,415 flaws. Second, both SCA tools produce a large number of false positives. Lastly, the author recommends that developers analyze their source code using Flawfinder tool because Flawfinder reports more valuable information to the developer than RATS. Our work differs from McLean et al. in that our work evaluates the SCA tools in the context of the software engineering metrics. Secondly, we focus on well-known weaknesses (i.e., CWEs) in the Juliet Test Suite.

V. Concluding Remarks

This paper presented our framework and work on evaluating five SCA tools in the context of different software engineering metrics that we can measure in the source code. As shown in our results, the performance (or quality) of an SCA tool can be impacted by the value of various software engineering metrics. Based on our experience gained from this research, we have the following conclusions and future directions for the research.

- Based on our current results the software engineering metrics can be listed in decreasing order based on how they affect the TP rate for each SCA tool. From the experimental results, we observed that most of the five SCA tools achieve lower value of TP rate when the source code has a high degree of complexity. This is because the source code will be more complicated, error-prone, and difficult-to-understand. The second type of software engineering metric that leads the SCA tools to find low number of flaws are object-oriented metrics such as CountOutput. For example, when the source code

includes functions with high coupling this will decrease an SCA tool's ability to find more flaws. Last, volume metrics have the least negative impact on the number of flaws identified by an SCA tool.

- Choosing an SCA tool for a given source code depends on several important factors, such as the weakness the developers want to test for, the value of one or more of the source code metrics, and finally the structure of the given source code. For example, the SCA Tool2 does not perform well with the source code that has a high degree of Knots. Tool 2, however, performs better when the source code has a high level of coupling (*e.g.*, CountOutput).
- The experimental results show that there is a relationship between the number of the uncovered flaws by SCA tools and the value of the software engineering metrics. Thus, we are planning to use a variety of Machine Learning techniques to predict the performance of an SCA tool (*i.e.*, ability to generate a TP, FP, or FN) on a given piece of source code in the context of different software engineering metrics.
- Five of SCA tools that support both the C++ and Java programing language and also its rules mapping to CWEs have been evaluated. In the future, we plan to extend this research to cover more open-source SCA tools that analyze Python, Ruby, C++, and Java source code.
- SCA tools results report a large number of false warnings (*e.g.*, FP). The manual inspections of the false warnings is an unavoidable, time-consuming, and costly process. In future, we plan to extend the functionality of proposed framework to address this problem either by improving the precision of the analysis or by post-processing the false warnings after they are generated.

REFERENCES

[1] Owasp. "Static Code Analysis." Internet: https://www.owasp.org/index.php/Static_Code_Analysis, Sep. 29,2017 [Oct. 15,2017].

[2] Wikipedia. "List of tools for static code analysisWikipedia, The Free Encyclopedia." Internet: https://en.wikipedia.org/w/index.php?title=List_of_tools _for_static_code_analysis&oldid=739038439, Sep. 12,2016 [Sep. 13,2017].

[3] U.S. Department of Homeland Security. "The Common Weakness Enumeration (CWE) Initiative, MITRE Corporation." Internet:http://cwe.mitre.org/index.html, Jan 16, 2018 [Jan 20, 2018].

[4] A. R. Knudsen. "Evaluating the ability of static code analysis tools to detect injection vulnerabilities." Bachelors thesis, Ume University, Sweden, 2016.

[5] T. Boland, and P. E. Black. "Juliet 1.1 C/C++ and Java Test Suite." IEEE CS, vol. 45, pp: 88-90, 10 - Oct.2012.

[6] R. K. McLean, "Comparing Static Security Analysis Tools Using Open Source Software." International Conference on Software Security and Reliability Companion, pp:68-74, 2012.

[7] W. R. C. Christopher. Openoffice 3.4: Writer: Black and White. CreateSpace Independent Publishing Platform ,2012, pp. 228.

[8] K. Dooley, and I. Brown. Cisco IOS Cookbook, 2nd Edition Field-Tested Solutions to Cisco Router Problems. O'Reilly Media, December 2008, pp: 1248.

[9] G. F. Lyon. Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning. Insecure.Com, LLC,January 1, 2009.

[10] Angela O., G. Ramirez, and J. Beale. Wireshark & Ethereal network protocol analyzer toolkit. Syngress, 2006.

[11] L. M. R. Velicheti, D. C. Feiock, M. Peiris, R. Raje, and J. H. Hill. "Towards Modeling the Behavior of Static Code Analysis Tools." 9th Annual Cyber and Information Security Research Conference, 2014, pp:17-20.

[12] D. Coupal, and P. N. Robillard. "Factor analysis of source code metrics." Journal of Systems and Software, Vol. 12, 1990, Pages 263-269.

[13] G. Botterweck, and C. Werner.Mastering Scale and Complexity in Software Reuse: 16th International Conference on Software Reuse, ICSR 2017, Salvador, Brazil, May 29-31, 2017.

[14] Owasp. "OWASP Benchmark Project." Internet: https://www.owasp.org/index.php/Benchmark, Oct. 10,2017 [Oct. 15,2017].

[15] Sci Tool. "Understand." Internet: https://scitools.com/, Oct. 10,2017 [Oct. 15,2017].

[16] M. Livny, and B. Miller. "The Case for an Open and Evolving Software Assurance Framework." Internet: http://continuousassurance.org/wp-content/uploads/2013/11/White-Paper-Evolving-Framework.pdf, Nov. 20,2017 [Nov. 28,2017].

[17] Center for Assured Software, National Security Agency. "Juliet Test Suite v1.1 for C/C++ User Guide." NIST, 2012.