Towards a General-purpose Framework for Evaluating the Quality of Static Code Analysis Tools

S²ERC-TR-312
Lakshmi Manohar Rao Velicheti, Dennis C. Feiock, Manjula Peiris, Rajeev Raje, and James Hill
Indianapolis University Purdue University
Indianapolis
February 2014
Towards a General-purpose Framework for Evaluating the Quality of Static Code Analysis Tools

Lakshmi Manohar Rao Velicheti, Dennis C. Feiock, Manjula Peiris, Rajeev Raje, and James H. Hill
Department of Computer and Information Science
Indiana University-Purdue University Indianapolis
Indianapolis, IN USA
Email: lveliche@cs.iupui.edu, dfeiock@iupui.edu, {tmpeiris, hillj}@cs.iupui.edu

Abstract—This paper presents a study of two Commercial off-the-shelf (COTS) static code analysis (SCA) tools. In our study we identify and assess the performance of each tool in detecting and identifying weaknesses and vulnerabilities in source code. The goal of this study is to evaluate how well different SCA tools locate flaws in source code (i.e., the quality of the tool) and model the behavior of SCA tools. Our initial results show that our framework is able to showcase how different software engineering metrics (e.g., cyclomatic complexity and max nesting) impact a SCA tools ability to locate flaws.

Index Terms—static code analysis, Juliet Test Suite, tools, evaluation, framework

I. INTRODUCTION

Static code analysis (SCA) [1] is the process of analyzing the source code of a program for flaws without executing the program. SCA tools aid developers in quickly identifying flaws that can jeopardize the security and integrity of the program [2]. Given the number of SCA tools available [3], a major challenge facing software developers (and organizations that produce software) is selecting a tool (or a set of tools) that will be able to detect potential flaws in their programs. This is important because failing to detect flaws in the source code could result in catastrophic events [4], [5].

To assist with this challenge, we have created an open-source, extensible Python framework called Static Code Analysis Tool Evaluator (SCATE). The goal of SCATE is to provide a standardized methodology for evaluating the quality of a SCA tool and model its behavior. By behavior modeling, we mean creating a knowledge base that is able to predict when a given SCA tool will identify a potential flaw in the source code. Unlike existing studies, we model behavior based on well-defined software engineering metrics, such as cyclomatic complexity, lines of code, maximum nesting that impact a SCA tools ability to analyze the code [6].

The main contributions of this paper are as follows:

• It proposes a method for identifying false positives and false negatives in reports generated by a SCA tool; and
• It quantitatively evaluates two COTS SCA tools against the Juliet Test Suite [7] from the National Institute of Standards and Technology (www.nist.gov), which are test cases that implement different Common Weakness Enumerations (CWEs) [8].

Due to licensing agreements, we do not include the names of the SCA tools we used in our study. Our initial results show that each tools perform different on the Juliet Test Suite, as expected. Although this is the case, we learned that if software engineers understand the quality of their source code based on well-defined software engineering metrics, then they will understand how to better apply a given SCA tool instead of searching for the SCA tool that is the “silver bullet”.

Paper organization. The remainder of this paper is organized as follows: Section II discusses the design and implementation of SCATE; Section III presents the current results of our study; Section IV compares our work to other related works; and Section V presents concluding remarks, lessons learned, and future research directions.

II. DESIGN AND IMPLEMENTATION OF SCATE FRAMEWORK

Figure 1 shows the architecture of the SCATE framework. As mentioned in Section I, SCATE is implemented using Python. We selected Python because it allows us to quickly architect a framework using software design patterns [9] so we can focus on evaluating the quality of different SCA tools. Likewise, the SCATE framework is designed to be extensible to many different SCA tools, code bases, and reporting formats.

As shown in this figure 1:

• Command is an interface that implements the Command [9] software design pattern. Implementation of this interface defines the different operations that SCATE can perform.
• TestSuite is an interface for defining Wrapper Facades [9] to different test suites used to evaluate a SCA tool.
Implementations of this interface convert source code with known flaws into abstractions specified in Figure 2.

- **Tool** is another interface for defining Wrapper Facades to different SCA tools. The tool facades must implement tool specific command objects so that SCATE understands how to interact with it.
- The **DataManager** is the abstraction that defines the different import/export formats supported by SCATE. The test suite objects uses the data manager to build the actionable knowledge base.

SCATE supports four commands: import, build, analyze and report. The import command builds the ground truth for the target test suite. The build command runs a SCA tool against all the test cases in the target test suite, and exports all flaws reported by the tool into an actionable knowledge base. The analyze command uses the ground truth to filter the tools’ results. Lastly, the report command uses the filtered results to generate a target document for reporting purposes. Currently, SCATE supports generating reporting documents in LaTeX, which can be used to generate a PDF, and CSV, which can be imported into Excel to create pivot charts [10].

### A. Normalizing SCA Tool Output

One of the challenges of evaluating different SCA tools under a single framework is each SCA tool typically generates output in a different format. For example, the output of one SCA tools will contain the filename, routine, and line number of the potential flaw. Another tool, on the other hand, may not include the line number. This can make it hard to compare and contrast different SCA tools against the same code base.

To address this problem, we use a hierarchical abstraction model to normalize a SCA tool’s output for comparison. As shown in Figure 2, the abstractions are a set of aggregated result elements (*i.e.*, the result set contains weaknesses, a weakness contains test suites, and a test suite contains flaws). The weaknesses in SCATE represent the Common Weakness Enumeration (CWEs) that can exist in source code, and are identified by SCA tools. The test suite is a set of different test cases for the weakness. Lastly, the flaw is the filename, routine, and line number in a specific test case. Not all SCA tools will support each attribute of a flaw, which can impact our ability to compare results between different SCA tools and identify false positives and false negatives.

We also use CodeDx [11], which is a visualization tool for identifying and managing weaknesses and vulnerabilities in

---

**Fig. 1.** Overview of the SCATE framework for evaluating the quality of a SCA tool.

**Fig. 2.** Hierarchical abstractions used to normalize the output of a SCA tool.
source code, to assist with normalizing SCA tool output. Code Dx imports different tool outputs so developers can visually look at different aspects of the results.

For example, Figure 3 showcases the weakness flow for different SCA tools based on CWE’s, severity levels, and error type (i.e., false positive and false negative). As shown in this figure, a weakness with severity level medium for CWE-404, which is improper resource shutdown or release, was identified by FindBugs [12].

B. Identifying Flaws In Test Suites for Training Purposes

One of the goals of the SCATE framework is to easily import different test suites for evaluation against different SCA tools. To accomplish this goal, however, we need a standard method for identifying flaws in source code. This will assist with training the model we plan to apply on unknown source code.

We use the approach currently practiced in the Juliet Test Suite [7] to define flaws for importing. Listing 1 shows a test case from the Juliet Test Suite. As shown in this listing, the flaw is identified in source code by the FLAW marker. The line proceeding the FLAW marker is considered the flaw in the source code.

```c
void CWE835_Infinite_Loop__while_true__01_bad() {
    int i = 0;

    /* FLAW: Infinite Loop -- while(true) with no break point */
    while(1) {
        printf("%d\n", i);
        i++;
    }
}
```

Listing 1. Example of a flaw in source code identified with the FLAW marker.

We also support the additional markers, as practiced in the Juliet Test Suite:

- **POTENTIAL FLAW** is location in source code where a flaw may, or may not, manifest itself depending upon how the routine is invoked.
- **INCIDENTAL FLAW** is a flaw that exist in the source code, but its does not represent the current CWE under evaluation. For example, source code for evaluating CWE-122, which is heap-based buffer overflow, can also contain dead code, which is CWE-561. In this case, the dead code in the source code under evaluation would be marked as an incidental flaw.
- **FIX** identifies the corrected version of the flaw. There may be one or more fix in a test case for any marked flaw.

Once all flaws in the test suite are marked, SCATE parses the test suite and builds a knowledge base of known flaws. We can then use the knowledge base by executing the SCA tool against the test suite, capturing the SCA tools output, and validating the captured output against the knowledge base by checking if the SCA tool found the known flaws.

C. Identifying False Positives and False Negatives

It is important to identify false positive and false negatives generated by SCA tools. This is because a false positive represents a location flagged in source code that has a potential flaw, but—in reality—it does not have a flaw. Likewise, a false negative is when a tools does not report a flaw when there is a flaw. False positives can cause developers to spend countless hours searching for a problems that do not exist, or a developer may ignore it if it is too many. False negatives can cause developers to believe their code has no flaws, when in fact it does contain flaws.

When a SCA tool detects a flaw of the correct type in a fixed version of the source code, then we consider the report a false positive. As per NIST’s nomenclature in the Juliet Test Suite, a tool reporting a target flaw type in a good function that has good in its name, or a class in a test case containing good in its filename, is considered a false positive [7].

When a SCA tool does not report a flaw of the correct type, then we consider it a false negative. As per NIST’s nomenclature, if a tool does not report a target flaw type in a bad function that has bad in its name, or a class in a test case containing bad in its filename, is considered a false negative [7].

III. USING SCATE TO EVALUATE SCA TOOLS

We validate SCATE and its ability to evaluate the quality of a SCA tool by using the Juliet Test Suite for C/C++ v1.1 from NIST. We selected the Juliet Test Suite because it follows best testing practices and provides standardization across multiple implementations of CWEs.
A. Experimental Setup

Using the import features of SCATE, we scanned to entire Juliet Test Suite and created a knowledge base on the known flaws. In total, we created a test suite that contained 177,031 test cases across 91 CWEs. We then installed each SCA tool in the System Integration Lab at IUPUI (www.emulab.cs.iupui.edu). The System Integration Lab is a test bed powered by Emulab [13] software. Each experimental node we used in the test bed was a Dell PowerEdge R415 experimental nodes with 4-core, 2.6 Ghz processors, 8 GB PC10600 RAM, 1 Gbps control network connection. Finally, we executed two COTS SCA tools against the Juliet Test Suite knowledge base we created using SCATE.

B. Discussion of General Results

Figure 4 shows the overall CWE results for both COTS SCA tools we used in our initial evaluation. As shown in this figure, Expected represents the number of flaws in the knowledge base. Tool 1 and Tool 2 represent the number of flaws per CWE identified by the corresponding COTS SCA tool.

As shown in this figure, the SCA tools do not exhibit the same behavior. By behavior, we mean the SCA tools ability to find the same weakness in the same test case. Because of this observation, it is hard to claim that one tool is better than another tool just by looking at the overall results produced by SCATE against the Juliet Test Suite. For example, can the tool that finds no CWE-392 in the test cases by considered a bad tool?

These are hard questions to answer, and really do not provide developers with any valuable answers—as we learned. This is why we believe that reporting how many flaws each SCA tool identified in general is not a valuable result. Instead, we focus on evaluating each tool in the context of software engineering metrics about each test case. We focus on software engineering metrics because (1) organizations focus on software engineering metrics to characterize the quality of its source code; (2) we believe software engineering metrics can remove bias associated the associated with the test suite, such as the test cases do not reflect real-world source code; and (3) it will provide software developers (and organizations) with powerful insight on what tool functions best on source code is written, or give motivation for improving how source code is written (i.e., changing the organization’s culture).

C. Discussion of CWE-specific Results

Although we are evaluating the tools against 91 CWEs in the Juliet Test Suite, we discuss the results of the following CWEs in detail:

- **CWE-121: Stack-based buffer overflow condition.** This is when a buffer is written beyond its size was allocated on the stack.
- **CWE-124: Buffer underflow condition.** This is when a buffer is written to using an index, or pointer, that references a memory location prior to the beginning of the buffer.
- **CWE-127: Buffer under read condition.** This is when a buffer is read using an index, or pointer, that references memory locations prior to the targeted buffer.
- **CWE-194: Unexpected sign extension.** This is when an operation on a number causes it to transform into a larger data type. When the original number is negative, this weakness can produce unexpected values.
- **CWE-390: Detection of Error Condition Without Action.** This is when the implementation in source code detects a specific error, but takes no actions to handle the error.
- **CWE-401: Memory Leak.** This is when the implementation in source code improperly releases (or fails to release) memory before removing the last reference of an object.
- **CWE-835: Infinite loop.** This is when the implementation in source code has an iteration, or loop, with an exit condition that cannot be reached.

We selected the following CWEs because it showcases what we were able to discover using SCATE.

1) Analysis via Flow Structure: As discussed in section I, the Juliet Test Suite from NIST was selected because of standardized approach to defining test cases. One component of the standardization is flaw metadata. The metadata we collect includes the CWE, short name of the flaw, functional variant, two digit flow structure number, and an optional subfile indicator. A flow structure number indicates the type of control or data flow used [7]. An optional subfile indicator is used when a test case is split between different files. This information helps us approximate a tool’s behavior because we can see how well the tool finds flaws for different permutations.
(i.e., flow structures).

Figure 5 illustrates the behavior of both COTS SCA tools for flow structures. As shown in this figure, we discovered that Tool 1 finds more of the different kinds of permutations. Likewise, and Tool 2 found more flaws that Tool 1 for permutation 1 and 12. We also discovered that Tool 1 did not perform well on permutation 5, 7, 9, and 12, which are flow structures that vary usage of “if” statements.

2) Analysis via Software Engineering Metrics: We used Understand (www.scitools.com) to calculate the following metrics of each test case from the Juliet Test Suite:

- **Cyclomatic complexity**, which is the measure of number of independent path in the program, and
- **Maximum nesting**, which is the level of control constructs.

We then combined the metrics above with the analysis results of the SCA tool and generated a comma-separated value (CSV) file using SCATE. We then imported the combined results into Excel and used pivot charts to filter the data and search for trends.

**Cyclomatic complexity.** Figure 6 highlights the results for CWE-121 when we focused on cyclomatic complexity. As shown in this figure, Tool 1 handles cyclomatic complexity better than Tool 2. Moreover, Tool 1 does a better job at handling source code with high cyclomatic complexity than Tool 2. Lastly, both tools find less flaws as the cyclomatic complexity of the source code increases, which is expected.

Figure 7 highlights the cyclomatic complexity results for CWE-124. In this figure, we see that initially Tool 2 finds more flaws than Tool 1. As cyclomatic complexity increases, Tool 1 begins to find more flaws when compared to Tool 2. Finally, both tools find fewer flaws as cyclomatic complexity increases, which is similar to the results for CWE-121.

From these results, a software developer cannot conclude that Tool 1 is better than Tool 2. Likewise, a software developer cannot conclude that Tool 2 is better than Tool 1. It depends on what CWE is under investigation, and the structure of the source code. If software developers are writing source code that has high complexity, then Tool 1 may be a better choice. If their source code has low cyclomatic complexity, then Tool 2 may suffice. By using SCATE, we are able to gather the necessary information and analyze it so software developers can understand how a SCA tool will perform in the context of how they implement source code.

**Maximum nesting.** As previously explained, maximum nesting is the level of control constructs. For example, if there is a single conditional loop, then its maximum nesting is 1. On the other hand, if there is a conditional loop inside another conditional loop, then maximum nesting is 2. When there are no conditional loops, the value of maximum nesting is 0.

Figure 8 showcase the results for CWE-127 based on the maximum nesting of each test case. As shown in this figure, Tool 1 is able to find more flaws when compared to Tool 2 as the value of maximum nesting increases. Both tools, however, have a decrease in number of flaws identified as maximum
nesting increases.

Figure 9, however, showcases the results for CWE-194 based on maximum nesting of each test case. As shown in these results, Tool 2 perform better than Tool 1 in when maximum nesting values are low. When the value of maximum nesting increases to 3, Tool 1 performs better than Tool 2. The results presented in Figure 9 almost tell a completely opposite story when compared to the results in Figure 8. This is because both SCA tools practically trade places for its behavior when maximum nesting values are low. Tool 1, however, is able to handle a higher maximum nesting value.

D. False Positive Results

The way we identify a false positive is by looking at what checker (i.e., the component responsible for identifying a flaw) reported the flaw, and determining if the checker corresponds to CWE in the test case. If the reporting checker is not designed to identify the CWE in the test case, then it is a false positive.

\[
I_F = F - (TP_F + FP_F + FN_F)
\]

As shown in this equation, we measure the number of incidental flaws \(I_F\) as the difference of the number of flaws \(F\) reported by a SCA tool and the sum of the number of true positives \(TP_F\), false positives \(FP_F\), and false negatives \(FN_F\). Table I shows the results for the 7 sampled CWE’s. As shown in this table, Tool 1 identifies more flaws than Tool 2, but Tool 1 generates more false positives than Tool 2. This, unfortunately, can degrade the SCA tool’s accuracy and effectiveness because it is “crying wolf” more than needed, or it does not want to miss any potential flaw. For only CWE-242, Tool 1 found all the flaws, but it generated many false positives. Finally, Table I also illustrates that the total number of incidental flaws is dependent on the number of reported flaws. This means that the SCA tools are reporting more of the secondary flaws, and less of the root causes.

For example, Listing 2 illustrates source code for CWE-401. Although the source code in Listing 2 is for CWE-401, Tool 1 identifies a flaw in `goodB2G2_sink` causing a memory leak because memory is being freed inconsistently.

```c
static void goodB2G2_sink(twoints * data) {
    if (goodB2G2_static) {
        /\* FIX: Deallocate memory */
        free(data);
    }
    else {
        /\* INCIDENTAL: CWE 561 Dead Code. *
        * the code below will never run */
        /\* POTENTIAL FLAW: No deallocation */
        /\* empty statement needed for *
        * some flow variants */
    }
}
```

Listing 2. Source code illustrating an example of CWE 401.

Listing 3 illustrates source code that fixes an instance of CWE-835. As shown in this listing, the fix allows the control loop to break after 10 iterations. Tool 2, however, reports that the fix actually contains a flaw—stating that control never exits the loop (i.e., an infinite loop).

1) Identifying Incidental Flaws: We define incidental flaws as flaws reported by the tools by do not belong to any target type. This is similar to the incidental flaw marker that appears in the Juliet Test Suite (as shown in Listing 2), but it also includes flaws not identified by an incidental flaw marker. We use Equation 1 to calculate the number of incidental flaws \(I_F\).

```
Listing 3. Source code illustrating an example of CWE-835.
```

As shown in this equation, we measure the number of incidental flaws \(I_F\) as the difference of the number of flaws \(F\) reported by a SCA tool and the sum of the number of true positives \(TP_F\), false positives \(FP_F\), and false negatives \(FN_F\).
### Table I

<table>
<thead>
<tr>
<th>CWE</th>
<th>Tool 1</th>
<th>True Positive</th>
<th>False Positive</th>
<th>False negative</th>
<th>Incidental Flaws</th>
<th>Total flaws returned by tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>CWE134</td>
<td></td>
<td>1325</td>
<td>1885</td>
<td>9480</td>
<td>40656</td>
<td>43866</td>
</tr>
<tr>
<td>Tool 2</td>
<td>15</td>
<td>25</td>
<td>12650</td>
<td>1878</td>
<td>1918</td>
<td></td>
</tr>
<tr>
<td>CWE242</td>
<td>Tool 1</td>
<td>19</td>
<td>30</td>
<td>0</td>
<td>132</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>Tool 2</td>
<td>0</td>
<td>0</td>
<td>49</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>CWE252</td>
<td>Tool 1</td>
<td>0</td>
<td>0</td>
<td>1715</td>
<td>1656</td>
<td>1656</td>
</tr>
<tr>
<td></td>
<td>Tool 2</td>
<td>61</td>
<td>0</td>
<td>1654</td>
<td>92</td>
<td>153</td>
</tr>
<tr>
<td>CWE367</td>
<td>Tool 1</td>
<td>19</td>
<td>30</td>
<td>49</td>
<td>431</td>
<td>480</td>
</tr>
<tr>
<td></td>
<td>Tool 2</td>
<td>0</td>
<td>0</td>
<td>98</td>
<td>65</td>
<td>65</td>
</tr>
<tr>
<td>CWE401</td>
<td>Tool 1</td>
<td>966</td>
<td>392</td>
<td>8446</td>
<td>22616</td>
<td>23974</td>
</tr>
<tr>
<td></td>
<td>Tool 2</td>
<td>680</td>
<td>0</td>
<td>9124</td>
<td>4629</td>
<td>5309</td>
</tr>
<tr>
<td>CWE476</td>
<td>Tool 1</td>
<td>145</td>
<td>50</td>
<td>1455</td>
<td>1984</td>
<td>2179</td>
</tr>
<tr>
<td></td>
<td>Tool 2</td>
<td>190</td>
<td>0</td>
<td>1480</td>
<td>233</td>
<td>423</td>
</tr>
<tr>
<td>CWE762</td>
<td>Tool 1</td>
<td>1648</td>
<td>432</td>
<td>16032</td>
<td>35203</td>
<td>37283</td>
</tr>
<tr>
<td></td>
<td>Tool 2</td>
<td>1954</td>
<td>0</td>
<td>16158</td>
<td>11581</td>
<td>15555</td>
</tr>
</tbody>
</table>

### IV. Related Work

#### A. The National Security Agency

The National Security Agency (NSA) has previously done similar work to automate, test, and evaluate SCA tools [14]. In their study they tested 9 tools against 177 CWEs. The report, however, is not publicly available. The NSA’s study concluded that using multiple tools would increase the rate of finding weaknesses and decrease the false positives. The primary difference between their work and our work is we analyze software engineering metrics (e.g., cyclomatic complexity and maximum nesting) against the results of the tools. This helps software engineers understand how their code impacts the ability of a static code analysis tool. We also provide an open-source framework for conducting repeatable experiments across different SCA tools. This framework helps anyone to easily automate and evaluate a SCA tool.

#### B. The National Institute of Standards and Technology

The Software Assurance Metrics And Tool Evaluation (SA-MATE) project of the National Institute of Standards and Technology (NIST) does Static Analysis Tool Exposition (SATE). Recently, they executed SATE IV to research on how well static code analysis tools can help write secure code. NIST evaluated 8 tools against 177 CWE’s and 4 test cases from the Common Vulnerabilities and Exposures (CVE) set. According to the study, two-thirds of the weaknesses are reported by only one tool and the overlap between the tools is minimal as different tools concentrate on different weaknesses. NIST also stated that as the size and complexity of the code base increases the detection rates of the tools may decrease, which our study supports. Our work complements the NIST study in that we evaluate a SCA tools ability to identify flaws in source code based on different software engineering metrics.

### V. Concluding Remarks

This paper discussed the design and implementation of SCATE, and how we used it to evaluate two COTS SCA tools against the Juliet Test Suite from NIST. The primary focus of our study was not to see how many flaws each SCA tool could locate the Juliet Test Suite. It was to understand how each SCA tool fares against different software engineering metrics that we can measure in the source code. Based on our experience performing the study, our lessons learned and future research directions include:

- Our current study consist of using only COTS SCA tools, and evaluating the SCA tool against C++ source code. We plan to extend the study to include open-source SCA tools. We also plan to evaluate SCA tools against Java source code, where applicable.
- SCATE helped evaluate the quality of a SCA tool. We plan to continue extending its functionality so we can improve our modeling capabilities and predict what SCA tool, or collection of SCA tools, will assist developers in achieving the desired level of assurance in their source code.
- SCA tools results report a large number of incidental flaws, i.e., flaws that do not represent the CWE under investigation. In the future, we plan to capture this information so that we can better model the behavior of a SCA tool. We also believe that SCA tools can improve reporting of incidental flaws by hierarchically reporting incidental flaws based on the root flaw, if it is found. If the root cause is not found, we believe SCA tools can learn the root cause of an incidental flaw when there are a large number of reported similar incidental flaws.
- We have evaluated the SCA tools against two software engineering metrics, but Understand reports 38 different software engineering metrics about the source code it analyzes. Future work is to perform the same analysis using the additional 36 software engineering metrics.
- In future, we intend to integrate our work into Software Assurance Marketplace (SWAMP) [15], which is a cloud environment for evaluating the quality of source code by executing it against different SCA tools.

### References


