Towards Modeling the Behavior of Static Code Analysis Tools

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

ABSTRACT

This paper presents preliminary results of an independent study to assess the performance of a static code analysis (SCA) tool’s ability to detect and identify weaknesses and vulnerabilities in source code. The goal of the study is to model the behavior of static code analysis tools, and predict what SCA tool, or set of SCA tools, should be applied against a given source code to identify weaknesses and vulnerabilities.

Categories and Subject Descriptors
D.2.9 [Software Engineering]: Management—Software quality assurance (SQA)

General Terms
Design, Experimentation, Performance, Security, Standardization

Keywords
static code analysis tools, evaluation, behavior

1. 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 [1], 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 [3, 4].

To assists with this challenge, we have created an open-source, extensible Python framework called Static Code Analysis Tool (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 (e.g., cyclomatic complexity, lines of code, and maximum nesting) that impact a SCA tools ability to analyze the code [5].

The main contributions of this paper are as follows:

• It quantitatively evaluates two Commercial Off-The-Shelf (COTS) SCA tools against the Juliet Test Suite [6] from the National Institute of Standards and Technology (www.nist.gov) with emphasis of software engineering metrics.

Paper organization. The remainder of this paper is organized as follows: Section 2 discusses the design and implementation of SCATE; Section 3 presents the current results of our study; Section 4 compares our work to other related works; and Section 5 presents concluding remarks, lessons learned, and future research directions.

2. DESIGN AND IMPLEMENTATION OF SCATE FRAMEWORK

Figure 1 shows the architecture of the SCATE framework. As mentioned in Section 1, SCATE is implemented using Python. We selected Python because it allows us to quickly architect a framework using software design patterns [7] so we can focus on evaluating the quality of different SCA tools.

As shown in this figure 1:

• Command is an interface that implements the Command [7] software design pattern. Implementation of this interface defines the operations that SCATE can perform.

• TestSuite is an interface for defining Wrapper Facades [7] to test suites used to evaluate a SCA tool (i.e., Juliet from NIST). Implementations of this interface convert source code with known flaws into standard abstractions.

• Tool is another interface for defining Wrapper Facades to SCA tools. The tool facades must implement tool specific command objects so that SCATE understands how to perform the necessary operations.

• DataManager is the abstraction that defines import and 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. SCATE supports generating reports in PDF via LaTeX and CSV, which can be imported into Excel to create pivot charts.
2.1 Normalizing SCA Tool Output

One of the challenges of evaluating 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. Abstractions are a set of aggregated result elements (i.e., a result set contains weaknesses, a weakness contains test suites, and a test suite contains flaws). A weakness 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, line number and type of flaw indicated by severity 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.

2.2 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, we need a standard method for identifying flaws in source code and its corresponding severity level. This will assist with training a model which can be applied on unknown source code.

We use the approach currently practiced in the Juliet Test Suite [6] to define flaws for importing. 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. 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 to evaluate a 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.

2.3 Identifying False Positives and False Negatives

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 [6].

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 [6].

3. 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.

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. Finally, we executed two COTS SCA tools against the Juliet Test Suite knowledge base.

3.1 Discussion of General Results

Figure 2 shows the overall results for both COTS SCA tools we used in our initial evaluation. As shown in this figure, 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 does not find CWE-392 in the test cases be considered a bad tool?
This is a hard question to answer, and really does 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 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 their source code, or give motivation for improving how source code is written (i.e., changing the organization’s culture).

### 3.2 Discussion of CWE-specific Results

Although we are evaluating the tools against 91 CWEs in the Juliet Test Suite, we discuss the results of few CWEs in detail as they showcases what we were able to discover using SCATE.

#### 3.2.1 Analysis via Software Engineering Metrics

We used Understand (www.scitools.com) to calculate cyclomatic complexity and maximum nesting. 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. However, in this paper we only discuss results for cyclomatic complexity.

**Cyclomatic complexity.** Figure 3 highlights the results for CWE-121 (Stack-based buffer overflow condition) when we focused on cyclomatic complexity. 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 4 highlights the cyclomatic complexity results for CWE-124 Buffer underflow condition. We see that at lower cyclomatic complexity 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. Depending upon the domain a developer is working on some CWEs might be more important than others. 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.

3.3 False Positive Results

We identify a false positive (FP) 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.

Table 1: Data table detailing the types of errors identified by each tool for sampled CWEs.

<table>
<thead>
<tr>
<th>CWE</th>
<th>Tool 1</th>
<th>Tool 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CWE134</td>
<td>1325</td>
<td>15</td>
</tr>
<tr>
<td>CWE401</td>
<td>966</td>
<td>680</td>
</tr>
</tbody>
</table>

Table 1 shows the results for the 2 sampled CWE’s. As shown in this table, Tool 1 identifies more true positives (TP) than Tool 2, but Tool 1 generates more false positives than Tool 2.

4. RELATED WORK

The National Security Agency (NSA) has previously done similar work to automate, test, and evaluate SCA tools [8]. The primary difference between their work and our work is that we analyze software engineering metrics (e.g., cyclomatic complexity and maximum nesting) against the results of the tools. Also National Institute of Standards and Technology (NIST) does Static Analysis Tool Exposition (SATE) to research on how well static code analysis tools can help write secure code. 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.

5. 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 from NIST. The primary focus of our study was to understand how each SCA tool fares against different software engineering metrics that we can measure in the Test Suite. It was to understand how a SCA tool will perform in the context of how they implement source code.

We identify a false positive (FP) 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.

Table 1 shows the results for the 2 sampled CWE’s. As shown in this table, Tool 1 identifies more true positives (TP) than Tool 2, but Tool 1 generates more false positives than Tool 2.

6. ACKNOWLEDGEMENTS

This works was sponsored in part by Northrop Grumman, Lockheed Martin, and the Department of Homeland Security through the Security and Software Engineering Research Center (S2ERC).

7. REFERENCES