CAS Static Analysis Tool Study - Methodology

Center for Assured Software
National Security Agency
9800 Savage Road
Fort George G. Meade, MD 20755-6738
cas@nsa.gov

December 2012

Warnings

Trade names or manufacturers' names are used in this report for identification only. This usage does not constitute an official endorsement, either expressed or implied, by the National Security Agency.

References to a product in this report do not imply endorsement by the National Security Agency of the use of that product in any specific operational environment, to include integration of the product under evaluation as a component of a software system.

References to an evaluation tool, technique, or methodology in this report do not imply endorsement by the National Security Agency of the use of that evaluation tool, technique, or methodology to evaluate the functional strength or suitability for purpose of arbitrary software analysis tools.

Citations of works in this report do not imply endorsement by the National Security Agency or the Center for Assured Software of the content, accuracy or applicability of such works.

References to information technology standards or guidelines do not imply a claim that the product under evaluation is in conformance or nonconformance with such a standard or guideline.

References to test data used in this evaluation do not imply that the test data was free of defects other than those discussed. Use of test data for any purpose other than studying static analysis tools is expressly disclaimed.

This report and the information contained in it may not be used in whole or in part for any commercial purpose, including advertising, marketing, or distribution.

This report is not intended to endorse any vendor or product over another in any way.

Trademark Information

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

MITRE is a registered trademark of The MITRE Corporation.

Apache is a registered trademark of the Apache Software Foundation.

Table of Contents

Abstract

The primary mission of the National Security Agency's (NSA) Center for Assured Software (CAS) is to increase the degree of confidence that software used within the Department of Defense (DoD) is free from exploitable vulnerabilities. Over the past several years, commercial and open source static analysis tools have become more sophisticated at being able to identify flaws that can lead to such vulnerabilities. As these tools become more reliable and popular with developers and clients, the need to fully understand their capabilities and shortcomings is becoming more important.

To this end, the CAS regularly conducts studies using a scientific, methodical approach that measures and rates effectiveness of these tools in a standard and repeatable manner. The CAS Static Analysis Tool Study Methodology is based on a set of artificially created known answer tests that comprise examples of "intentionally flawed code." Each flawed example, with the exception of specific test cases, has at least one corresponding construct that is free from that specific flaw. In applying the methodology, the tester analyzes all tools using this common testing corpus. The methodology then offers a common way to "score" the tools' results so that they are easily compared. With this known answer approach, testers can have full insight into what a tool should report as a flaw, what it misses, and what it actually reports. The CAS has created and released the test corpus to the community for analysis, testing, and adoption.[1]

This document provides a step by step description of this methodology in the hope that it can become part of the public discourse on the measurement and performance of static analysis technology. In addition, this document shows the various reporting formats used by the CAS and provides an overview of how the CAS administers its tool study. It is available for public consumption, comment and adoption. Comments and suggestions on the methodology can be sent to CAS@nsa.gov.

[1] This test suite is available as the "Juliet Test Suite" and is publicly available through the National Institute for Standards and Technology (NIST) at http://samate.nist.gov/SRD/testsuite.php.

Section 1: Introduction

1.1 Background

Software systems support and enable mission-essential capabilities in the Department of Defense. Each new release of a defense software system provides more features and performs more complex operations. As the reliance on these capabilities grows, so does the need for software that is free from intentional or accidental flaws. Flaws can be detected by analyzing software either manually or with the assistance of automated tools.

Most static analysis tools are capable of finding multiple types of flaws, but the capabilities of tools are not necessarily uniform across the spectrum of flaws they detect. Even tools that target a specific type of flaw are capable of finding some variants of that flaw and not others. Tools' datasheets or user manuals often do not explain which specific code constructs they can detect, or the limitations and strengths of their code checkers. This level of granularity is needed to maximize the effectiveness of automated software evaluations.

1.2 Center for Assured Software (CAS)

In order to address the growing lack of Software Assurance in the DoD, the NSA's CAS was created in 2005. The CAS's mission is to improve the assurance of software used within the DoD by increasing the degree of confidence that software is free from intentional and unintentional vulnerabilities. The CAS accomplishes this mission by assisting organizations in deploying processes and tools to address assurance throughout the Software Development Life Cycle (SDLC).

As part of an overall secure software development process, the CAS advocates the use of static analysis tools at various stages in the SDLC, but not as a replacement for other software assurance efforts, such as manual code reviews. The CAS also believes that some organizations and projects warrant a higher level of assurance that can be gained through the use of more than one static analysis tool.

The CAS is responsible for performing an annual study on the capabilities of automated, flaw-finding static analysis tools. The results of these studies assist software development teams with the selection of a tool for use in their SDLC.

1.3 Feedback

The CAS continuously tries to improve its methodology for running these studies. As you read this document, if you have any feedback or questions on the information presented, please contact the CAS via email at cas@nsa.gov.

Section 2: Methodology

2.1 Overview

The CAS methodology consists of using artificial code in the form of test cases to perform static analysis tool evaluations. Each test case targets a specific flaw. These test cases are grouped with similar flaw-types into Weakness Classes. After each tool scans the test suite, the results are then scored, analyzed, and presented in a programming language-specific report. The following paragraphs explain each aspect of the methodology in greater detail.

2.2 Test Cases

In order to study static analysis tools, evaluators need software for the tools to analyze. There are two types of software to choose from: natural and artificial. Natural software is software that was not created to test static analysis tools. Open source software applications, such as the Apache web server (httpd.apache.org) or the OpenSSH suite (www.openssh.com) are examples of natural software. Artificial software contains intentional flaws and is created specifically to test static analysis tools.

The CAS decided that the benefits of using artificial code outweighed the disadvantages and therefore created artificial code to study static analysis tools. The CAS generates the source code as a collection of test cases. Each test case contains exactly one intentional flaw and typically at least one non-flawed construct similar to the intentional flaw. The non-flawed constructs are used to determine if the tools could discriminate flaws from non-flaws. For example, one test case illustrates a type of buffer overflow vulnerability. The flawed code in the test case passes the C *strcpy* function a destination buffer that is smaller than the source string. The non-flawed construct passes a large enough destination buffer to *strcpy*.

The test cases created by the CAS and used to study static analysis tools are called the Juliet Test Suites. They are publicly available through the National Institute for Standards and Technology (NIST) at http://samate.nist.gov/SRD/testsuite.php.

2.3 Weakness Classes

To help understand the areas in which a given tool excelled, similar test cases are grouped into a Weakness Class. Weakness Classes are defined using the MITRE Common Weakness Enumeration (CWE)[2] entries that contain similar flaw types. Since each Juliet test case is associated with the CWE entry in its name, each test case is contained in a Weakness Class.

For example, Stack-based Buffer Overflow (CWE-121) and Heap-based Buffer Overflow (CWE-122) are both placed in the Buffer Handling Weakness Class. Therefore, all of the test cases associated with CWE entries 121 and 122 are mapped to the Buffer Handling Weakness

[2] The MITRE CWE is a community-developed dictionary of software weakness types and can be found at http://cwe.mitre.org

Class. Table 1 provides a list of all the Weakness Classes used in the study, along with an example of each.

Weakness Class	Example Weakness (CWE Entry)
Authentication and Access Control	CWE-259: Use of Hard-coded Password
Buffer Handling (C/C++ only)	CWE-121: Stack-based Buffer Overflow
Code Quality	CWE-561: Dead Code
Control Flow Management	CWE-483: Incorrect Block Delimitation
Encryption and Randomness	CWE-328: Reversible One-Way Hash
Error Handling	CWE-252: Unchecked Return Value
File Handling	CWE-23: Relative Path Traversal
Information Leaks	CWE-534: Information Exposure Through Debug Log Files
Initialization and Shutdown	CWE-404: Improper Resource Shutdown or Release
Injection	CWE-134: Uncontrolled Format String
Malicious Logic	CWE-506: Embedded Malicious Code
Number Handling	CWE-369: Divide by Zero
Pointer and Reference Handling	CWE-476: NULL Pointer Dereference

Table 1 – Weakness Classes

The Miscellaneous Weakness Class, which was used to hold a collection of individual weaknesses that did not fit into the other classes, was re-evaluated in 2012. Based upon their implementation and to alleviate any confusion regarding the nature of the test cases in this category, the CAS felt these test cases could be re-categorized within the other Weakness Classes.

The following paragraphs provide a brief description of the Weakness Classes defined by the CAS.

2.3.1 Authentication and Access Control

Attackers can gain access to a system if the proper authentication and access control mechanisms are not in place. An example would be a hardcoded password or a violation of the least privilege principle. The test cases in this Weakness Class test the tools' ability to check whether or not the source code is preventing unauthorized access to the system.

2.3.2 Code Quality

Code quality issues are typically not security related; however, they can lead to maintenance and performance issues. An example would be unused code. While this is not an inherent security risk, it may lead to maintenance issues in the future. The test cases in this Weakness Class test the tools' ability to find poor code quality issues in the source code.

The test cases in this Weakness Class cover some constructs that may not be relevant to all audiences. The test cases are all based on weaknesses in CWEs, but even persons interested in

code quality may not consider some of the tested constructs to be weaknesses. For example, this Weakness Class includes test cases for flaws such as an omitted break statement in a switch, a missing default case in a switch, and dead code.

2.3.3 Control Flow Management

Control flow management deals with timing and synchronization issues that can cause unexpected results when the code is executed. An example would be a race condition. One possible consequences of a race condition is a deadlock which leads to a denial of service. The test cases in this Weakness Class test the tools' ability to find issues in the order of execution within the source code.

2.3.4 Encryption and Randomness

Encryption is used to provide data confidentiality. However, if a weak or wrong encryption algorithm is used, an attacker may be able to convert the ciphertext into its original plain text. An example would be the use of a weak Pseudo Random Number Generator (PRNG). Using a weak PRNG could allow an attacker to guess the next number that is generated. The test cases in this Weakness Class test the tools' ability to check for proper encryption and randomness in the source code.

2.3.5 Error Handling

Error handling is used when a program behaves unexpectedly. However, if a program fails to handle errors properly it could lead to unexpected consequences. An example would be an unchecked return value. If a programmer attempts to allocate memory and fails to check if the allocation routine was successful, then a segmentation fault could occur if the memory failed to allocate properly. The test cases in this Weakness Class test the tools' ability to check for proper error handling within the source code.

2.3.6 File Handling

File handling deals with reading from and writing to files. An example would be reading from a user-provided file on the hard disk. Unfortunately, adversaries can sometimes provide relative paths that contain periods and slashes. An attacker can use this method to read to or write to a file in a different location on the hard disk than the developer expected. The test cases in this Weakness Class test the tools' ability to check for proper file handling within the source code.

2.3.7 Information Leaks

Information leaks can cause unintended data to be made available to a user. For example, developers often use error messages to inform users that an error has occurred. Unfortunately, if sensitive information is provided in the error message an adversary could use it to launch future attacks on the system. The test cases in this Weakness Class test the tools' ability to check for information leaks within the source code.

2.3.8 Initialization and Shutdown

Initializing and shutting down resources occurs often in source code. For example, in C/C++ if memory is allocated on the heap it must be deallocated after use. If the memory is not deallocated, it could cause memory leaks and affect system performance. The test cases in this Weakness Class test the tools' ability to check for proper initialization and shutdown of resources in the source code.

2.3.9 Injection

Code injection can occur when user input is not validated properly. One of the most common types of injection flaws is cross-site scripting (XSS). An attacker can place query strings in an input field that could cause unintended data to be displayed. This can often be prevented using proper input validation and/or data encoding. The test cases in this Weakness Class test the tools' ability to check for injection weaknesses in the source code.

2.3.10 Malicious Logic

Malicious logic is the implementation of a program that performs an unauthorized or harmful action. In source code, unauthorized or harmful actions can be indicators of malicious logic. Examples of malicious logic include Trojan horses, viruses, backdoors, worms, and logic bombs. The test cases in this Weakness Class test the tools' ability to check for malicious logic in the source code.

2.3.11 Number Handling

Number handling issues include incorrect calculations as well as number storage and conversions. An example is an integer overflow. On a 32-bit system, a signed integer's maximum value is 2,147,483,647. If this value is increased by one, its new value will be a negative number rather than the expected 2,147,483,648 due to the limitation of the number of bits used to store the number. The test cases in this Weakness Class test the tools' ability to check for proper number handling in the source code.

2.3.12 Pointer and Reference Handling

Pointers are often used in source code to refer to a block of memory without having to reference the memory block directly. One of the most common pointer errors is a NULL pointer dereference. This occurs when the pointer is expected to point to a block of memory, but instead it points to the value of NULL. This can cause an exception and lead to a system crash. The test cases in this Weakness Class test the tools' ability to check for proper pointer and reference handling.

2.4 Assessment

2.4.1 Tool Execution

The CAS regularly evaluates commercial and open source static analysis tools with the use of the Juliet Test Suites. The tools are installed and configured on separate hosts with a standardized set of resources in order to avoid conflicts and allow independent analysis. Every tool is executed using its command-line interface (CLI) and the results are exported upon completion.

2.4.2 Scoring Results

In order to assess the tool's performance, tool results are scored using result types. Table 2 contains the various result types that can be assigned as well as their definitions.

Result Type	Explanation
True Positive (TP)	Tool reports the intentionally-flawed code.
False Positive (FP)	Tool reports the non-flawed code.
False Negative (FN)	Tool fails to report the intentionally-flawed code.

Table 2 – Summary of Result Types

For example, consider a test case that targets a buffer overflow flaw. The test case contains intentionally-flawed code that attempts to place data from a large buffer into a smaller one. If a tool reports a buffer overflow in this code then the result is marked as a *True Positive*. The test case also contains non-flawed code that is similar to the flawed code, but prevents a buffer overflow. If a tool reports a buffer overflow in this code then the result is marked as a *False Positive*. If the tool fails to report a buffer overflow in the intentionally-flawed code, then the result is marked as a *False Negative*. If a tool reports any other type of flaw, for example a memory leak, in the intentionally-flawed or non-flawed code, then the result type is considered an incidental flaw as it is not the target of the test case. Incidental flaws are excluded from scoring.

2.5 Metrics

Metrics are used to perform analysis of the tool results. After the tool results have been scored, specific metrics can be calculated. Several metrics used by the CAS are described in the following paragraphs.

2.5.1 Precision, Recall, and F-Score

One set of metrics contains the Precision, Recall, and F-Scores of the tools based on the number of True Positive (TP), False Positive (FP), and False Negative (FN) findings for that tool. The following paragraphs describe these metrics in greater detail.

Precision

In the context of the methodology, Precision (also known as "positive predictive value") is the ratio of weaknesses reported by a tool to the set of actual weaknesses in the code analyzed. It is defined as the number of weaknesses correctly reported (True Positives) divided by the total number of weaknesses actually reported (True Positives plus False Positives).

$$Precision = \frac{\#TP}{\#TP+\#FP}$$

Precision is synonymous with the True Positive rate and is the complement of the False Positive rate. It is also important to highlight that Precision and Accuracy are not the same. In this methodology, Precision describes how well a tool identifies flaws, whereas accuracy describes how well a tool identifies flaws and non-flaws as well.

Note that if a tool does not report any weaknesses, then Precision is undefined, i.e. 0/0. If defined, Precision is greater than or equal to 0, and less than or equal to 1. For example, a tool that reports 40 issues (False Positives and True Positives), of which only 10 are real flaws (True Positives), has a Precision of 10 out of 40, or 0.25.

Precision helps users understand how much trust can be given to a tool's report of weaknesses. Higher values indicate greater trust that issues reported correspond to actual weaknesses. For example, a tool that achieves a Precision of 1 only reports issues that are real flaws in the test cases. That is, it does not report any False Positives. Conversely, a tool that has a Precision of 0 always reports issues incorrectly. That is, it only reports False Positives.

Recall

The Recall metric (also known as "sensitivity" or "soundness") represents the fraction of real flaws reported by a tool. Recall is defined as the number of real flaws reported (True Positives), divided by the total number of real flaws – reported or unreported – that exist in the code (True Positives plus False Negatives).

$$Recall = \frac{\#TP}{\#TP+\#FN}$$

Recall is always a value greater than or equal to 0, and lesser than or equal to 1. For example, a tool that reports 10 real flaws in code that contains 20 flaws has a Recall of 10 out of 20, or 0.5.

A high Recall means that the tool correctly identifies a high number of the target weaknesses within the test cases. For example, a tool that achieves a Recall of 1 reports every flaw in the test cases. That is, it has no False Negatives. (While a Recall of 1 indicates all of the flaws in the test cases were detected, notice that this metric does not account for the number of False Positives that may have been produced by the tool.) In contrast, a tool that has a Recall of 0 reports none of the real flaws. That is, it has a high False Negative rate.

F-Score

In addition to the Precision and Recall metrics, an F-Score is calculated by taking the harmonic mean of the Precision and Recall values. Since a harmonic mean is a type of average, the value of the F-Score will always be between the Precision and Recall values (unless the Precision and Recall values are equal, in which case the F-Score will be that same value). Note that the harmonic mean is always less than the arithmetic mean (again, unless the Precision and Recall are equal).

The F-Score provides weighted guidance in identifying a good static analysis tool by capturing how many of the weaknesses are found (True Positives) and how much noise (False Positives) is produced. An F-Score is computed using the following formula:

$$F\text{-}Score = 2 \times \left(\frac{Precision \times Recall}{Precision + Recall} \right)$$

A harmonic mean is desirable since it ensures that a tool must perform reasonably well with respect to both Precision and Recall metrics. In other words, a tool will not get a high F-Score with a very high score in one metric but a low score in the other metric. Simply put, a tool that is very poor in one area is not considered stronger than a tool that is average in both.

2.5.2 Discriminations and Discrimination Rate

Another set of metrics measures the ability of tools to discriminate between flaws and non-flaws. This set of metrics is helpful in differentiating unsophisticated tools performing simple pattern matching from tools that conduct a more complex analysis.

For example, consider a test case for a buffer overflow where the flaw uses the *strcpy* function with a destination buffer smaller than the source data. The non-flaw on this test case may also use *strcpy*, but with a sufficiently large destination buffer. A tool that simply searches for the use of *strcpy* would correctly report the flaw in this test case, but also produce a False Positive when reporting the non-flaw.

If a tool behaved in this way on all test cases in a certain area, the tool would have a Recall of 1, a Precision of .5, and an F-Score of .67 (assuming that each test case had only one "good" or non-flawed construct). These scores don't accurately reflect the tool's unsophisticated behavior. In particular, the tool is "noisy" (generates many False Positive results), which is not reflected in its Precision of .5.

Discriminations

To address the issue described above, the CAS defines a metric called "Discriminations." A tool is given credit for a Discrimination when it correctly reports the flaw (a True Positive) in a test case without incorrectly reporting the flaw in non-flawed code (that is, without any False Positives). For every test case, each tool receives 0 or 1 Discriminations.

In the example above, an unsophisticated tool that is simply searching for the use of *strcpy* will not get credit for a Discrimination on the test case because while it correctly reports the flaw, it also reports a False Positive.

Discriminations must be determined for each test case individually. The number of Discriminations in a set of test cases cannot be calculated from the total number of True Positives and False Positives.

Over a set of test cases, a tool can report as many Discriminations as there are test cases (an ideal tool would report a Discrimination on each test case). The number of Discriminations will always be less than or equal to the number of True Positives over a set of test cases (because a True Positive is necessary, but not sufficient, for a Discrimination).

Discrimination Rate

Over a set of test cases, the Discrimination Rate is the fraction of test cases in which the tool reports a Discrimination.

$$Discrimination\,Rate = \frac{\#\,Discriminations}{\#\,Flaws}$$

The Discrimination Rate is always a value greater than or equal to 0, and less than or equal to 1.

Over a set of test cases, the Discrimination Rate is always less than or equal to the Recall. This is because Recall is the fraction of test cases where the tool reported True Positives, regardless of whether it reported False Positives. Every test case where the tool reports a Discrimination "counts" toward a tool's Recall and Discrimination Rate, but other, non-Discrimination test cases may also count toward Recall (but not toward Discrimination Rate).

Section 3: Reporting

Graphs and tables are used to show the tool results for various metrics including Recall, Precision, Discrimination Rate, and F-Score. Examining just a single metric does not give the whole picture and obscures details that are important in understanding a tool's overall strengths. For example, just looking at Recall tells the analyst how many flaws are found, but these flaws can be hidden in a sea of False Positives, making it extremely time-consuming to interpret the results. By examining the values for Recall, Precision, and Discrimination Rate, the analyst can get a better picture of the tool's overall strengths.

3.1 Results by Tool

3.1.1 Precision, Recall, and F-Score Table

To summarize each tool's Precision, Recall, and F-Score on the test cases, a table is produced to show how the tool performs regarding each metric. However, when interpreted in isolation, each metric can be deceptive in that it does not reflect how well a tool performs with respect to other tools, i.e., it is not known what is a "good" number. As an example, it is impossible for an analyst to know whether a Precision of .45 is a good value or not. If every other tool has a Precision value of .20, then a value of .45 would suggest that the tool outperforms its peers. On the other hand, if every other tool has a Precision of .80, then a value of .45 would suggest that the tool underperforms in this metric.

For Precision, if a tool does not report any findings in a given Weakness Class, it is excluded from the calculation of the average for that Weakness Class. For Recall and F-Score, all results are used to calculate the average, even if a tool does not report any findings for that Weakness Class.

A small green triangle pointing up is used to indicate that the tool's value is .05 or more above the average for all the tools in that Weakness Class. A small red triangle pointing down is used to indicate that the tool's value is .05 or more below the average for that Weakness Class. If the tool's value is within .05 of the average, then no icon is used meaning the tool results are close to average.

		Sample Size	Tool Results		
Weakness Class	**Abbr.**	**# of Flaws**	**Precision**	**F-Score**	**Recall**
Weakness Class A	AAA	511	▼ .25	▼ .20	▼ .17
Weakness Class B	BBB	953	-	0	0
Weakness Class C	CCC	433	▲ .96	▲ .72	▲ .58
Weakness Class D	DDD	720	▼ .56	.57	▲ .58
Weakness Class E	EEE	460	1	.29	.17
Legend:	▲ = .05 or more above average		▼ = .05 or more below average		

Table 3 – Precision-Recall Results for SampleTool by Weakness Class

In this example, Table 3 shows that SampleTool has a Precision of .96, an F-Score of .72, and a Recall of .58, which are all more than .05 above the average with respect to Precision, Recall, and F-Score for Weakness Class C. Its performance is mixed in Weakness Class D, with below-average Precision of .56 and an above-average Recall of .58. Within Weakness Class E, SampleTool has average results, indicated by the absence of triangles.

Note that if a tool does not produce any findings for a given Weakness Class, then Precision cannot be calculated because of division by zero, which is reflected as a dash ('-') in the table. This can be an indication either a tool does not have the capability to test for flaw types in a given Weakness Class, or a tool has the ability to detect these flaw types but failed to report any. In this methodology, when Precision is undefined, F-Score is defined as zero.

A Recall value of 0 indicates that the tool did not report any True Positives for that Weakness Class. A Recall of .00 signifies a non-zero value (i.e. the tool found at least one True Positive in that Weakness Class); however, the value is too small to be displayed in the table to within two decimal places.

Also note that the number of flaws represents the total number of test cases in that Weakness Class. This represents a sample size and gives the analyst an idea of how many opportunities a tool had to produce findings. In general, when there are more opportunities, one can have more statistical confidence in the metric.

The tool results and averages for Precision, Recall, and F-Score displayed in the tables and graphs are all weighted according to the data and control flow complexity of the test case. (See Section 4.1.9 for more information on weighting.)

3.1.2 Precision Graph by Weakness Class

Figure 1 shows an example of a Precision Graph for a single tool. Even when all values are relatively small, the Y-axis scale remains consistent in order to compare tools equally. The tool's Precision for a given Weakness Class is indicated by the length of a blue bar, whereas the average Precision among all tools for a given Weakness Class is indicated by the length of a gray bar. If a tool did not report at least one finding for a given Weakness Class, then Precision is undefined and is indicated on the graph by a bar length of zero.

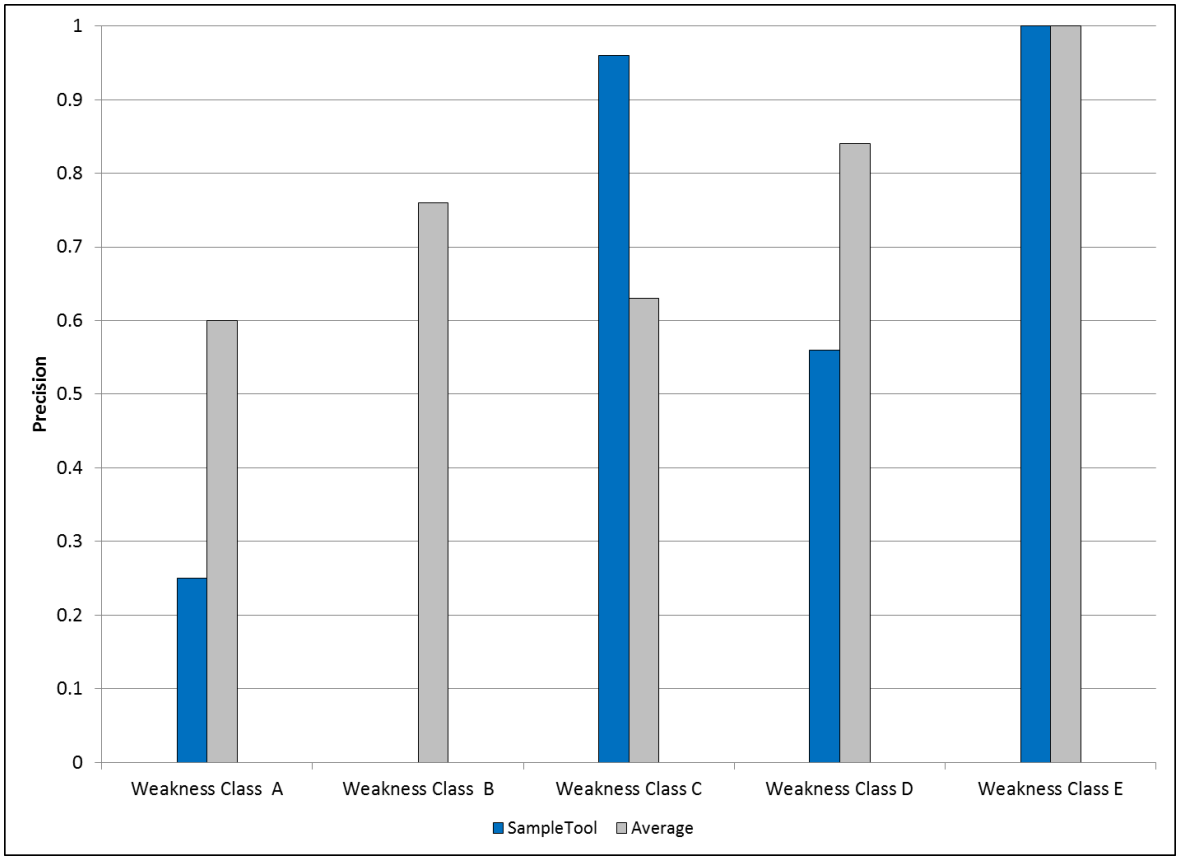

Figure 1 – Example Precision Graph for a single tool

The example graph in Figure 1 shows that when compared to the other tools, SampleTool is relatively strong when focusing on Weakness Class C, and less effective in Weakness Classes A and D. SampleTool performed average in Weakness Class E. Also, the tool did not produce any findings for Weakness Class B.

3.1.3 Recall Graph by Weakness Class

Figure 2 shows an example of a Recall Graph for a single tool. Even when all values are relatively small, the Y-axis scale remains consistent in order to compare tools equally. The tool's Recall for a given Weakness Class is indicated by the length of a purple bar, whereas the average Recall among all tools for a given Weakness Class is indicated by the length of a gray bar.

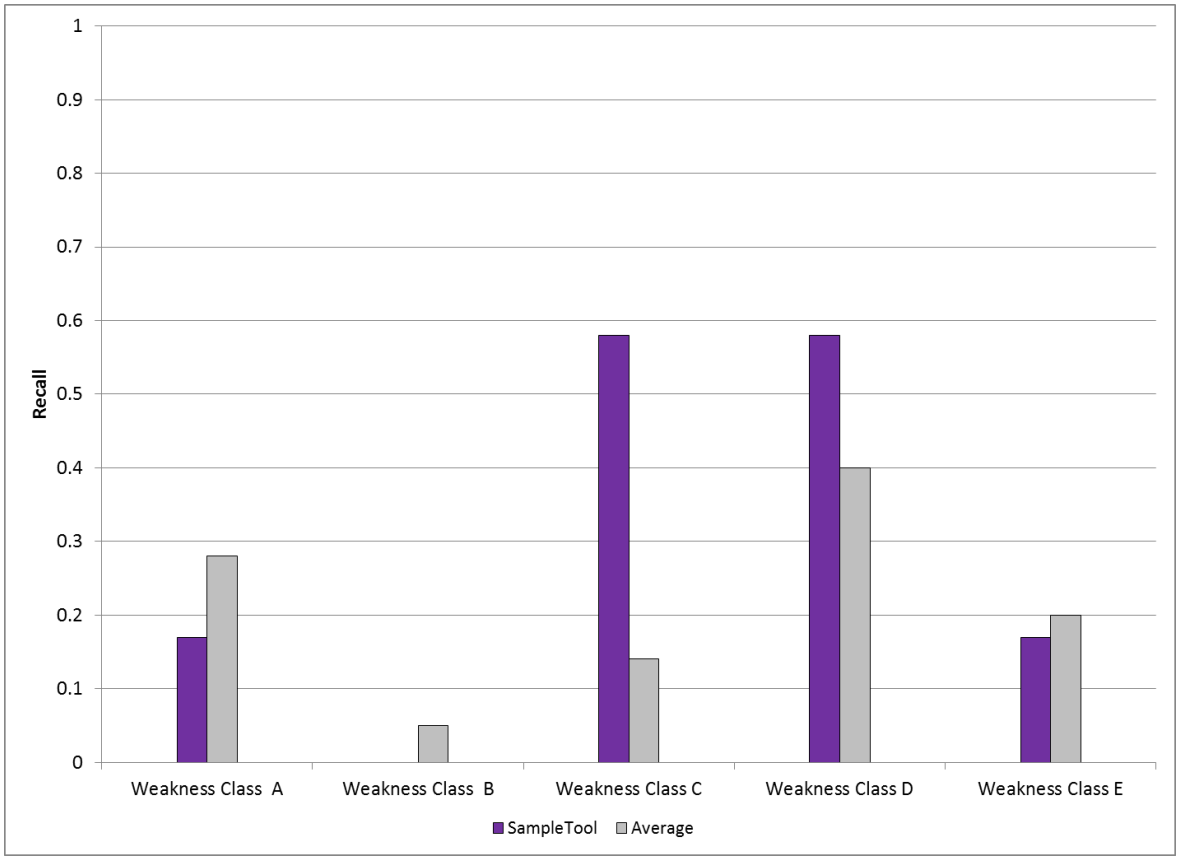

Figure 2 – Example Recall Graph for a single tool

The example graph in Figure 2 shows that when compared to the other tools, SampleTool is stronger in Weakness Classes C and D, but less effective in Weakness Classes A and E. Also, the tool did not report any True Positives for Weakness Class B, as evident by the Recall value of zero.

3.1.4 Precision-Recall Graph by Tool

Figure 3 displays an example of a Precision-Recall graph illustrating a tool's performance compared to the average for each Weakness Class. Notice that the Precision metric is mapped to the vertical axis and the Recall metric is mapped to the horizontal axis. A tool's relation to both metrics is represented by a point on the graph. If a tool did not report at least one True Positive for a given Weakness Class, then it is not shown on the graph. The closer the point is to the top right, the stronger the tool is in the given area.

The circle marker represents the tool's actual metric values for the specified Weakness Class. A green circle indicates that the tool performed better than average in both Precision and Recall as compared to the average. A red circle indicates that the tool performed below average in both Precision and Recall as compared to the average. A gray circle indicates that the tool did not perform better or worse in both Precision and Recall. For example, this could indicate that a tool had better than average Precision, but worse than average Recall for a given Weakness Class. The black diamond represents the average metric values for all the tools that produced findings for the given Weakness Class. The labels contain the abbreviated Weakness Class names.

A solid line is drawn between the two related points and helps visually state how a given tool compared to the average. Note that the longer the line, the greater the difference between the tool and the average. In general, movement of a specific tool away from the average toward the upper right demonstrates a relatively greater capability in the given area (as indicated by the arrow in the graph's background).

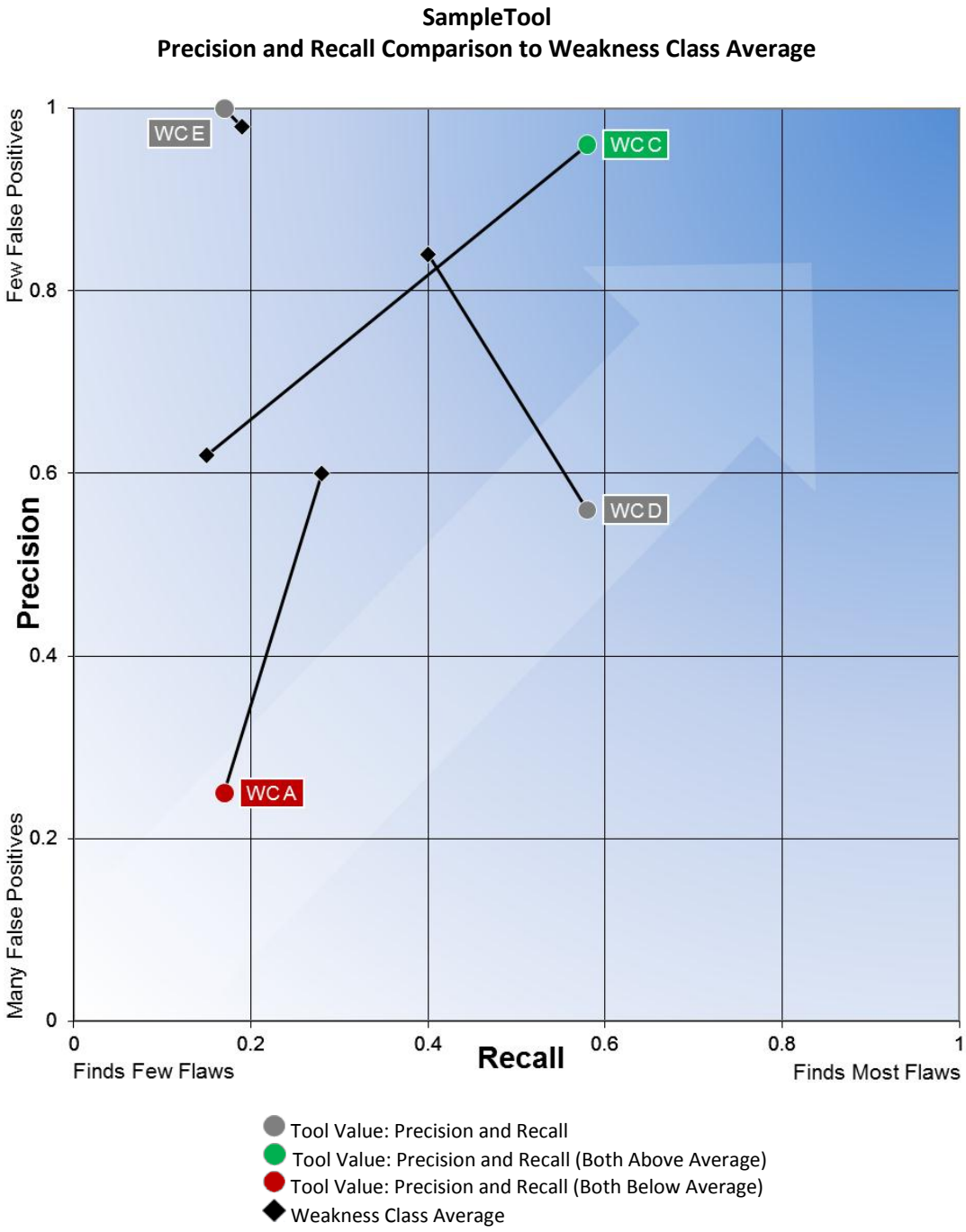

Figure 3 – Example Precision-Recall Graph for single tool

In the example in Figure 3, the graph shows that the SampleTool is stronger than the average of all tools when focusing on Weakness Class C. Both Precision and Recall for the tool are above average as evident by the green data point and by moving from the black diamond upward, meaning higher Precision, and to the right, indicating higher Recall. SampleTool is less effective compared to the average of all tools relating to Weakness Class A as indicated by the red data point. Both metrics are below average, and the line moves from the black diamond downward, meaning less Precision, and to the left, indicating less Recall.

For the results associated with Weakness Class E and Weakness Class D, where the line moves to the upper left or the lower right, more analysis is often needed. In these situations, the tool is above average for one metric but below average for the other.

3.1.5 Discriminations and Discrimination Rate Table by Weakness Class

A table is produced that summarizes each tool's ability to discriminate between flaws and non-flaws. Similar to Precision, Recall, and F-Score, when interpreted in isolation, this metric can be deceptive in that it does not reflect how a tool performs with respect to other tools, i.e., it is not known what a "good" number is.

For Discrimination Rates, all tools are included in the calculation of the average for that Weakness Class.

If the tool has a value .05 or more above the average, a small green triangle pointing up was used. For values .05 or more below the average, a small red triangle pointing down is used. If the value is within .05 of the average then no icon is used, meaning the tool results are close to average. A Discrimination Rate of .00 indicates that the tool made at least one Discrimination; however, the value, when rounded to two decimal places, displays as 0.

A tool's Discriminations and Discrimination Rates on each Weakness Class are shown in a table like Table 4.

		Sample Size	Tool Results	
Weakness Class	**Abbr.**	**# of Flaws**	**Discriminations**	**Disc. Rate**
Weakness Class A	AAA	511	50	▼ .10
Weakness Class B	BBB	953	0	0
Weakness Class C	CCC	433	234	▲ .54
Weakness Class D	DDD	720	150	.21
Weakness Class E	EEE	460	15	.03
Legend:	▲ = .05 or more above average		▼ = .05 or more below average	

Table 4 – Discrimination Results for SampleTool by Weakness Class

In this example, Table 4 shows that SampleTool has a Discrimination Rate of .10 for Weakness Class A, which is at least .05 below the average of all tools, and a Discrimination Rate of .54 for Weakness Class C, which is at least .05 above the average. For Weakness Classes D and E, with Discrimination Rates of .21 and .03, respectively, SampleTool has average results. It also indicates that the tools overall performed poorly on Weakness Class E with respect to Discriminations since the average Discrimination Rate can be no higher than .08 (Weakness Class E's rate of .03 plus .05) . SampleTool did not report any Discriminations for Weakness Class B, indicating poor complex analysis in that area.

3.1.6 Discrimination Rate Graph by Weakness Class

Discrimination Rate Graphs like Figure 4 are used to show the Discrimination Rates for a single tool across different Weakness Classes. Even when all values are relatively small, the Y-axis scale on these graphs is not adjusted in order to compare tools to the ideal Discrimination Rate of 1.

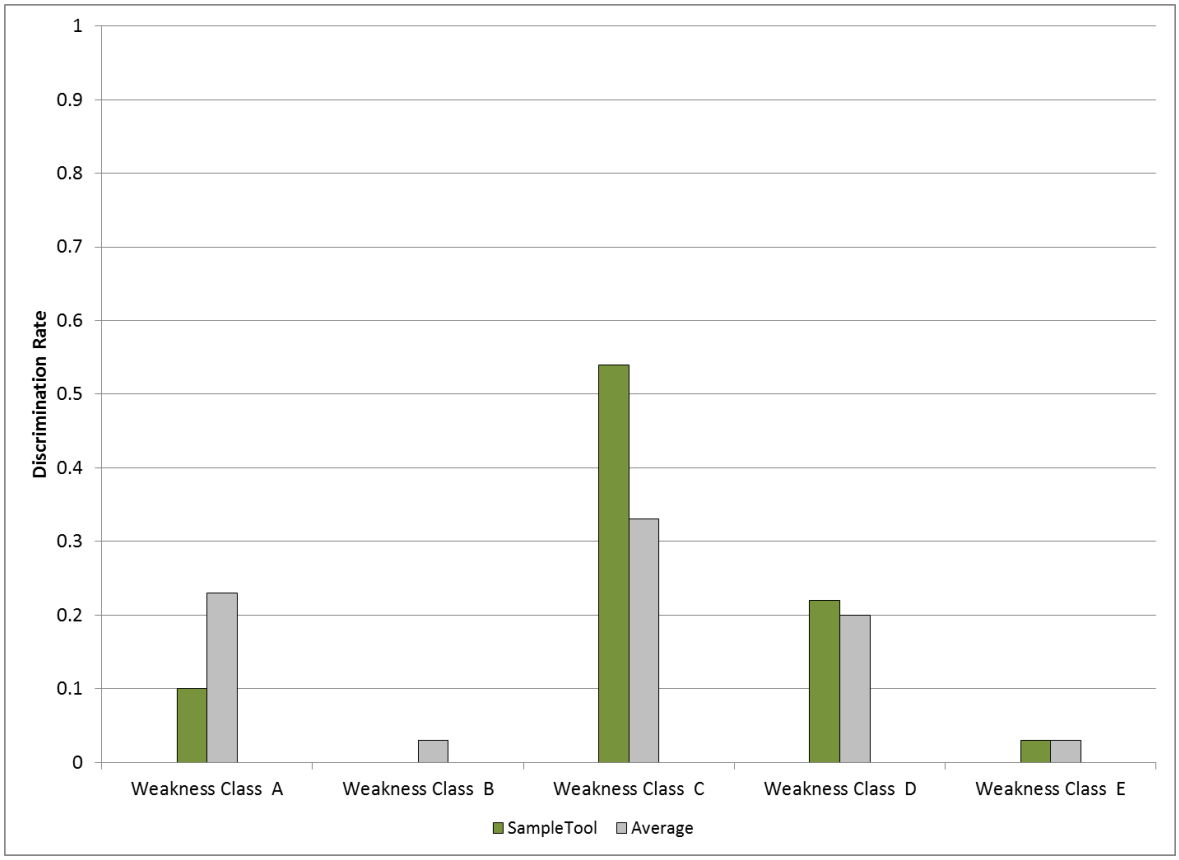

Figure 4 – Example Discrimination Rate Graph for a single tool

In the example in Figure 4, the graph shows that SampleTool has an above average Discrimination Rate in Weakness Classes C and D, and below average Discrimination Rates in Weakness Classes A and B. SampleTool has an average rate in Weakness Class E.

3.2 Results by Weakness Class

3.2.1 Precision Graph by Weakness Class

Figure 5 shows an example of a Precision Graph for a single Weakness Class. Even when all values are relatively small, the Y-axis scale on these graphs is not adjusted in order to compare tools to the ideal Precision of 1. An individual tool's Precision for the Weakness Class is indicated with a blue bar. The average Precision for all tools across the Weakness Class is indicated by a red bar and is always located on the far right side of the graph. The average is calculated using only the tools that cover this Weakness Class. In the example below, SampleTool2 is not included in the average Precision calculation.

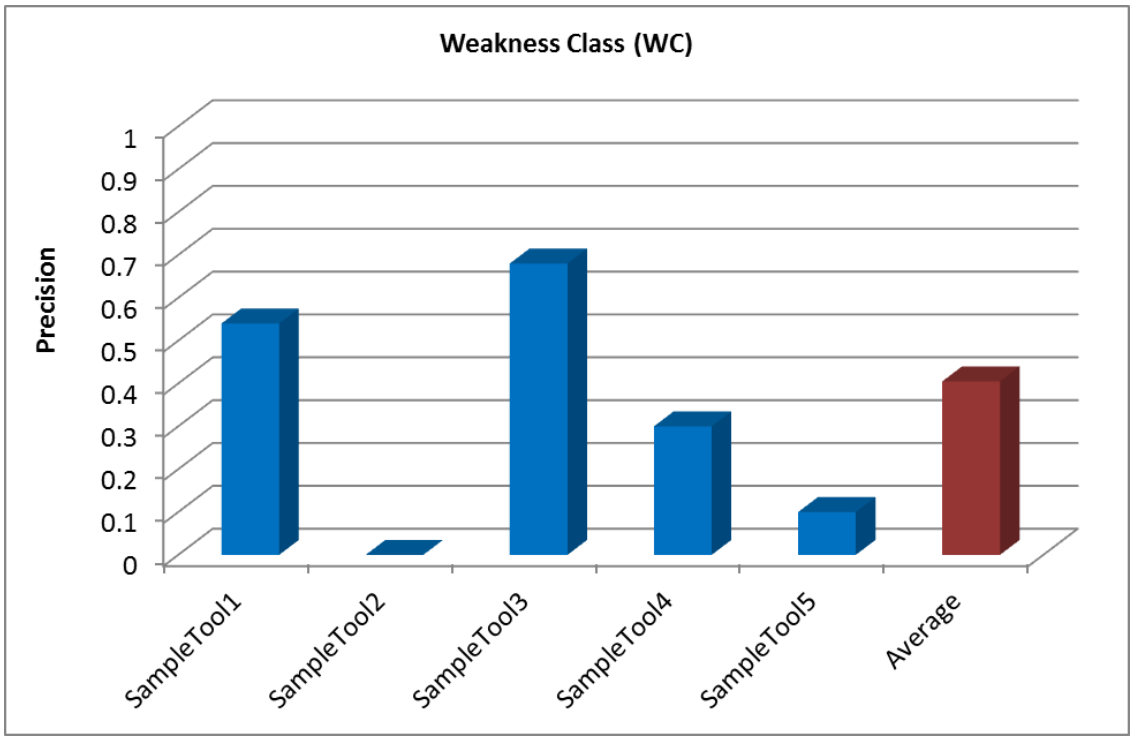

Figure 5 – Example Precision Graph for a single Weakness Class

In the example in Figure 5, the graph shows that SampleTool1 and SampleTool3 performed above average in this Weakness Class. SampleTool4 and SampleTool5 performed below average and SampleTool2 did not cover this Weakness Class.

3.2.2 Recall Graph by Weakness Class

Figure 6 shows an example of a Recall Graph for a single Weakness Class. Even when all values are relatively small, the Y-axis scale on these graphs is not adjusted in order to compare tools to the ideal Precision of 1. An individual tool's Recall for the Weakness Class is indicated with a purple bar. The average Recall for all tools across the Weakness Class is indicated by a red bar and is always located on the far right side of the graph. The average is calculated using all tools, which includes those that do not cover this Weakness Class.

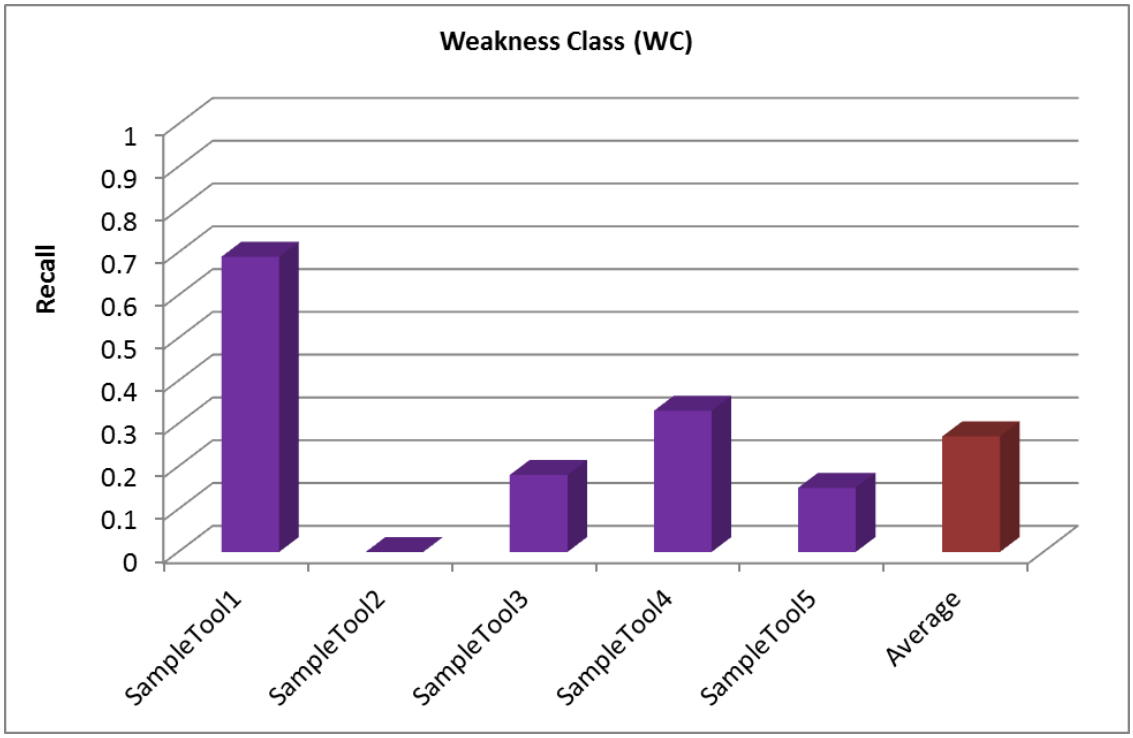

Figure 6 – Example Recall Graph for a single Weakness Class

In the example in Figure 6, the graph shows that SampleTool1 and SampleTool4 performed above average in this Weakness Class. SampleTool3 and SampleTool5 performed below average and SampleTool2 did not cover this Weakness Class.

3.2.3 Precision-Recall Graph by Weakness Class

Figure 7 illustrates an example of a Precision-Recall graph showing each tool's performance compared to the average for a given Weakness Class. Notice that the Precision metric is mapped to the vertical axis and the Recall metric is mapped to the horizontal axis. A tool's relation to both metrics is represented by a point on the graph. If a tool did not report at least one True Positive for this Weakness Class then it is not shown on the graph. The closer the point is to the top right, the stronger the tool is in the given area.

The circle marker represents a tool's actual metric value for the specified Weakness Class. A green circle indicates that the tool performed better than average in both Precision and Recall as

compared to the average. A red circle indicates that the tool performed below average in both Precision and Recall as compared to the average. A gray circle indicates that the tool did not perform better or worse in both Precision and Recall. For example, this could indicate that a tool had better than average Precision, but worse than average Recall for the Weakness Class. The black diamond represents the average values for all the tools that produced findings for the given Weakness Class. The labels at each circle contain the tool names and the label at the diamond contains the abbreviated Weakness Class name.

In general, movement of a specific tool away from the average toward the upper right demonstrates a relatively greater capability in the given area (as indicated by the arrow in the graph's background).

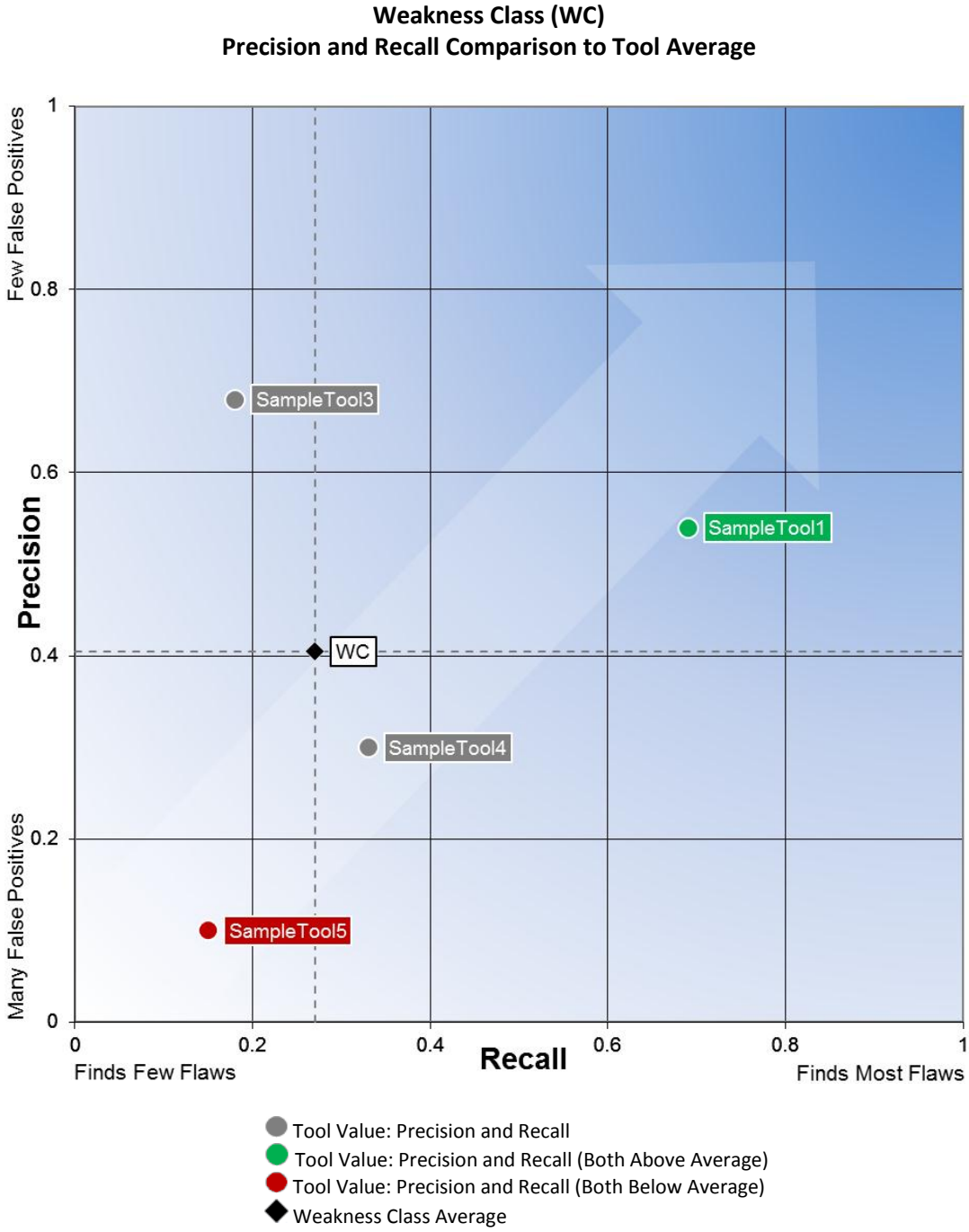

Figure 7 – Example Precision-Recall Graph for Weakness Class (WC)

In the example in Figure 7, the graph shows that the SampleTool1 is stronger than the average of all tools when focusing on this Weakness Class. Both Precision and Recall for the tool are above average as evident by the green data point and by moving from the black diamond upward, meaning higher Precision, and to the right, indicating higher Recall. SampleTool5 is weaker compared to the average of all tools relating to this Weakness Class as indicated by the red data

point. Both metrics are below average, and the line moves from the black diamond downward, meaning less Precision, and to the left, indicating less Recall.

For the results associated with SampleTool3 and SampleTool4, where the line moves to the upper left or the lower right, more analysis is often needed. In these situations, the tool is above average for one metric but below average for the other.

3.2.4 Discrimination Rate Graph by Weakness Class

Figure 8 shows an example of a Discrimination Rate Graph for a single Weakness Class. Even when all values are relatively small, the Y-axis scale on these graphs is not adjusted in order to compare tools to the ideal Discrimination Rate of 1. An individual tool's Discrimination Rate for the Weakness Class is indicated with a green bar. The average Discrimination Rate for all tools across the Weakness Class is indicated by a red bar and is always located on the far right side of the graph. The average is calculated using all tools, which includes those that do not cover this Weakness Class.

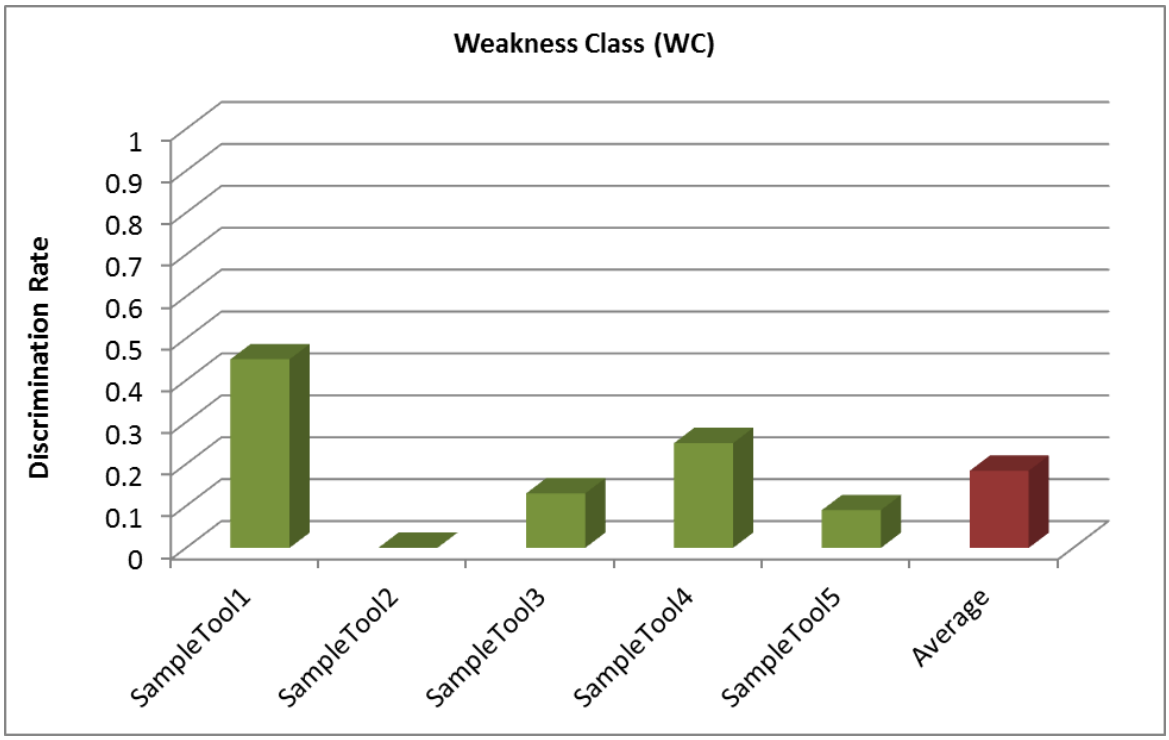

Figure 8 – Example Discrimination Rate Graph for a single Weakness Class

In the example in Figure 8, the graph reveals SampleTool1 and SampleTool4 performed above average in this Weakness Class. SampleTool3 and SampleTool5 performed below average and SampleTool2 did not cover this Weakness Class.

3.2.5 Precision-Recall and Discrimination Results by Weakness Class

Table 5 shows an example of a Precision-Recall and Discrimination results chart. This chart can be used to view the results of all tools in a Weakness Class. This table supports the graphs shown in the previous sections.

Note that a value of .00, not shown here, is used to indicate a non-zero value for Recall and Discrimination Rate that is too small to be presented in the table, and when rounded to two decimal places would otherwise display inaccurately as 0.

Tool	Precision	F-Score	Recall	Disc. Rate
SampleTool1	.54	.61	.69	.45
SampleTool2	-	0	0	0
SampleTool3	.68	.28	.18	.13
SampleTool4	.30	.31	.33	.25
SampleTool5	.10	.12	.15	.09
Average	.41	.26	.27	.18

Table 5 – Weakness Class Precision-Recall and Discrimination Results

3.3 Combined Tool Results

Since there are a variety of commercial and open source static analysis tools available, developers can use more than one tool to analyze a code base. The purpose of combining two tools is to show how adding a second tool might complement the tool already in use. Since tools can have some overlap, the goal would be to use a second tool that is stronger in the areas where the current tool is lacking.

This section describes the tables and graphs used to show the effects of combining two tools.

3.3.1 Combination of Two Tools

Table 6 shows the combined Discrimination Rate and Recall results when combining two sample tools. The tool's own results are shown in the light gray boxes. The green boxes indicate the highest combined values in the table for both Discrimination Rate and Recall.

Tool A / Tool B	SampleTool1		SampleTool2		SampleTool3		SampleTool4		SampleTool5	
	Disc. Rate	Recall	Disc. Rate	Recall	Disc. Rate	Recall	Disc. Rate	Recall	Disc. Rate	Recall
SampleTool1	.25	.53	.27	.59	.60	.77	.45	.73	.26	.59
SampleTool2	.27	.59	.06	.11	.47	.70	.35	.52	.07	.25
SampleTool3	.60	.77	.47	.70	.45	.67	.80	.91	.47	.80
SampleTool4	.45	.73	.35	.52	.80	.91	.33	.50	.34	.62
SampleTool5	.26	.59	.07	.25	.47	.80	.34	.62	.02	.22
Legend:	Individual Tool Result		Highest Discrimination Rate / Recall in Table							

Table 6 – Combined Discrimination Rate and Recall for Two Tools

In the example in Table 6, the table shows that SampleTool3 performs the best of all individual tools studied, but when combined with SampleTool4, yields the best results for both Discrimination Rate and Recall. Notice that the data in the table is symmetric, so the results for each tool are the same whether you are looking down a column or across a row.

Figure 9 shows combined Discrimination Rate and Recall graphs. Each bar chart shows the overlap between the tool named below the chart (the "base" tool) and each of the other tools (the "additional" tools). Each vertical bar shows the overlap between the base tool and one additional tool. The lowest, blue segment shows the results (Discrimination Rate or Recall) for the base tool only; the topmost, green segment shows the results for the additional tool only; and the middle segment shows the overlap, or the same results reported by both tools. That is, the bottom two segments combined indicate the overall results for the base tool, while the top two segments combined indicate the overall results for the additional tool.

Note that in each multiple bar chart, a "water line" appears at the same height for each bar. This shows the base tool's own results (although its overlap, and hence its results for flaws reported only by the base tool, varies).

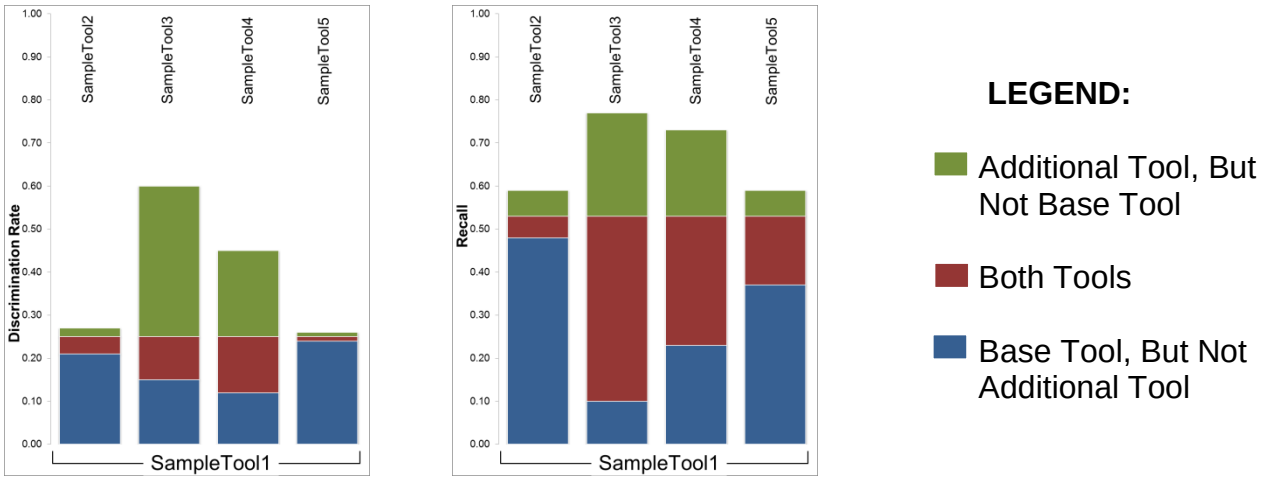

Figure 9 – Combined Discrimination Rate and Recall for Two-Tool Combinations

The Discrimination Rate chart example on the left in Figure 9 shows that combining SampleTool1 with SampleTool3 yields the best results. In fact, adding SampleTool3 more than doubles the Discrimination Rate as opposed to using SampleTool1 alone. Combining SampleTool1 with SampleTool2 or SampleTool5 is ineffective as the Discrimination Rate shows a very small increase using these tool combinations.

The Recall chart example on the right in Figure 9 shows that combining SampleTool1 with SampleTool3 yields the best results. However, there is a large amount of overlap when combining these two tools. Although the combination of SampleTool1 and SampleTool4 does not generate the highest combined Recall, the overlap between the two tools is smaller than the overlap between SampleTool1 and SampleTool3 and should be taken into consideration.

An ideal two-tool combination chart would show a bar whose total value is close to 1 and whose overlap, or red shaded area, is barely visible. This would indicate that the two tools combined to find nearly all of the flaws (Recall) or discriminate across nearly all of the test cases (Discrimination Rate) with very little overlap. A two-tool combination resulting in a large amount of overlap serves little value.

3.3.2 Multiple Tool Coverage

Table 7 shows the overall coverage for all tools across all of the test cases. For each number of tools, the sum of test cases where exactly that number of tools correctly reported the flaw (True Positive) is shown. This table also shows the sum of test cases where exactly that number of tools reported the flaw without reporting any False Positives (Discrimination).

In addition, Table 7 also shows the number of test cases where no tool reported the flaw or a made a Discrimination. Figure 10 shows the percentage of flaws and Figure 11 shows the percentage of Discriminations shown in Table 7 as pie charts.

Number of Tool(s)	# of Flaws Found by Tool(s)	% of All Flaws Found by Tool(s)	# of Discriminations Found By Tool(s)	Discrimination Rate %
No Tools	660	34.6%	990	51.8%
Exactly One Tool	625	32.7%	550	28.8%
Exactly Two Tools	350	18.3%	250	13.1%
Exactly Three Tools	200	10.5%	80	4.2%
Exactly Four Tools	50	2.6%	40	2.1%
Exactly Five Tools	25	1.3%	0	0.0%
Total	**1910**	**100%**	**1910**	**100%**

Table 7 – Multiple Tool Overlap Results

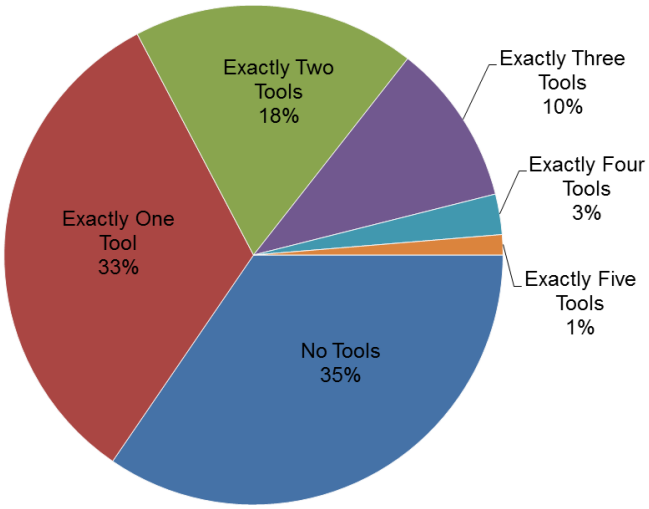

Figure 10 – True Positive Overlap Graph

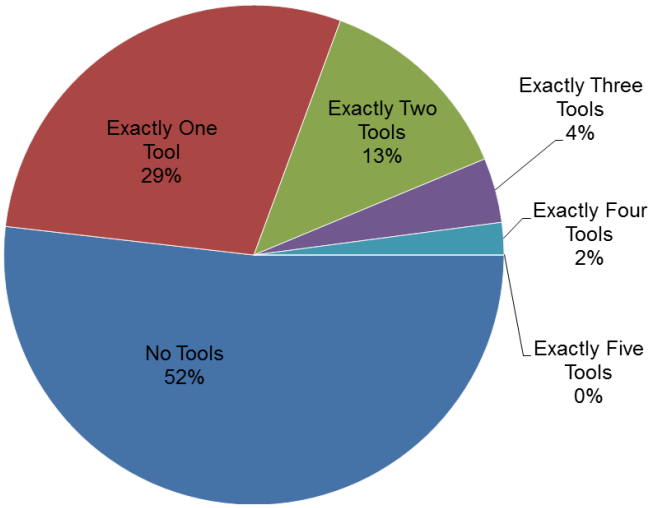

Figure 11 – Test Case Discrimination Coverage Graph

Table 8 shows how well the tools performed collectively with finding flaws in the test cases for each Weakness Class. In this example, the tools showed the worst coverage, based upon averages, in Weakness Class E (15.7%). The tools had the best coverage and found nearly all of the flaws in Weakness Class D (92.2%).

All together, the tools found more than half of the flaws in 4 out of 5 Weakness Classes.

Weakness Class (WC)	# Total Test Cases in WC	# Test Cases Not Found By Any Tool	% in WC Not Found By Any Tool	# Test Cases Found By At Least One Tool	% in WC Found By At Least One Tool
Weakness Class A	511	173	33.9%	338	66.1%
Weakness Class B	953	284	29.8%	669	70.2%
Weakness Class C	439	95	21.6%	344	78.4%
Weakness Class D	720	56	7.8%	664	92.2%
Weakness Class E	460	388	84.3%	72	15.7%

Table 8 – Percentage of Flaws Found in Test Cases by Weakness Class

Section 4: CAS Tool Study

The CAS uses the methodology described in Section 2 to perform its annual static analysis tool studies. The following paragraphs describe in more detail how tools are analyzed.

4.1 Tool Run

The CAS follows a standard procedure for running each tool on the Juliet test cases. The purpose is to provide a consistent process that can be used in future studies or, if the need arises, for retesting a specific tool. This section provides an overview of the process. Detailed, step-by-step records of each tool run are documented during the study.

4.1.1 Test Environment

The CAS configures a "base" virtual machine (VM) running the Windows operating system with all software needed to compile and run the test cases from the Juliet Test Suite. This base VM is then converted into a template. For each tool, the CAS uses this template to deploy a new VM. All steps for the tool runs are performed in separate virtual environments using the local administrative account in order to prevent conflicts and test each tool in isolation.

4.1.2 Tool Installation and Configuration

Each static analysis tool is installed on its own VM in accordance with the vendor's specifications, along with the current version of the Juliet test cases. There are strict guidelines established that limit the adjustment of settings to ensure that all tools are treated fairly. For tools in which no vendor input can be obtained, the approach is to turn on all of the rules related to the test cases.

4.1.3 Tool Execution and Conversion of Results

Each tool is executed from the command line via CAS-created scripts. For tools that cover multiple languages, each analysis of the test cases for a specific language family is considered as a separate run. If errors are found during or after the analysis, the tool can be run on individual test cases without running the entire tool suite again.

The CAS exports the results of the analysis from each tool from its VM to a separate analysis system. In order to perform analysis, all the results must be in a similar format. Because each tool exports its results differently, the CAS has developed its own data format to normalize all output.

4.1.4 Scoring of Tool Results

The final step in the tool run process is to establish which results represent real flaws in the test cases (True Positives) and which do not (False Positives). After True Positives and False

Positives have been scored, False Negatives are then determined. For each test case in which a tool did not correctly identify the flaw and register a True Positive, a False Negative is added.

The "scoring" of results only includes result types that are related to the test case in which they appear. Tool results indicating a weakness that is not the focus of the test case (such as an unintentional flaw in the test case) are ignored. Results are scored using a CAS created tool which automates the scoring process.

4.1.5 Metrics

Using the factors described in Section 2.5, the CAS generates metrics by adhering to certain rules, detailed in the following sections, as part of its overall analysis strategy.

4.1.6 Precision

When calculating the Precision, duplicate True Positive values are ignored. That is, if a tool reports two or more True Positives on the same test case, the Precision is calculated as if the tool reports only one True Positive. However, duplicate False Positive results are included in the calculation.

Some of the Juliet test cases are considered "bad-only," meaning they only contain a flawed construct. Since the bad-only test cases can impact the results, they are excluded from all Precision calculations.

4.1.7 Recall

As in the calculation for Precision, duplicate True Positive values are also ignored when determining Recall. If a tool reports two or more True Positives on the same test case, the Recall is calculated as if the tool reports only one True Positive.

4.1.8 F-Score

In the 2012 tool study, equal weighting was used for both Precision and Recall when calculating the F-Score. However, alternate F-Scores could be calculated by using higher weights for Precision (thus establishing a preference that tool results will be correct) or by using higher weights for Recall (thus establishing a preference that tools will find more weaknesses).

As previously explained in 4.1.6, some of the Juliet test cases are considered bad-only, meaning they only contain a flawed construct. Since the bad-only test cases can impact the results, they are also excluded from all F-Score calculations.

4.1.9 Weighting

The Juliet test cases are designed to test a tool's ability to analyze different control and data flows. However, for each flaw, a single test case is created that contains no control or data flows and is generated to test the most basic, or simplest, form of the flaw. These are referred to as the "baseline" test cases.

The total weight for all test cases covering a given flaw is equal to 1. Because the baseline test case should be the easiest to find, it is given a weight of 0.5 and the remaining weight (0.5) is distributed equally among the remaining test cases. Therefore, all control and data flow variants are weighted equally for a given flaw. Some flaws do not allow for additional control and data flow test cases, such as class-based flaws. In these instances, there is a single test case for which the weight is equal to 1.

Table 9 shows sample tool results for a given flaw containing a baseline, two data flow, and three control flow test cases. Table 10 shows the Precision, Recall, and F-Score calculation differences when adding weights to the test cases. The sample values for the metrics increase when using weighted values; however, this is not always the case when using real data.

	#TPs	#FPs	#FNs	Weight	Weighted TPs	Weighted FPs	Weighted FNs
Baseline	1	0	0	0.5	0.5	0	0
Data Flow1	0	1	1	0.1	0	0.1	0.1
Data Flow2	1	2	0	0.1	0.1	0.2	0
Control Flow1	1	1	0	0.1	0.1	0.1	0
Control Flow2	1	2	0	0.1	0.1	0.2	0
Control Flow3	0	2	1	0.1	0	0.2	0.1
Totals	4	8	2	1	0.8	0.8	0.2

Table 9 - Sample Tool Results for a Single Flaw Type

	Unweighted	Weighted
Precision	4 / (4 + 8) = **0.33**	0.8 / (0.8 + 0.8) = **0.50**
Recall	4 / (4 + 2) = **0.67**	0.8 / (0.8 + 0.2) = **0.80**
F-Score	2 * ((0.33 * 0.67) / (0.33 + 0.67)) = **0.44**	2 * ((0.50 * 0.80) / (0.50 + 0.80)) = **0.62**

Table 10 - Sample Precision, Recall, and F-Score values for a Single Flaw Type

Appendix A: Juliet Test Case CWE Entries and Weakness Classes

Table 11 shows the CWE entries included in each Weakness Class as defined by the CAS and ordered by Weakness Class.

Weakness Class	CWE ID	CWE Name
Authentication and Access Control	15	External Control of System or Configuration Setting
	222	Truncation of Security-relevant Information *
	223	Omission of Security-relevant Information *
	247	Reliance on DNS Lookups in a Security Decision
	256	Plaintext Storage of a Password
	259	Use of Hard-coded Password
	272	Least Privilege Violation
	284	Improper Access Control
	491	Public cloneable() Method Without Final ('Object Hijack')
	500	Public Static Field Not Marked Final
	549	Missing Password Field Masking
	566	Authorization Bypass Through User-Controlled SQL Primary Key
	582	Array Declared Public, Final, and Static
	605	Multiple Binds to the Same Port *
	607	Public Static Final Field References Mutable Object
	613	Insufficient Session Expiration
	620	Unverified Password Change
Buffer Handling	121	Stack-based Buffer Overflow
	122	Heap-based Buffer Overflow
	123	Write-what-where Condition
	124	Buffer Underwrite ('Buffer Underflow')
	126	Buffer Over-read
	127	Buffer Under-read
	176	Improper Handling of Unicode Encoding
	188	Reliance on Data/Memory Layout *
	242	Use of Inherently Dangerous Function
	464	Addition of Data Structure Sentinel *
	680	Integer Overflow to Buffer Overflow
	758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior *
	785	Use of Path Manipulation Function without Maximum-sized Buffer

Weakness Class	CWE ID	CWE Name
Code Quality	398	Indicator of Poor Code Quality
	477	Use of Obsolete Functions
	478	Missing Default Case in Switch Statement
	480	Use of Incorrect Operator *
	481	Assigning instead of Comparing *
	482	Comparing instead of Assigning *
	484	Omitted Break Statement in Switch
	486	Comparison of Classes by Name *
	561	Dead Code
	563	Unused Variable
	570	Expression is Always False
	571	Expression is Always True
	579	J2EE Bad Practices: Non-serializable Object Stored in Session *
	581	Object Model Violation: Just One of Equals and Hashcode Defined *
	585	Empty Synchronized Block
	597	Use of Wrong Operator in String Comparison *
	676	Use of Potentially Dangerous Function
	685	Function Call With Incorrect Number of Arguments *
	688	Function Call With Incorrect Variable or Reference as Argument *
Control Flow Management	364	Signal Handler Race Condition
	366	Race Condition within a Thread
	367	Time-of-check Time-of-use (TOCTOU) Race Condition
	382	J2EE Bad Practices: Use of System.exit()
	383	J2EE Bad Practices: Direct Use of Threads
	479	Signal Handler Use of a Non-reentrant Function
	483	Incorrect Block Delimitation
	572	Call to Thread run() instead of start()
	606	Unchecked Input for Loop Condition
	609	Double-Checked Locking
	666	Operation on Resource in Wrong Phase of Lifetime *
	667	Improper Locking
	674	Uncontrolled Recursion
	698	Execution After Redirect (EAR)
	764	Multiple Locks of a Critical Resource
	765	Multiple Unlocks of a Critical Resource
	832	Unlock of a Resource that is not Locked
	833	Deadlock
	835	Loop with Unreachable Exit Condition ('Infinite Loop')

Weakness Class	CWE ID	CWE Name
Encryption and Randomness	315	Plaintext Storage in a Cookie
	319	Cleartext Transmission of Sensitive Information
	327	Use of a Broken or Risky Cryptographic Algorithm
	328	Reversible One-Way Hash
	329	Not Using a Random IV with CBC Mode
	336	Same Seed in PRNG
	338	Use of Cryptographically Weak PRNG
	523	Unprotected Transport of Credentials
	759	Use of a One-Way Hash without a Salt
	760	Use of a One-Way Hash with a Predictable Salt
	780	Use of RSA Algorithm without OAEP
Error Handling	248	Uncaught Exception
	252	Unchecked Return Value
	253	Incorrect Check of Function Return Value
	273	Improper Check for Dropped Privileges
	390	Detection of Error Condition Without Action
	391	Unchecked Error Condition
	396	Declaration of Catch for Generic Exception
	397	Declaration of Throws for Generic Exception
	440	Expected Behavior Violation
	584	Return Inside Finally Block
	600	Uncaught Exception in Servlet
	617	Reachable Assertion
File Handling	23	Relative Path Traversal
	36	Absolute Path Traversal
	377	Insecure Temporary File
	378	Creation of Temporary File With Insecure Permissions
	379	Creation of Temporary File in Directory with Incorrect Permissions
	675	Duplicate Operations on Resource
Information Leaks	209	Information Exposure Through an Error Message
	226	Sensitive Information Uncleared Before Release
	244	Improper Clearing of Heap Memory Before Release ('Heap Inspection')
	499	Serializable Class Containing Sensitive Data
	526	Information Exposure Through Environmental Variables
	533	Information Exposure Through Server Log Files
	534	Information Exposure Through Debug Log Files
	535	Information Exposure Through Shell Error Message
	539	Information Exposure Through Persistent Cookies
	591	Sensitive Data Storage in Improperly Locked Memory

Weakness Class	CWE ID	CWE Name
Information Leaks (cont.)	598	Information Exposure Through Query Strings in GET Request
	614	Sensitive Cookie in HTTPS Session Without 'Secure' Attribute
	615	Information Exposure Through Comments
Initialization and Shutdown	400	Uncontrolled Resource Consumption ('Resource Exhaustion')
	401	Improper Release of Memory Before Removing Last Reference ('Memory Leak')
	404	Improper Resource Shutdown or Release
	415	Double Free
	416	Use After Free
	457	Use of Uninitialized Variable
	459	Incomplete Cleanup
	568	finalize() Method Without super.finalize()
	580	clone() Method Without super.clone()
	586	Explicit Call to Finalize()
	590	Free of Memory not on the Heap
	665	Improper Initialization
	672	Operation on a Resource after Expiration or Release
	761	Free of Pointer not at Start of Buffer
	762	Mismatched Memory Management Routines
	772	Missing Release of Resource after Effective Lifetime
	773	Missing Reference to Active File Descriptor or Handle
	775	Missing Release of File Descriptor or Handle after Effective Lifetime
	789	Uncontrolled Memory Allocation
Injection	78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
	80	Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)
	81	Improper Neutralization of Script in an Error Message Web Page
	83	Improper Neutralization of Script in Attributes in a Web Page
	89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
	90	Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')
	113	Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')
	129	Improper Validation of Array Index
	134	Uncontrolled Format String
	426	Untrusted Search Path
	427	Uncontrolled Search Path Element
	470	Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')

Weakness Class	CWE ID	CWE Name
Injection (cont.)	601	URL Redirection to Untrusted Site ('Open Redirect')
	643	Improper Neutralization of Data within XPath Expressions ('XPath Injection')
Malicious Logic	111	Direct Use of Unsafe JNI
	114	Process Control
	321	Use of Hard-coded Cryptographic Key
	325	Missing Required Cryptographic Step
	506	Embedded Malicious Code
	510	Trapdoor
	511	Logic/Time Bomb
	546	Suspicious Comment
Number Handling	190	Integer Overflow or Wraparound
	191	Integer Underflow (Wrap or Wraparound)
	193	Off-by-one Error
	194	Unexpected Sign Extension
	195	Signed to Unsigned Conversion Error
	196	Unsigned to Signed Conversion Error
	197	Numeric Truncation Error
	369	Divide By Zero
	681	Incorrect Conversion between Numeric Types
Pointer and Reference Handling	395	Use of NullPointerException Catch to Detect NULL Pointer Dereference
	467	Use of sizeof() on a Pointer Type
	468	Incorrect Pointer Scaling
	469	Use of Pointer Subtraction to Determine Size
	475	Undefined Behavior For Input to API *
	476	NULL Pointer Dereference
	562	Return of Stack Variable Address
	587	Assignment of a Fixed Address to a Pointer
	588	Attempt to Access Child of a Non-structure Pointer
	690	Unchecked Return Value to NULL Pointer Dereference
	843	Access of Resource Using Incompatible Type ('Type Confusion')

**Previously categorized under the "Miscellaneous Weakness Class"*

Table 11 – CWE Entries and Test Cases in each Weakness Class