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 2011

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 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

All company and product names used in this document are registered trademarks or trademarks of their respective owners in the United States of America and/or other countries.

Table of Contents

Abstract

Part of the mission for the National Security Agency's Center for Assured Software (CAS) is to increase the degree of confidence that software used in the 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 NSA CAS regularly conducts studies using a scientific, methodical approach that measures and rates effectiveness of these tools in a standard and repeatable manner. The methodology (termed the CAS Static Analysis Tool Study Methodology) is based on a set of artificially created "known answer tests" that comprise examples of "good code" as well as "flawed code". In applying the methodology, the tester tests all tools using the common "testing corpus". The methodology then offers a common way to "score" the tools 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.[a]

This report 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. It is available for public consumption, comment and adoption. Comments on the methodology are welcome and can be sent to cas@nsa.gov.

[a] This test suite is available as "Juliet Test Suite" published as part of the National Institute of Standards and Technology Software Assurance Technology Exposition (SATE) project.

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 Department of Defense (DoD), the National Security Agency'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 used is free from intentional and unintentional exploitable vulnerabilities. The CAS accomplishes this mission by assisting organizations in deploying processes and tools to address assurance throughout the Software Development Lifecycle (SDLC).

As part of an overall secure 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.

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: CAS Methodology

The CAS methodology requires the use of test cases to perform tool evaluations. Upon completion, the tool results are assigned a result type that can be used for further analysis.

2.1 Juliet Test Cases

In order to study static analysis tools, users need software for 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 outweigh 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 *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.

2.2 Assessment

2.2.1 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 in order to avoid conflicts and to allow independent analysis. It is important that each tool is treated the same and thus an equal amount of hardware resources is given to each one. Every tool is executed using its command line interface (CLI) and the results are exported upon completion.

2.2.2 Scoring Results

In order to determine the tool's performance, tool results are scored using result types. Table 1 contains the various result types that can be assigned as well as their definitions.

Result Type	Explanation
True Positive (TP)	Tool correctly reports the flaw that is the target of the test case.
False Positive (FP)	Tool reports a flaw with a type that is the target of the test case, but the flaw is reported in non-flawed code.
False Negative (FN)	This row is not a tool result. A false negative result is added for each test case for which there is no true positive.
(blank)	This row is a tool result where none of the result types above apply. More specifically, either: • The tool result is not in a test case file • The tool result type is not associated with the test case in which it is reported

Table 1 – Summary of Result Types

For example, consider a test case that targets a buffer overflow flaw. The test case contains flawed code in which data in a large buffer is attempted to be placed 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 in which a buffer overflow cannot occur. 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 flawed code, then a result should be added that is considered a false negative. If a tool reports any other type of flaw, for example a memory leak, in the flawed or non-flawed code, then the result type should be left blank as this type of flaw is not the target of the test case and is considered an incidental flaw.

2.3 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 sections.

2.3.1 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 on the test cases. The following sections describe these metrics in greater detail.

Precision

In the context of the methodology, Precision (also known as “positive predictive value") means 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 more 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 on 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 a piece of code that contains 20 flaws has a Recall 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. 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 for the F-Score will always be between the values for Precision and Recall (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.

2.3.2 Discriminations and Discrimination Rate

Another set of metrics looks for areas where a tool showed it could discriminate between flaws and non-flaws. This section describes these metrics in greater detail.

The purpose of discriminations and the discrimination rate is to differentiate unsophisticated tools doing simple pattern matching from tools that perform 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 report a false positive on 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 where the tool reports a Discrimination. That is:

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

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 positive(s), regardless of whether it reported false positive(s). 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: CAS Tool Study

The CAS uses the methodology described in Section 2 during its regular static analysis tool studies. The following sections describe how tools are analyzed in more detail.

3.1 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.

3.1.1 Test Environment

The CAS sets up a "base" virtual machine running Microsoft Windows with all software needed to compile and run the Juliet test cases. For each tool run, the CAS either copies the "base" virtual machine or creates a new snapshot within the "base" virtual machine. All steps for the tool runs are performed in separate virtual environments using the local administrative account in order to prevent conflicts and to test each tool in isolation.

3.1.2 Tool Installation and Configuration

Each static analysis tool is installed on its own virtual machine 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 where no vendor input can be obtained, the approach is to turn on all of the rules related to the test cases.

3.1.3 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 a separate run. If errors are found during or after the analysis, the tool can be run on individual test cases.

The CAS then exports the results of the analysis from each tool. 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.

3.1.4 Scoring of Tool Results

The final step in the tool run process is to determine which results represent real flaws in the test cases (true positives) and which do not (false positives).

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 in scoring and analysis and their result type is left blank. Results are scored using a CAS created tool which automates the scoring process.

Section 4: CAS Tool Analysis

4.1 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 CWE entries that contain similar weaknesses. 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 Class. Table 2 provides a summary list of Weakness Classes used in the study, along with an example weakness in that Weakness Class.

The Miscellaneous Weakness Class is used to hold a collection of individual CWE entries that do not fit into the other classes. Therefore, the weaknesses in the Miscellaneous Weakness Class do not have a common theme.

Weakness Class	Example Weakness (CWE Entry)
Authentication and Access Control	CWE-620: Unverified Password Change
Buffer Handling (C/C++ only)	CWE-121: Stack-based Buffer Overflow
Code Quality	CWE-561: Dead Code
Control Flow Management	CWE-362: Race Condition
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 Leak Through Debug Log Files
Initialization and Shutdown	CWE-415: Double Free
Injection	CWE-89: SQL Injection
Malicious Logic	CWE-506: Embedded Malicious Code
Miscellaneous	CWE-480: Use of Incorrect Operator
Number Handling	CWE-369: Divide by Zero
Pointer and Reference Handling	CWE-476: Null Pointer Dereference

Table 2 – Weakness Classes

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

4.1.1 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 a tool's ability to check whether or not the source code is preventing unauthorized access to the system.

4.1.2 Buffer Handling

Improper buffer handling can lead to attackers crashing or gaining complete control of a system. An example would be a buffer overflow that allows an adversary to execute his/her code. The test cases in this Weakness Class test a tool's ability to find buffer access violations in the source code. This Weakness Class is only valid for the C/C++ language family.

4.1.3 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. This is not an inherent security risk; however it may lead to maintenance issues in the future. The test cases in this Weakness Class test a tool's 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 (CWE-484), an omitted default case in a switch (CWE-478), and dead code (CWE-561).

4.1.4 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 consequence of a race condition is a deadlock which leads to a denial of service. The test cases in this Weakness Class test a tool's ability to find issues in the order of execution within the source code.

4.1.5 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 a tool's ability to check for proper encryption and randomness in the source code.

4.1.6 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 a tool's ability to check for proper error handling within the source code.

4.1.7 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 to a file 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 a tool's ability to check for proper file handling within the source code.

4.1.8 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 a tool's ability to check for information leaks within the source code.

4.1.9 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 a tool's ability to check for proper initialization and shutdown of resources in the source code.

4.1.10 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 inject malicious scripts to acquire elevated privileges or steal session tokens in order to gain access to sensitive information. This can often be prevented using proper input validation and/or data encoding. The test cases in this Weakness Class test a tool's ability to check for injection weaknesses in the source code.

4.1.11 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 a tool's ability to check for malicious logic in the source code.

4.1.12 Miscellaneous

The weaknesses in this class do not fit into any of the other Weakness Classes. An example would be an assignment instead of a comparison. Control flow statements allow developers to compare variables to certain values in order to determine the proper path to execute. Unfortunately, they also allow developers to make variables assignments. If an assignment is made where a comparison was intended then it could lead to the wrong path of execution.

4.1.13 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 a tool's ability to check for proper number handling in the source code.

4.1.14 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. If referenced, this can cause an exception and lead to a system crash. The test cases in this Weakness Class test a tool's ability to check for proper pointer and reference handling.

4.2 Metrics

Using the factors described in Section 2.3, the CAS generates metrics by adhering to certain rules, detailed in the following sections, as part of its overall analysis strategy.

4.2.1 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 on that test case. Duplicate false positive results are included in the calculation, however.

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.

4.2.2 Recall

Like with Precision, duplicate true positive values are ignored when calculating Recall. That is, 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 on that test case.

4.2.3 F-Score

Equal weighting is used for both Precision and Recall when calculating the F-Score. Alternate F-Scores can 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, 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 F-Score calculations.

4.2.4 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 basic form of the flaw ("baseline" test cases).

In the past, all test cases for a given flaw were weighted equally. Therefore, the Precision, Recall, and F-Score for baseline test cases were calculated using the same weighting as the test cases containing control and data flows. Based upon feedback from the software assurance community and further research by the CAS, it was determined that the baseline test cases should

merit more "weight" than the control and data flow test cases. In general, if a tool is able to find a given flaw then it should be able to at least find it in the baseline test case.

Instead of each test case being weighted equally, the total weight for all test cases covering a given flaw is now equal to 1. The baseline test case 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 1 shows sample tool results for a given flaw containing a baseline, two data flow, and three control flow test cases. Table 2 shows the Precision, Recall, and F-Score calculation differences when adding weights to the test cases.

	#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 1 – 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 2 – Sample Precision, Recall, and F-Score values for a Single Flaw Type

As shown in the tables above, the sample values for Precision, Recall, and F-Score increase when using weighted values; however, this is not always the case when using real data.

Section 5: CAS 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 issues are found, but these issues can be hidden in a sea of false positives, making it extremely time-consuming to interpret the

results. The Recall metric alone does not give the analyst this perspective. By examining both the Recall and Precision values, the analyst can get a better picture of the tool's overall strengths.

5.1 Results by Tool

5.1.1 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 a tool performs with respect to other tools, i.e., it is not known what a "good" number is. It is impossible for an analyst to know if 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 on 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.

If the tool has a higher value than the average, a small green triangle pointing up was used. For values below 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.

	Sample Size	Tool Results		
Weakness Class	**# of Flaws**	**Precision**	**F-Score**	**Recall**
Weakness Class A	511	▼ .25	▼ .20	▼ .17
Weakness Class B	953	-	-	0
Weakness Class C	433	▲ .96	▲ .72	▲ .58
Weakness Class D	720	▼ .56	.57	▲ .58
Weakness Class E	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 0.05 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 – even though the Precision was 1 – which suggests that all tools had high Precision within this Weakness Class.

Note that if a tool does not produce any true positive findings or false positive 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 interpreted as an indication that the tool does not attempt to find flaws related to the given Weakness Class, or at least does not attempt to find the flaw types represented by the test cases for that Weakness Class.

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 has to produce findings. In general, when there are more opportunities, one can have more statistical confidence in the metric.

To help understand the Precision, Recall, and F-Score results for a tool, each metric is compared against the weighted average of all the tools. The weighted average was calculated for each functional variant with one-half weighting placed on the baseline flow variant, and the other half of the weighting being distributed equally across the remaining flow variants.

5.1.2 Precision Graph by Tool

Figure 1 shows an example of a Precision Graph for a single tool. 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. The tool's Precision for a given Weakness Class is indicated by a blue square. The average Precision for a given Weakness Class is indicated by a red square. The dotted line connecting the averages is shown as a visual aid; however, the averages are not related. If a tool did not report at least one true positive for a given Weakness Class, the Precision is undefined and is indicated on the graph by a black line-filled square at 0 on the x-axis.

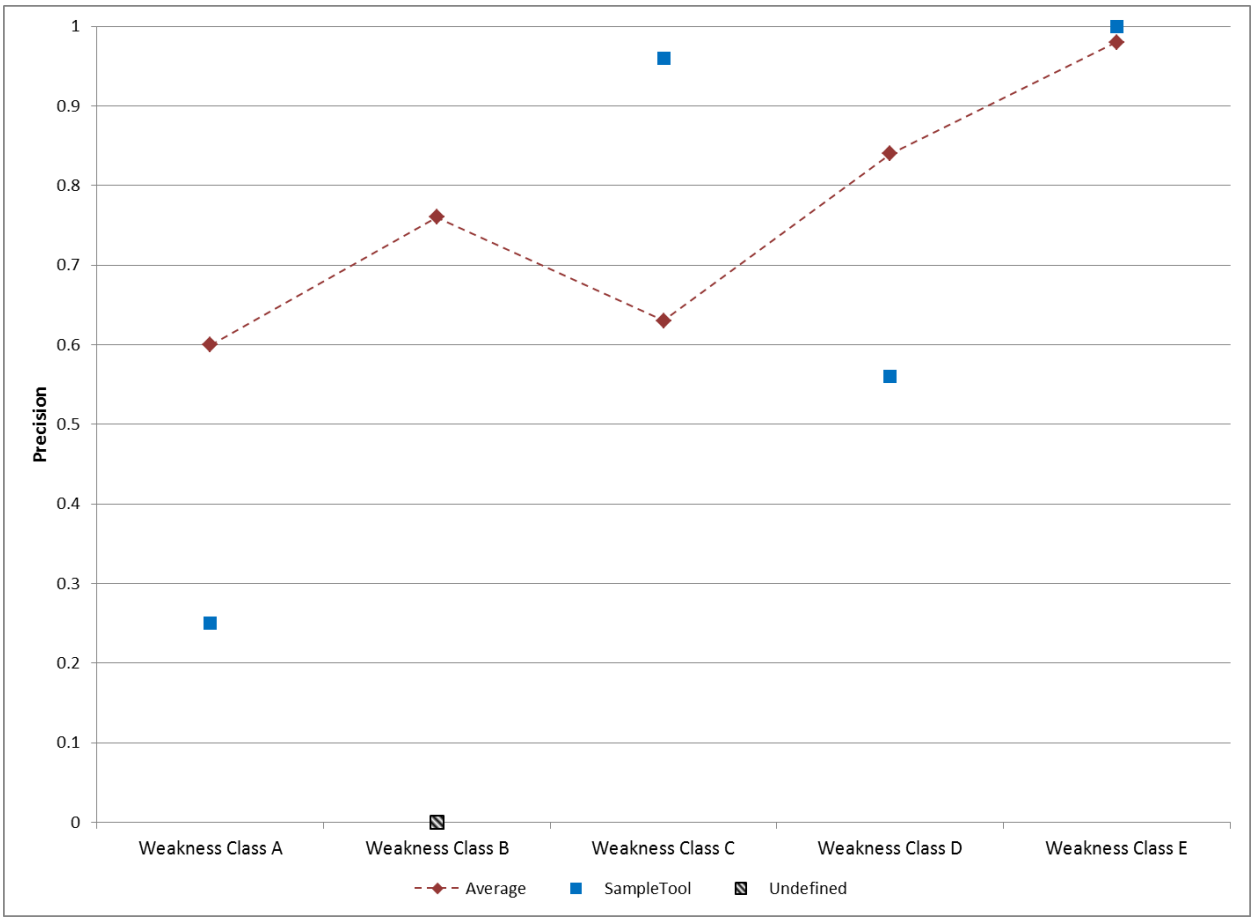

Figure 1 – Example Precision Graph for a single tool

In the example in Figure 1, the graph shows that the SampleTool is relatively strong when focusing on Weakness Class C, and less strong, when compared to the other tools, for Weakness Class A and Weakness Class D.

5.1.3 Recall Graph by Tool

Figure 2 shows an example of a Recall Graph for a single tool. 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 Recall of 1. The tool's Recall for a given Weakness Class is indicated by a purple square. The average Recall for a given Weakness Class is indicated by a red square. The dotted line connecting the averages is shown as a visual aid; however, the averages are not related.

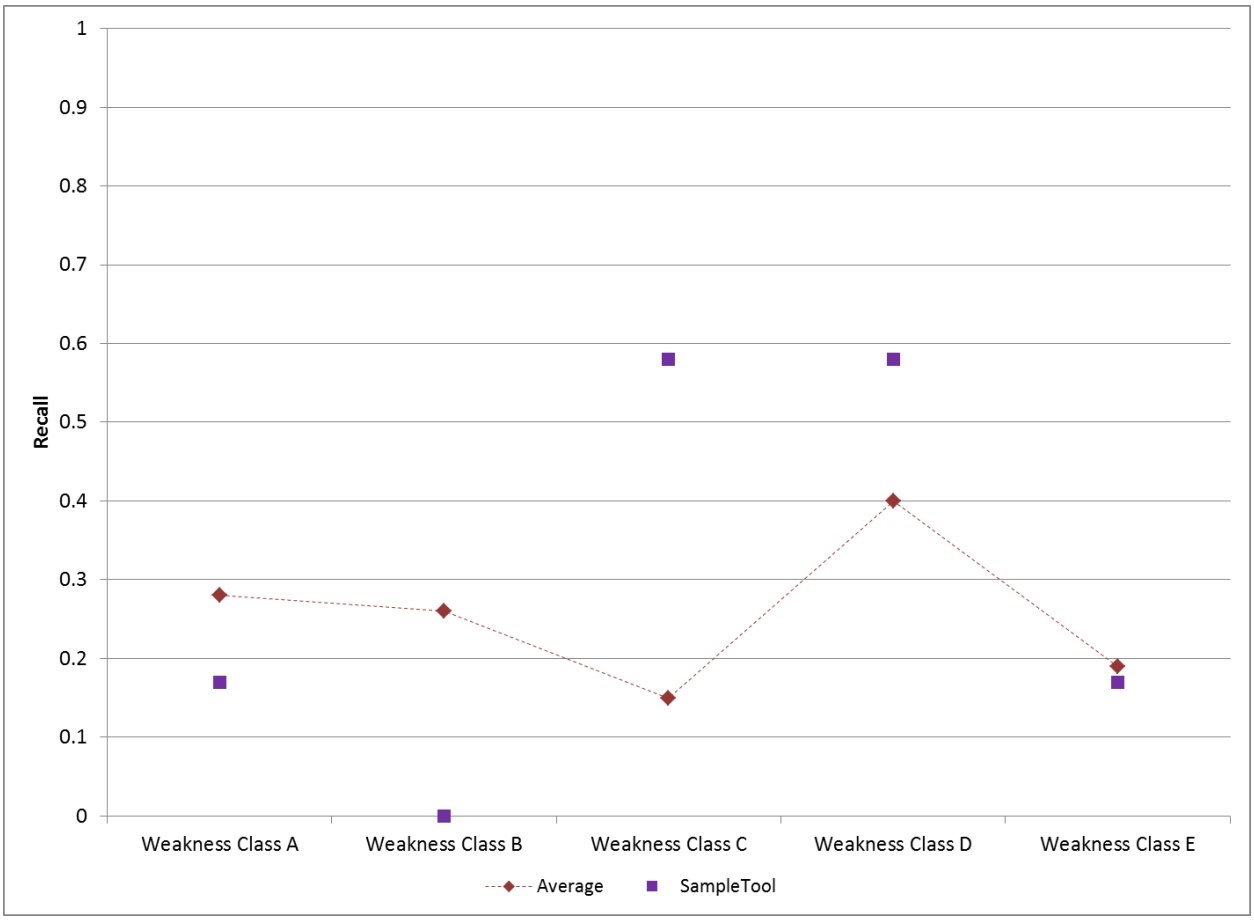

Figure 2 – Example Recall Graph for a single tool

In the example in Figure 2, the graph shows that the SampleTool is strong when focusing on Weakness Class C and Weakness Class D, but not strong related to Weakness Class A and Weakness Class B. The graph also indicates the tool found 0 true positives for Weakness Class B.

5.1.4 Precision-Recall Graphs by Tool

Figure 3 shows an example of a Precision-Recall graph. 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 square marker (in white) represents the tool's actual metric values for the specified Weakness Class as calculated by the scoring function. The other point, shown as a black circle, represents the average metric values for all the tools that produced findings for the given Weakness Class.

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.

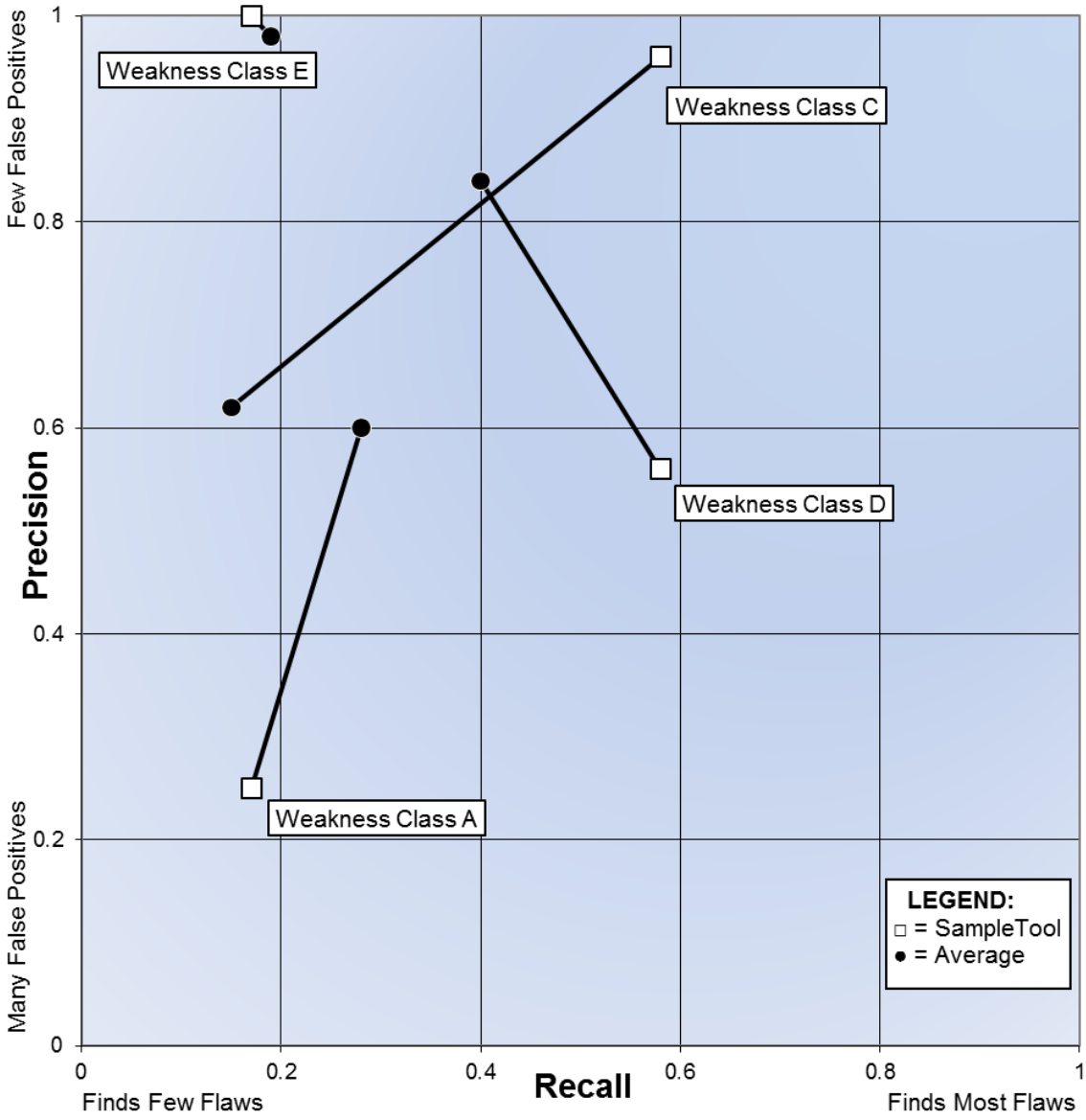

Figure 3 – Example Precision-Recall Graph for single tool

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 moving from the black dot upward, meaning higher Precision, and to the right, indicating higher Recall. SampleTool is less strong compared to the average of all tools relating to Weakness Class A. Both metrics are below average, and the line moves from the black dot 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.

5.1.5 Discriminations and Discrimination Rate Table by Tool

To summarize each tool's Discrimination Rate on the test cases, a table is produced to show how the tool performs regarding Discriminations. However, similar to Precision, Recall, and F-Score, when interpreted in isolation, each 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 higher value than the average, a small green triangle pointing up was used. For values below 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 tool's Discriminations and Discrimination Rates on each Weakness Class are shown in a table like Table 4.

	Sample Size	Tool Results	
Weakness Class	**# of Flaws**	**Discriminations**	**Disc. Rate**
Weakness Class A	511	50	▼ .10
Weakness Class B	953	0	0
Weakness Class C	433	234	▲ .54
Weakness Class D	720	150	.21
Weakness Class E	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.

5.1.6 Discrimination Rate Graphs by Tool

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.

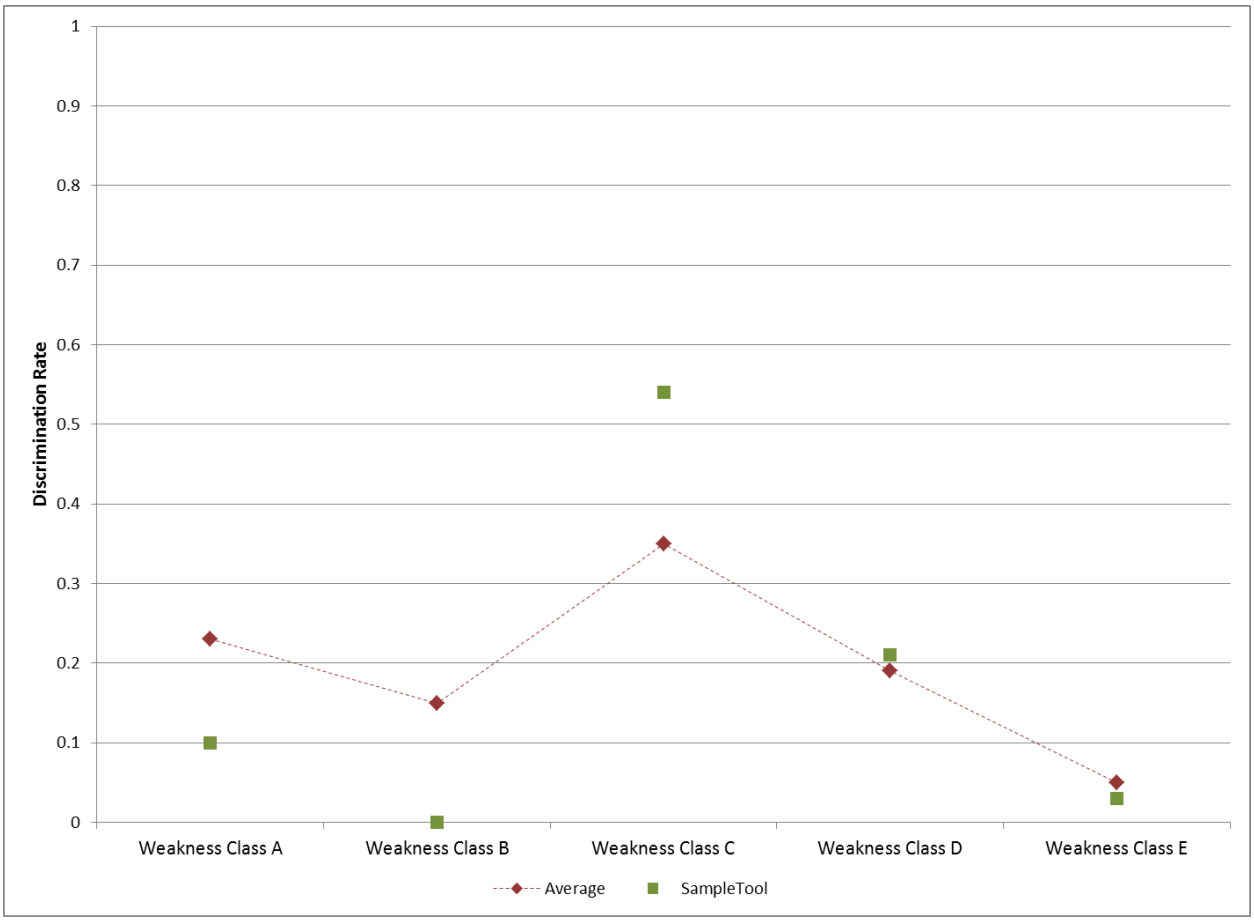

Figure 4 – Example Discrimination Rate Graph for a single tool

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

5.2 Results by Weakness Class

5.2.1 Precision Graphs 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.

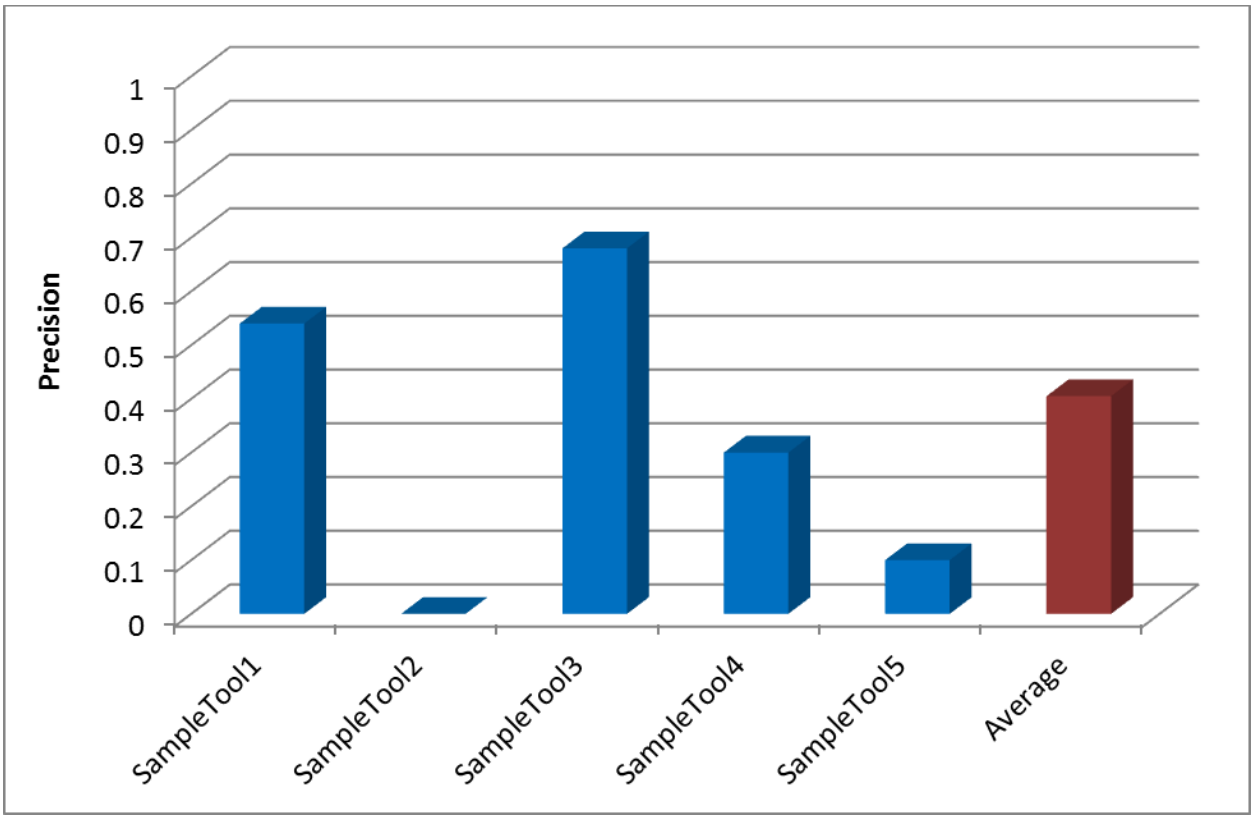

Figure 5 – Example Precision Graph for a single Weakness Class

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.

5.2.2 Recall Graphs 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.

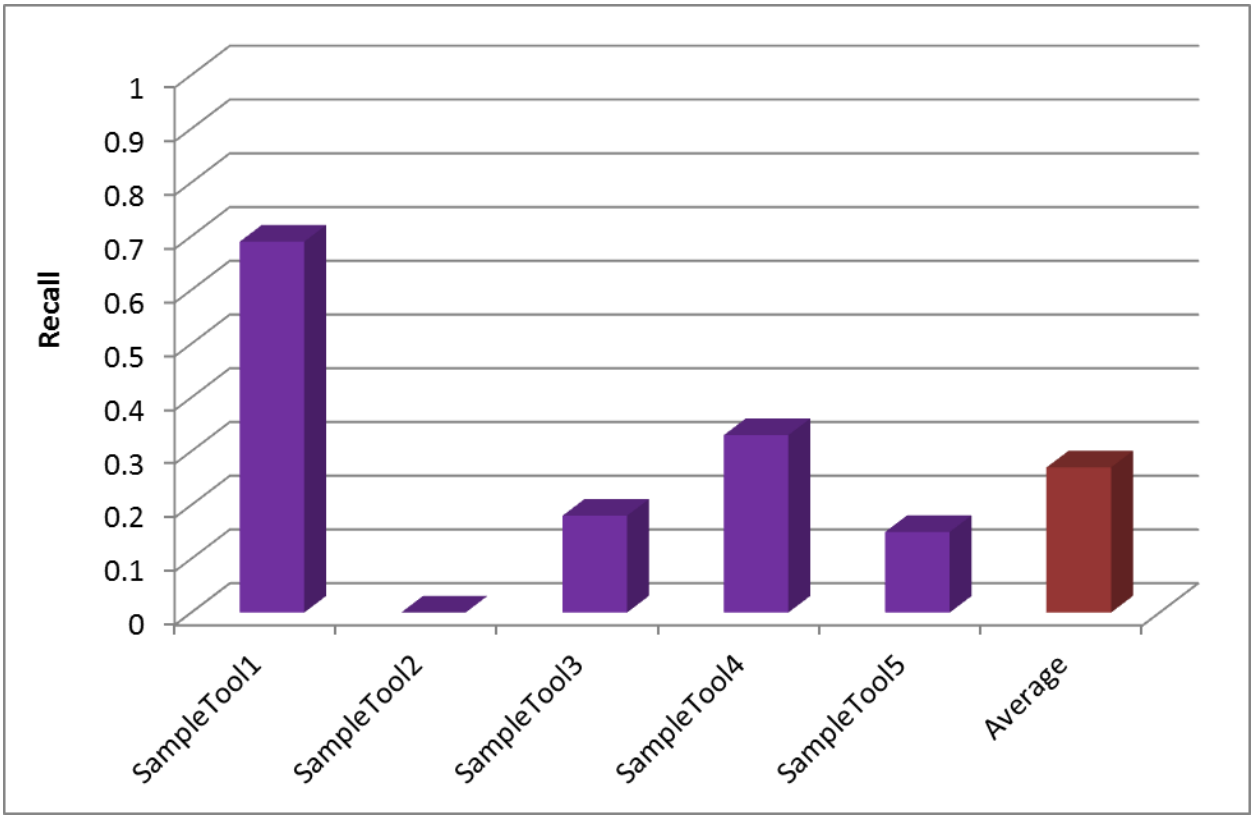

Figure 6 – Example Recall Graph for a single Weakness Class

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.

5.2.3 Discrimination Rate Graphs by Weakness Class

Figure 7 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 purple 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.

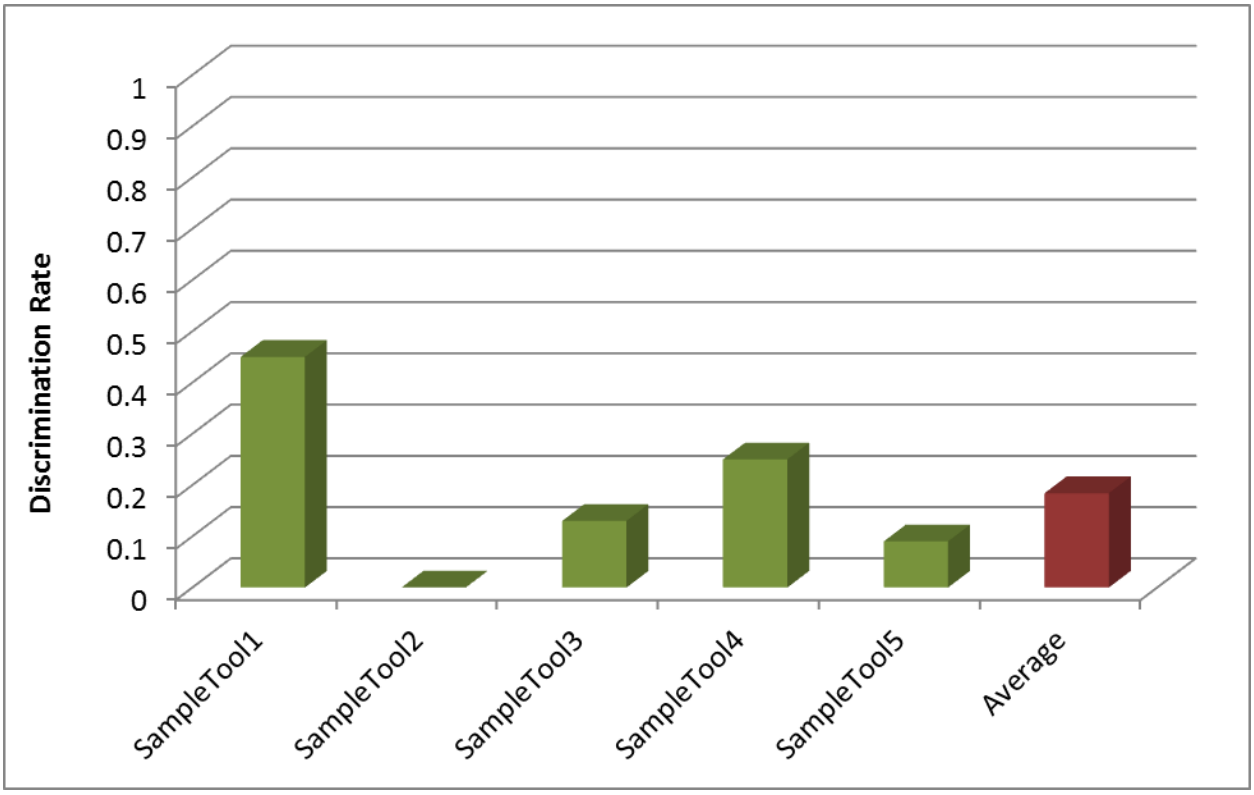

Figure 7 – Example Discrimination Rate Graph for a single Weakness Class

In the example in Figure 7, 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.

5.2.4 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.

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

5.3 Results for Two Tool Combinations

Since there are a variety of commercial and open source static SCA tools available, developers can use more than one tool to analyze a code base. The purpose of comparing 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.

5.3.1 Combined Discrimination Rate and Recall Table

Table 6 shows the combined Discrimination Rate and Recall results when combining two sample static SCA 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.

5.3.2 Combined Discrimination Rate and Recall Graphs

Figure 8 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).

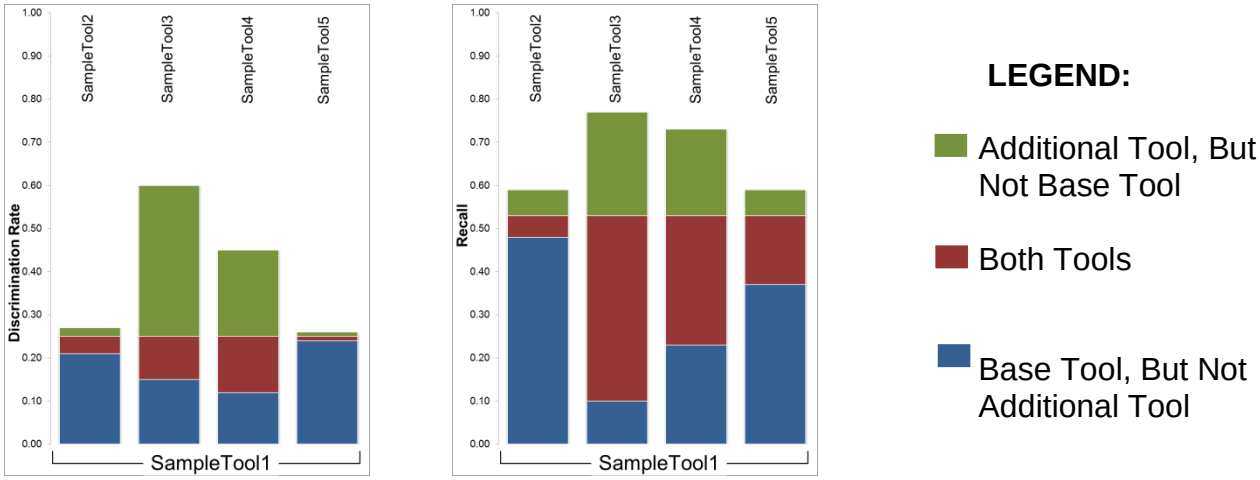

Figure 8 – Combined Discrimination Rate and Recall for Two-Tool Combinations

The Discrimination Rate chart example 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 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.

5.3.3 Tool Coverage

Table 7 shows the overall coverage for all tools across all of the test cases. For each number of tools, the number of test cases where exactly that number of tools reported the flaw (a "True Positive") and reported the flaw without reporting any False Positives (a "Discrimination") is shown. This table also shows the number of test cases where no tool reported the flaw or a Discrimination. Figure 9 shows the percentage of flaws and Figure 10 shows the percentage of discriminations shown in Table 7 as pie charts.

Tool	# Test Cases with True Positives	True Positive %	# Test Cases with Discriminations	Discrimination %
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%
No Tools	660	34.6%	990	51.8%
Total	**1910**	**100%**	**1910**	**100%**

Table 7 – Tool Coverage Results

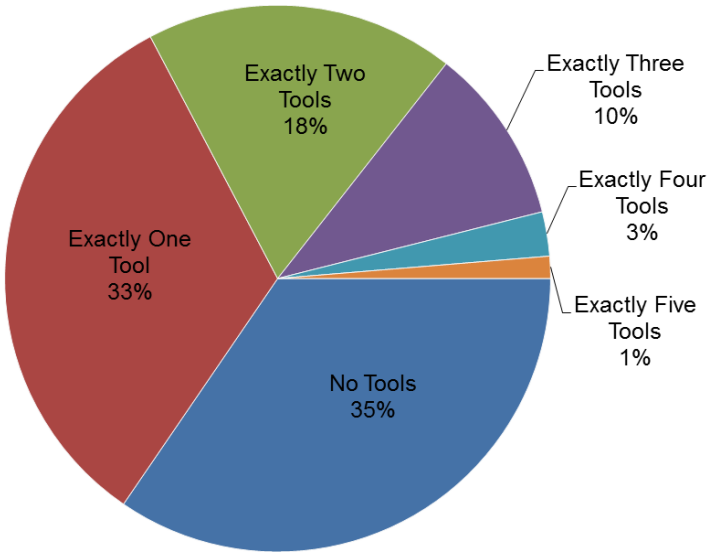

Figure 9 – Tool True Positive Coverage Graph

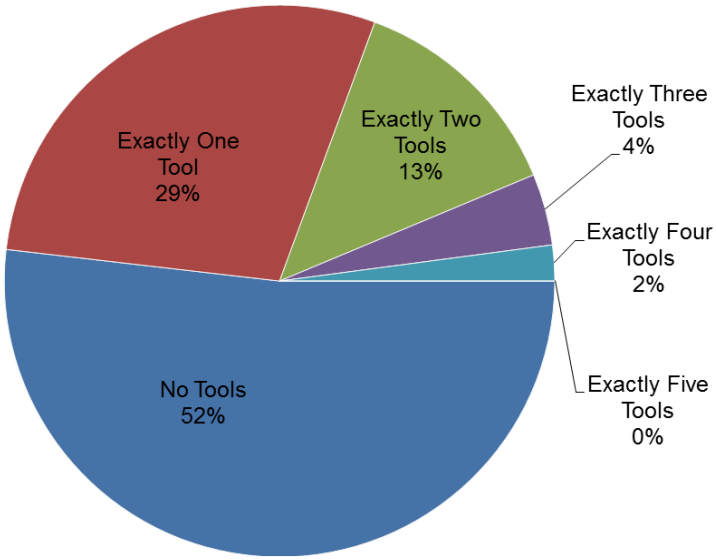

Figure 10 – Tool Discrimination Coverage Graph

Appendix A: Juliet Test Case CWE Entries and CAS Weakness Classes

Table 8 shows the CWE entries included in each Weakness Class defined by the CAS ordered by Weakness Class.

Weakness Class	CWE Entry ID	CWE Entry Name
Authentication and Access Control	15	External Control of System or Configuration Setting
	247	Reliance on DNS Lookups in a Security Decision
	256	Plaintext Storage of a Password
	259	Use of Hard-coded Password
	272	Least Privilege Principal
	284	Access Control (Authorization) Issues
	491	Public cloneable() Method Without Final ('Object Hijack')
	500	Public Static Field Not Marked Final
	549	Missing Password Field Masking
	560	Use of umask() with chmod-style Argument
	566	Access Control Bypass Through User-Controlled SQL Primary Key
	582	Array Declared Public, Final, and Static
	607	Public Static Final Field References Mutable Object
	613	Insufficient Session Expiration
	614	Sensitive Cookie in HTTPS Session Without 'Secure' Attribute
	620	Unverified Password Change
	784	Reliance on Cookies without Validation and Integrity Checking in a Security Decision
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
	242	Use of Inherently Dangerous Function
	680	Integer Overflow to Buffer Overflow
	785	Use of Path Manipulation Function without Maximum-sized Buffer
Code Quality	398	Indicator of Poor Code Quality
	477	Use of Obsolete Functions
	478	Missing Default Case in Switch Statement

Weakness Class	CWE Entry ID	CWE Entry Name
	484	Omitted Break Statement in Switch
	547	Use of Hard-coded Security-relevant Constants
	561	Dead Code
	563	Unused Variable
	570	Expression is Always False
	571	Expression is Always True
	585	Empty Synchronized Block
	676	Use of Potentially Dangerous Function
Control Flow Management	180	Incorrect Behavior Order: Validate Before Canonicalize
	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()
	584	Return Inside Finally Block
	606	Unchecked Input for Loop Condition
	609	Double-Checked Locking
	674	Uncontrolled Recursion
	698	Redirect Without Exit
	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')
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
	330	Use of Insufficiently Random Values
	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

Weakness Class	CWE Entry ID	CWE Entry Name
	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
	392	Missing Report of Error Condition
	396	Declaration of Catch for Generic Exception
	397	Declaration of Throws for Generic Exception
	440	Expected Behavior Violation
	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 Insecure Permissions
	675	Duplicate Operations on Resource
Information Leaks	204	Response Discrepancy Information Leak
	209	Information Exposure Through an Error Message
	226	Sensitive Information Uncleared Before Release
	244	Improper Clearing of Heap Memory Before Release (‘Heap Inspection’)
	497	Exposure of System Data to an Unauthorized Control Sphere
	499	Serializable Class Containing Sensitive Data
	533	Information Leak Through Server Log Files
	534	Information Leak Through Debug Log Files
	535	Information leak Through Shell Error Message
	591	Sensitive Data Storage in Improperly Locked Memory
	598	Information Leak Through Query Strings in GET Request
	615	Information Leak 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

Weakness Class	CWE Entry ID	CWE Entry Name
	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 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	Failure to Sanitize CRLF Sequences in HTTP Headers (aka 'HTTP Response Splitting')
	129	Improper Validation of Array Index
	134	Uncontrolled Format String
	436	Untrusted Search Path
	427	Uncontrolled Search Path Element
	470	Use of Externally-Controlled Input to Select Classes or Code (aka 'Unsafe Reflection')
	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
	304	Missing Critical Step in Authentication
	321	Use of Hard-coded Cryptographic Key
	325	Missing Required Cryptographic Step

Weakness Class	CWE Entry ID	CWE Entry Name
	506	Embedded Malicious Logic
	510	Trapdoor
	511	Logic/Time Bomb
	514	Covert Channel
	546	Suspicious Comment
Miscellaneous	188	Reliance on Data/Memory Layout
	222	Truncation of Security-relevant Information
	223	Omission of Security-relevant Information
	464	Addition of Data Structure Sentinel
	475	Undefined Behavior For Input to API
	480	Use of Incorrect Operator
	481	Assigning instead of Comparing
	482	Comparing instead of Assigning
	486	Comparison of Classes by Name
	489	Leftover Debug Code
	579	J2EE Bad Practices: Non-serializable Object Stored in Session
	581	Object Model Violation: Just One of Equals and Hashcode Defined
	597	Use of Wrong Operator in String Comparison
	605	Multiple Binds to Same Port
	666	Operation on Resource in Wrong Phase of Lifetime
	685	Function Call With Incorrect Number of Arguments
	688	Function Call With Incorrect Variable or Reference as Argument
	758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior
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	374	Passing Mutable Objects to an Untrusted Method
	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

Weakness Class	CWE Entry ID	CWE Entry Name
	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')

Table 8 – CWE Entries and Test Cases in each Weakness Class