# On Analyzing Static Analysis Tools

National Security Agency Center for Assured Software (cas@nsa.gov)
July 26, 2011

The National Security Agency (NSA) Center for Assured Software (CAS) conducted a study of static analysis tools for C/C++ and Java in 2010. The purpose of this study was to determine the capabilities of commercial and open source static analysis tools for C/C++ and Java in order to provide objective information to organizations that are looking to purchase, deploy, or make the best use of static analysis tools.

This document details the methodology of this study, including:

- Purpose and scope of the study
- Tool criteria and selection process
- Software analyzed (Test Cases)
- Environment and procedure used to run tools
- Techniques used to automatically mark results as "True Positive", "False Positive", and "False Negative"
- Grouping of results for analysis (Weakness Classes)
- Analysis metrics calculated (Precision, Recall, F-Score, Discrimination Rate)
- Visualizations employed

# 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. This study focused on the capabilities of automated, flaw-finding, static analysis 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.

In order to identify the capabilities of static analysis tools, the National Security Agency's Center for Assured Software performed this study in 2010.

## 1.2 Center for Assured Software

In order to address the growing lack of Software Assurance in the U.S. Government, the National Security Agency's Center for Assured Software (CAS) was created in 2005. The CAS's mission is to improve the Assurance of software used by the U.S. Government. The CAS accomplishes this mission by assisting organizations in deploying processes and tools to address Assurance throughout the software development life cycle.

As part of an overall secure development process, the CAS advocates the use of static analysis tools. 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 Purpose of the Study

The purpose of this study was to determine the capabilities of commercial and open source static analysis tools for C/C++ and Java in order to provide objective information to organizations that are looking to purchase, deploy, or make the best use of static analysis tools. By identifying the strengths of the tools, this study also aimed to determine how tools could be combined to provide a more thorough analysis of software by using strong tools in each area analyzed.

The goal of this study was not to choose a single "best" tool, to create a benchmark for all tools, or to create an overall tool assessment that combined results across diverse weaknesses types. This study focused solely on tool results. Other factors that an organization should use

in choosing a static analysis tool, such as cost, performance, ease of use, and ability to customize, were not considered.

## 1.4 Scope of the Study

### 1.4.1 Static Analysis Tools

This study examined "static analysis" tools. That is, tools that analyze software for flaws without executing the software. Tools of this type are sometimes called "Static Application Security Testing (SAST) Tools" or "Code Weakness Analysis Tools".

Many static analysis tools perform analysis on the source code to software. Some tools analyze the compiled form of the software, either as native executables, libraries, or program object files or as Java bytecode (an intermediate representation that cannot be executed on most processors directly, but instead must be interpreted on a Java Virtual Machine or compiled into machine code by a Just-in-Time compiler). Tools in this study analyzed the source code and/or the compiled form of software and some tools did not specify how analysis was performed. This study included tools which analyzed binaries compiled and linked with specific, tool-defined options to aid analysis.

### 1.4.2 Languages Covered

Static analysis tools exist for many languages, but the CAS limited this study to two language families due to resource constraints. The language families of C/C++ and Java were chosen because the CAS believed that they are the unmanaged and managed languages most commonly used in software for the US Government.

Although C and C++ are different programming languages, this study examines tool results on the two as a single unit. This decision was made because C++ is a generally a superset of C. All of the static analysis tools in this study which covered C or C++ supported both languages and supported analysis of a single project containing both C and C++ code.

### 1.4.3 Minimum Tool Criteria

Tools were required to meet the following minimum criteria in order to be considered for study, though not all tools meeting these criteria were included in the study:

- Tool must have provided automated analysis. Tools that only assisted a manual analysis were not included.
- Tool must not have required annotations or other changes to the source code in order to perform analysis. No annotations or comments designed to assist the tools were present in the code analyzed in this study. As described in Section 1.4.1, tools that analyzed binaries built with specific configuration options were allowed in this study since building binaries using those configuration options did not require changes to the application's source code.
- Tool must have possessed the ability to identify security-related flaws.
- Tool must have been in the "beta" or later stage of development in September 2010 when the CAS chose the tools used in this study.
- Tool must have run on the Windows operating system.
- Tool must have provided an export or report of results in a format that could be manipulated outside of the tool.
- The tool must have been available to the CAS, either as a product already licensed by the CAS, under a trial license from the vendor, or as an open source or free tool.

## 1.5 Related Work

The CAS is aware of several projects that are related to this study:

- F. Michaud and R. Carbone at Defence Research and Development Canada Valcartier conducted a study of static analysis tools and released a report titled "Practical verification & safeguard tools for C/C++" in November 2007. The report is available at handle.dtic.mil/100.2/ADA479348.
- In 2008, James Walden, Adam Messer, and Alex Kuhl of the Northern Kentucky University investigated the effect of code complexity on static analysis. Results of their study are described in the paper titled "Measuring the Effect of Code Complexity on Static Analysis Results", which is available at faculty.cs.nku.edu/~waldenj/papers/essos2009-long.pdf.
- Martin Johns at the University of Hamburg, Germany, and others, conducted an evaluation of static analysis tools called "Scanstud" in 2007-2008. Slides on this project are available at www.owasp.org/images/7/76/Johns_jodeit_-_ScanStud_OWASP_ Europe_2008.pdf.
- James A. Kupsch and Barton P. Miller of the University of Wisconsin, Madison released a paper titled "Manual vs. Automated Vulnerability Assessment: A Case Study" in June 2009. This paper is available at pages.cs.wisc.edu/~kupsch/vuln_assessment/ManVsAutoVulnAssessment.pdf.
- During 2008, 2009, and 2010, the National Institute for Standards and Technology (NIST) Software Assurance Metrics And Tool Evaluation (SAMATE)

project sponsored the Static Analysis Tool Exposition (SATE, samate.nist.gov/index.php/SATE.html), which examined the performance of static analysis tools on open source applications in C/C++ and Java. This study differed from SATE in three main ways: this study focused on synthetic test cases, this study attempted to systematically identify tool capabilities across a broad spectrum of weakness classes, and this study attempted to determine tool capabilities on both results reported by the tools (true and false positives) and constructs not reported by the tools (true and false negatives).

# Section 2: Study Procedure

This section describes the procedure used to perform this study.

## 2.1 Software Analyzed

In order to study static analysis tools, the CAS needed software for the tools to analyze. The CAS considered using "natural" software or "artificial" software in the study. Natural software is software that was not created to test software analysis tools. Open source projects such as the Apache web server (httpd.apache.org) or the OpenSSH suite (www.openssh.com) could have been used as natural software. Artificial software, on the other hand, is software that contains intentional flaws and that was created to test software analysis tools.

### 2.1.1 Limitations of Natural Code

During the 2006 static analysis tool study, the CAS used a combination of natural and artificial code. In addition, the CAS has followed the National Institute of Standards and Technology (NIST) Static Analysis Tool Exposition (SATE) which examined the performance of static analysis tools on natural code.

Experience from these efforts indicated that using natural code presents specific challenges, such as:

- Evaluating tool results to determine their correctness – When a static analysis tool is run on natural code, each result needs to be reviewed to determine if the code in fact has the specified type of flaw at the specified location (i.e. if the result is correct or a "false positive"). This review is non-trivial for most results on natural code and often the correctness of a given result cannot be determined with a high degree of certainty in a reasonable amount of time.
- Comparing results from different tools – Comparing tool results on natural code is complicated because different static analysis tools report results in different manners. For example, many flaws involve a "source" of tainted data and a "sink" where that data is used in a dangerous manner. Some tools may report the source where others report the sink. Sometimes multiple sources of tainted data all lead to one sink, which may cause different tools to report a different number of results.
- Identifying flaws in the code that no tools find – When evaluating static analysis tools, a "standard" list of all flaws in the code is needed in order to identify which flaws each tool failed to report. With natural code, creating this "standard" is difficult, especially identifying flaws that are not reported by any automated tool and therefore can only be found with manual code review.
- Evaluating tool performance on constructs that do not appear in the code – Natural code has the limitation that even a combination of different projects will likely not contain all flaws and non-flawed constructs that the CAS wants to test. Even flaw types that appear in the code may be obfuscated by complex data and control flows such that tools that report some flaws of that type will not report the flaws in the natural code. To address this issue, the CAS considered using a "seeding" method to embed flaws and non-flaws into natural code. Ultimately, "seeding" was not used in this study because the CAS believed that performing the study using "seeded" code would be overly complex and result in testing fewer constructs than desired.

Based on these experiences and challenges, the CAS decided to develop artificial code for this study. Using artificial code simplified the study because the CAS could control which flaws and non-flaws were included in the code and the location of each flaw.

### 2.1.2 Limitations of Artificial Code

Although the use of artificial code simplified this study and allowed for studying tool results on a large number of flawed and non-flawed constructs, it may limit the applicability of the study results in the following two ways:

- Artificial code is simpler than natural code – The first limitation of the artificial code used in this study is the code's relative simplicity. Some test cases are intentionally the simplest form of the flaw being tested. Even test cases which include data or control flow complexity are relatively simple compared to natural code, both in terms of the number of lines of code and in terms of the number and types of branches, loops, and function calls. This simplicity may have inflated the study results in that tools may have reported flaws during this study that they would rarely report in natural, non-trivial code.

- Frequencies of flaws and non-flawed constructs in the test cases may not reflect their frequencies in natural code – The second limitation of the test cases is that the frequencies of the flaws and non-flawed constructs is likely very different from their frequencies in natural code. Each type of flaw is tested once in the test cases, regardless of how common or rare that flaw type may be in natural code. For this reason, two tools that have similar flaw reporting results on the test cases may provide very different results on natural code, such as if one tool finds common flaws and the other tool only finds rare flaws. Even a tool with poor results on the test cases may have good results on natural code. Similarly, each non-flawed construct also appears only once in the test cases, regardless of how common the construct is in natural code. Therefore, the false positive rates on the test cases may be much different from the rates the tools would have on natural code.

## 2.1.3 Test Case Design

The CAS decided that the benefits of using artificial code outweighed the disadvantages and created artificial code for this study. The CAS built the artificial code as a collection of "test cases". Each test case contained exactly one intentional flaw and contained one or more non-flawed constructs similar to the intentional flaw. The CAS used the non-flawed constructs to determine if the tools could discriminate flaws from non-flaws.

For example, one test case the CAS created illustrated a type of buffer overflow vulnerability. The flawed code in the test case passed the strcpy function a destination buffer that was smaller than the source string. The non-flawed construct passed a large enough destination to strcpy.

The CAS created two sets of test cases for this study, one for C/C++ and one for Java. These test cases were publicly released in March 2011 through the National Institute for Standards and Technology (NIST) as the Juliet Test Suites at samate.nist.gov/SRD/testsuite.php.[a]

## 2.1.4 Test Case Scope

The test cases used in this study focused on functions available on the underlying platform rather than the use of third-party libraries. This section provides further details on the scope of the test cases, including the types of control and data flows studied.

[a] The Juliet Test Suites are not the exact test cases used in the CAS's 2010 study (they incorporate minor bug fixes).

### 2.1.4.1 C/C++ Test Case Scope

Wherever possible, the C/C++ test cases used only Application Programming Interface (API) calls to the C standard library, which is available on all platforms. In order to cover more issues, some test cases targeted the Microsoft Windows platform (using Windows-specific API functions). No third-party C or C++ library functions are used.

The C test case code targeted the C89 standard so that the test cases could be compiled and analyzed using a wide variety of tools that may not support newer versions of the C language.

The test cases limited the use of C++ constructs and features to only the test cases that require them (such as test cases related to C++ classes or the "new" operator). Unless necessary for the flaw type targeted, test cases did not use the C++ standard library.

### 2.1.4.2 Java Test Case Scope

The Java test cases limited the use of features to those found in Java 1.4 unless features introduced in later versions were necessary for the flaw being tested. The test cases covered issues that could affect standalone Java applications or Java Servlets. No test cases specifically covered Java Applets or Java Server Pages (JSPs).

The Java Servlet test cases made use of the Java Servlet API version 2.4 or above. The test cases were developed and distributed with Apache's implementation of this API.

Some of the Java Servlet test cases made use of the StringEscapeUtils class from the Apache Commons Lang library in order to prevent incidental security issues. No other third party library functions were used in the test cases.

### 2.1.4.3 Data and Control Flows Studied

The test cases aimed to exercise the ability of tools to follow various control and data flows in order to properly report flaws and properly disregard non-flaws. The type of control or data flow present in a test case was specified by the "Flow Variant" number included in the name of each test case. Test cases with the same Flow Variant number (but testing different flaw types) used the same type of control or data flow.

Test cases with a flow variant of "01" were the simplest form of the flaws and did not contain added control or data flow complexity. This set of test cases is referred to as the "Baseline" test cases.

Test cases with a flow variant from "02" to "19" (inclusive) covered various types of control flow constructs and are referred to as the "Control Flow" test cases. Test cases with a flow variant of "31" or greater

covered various types of data flow constructs and are referred to as the "Data Flow" test cases.

Not all flaw types had test cases with all flow variants. There were several reasons for this:

- Some flaw types do not involve "data" and therefore could not be used in Data Flow test cases.
- Some flaw types are inherent in a C++ or Java class and could not be placed in Control or Data flows (only a Baseline test case was possible for these flaw types).
- Some flaw types could not be generated by the CAS's custom Test Case Template Engine. Test cases for those flaw types were manually created. Only Baseline ("01" flow variant) test cases were created for these flaw types due to resource constraints.
- Some flaw types are incompatible with some control and data flows in that the CAS's engine created a test case that would not compile or did not function appropriately. Some of these issues are unavoidable because the problem is inherent in the combination of the flaw type and the flow variant. Others of these issues were limitations of the CAS's engine.

### 2.1.5 Test Case Selection

The CAS used several sources when selecting flaw types for test cases:

- The test case development team's experiences in Software Assurance
- Flaw types used in the CAS's 2009 tool study
- Vendor information regarding the types of flaws their tools identify
- Weakness information in MITRE's Common Weakness Enumeration (CWE)

While each test case used a CWE identifier as part of its name, a specific CWE entry for a flaw type was not required in order to create a test case. Test cases were created for all appropriate flaw types and each test case was named using the most relevant CWE entry (which may have been rather generic and/or abstract).

### 2.1.6 Test Case Statistics

This tool study was conducted using the 2010 version of the test case suite. The suite contained two projects: one for C/C++ and one for Java. Table 1 contains statistics on the size and scope of the tested version of the test cases.

| | CWE Entries Covered | Flaw Types | Test Cases | Lines of Code[b] |
|---|---|---|---|---|
| **C/C++** | 116 | 1,432 | 45,324 | 6,338,548 |
| **Java** | 106 | 527 | 13,801 | 3,238,667 |
| **All Test Cases** | **177** | **1,959** | **59,125** | **9,577,215** |

Table 1 – Test Case Statistics

The test cases cover twenty of the 2010 CWE/SANS Top 25 Most Dangerous Programming Errors[c]. Of the five CWE entries on the Top 25 that the test cases do not cover, four are design issues that do not fit into the structure of the CAS test cases and one is an issue specific to the PHP language (which is not in scope for this study).

## 2.2 Tools Studied

The CAS selected tools for this study based on several factors:

- Evaluaton of promotional material supplied by the tool vendors
- Online reviews
- Evaluation of tool methodology and available rule sets
- Results on a small amount of sample code

The criteria that the tools were required to meet for this study are contained in Section 1.4. The tool versions considered were the most recent release available as of September 1, 2010.

Although there are numerous commercial and open source/free static analysis tools available, the resource limitations led the CAS to test only a limited number of tools. For C and C++, the CAS chose six commercial and one open source and/or free static analysis tools. For Java, the CAS chose five commercial and two open source and/or free static analysis tools. Table 2 contains an anonymized list of tools and versions studied.

[b] Lines that are not blank or only comments. Counted using CLOC (cloc.sourceforge.net).

[c] Christey, Steve, ed., "2010 CWE/SANS Top 25 Most Dangerous Programming Errors", MITRE Corporation, http://cwe.mitre.org/top25/ (accessed February 15, 2011).

| Tool | License Model | C/C++ | Java |
|---|---|---|---|
| Tool 1 | Commercial | ✓ | ✓ |
| Tool 2 | Commercial | ✓ | ✓ |
| Tool 3 | Commercial | ✓ | ✓ |
| Tool 4 | Commercial | ✓ | ✓ |
| Tool 5 | Commercial | ✓ | ✓ |
| Tool 6 | Commercial | ✓ | |
| Tool 7 | Open Source | ✓ | |
| Tool 8 | Open Source | | ✓ |
| Tool 9 | Open Source | | ✓ |

Table 2 – Static Analysis Tools Studied

## 2.3 Tool Run Process

The CAS followed a standard process for running each tool on the test cases. This section provides an overview of the process. Detailed, step-by-step records of each tool run were documented during the study.

### 2.3.1 Test Environment

Using VMware, the CAS set up a "base" virtual machine running the 32-bit version of the Microsoft Windows XP operating system. The "base" virtual machine contained software needed to compile and run the test cases. The software installed on the "base" virtual machine included 7zip, Apache Ant, CmdHere Powertoy, Sun Java JDK 6, Sun Java JRE 6, Microsoft File Checksum Integrity Verifier, Microsoft Visual Studio 2008 Professional, Notepad++, Python 3.2.1, and VMware Tools.

For each tool run, the CAS either copied the "base" virtual machine or created a new snapshot in the "base" virtual machine. The virtual machine was not connected to the Internet. All steps for the tool run were performed as the local administrative account.

### 2.3.2 Tool Installation, Execution, and Results Export

Each static analysis tool was installed into a separate virtual machine or snapshot in order to prevent conflicts and test each tool in isolation. The CAS copied the installation and license files for the static analysis tool into the virtual machine. Next, the tool was installed using default settings and according to the installation instructions provided with the tool.

The CAS then ran the static analysis tool using the tools' command line interface. The CAS ran the analysis using the default configuration of the tool and in accordance with the tool's documentation. No adjustments were made to the tool configuration and no changes or annotations were made to the source code of the test cases. The CAS did not consult with tool vendors to optimize or configure the tool.

After the analysis finished, the CAS exported the results of the analysis from the tool. Unfortunately, each tool exported the results in a different format. The exported results were then transformed into standard comma separated value (CSV) format by a Python script or Extensible Stylesheet Language (XSL) transform created by the CAS.

### 2.3.3 Scoring of Tool Results

The next step in the tool run process was to determine which results represented real flaws in the test cases (true positives) and which did not (false positives). A new column named "result type" was added to the result CSV file to hold the result of the scoring process.

The "scoring" of results only included result types that were related to the test case in which they appear. Tool results indicating a weakness that was not the focus of the test case (such as a tool result indicating a memory leak in a buffer overflow test case) were ignored in scoring and analysis.

In past studies, each tool result was scored manually. In the 2010 study, most results were scored using a CAS created tool called the AutoScorer.

#### 2.3.3.1 Weakness ID Mappings

For each tool, the CAS created an XML file to map the tool's weakness IDs to the test cases. For example, if a tool had a weakness ID that indicated occurrences of memory leaks, then it would be mapped to the CWE-401 (Memory Leak) test cases. Using these mappings, the CAS tool was able to determine which result types were related to which test cases.

#### 2.3.3.2 Automated Scoring Process (Pass 1)

The AutoScorer was run two times on each individual tool's results. During the first run, most tool results related to the test case in which they appear are marked as true positives and false positives. The AutoScorer is able to do this by using the Weakness ID Mapping to determine which result types are related to which test cases. All of these related results are "positives". The AutoScorer scores those results in "bad" functions and classes as true positives and results in "good" functions and classes as false positives.

#### 2.3.3.3 Manual Scoring

Some tools report findings for some test cases in locations outside of a function or class, such as in a typedef. In these cases, the AutoScorer could not determine whether a result should be scored as a True Positive or False Positive, and therefore it was scored as an "error". These "error" results were scored manually by the CAS as either a True Positive or False Positive.

#### 2.3.3.4 Automated Scoring Process (Pass 2)

Upon completion of the manual scoring, the AutoScorer was once again run on the tool results CSV. During this second run, the AutoScorer added rows to the CSV for False Negative results (test cases where the tool did not report a True Positive).

During analysis of the results, the CAS also identified a small set of test cases[d] that were invalid (did not exhibit the intended flaw). These 38 C and 36 Java test cases were excluded from analysis by scoring tool results for those test cases as "ITC" in this second pass of the AutoScorer.

#### 2.3.3.5 Summary of Result Types

At the end of the scoring process, each row in the result CSV file was assigned one of the result types in Table 3.

| Result Type | Explanation |
|---|---|
| True Positive (TP) | Tool correctly reported the flaw that was the target of the test case. |
| False Positive (FP) | Tool reported a flaw with a type that is the target of the test case, but the flaw was reported in non-flawed code. |
| False Negative (FN) | This row is not a tool result. It was added by the AutoScorer to indicate that the tool failed to report the target flaw in a test case. |
| Invalid Test Case (ITC) | The test case in which this result appeared contained an error. This row was not included in the data analysis. |
| (blank) | This row is a tool result where none of the result types above apply. More specifically, either:<br>• The tool result was not in a test case file<br>• The tool result type was not associated with the test case in which it was reported<br>This row was not included in the data analysis. |

Table 3 – Summary of Results Types

## 2.4 Data Analysis

With the study of each individual static code analysis tool complete, the data collected was analyzed to provide an overview of the results at a more abstract level. This analysis is the basis for this paper and enables an understanding of the strengths of each tool.

### 2.4.1 Weakness Classes

To help understand the areas in which a given tool excelled, similar test cases were grouped into Weakness Classes. Weakness classes are defined using CWE entries that contain similar weaknesses. Since each test case is associated with the CWE entry in its name, each test case is contained in a Weakness Class.

Note that buffer handling errors were only represented in C/C++ test cases. The Miscellaneous Weakness Class was used to hold a collection of individual weaknesses that did not fit into the other twelve classes. Therefore, the weaknesses in the Miscellaneous Weakness Class did not have a common theme.

For example, Stack-based Buffer Overflow (CWE-121) and Heap-based Buffer Overflow (CWE-122) were both placed in the Buffer Handling Weakness Class. Therefore, all of the test cases associated with CWE entries 121 and 122 were mapped to the Buffer Handling Weakness Class. Table 4 provides a summary list of Weakness Classes used in this study, along with an example weakness and the number of test cases in that Weakness Class for each language family.

| Weakness Class | Example Weakness (CWE Entry) | C/C++ Test Cases | Java Test Cases |
|---|---|---|---|
| Authentication and Access Control | CWE-620: Unverified Password Change | 604 | 422 |
| Buffer Handling | CWE-121: Stack-based Buffer Overflow | 11,386 | - |
| Code Quality | CWE-561: Dead Code | 440 | 410 |
| Control Flow Management | CWE-362: Race Condition | 598 | 527 |
| Encryption and Randomness | CWE-328: Reversible One-Way Hash | 298 | 950 |
| Error Handling | CWE-252: Unchecked Return Value | 2,790 | 437 |
| File Handling | CWE-23: Relative Path Traversal | 2,520 | 718 |
| Information Leaks | CWE-534: Information Leak Through Debug Log Files | 283 | 468 |
| Initialization and Shutdown | CWE-415: Double Free | 9,894 | 450 |
| Injection | CWE-89: SQL Injection | 6,882 | 5,970 |
| Miscellaneous | CWE-480: Use of Incorrect Operator | 2,304 | 222 |
| Number Handling | CWE-369: Divide by Zero | 6,017 | 2,802 |
| Pointer and Reference Handling | CWE-476: Null Pointer Dereference | 1,308 | 425 |

Table 4 – Weakness Classes

The following sections provide a brief description of the Weakness Classes used in the 2010 tool study.

#### 2.4.1.1 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 the tools' ability to check whether or not the source code is preventing unauthorized access to the system.

#### 2.4.1.2 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

[d] These invalid test cases were fixed prior to the public release of the test cases as the Juliet Test Suites.

the tools' ability to find buffer access violations in the source code.

#### 2.4.1.3 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 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 CWE, 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 a suspicious comment (CWE-546).

#### 2.4.1.4 Control Flow Management

Control flow management deals with timing and synchronization issues that can cause unexpected results when the code 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.

#### 2.4.1.5 Encryption and Randomness

Encryption is used to provide data confidentiality. However, if the wrong or a weak encryption algorithm is used an attacker may be able to covert 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.

#### 2.4.1.6 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 the tools' ability to check for proper error handling within the source code.

#### 2.4.1.7 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 the tools' ability to check for proper file handling within the source code.

#### 2.4.1.8 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 the tools' ability to check for information leaks within the source code.

#### 2.4.1.9 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 the tools' ability to check for proper initialization and shutdown of resources in the source code.

#### 2.4.1.10 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. 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.

#### 2.4.1.11 Miscellaneous

The weaknesses in this class do not fit into the previously detailed Weakness Classes. An example would be a logic/time bomb. An attacker or devious developer can place code into the application that will cause the program to crash at a certain point in time or when a certain logical condition is met. Although this is a serious flaw, it does not fit into the other Weakness Classes.

#### 2.4.1.12 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.

#### 2.4.1.13 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 (NPD). 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.

### 2.4.2 Precision and Recall

One set of analysis performed on the test case results calculated the Precision and Recall 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 Precision and Recall in greater detail.

#### 2.4.2.1 Precision

In the context on this report, 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 study, Precision describes how well does a tool identifies flaws, whereas accuracy describes how well does 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 reported 40 issues (false positives and true positives), of which only 10 were real flaws (true positives), would have a Precision of 10 out of 40, or 0.25.

Precision helps 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 reported issues that are real flaws on the test cases. That is, it did not report any false positives. Conversely, a tool that has a Precision of 0 always reported issues incorrectly. That is, it only reported false positives.

When calculating the Precision values in this study, duplicate true positive values were ignored. That is, if a tool reported two or more true positives on the same test case, the Precision is calculated as if the tool reported only one true positive on that test case. Duplicate false positive results are included in the calculation, however.

#### 2.4.2.2 Recall

The Recall metric (also known as "sensitivity" or "soundness") represents the fraction of real flaws that were reported by a tool. Recall is defined as the number of real flaws that a tool reported (true positives), divided by the total number of real flaws – reported or unreported – that existed 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 reported 10 real flaws in a piece of code that contained 20 flaws would have a Recall 10 out of 20, or 0.5.

A high Recall means that the tool correctly identified a high number of the target weaknesses within the test cases. For example, a tool that achieves a Recall of 1 reported every flaw in the test cases. That is, it had no false negatives. In contrast, a tool that has a Recall of 0 reported none of the real flaws. That is, it had a high false negative rate.

Like with Precision, duplicate true positive values were ignored when calculating Recall in this study. That is, if a tool reported two or more true positives on the same test case, the Recall was calculated as if the tool reported only one true positive on that test case.

#### 2.4.2.3 F-score

In addition to the Precision and Recall metrics, an F-score was calculated by taking the harmonic mean of the Precision and Recall values. Since a harmonic mean is a type of average, the value for F-score will always be between the values for Precision and Recall (unless Precision and Recall 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 were found (true positives) and how much

noise (false positives) was produced. An F-score is computed using the following formula:

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

Table 5 shows example F-scores for some example Precision and Recall values:

| Weakness Class | Precision | F-score | Recall |
|---|---|---|---|
| Weakness Class A | .80 | .80 | .80 |
| Weakness Class B | .80 | .69 | .60 |
| Weakness Class C | .80 | .53 | .40 |
| Weakness Class D | .80 | .32 | .20 |
| Weakness Class E | .80 | .18 | .10 |
| Weakness Class F | .20 | .32 | .80 |
| Weakness Class G | .20 | .20 | .20 |
| Weakness Class H | .50 | .50 | .50 |

Table 5 – Example F-score Values

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 would not be considered stronger than a tool that is average in both. See Weakness Class F and Weakness Class H in Table 5 above for an example of this point.

Note that this report uses equal weighting for both Precision and Recall when calculating the F-score. 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).

#### 2.4.2.4 Precision-Recall Tables and Averages

To summarize each tool's Precision, Recall, and F-score on the test cases, a table was produced to show how the tool performed regarding each metric. However, when interpreted in isolation, each metric can be deceptive in that it does not reflect how a tool performed 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 0.45 is a good value or not. If every other tool had a Precision value of 0.20, then a value of 0.45 would suggest that the tool outperformed its peers. On the other hand, if every other tool had a Precision of 0.80, then a value of 0.45 would suggest that the tool underperformed on this metric.

To help understand the Precision, Recall, and F-score results for a tool, each metric was compared against the average of the all tools. Note that when a tool did not report any findings in a given Weakness Class, it was excluded from the calculation of the average for that Weakness Class. This kept the average focused on the tools that demonstrated a capability for finding flaws within the given Weakness Class. The average allowed the CAS to relate a single tool's metric in the context of the larger group of tools that were capable of finding the same kind of weaknesses. The average also effectively anonymizes the performance of individual tools.

If the tool had a higher value than the average, a small green triangle pointing up was used. For values below average, a small red triangle pointing down was used. If the value was within 0.05 of the average, then no icon was used and the tool results were 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 6 – Precision-Recall Results for SampleTool by Weakness Class

In this example, Table 6 shows that SampleTool was more than 0.05 above the average with respect to Precision, Recall, and F-score for Weakness Class C. Its performance was mixed in Weakness Class D, with below-average Precision and above-average Recall. Within Weakness Class E, SampleTool had 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 had to produce findings. In general, when there are more opportunities, one can have more statistical confidence in the metric.

#### 2.4.2.5 Precision-Recall Graphs

Precision-Recall Graphs are used to show the tool results for both the Recall and Precision metrics. Examining just a single metric does not give the whole picture and obscures details that were important in understanding a tool's strengths. For example, just looking at Recall would tell the analyst how many issues are found, but

these issues could be hidden in a sea of false positives, making it extremely time-consuming to interpret the results. The Recall metric alone did not give the analyst this perspective. By examining both the Recall and Precision values on the same graph, the analyst could get a better picture of the tool's overall strengths.

Figure 1 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. The closer the point is to the top right, the stronger the tool is the given area. The background of the graph is shaded according to the F-Score at each point. An F-Score of 1 is indicated by a white background and an F-Score of 0 is indicated by a dark grey background.

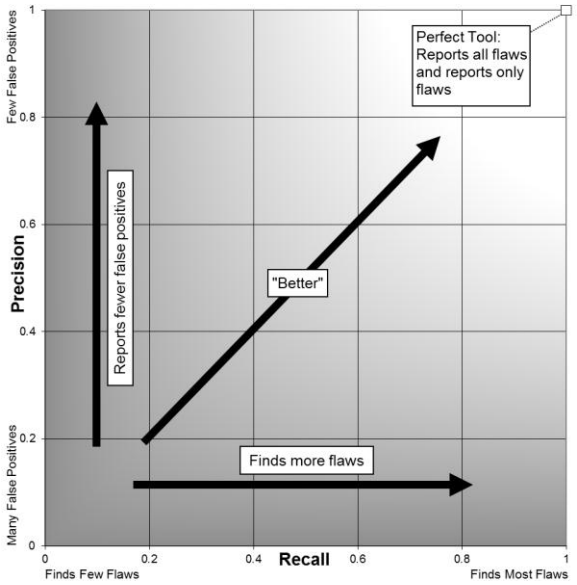


Figure 1 – Example Precision-Recall Graph

When graphing results for different tools on the same Weakness Class, square markers (in white) are used on the Precision-Recall Graph to represent the Precision and Recall for each tool and a black circle is used to represent the average Precision and Recall for all tools that produced findings for the Weakness Class

When used in to graph results for a single tool across different Weakness Classes, a Precision-Recall Graph shows two distinct points for each Weakness Class. A square marker (in white) represents the tool's Precision and Recall for the specified Weakness Class. Another point, shown as a black circle, represents the average Precision and Recall 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. Further explanations are provided next.

In the example in Figure 2, the graph shows that the SampleTool is strong when focusing on Weakness Class C. Both Precision and Recall for the tool are above average, and the line moves from the black dot toward the upper right. SampleTool is not strong related to Weakness Class A. Both metrics are below average, and the line moves from the black dot toward the lower left.

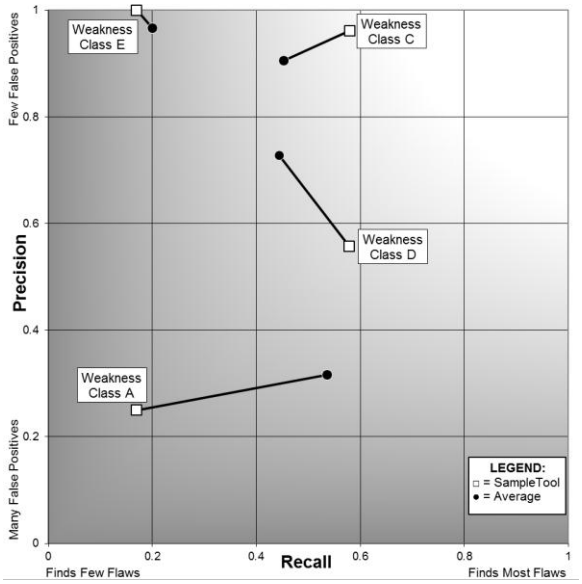


Figure 2 – Precision-Recall Graph for SampleTool by Weakness Class

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.

### 2.4.3 Discriminations and Discrimination Rate

Another analysis performed on the test case results looked for areas where a tool showed it could discriminate between flaws and non-flaws. This section describes this analysis in greater detail.

The purpose of this analysis was 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 Weakness Class, 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.

This reflects a limitation in the test cases that there are typically only one or two non-flaws for every flaw. Noisy tools can have good Recall and a respectable Precision on the test cases because there are far fewer opportunities for false positives in the test cases than there would be in real software.

#### 2.4.3.1 Discriminations

To address the issue described above, the CAS defined 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" would not get credit for a Discrimination on the test case because while it correctly reported the flaw, it also incorrectly reported a false positive.

Discriminations must be determined for each test case individually. The number of Discriminations in a Weakness Class (or other 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).

#### 2.4.3.2 Discrimination Rate

Over a Weakness Class (or other set of test cases), the Discrimination Rate is the fraction of test cases where the tool reported a Discrimination. That is:

$$DiscriminationRate = \frac{\#Discriminations}{\#Flaws}$$

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 will always be 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 if it reported false positive(s). Every test case where the tool reported 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).

#### 2.4.3.3 Discrimination Results Tables

A tool's Discriminations and Discrimination Rates on each Weakness Class are shown in a table like Table 7.

| | Sample Size | Tool Results | |
|---|---|---|---|
| **Weakness Class** | **# of Flaws** | **Discrimin-ations** | **Disc. Rate** |
| Weakness Class A | 511 | 50 | 0.10 |
| Weakness Class B | 953 | 0 | 0.00 |
| Weakness Class C | 433 | 234 | 0.54 |
| Weakness Class D | 720 | 150 | 0.21 |
| Weakness Class E | 460 | 15 | 0.03 |

Table 7 – Discrimination Results for SampleTool by Weakness Class

#### 2.4.3.4 Discrimination Rate Graphs

Discrimination Rate Graphs like Figure 3 are used to show the Discriminations Rates for a 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.

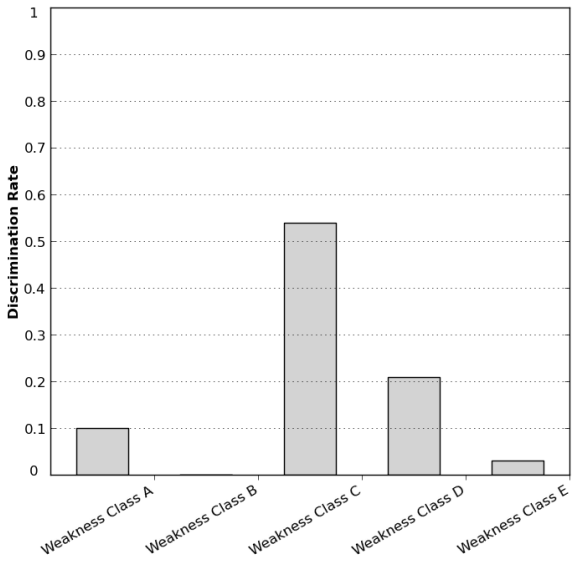


Figure 3 – Discrimination Graph for SampleTool by Weakness Class

# Section 3: Conclusion

## 3.1 Study Assessment

Overall, the CAS and its customers found the 2010 study of static analysis tools to be a success. This study accomplished the purpose described in Section 1.3 in that it determined the capabilities of commercial and open source static analysis tools for C/C++ and Java and provided objective information to organizations looking to purchase, deploy, or make the best use of static analysis tools. By identifying the strengths of the tools, this study also determined how tools could be combined to provide a more thorough analysis of software by using strong tools in each area analyzed.

## 3.2 Suggestions for Further Study

There are several follow on projects could be undertaken based on this study:

- Tool vendors are continually improving their products and releasing new versions. In addition, new tools may enter the static analysis market in the future. Therefore, additional studies of new tools and tool versions will be required as the results of this study are only valid for the specific tools and versions tested.
- As described in Section 2.3.2, this study was conducted using the default installation and configuration of each tool. It may be possible to obtain improved results from the tools by adjusting their settings, either through testing and experimentation or through interactions with the tool vendors or developers.
- As described in Section 2.1.4, there are limitations to the scope of test cases used in this study. Future studies could expand the scope of test cases to cover additional languages, flaw types, data flows, and/or control flows.
- As described in Section 2.1.2, there are limitations to the results in this study due to the fact that the relative frequencies of flaws and non-flaws in the test cases likely vary from those frequencies in natural software. Further study on natural code or natural code with intentionally introduced vulnerabilities may provide results that more accurately predict results when using the tools outside of controlled studies.