NIST Special Publication 500-326

SATE V Report: Ten Years of Static Analysis Tool Expositions

Aurelien Delaitre
Bertrand Stivalet
Paul E. Black
Vadim Okun
Athos Ribeiro
Terry S. Cohen

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.500-326

National Institute of Standards and Technology
U.S. Department of Commerce

NIST Special Publication 500-326

SATE V Report: Ten Years of Static Analysis Tool Expositions

Aurelien Delaitre
Prometheus Computing LLC

Bertrand Stivalet
Paul E. Black
Vadim Okun
Athos Ribeiro
Terry S. Cohen
Information Technology Laboratory
Software and Systems Division

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.500-326

October 2018

U.S. Department of Commerce
Wilbur L. Ross, Jr., Secretary

National Institute of Standards and Technology
Walter Copan, NIST Director and Under Secretary of Commerce for Standards and Technology

National Institute of Standards and Technology Special Publication 500-326
Natl. Inst. Stand. Technol. Spec. Publ. 500-326, 180 pages (October 2018)
CODEN: NSPUE2

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.500-326

Abstract

Software assurance has been the focus of the National Institute of Standards and Technology (NIST) Software Assurance Metrics and Tool Evaluation (SAMATE) team for many years. The Static Analysis Tool Exposition (SATE) is one of the team's prominent projects to advance research in and adoption of static analysis, one of several software assurance methods. This report describes our approach and methodology. It then presents and discusses the results collected from the fifth edition of SATE.

Overall, the goal of SATE was not to rank static analysis tools, but rather to propose a methodology to assess tool effectiveness. Others can use this methodology to determine which tools fit their requirements. The results in this report are presented as examples and used as a basis for further discussion.

Our methodology relies on metrics, such as recall and precision, to determine tool effectiveness. To calculate these metrics, we designed test cases that exhibit certain characteristics. Most of the test cases were large pieces of software with cybersecurity implications. Fourteen participants ran their tools on these test cases and sent us a report of their findings. We analyzed these reports and calculated the metrics to assess the tools' effectiveness.

Although a few results remained inconclusive, many key elements could be inferred based on our methodology, test cases, and analysis. In particular, we were able to estimate the propensity of tools to find critical vulnerabilities in real software, the degree of noise they produced, and the type of weaknesses they were able to find. Some shortcomings in the methodology and test cases were also identified and solutions proposed for the next edition of SATE.

Key words

Security Weaknesses; Software Assurance; Static Analysis Tools; Vulnerability.

Caution on Interpreting and Using the SATE Data

SATE V, as well as its predecessors, taught us many valuable lessons. Most importantly, our analysis should NOT be used as a basis for rating or choosing tools; this was never the goal.

There is no single metric or set of metrics that is considered by the research community to indicate or quantify all aspects of tool performance. We caution readers not to apply unjustified metrics based on the SATE data.

Due to the nature and variety of security weaknesses, defining clear and comprehensive analysis criteria is difficult. While the analysis criteria have been much improved since the first SATE, further refinements are necessary.

The test data and analysis procedure employed have limitations and might not indicate how these tools perform in practice. The results may not generalize to other software because the choice of test cases, as well as the size of test cases, can greatly influence tool performance. Also, we analyzed only a small subset of tool warnings.

The procedure that we used for finding CVE locations in the CVE-selected test cases and selecting related tool warnings has limitations, so the results may not indicate tools' actual abilities to find important security weaknesses.

Synthetic test cases are much smaller and less complex than production software. Weaknesses may not occur with the same frequency in production software. Additionally, for every synthetic test case with a weakness, there is one test case without a weakness, whereas, in practice, sites with weaknesses appear much less frequently than sites without weaknesses. Due to these limitations, tool results, including false positive rates, on synthetic test cases may differ from results on production software.

The tools were used differently in this exposition from their typical use. We analyzed tool warnings for correctness and looked for related warnings from other tools. Developers, on the other hand, use tools to determine what changes need to be made to software. Auditors look for evidence of assurance. Also, in practice, users write special rules, suppress false positives, and write code in certain ways to minimize tool warnings.

We did not consider the tools' user interfaces, integration with the development environment, and many other aspects of the tools, which are important for a user to efficiently and correctly understand a weakness report.

Teams ran their tools against the test sets in June through September 2013. The tools continue to progress rapidly, so some observations from the SATE data may already be out of date.

Because of the stated limitations, SATE should not be interpreted as a tool testing exercise. The results should not be used to make conclusions regarding which tools are best for a given application or the general benefit of using static analysis tools.

Table of Contents

List of Tables

List of Figures

1. Introduction

Nowadays software is ubiquitous. Most critical infrastructures heavily rely on software; we use it to control air traffic, computerize self-driving vehicles, and manage power plants. These successes depend upon our trust in software, noting that the more elaborate the system, the more complex and inevitable the defects.

Software assurance is a set of methods and processes to prevent, mitigate or remove vulnerabilities and ensure that the software functions as intended. Multiple techniques and tools have been used for software assurance [1]. One technique is static analysis, which examines software for weaknesses without executing it [2]. As sophisticated static security analysis tools were beginning to appear in the mid-2000s, users required a better understanding of their effectiveness. The National Institute of Standards and Technology (NIST) Software Assurance Metrics and Tool Evaluation (SAMATE) project has been evaluating these tools. The SAMATE team initially built a specification listing weakness classes which should be reported by static analysis tools [3]. Bill Pugh, Professor Emeritus at the University of Maryland and author of the static analysis tool Findbugs, proposed following the NIST Text REtrieval Conference (TREC) approach as a more practical way for testing tools [4, 5]. Instead of directing toolmakers to find specific weakness classes, we shifted our focus to determining what weaknesses existed in real software and could be found by tools.

In 2008, we initiated the first large-scale public event, inviting toolmakers to demonstrate the use of their tools. We labeled it the Static Analysis Tool Exposition (SATE) and refined it over five instances [6–9]. SATE is designed to advance research in static analysis tools that find security-relevant weaknesses in source code.

Definition and classification of such security weaknesses in software are necessary to communicate and analyze security findings. While many classifications have been proposed, *Common Weakness Enumeration (CWE)* is the most prominent effort [10].

We explain the SATE procedure, including the use of CWEs, and present the results of SATE V in this report.

1.1. Goals

A number of studies have compared static analysis tools [11–17]. SATE chose to encourage participation by creating a neutral space for sharing, rather than competing, to advance research in static analysis tools. This broader participation brings more results, on which we build and assess stronger metrics. We use these indicators to measure the strengths of tools and understand how to leverage their value. In addition, we identify their shortcomings and the challenges they face.

Users want to understand how effective tools are in meeting their requirements. The SATE metrics provide assessments of tools’ features. Such features include weakness types, the accuracy in detecting such weaknesses, and the rate of missing weaknesses in source code.

As a by-product, the exposition provides participating toolmakers with quality feedback, enabling them to assess their strengths and weaknesses. The results produced by their tools are partially reviewed and rated by experts. In one type of analysis, the tool warnings are matched to real vulnerabilities from the *Common Vulnerabilities and Exposures* (CVE) database [18].

Finally, demonstrating the use of tools on production software fosters their adoption by the user community. In fact, several toolmakers informally reported that their current and prospective customers demanded that they participate in SATE.

1.2. Scope

Due to high cost of security incidents, SATE focuses on tools capable of finding security defects. Although its parent project, SAMATE, considers all types of software assurance tools, SATE is only concerned with tools that statically analyze software, i.e., without executing the code.

1.3. Target Audience

The target audiences for this report are static analysis toolmakers, security researchers, and tool users.

1.4. Terminology

This report uses the concepts defined in Table 1.

Table 1. Glossary of Terms.

Term	Definition
Weakness, flaw, defect, bug	Defect in a system that may (or may not) lead to a vulnerability.
Vulnerability	A weakness in system security requirements, design, implementation, or operation, that could be accidentally triggered or intentionally exploited and result in a violation of the system's security policy [19].
Site	Conceptual place in a program where an operation is performed.
Finding, claim	A definitive statement provided by a tool about a site, e.g., the presence or absence of a weakness.
Warning	Claim reporting the presence of a potential weakness.
Report	Collection of warnings reported by a tool on a specific test case.
Location	A representation of a site, e.g., by file name and line number in source code.
Complexity	Code construct encapsulating a site, making the latter more or less difficult to analyze.
Control flow complexity	Amount of control flow statements, e.g., conditionals, loops, and function calls, that make a program more or less difficult to analyze.
Data flow complexity	Amount of data flow transfers, e.g., copying data, passing parameters to a function, and validation, that make a program more or less difficult to analyze.
Synthetic code	Artificial code generated and documented automatically.

Term	Definition
True positive (TP)	Flawed code reported correctly by a tool.
True negative (TN)	Non-flawed code not reported by a tool.
False positive (FP)	Non-flawed code reported by a tool as flawed.
False negative (FN)	Flawed code not reported by a tool.
Ground truth	Knowledge of all weaknesses in a test case, including their location in code and weakness class.
Track	An area of focus, such as a programming language (C/C++, Java, and PHP[1]), sometimes collectively called "classic tracks," or methodology (Ockham Criteria).
Good code, fixed code, non-buggy code	Code that should not contain any weakness.
Bad code, flawed code, buggy code	Code that contains at least one weakness.

1.5. Metrics

Since we have assembled a large set of test cases, we need to establish an objective way of measuring the tools' outputs. The following metrics address some basic questions:

- **Coverage** – What kinds of weaknesses can a tool find?

Coverage is determined by the types of weaknesses found by a tool. It is measured by the number of unique weakness types reported over the total number of weakness types tested.

- **Recall** – What proportion of weaknesses can a tool find?

Recall is defined by the number of correct findings by a tool compared with the total number of weaknesses present in the code. It is calculated by dividing the number of *True Positives (TP)* by the total number of weaknesses, i.e., the sum of the number of *True Positives (TP)* and the number of *False Negatives (FN)*.

$$Recall = \frac{TP}{(TP + FN)} \tag{1}$$

- **Applicable Recall** – What proportion of *covered* weaknesses can a tool find?

Applicable Recall is recall reduced to the types of weaknesses a tool can find. It is calculated by dividing the number of *True Positives (TP)* by the number of weaknesses covered by a tool. We consider *False Negatives (FN)* only if they belong to a weakness class the tool supports, i.e., *Applicable False Negatives (App.FN)*. In other words, a tool's performance is not penalized if it does not report weaknesses for which it does not search.

$$App.Recall = \frac{TP}{(TP + App.FN)} \tag{2}$$

[1] PHP: Hypertext Preprocessor, a recursive acronym for PHP, an open-source scripting language (http://php.net/manual/en/intro-whatis.php)

- **Precision** – How much can I trust a tool?

Precision is the proportion of correct warnings produced by a tool and is calculated by dividing the number of *True Positives (TP)* by the total number of *warnings*. The total number of warnings is the sum of the number of *True Positives (TP)* and the number of *False Positives (FP)*.

$$Precision = \frac{TP}{(TP + FP)} \tag{3}$$

Note that we calculate precision differently for production software. We call this "useful precision," as described in Sec. 3.1.3.2. Also, precision for synthetic test cases is based on 50 % prevalence of weaknesses, as described in Sec. 3.3.5.4.

- **Discrimination Rate** – How smart is a tool?

Buggy and good code often look similar. It is useful to determine whether the tools can differentiate between the two. Although precision captures that aspect of tool efficiency, it is relevant only when good sites dominate buggy sites. When there is parity in the number of good and bad sites, e.g., in some synthetic test suites, a tool could indiscriminately flag both good and bad sites as flawed and still achieve a precision of 50 %. *Discrimination*, however, recognizes a true positive on a specific flawed test case only if a tool did not report a false positive on the corresponding fixed test case [11]. For each weakness instance, a tool is assigned a discrimination of 1 if the tool reports a weakness for a bad site but not for the corresponding good site; otherwise it is assigned a discrimination of 0. Over a set of test cases, the *Discrimination Rate* is the number of discriminations divided by the total number of weakness instances. A tool that flags all sites (good and bad) indiscriminately would achieve a discrimination rate of 0 %.

- **Overlap** – Can the findings be confirmed by other tools?

Overlap represents the proportion of weaknesses found by more than one tool. This metric identifies which tools behave similarly and which weaknesses are easy or difficult for tools to find. The use of multiple tools would find more weaknesses (higher recall), whereas the use of independent tools would provide a better confidence in the common warnings' accuracy.

1.6. Types of Test Cases

The only way to understand how static analysis tools behave in any given situation is to run them on all existing software and analyze their outputs. This would be a colossal effort, so we should start with a few examples. But which examples should we choose as our test cases?

We want to generalize the knowledge acquired by running the tools on our test cases. Therefore, we must select programs that are representative of real, existing software. For example, their development should follow industry practices. Their size should align with similar software. Their programming language should be widely used for their purpose.

We also need a sufficient number and diversity of weaknesses in code to achieve statistical significance. The results must demonstrate all the capabilities of the tools and in different instances. If some features remain unexposed, the generalization would be inaccurate.

Lastly, we must know all the defect locations in the test cases, i.e., the ground truth. This enables faster tool warning evaluations and, more importantly, the identification of undetected weaknesses.

However, thoroughly analyzing large production software to find all defects is impractical. Consequently, no candidate test case exhibits the three ideal characteristics: 1) representative of real, existing code, 2) large amounts of test data to yield statistical significance, and 3) ground truth. But software showing two of the three characteristics exists (Fig. 1). There are three possible combinations of two features, corresponding to three types of test cases.

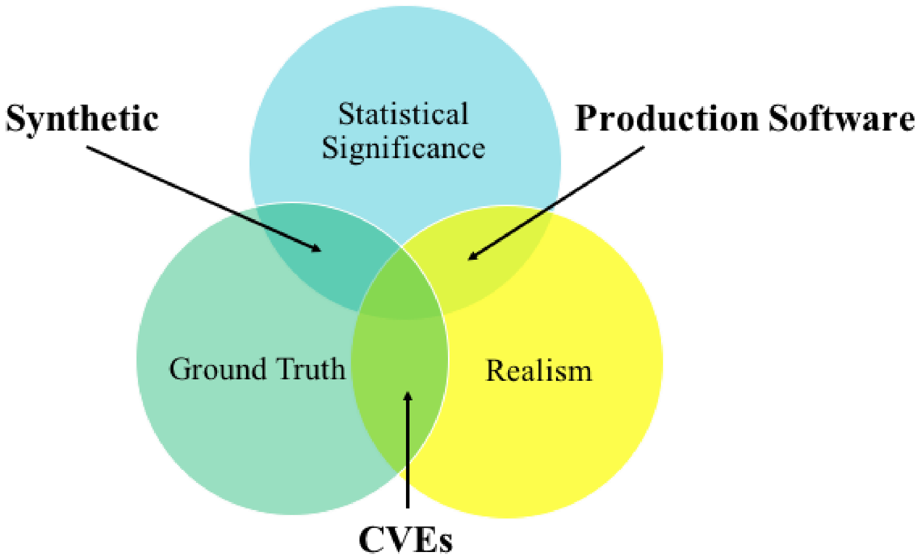

Figure 1. Types of Test Cases.

The first type of test cases is production software. It is large enough for statistical significance and is representative of real-world software. However, the defects it contains are only partially known. Section 3.1 describes the procedure and results obtained from this type of test case. We refer to these test cases as *Production Software* test cases.

Publicly reported vulnerabilities from the Common Vulnerabilities and Exposures (CVE) database [18] form a prime source of known defects in production software. Unfortunately, they are still too few to achieve statistical significance. In Sec. 3.2, we discuss the performance of tools in finding these genuine vulnerabilities. We refer to these test cases as *Software with CVEs* test cases.

Computer-assisted code generation provides us with large sets of test cases, containing known weaknesses of many types. Because these programs are usually short and artificially express a pre-determined flaw, they are not representative of real-world software. Section 3.3 discusses the performance of tools on these *Synthetic Test Cases*.

To calculate the metrics described in Sec. 1.5 we must select appropriate test cases. Table 2 summarizes the applicability of the metrics to the three types of test cases. Label *Applicable* means that the metric can be calculated, *Limited* states that there are some limitations with the calculation, and *N/A* (not applicable) means that the calculation is not possible.

Recall cannot be calculated on the *Production Software* test cases, because we do not know all the weaknesses contained in the applications. Discrimination rate requires a flawless version of a test case to check whether tools report the same warnings in the corresponding vulnerable version, but none exist in *Production Software*.

While weakness categories reported by a tool give an idea of tool's coverage, the coverage calculation is limited to the types of weaknesses present in *Production Software*. Similarly, coverage for the *Software with CVEs* is limited to the types of weaknesses represented by the CVEs.

The precision of the tools cannot be measured using the *Software with CVEs* test cases, because we have too few CVEs per program to achieve statistical significance.

While discrimination can usually be determined for the *Software with CVEs*, there are practical limitations. For example, the buggy code is sometimes removed in the fixed version, or heavily modified.

We can apply all the metrics to the *Synthetic Test Cases*, but we should keep in mind that these test cases were automatically generated. The tools' behavior on artificial code could differ from their behavior on production code.

Table 2. Mapping Metrics to Test Case Types.

Metric	Production Software	Software with CVEs	Synthetic Test Cases
Coverage	Limited	Limited	Applicable
Recall	N/A	Applicable	Applicable
Precision	Applicable	N/A	Applicable
Discrimination	N/A	Limited	Applicable
Overlap	Applicable	Applicable	Applicable

1.7. Related Work

In Sec. 1.6, we described the three properties of an ideal test suite: realism, ground truth, and statistical significance. In this section, we review some relevant test suites and their use in evaluating static analysis tools with respect to the desired properties.

Synthetic test suites satisfy the requirements of having ground truth and statistical significance. Kendra Kratkiewicz and Richard Lippmann [12] developed a comprehensive taxonomy of buffer overflows and created 291 test cases, comprised of small C programs, to evaluate tools for detecting buffer overflows. Each test case has three vulnerable versions with buffer overflows just outside, moderately outside, and far outside the buffer, in addition to a fourth, fixed, version. Kratkiewicz's taxonomy [12] lists different attributes, or code complexities, including aliasing, control flow, and loops, which may complicate analysis by the tools.

The largest synthetic test suite in the NIST Software Assurance Reference Dataset (SARD) [20] was created by the U.S. National Security Agency's (NSA) Center for Assured Software (CAS). Juliet 1.0 consists of about 60 000 synthetic test cases, covering 177 CWEs and a wide range of code complexities [11]. CAS ran nine tools on the test suite and found that static analysis tools differed significantly with respect to precision and recall. Also, tools' precision and recall ranking varied for different weaknesses. CAS concluded that sophisticated use of multiple tools would increase the rate of finding weaknesses and decrease the false positive rate. A newer version of the test suite, Juliet 1.2, correcting several errors and covering a wider range of CWEs and code constructs, was used in SATE V. (Since then Juliet 1.3 has been released. It has additional coverage and corrects many errors in version 1.2 [21].)

Wagner and Sametinger [22] evaluated several source code analysis tools on the Juliet test suite. Most of these tools were free and open source tools. Tools detected a minority of weaknesses only. Using a security rule set significantly improved the performance of one of the tools, PMD. Testing tools on a synthetic test suite provides an overview of their capabilities. However, these results may differ from the results obtained when running these tools on real-world software.

Evaluating tools on production software has the advantages of realism and statistical significance. Rutar et. al. [17] ran five static analysis tools on five open source Java programs, including Apache Tomcat, of varying size and functionality. Due to many tool warnings, Rutar et al. did not categorize every false positive and false negative reported by the tools. Instead, the tool outputs were cross-checked with each other. Additionally, a subset of warnings was examined manually. SATE also analyzed a subset of tool warnings for production software. One of the conclusions of Rutar et al. was that there was little overlap among warnings from different tools. Another conclusion was that a meta-tool combining and cross-referencing output from multiple tools could be used to prioritize warnings.

Several tool evaluation studies identified ground truth in production software. The earliest such effort was by Zitser et al. [13]. At the time of their 2004 publication, sophisticated tools could not handle realistic software, so they extracted source code for model programs. They created fourteen small model programs from three popular, open source, Internet server programs (BIND, Sendmail, and WU-FTP), which contained publicly known, exploitable buffer overflows. The model programs had both vulnerable and patched source code. Complexity of the model programs related to the buffer overflows was similar to the real programs, while the size was much smaller. Now, many sophisticated tools can handle large software out of the box or with minimal configuration. The study analyzed different characteristics of buffer overflows and evaluated true positive rates, false positive rates, and discrimination counts of static analysis tools.

Walden et al. [23] measured the effect of code complexity on the quality of static analysis on open source software. Thirty-five format string vulnerabilities were selected, and both vulnerable and fixed versions of the software were analyzed. We took a similar approach with the CVE-selected test cases. Walden et al. concluded that detection rates of format string vulnerabilities decreased with an increase in code size or code complexity.

Kupsch and Miller [24] evaluated the effectiveness of static analysis tools by comparing their results with the results of an in-depth manual vulnerability assessment. Of the vulnerabilities found by manual assessment, the tools found simple implementation bugs, but did not find any vulnerabilities requiring a deep understanding of the code or design.

For SATE 2009, SATE 2010, and SATE IV, we used a similar approach [7–9]. Security experts performed time-limited analyses of some of the test cases to identify the most important weaknesses. We evaluated the tool outputs to correlate warnings with these manual findings.

The Intelligence Advanced Research Projects Activity (IARPA) attempted to combine all three properties of an ideal test suite in its Securely Taking On New Executable Software of Uncertain Provenance STONESOUP program [25, 26]. IARPA created 7770 test cases by injecting small code snippets, containing weaknesses, into sixteen open source base programs. Input/output pairs were also created as part of the test case generation process. Although the base programs were real-world software, the inserted code snippets, or cysts, were unrelated to the control and data flow of the base programs. The resulting weaknesses were not representative of bugs made by real programmers. Thus, further improvement is still needed to satisfy the property of realism.

IARPA STONESOUP and many of the test cases mentioned above are available from the SARD [20].

SATE and most of the above-mentioned studies analyze tool outputs on a selected set of programs. A different approach to studying tools is gathering software development data over a period of time. This takes into consideration additional factors, such as development and failure history. Zheng et. al [27] analyzed the effectiveness of static analysis tools by looking at test and customer-reported failures for three large-scale network service software systems. One of the conclusions in Ref. [27] was that static analysis tools are effective at identifying code-level defects.

1.8. Evolution of SATE

Test cases in SATE 2008 were production software only: three C and three Java open source programs [6]. We analyzed a subset of warnings, focusing on the high severity warnings. The large number of tool warnings and the lack of the ground truth complicated our analysis.

To address this problem in SATE 2009 [7] and the following SATEs, we randomly selected a subset of thirty warnings from each tool report, based on weakness category and severity. The selection procedure assigned higher weight to higher severity warnings. We then analyzed the selected warnings for correctness. We also searched for related warnings from other tools, which allowed us to study overlap of warnings between tools.

We found that a binary true/false positive verdict on tool warnings did not provide adequate resolution to communicate the relationship of the warning to the underlying weakness. We expanded the number of correctness categories to four in SATE 2009 [7] and five in SATE 2010 [8]: true security, true quality, true but insignificant, unknown, and false. At the same time, we improved the warning analysis criteria.

Also in SATE 2009, we asked security experts to perform a time-limited analysis of some of the test cases. The expert analysis identified both design and source code weaknesses, focusing on the latter weaknesses. The expert analysis combined multiple weaknesses with the same root cause. That is, the security experts did not look for every weakness instance, but instead identified one or more instances per root cause. Threat modeling was used to guide specific testing activities, including code review, automated analysis, penetration testing, and fuzzing. Tools were used to aid expert analysis, but tools were not the main source of manual findings. We then selected tool warnings related to findings by security experts. Expert analysis was also used in SATE 2010 and SATE IV.

In SATE 2010, we included an additional approach to this problem: CVE-selected test cases. The CVE-selected test cases are pairs of programs: an older vulnerable version with publicly reported vulnerabilities (CVEs) and a fixed version, i.e., a newer version where some or all of the CVEs were fixed. For the CVE-selected test cases, we focused on tool warnings that corresponded to the CVEs.

Overall, we used three methods to select tool warnings for analysis from natural (non-synthetic) software: 1) random selection, 2) selection of warnings related to manual findings by experts, and 3) selection of warnings related to CVEs.

We used three different degrees of association or relation: equivalent (same weakness category and location or path), strongly-related (same weakness category and similar path) or weakly-related (Weakness categories are similar; weakness paths have an important attribute, e.g., a filter location, in common). The degrees of association are described in detail in the SATE IV report [9].

In the first three SATEs, weakness categories used for matching tool warnings were based on weakness names assigned by tools. In SATE IV, we started using a more systematic approach, based upon groups of CWE IDs.

In SATE IV, we introduced a large number of synthetic test cases, called the Juliet 1.0 test suite, which contain precisely characterized weaknesses. Thus, warnings for these weaknesses were amenable to mechanical analysis.

In SATE V, we introduced the Ockham Criteria [28] to evaluate sound static analysis tools. Sound tools in theory never report incorrect findings. This and other changes introduced in SATE V are explained later.

Table 3 presents a summary of the evolution of SATE over its five editions.

Table 3. Evolution of SATEs.

SATE	2008	2009	2010	IV	V
Production	Yes	Yes	Yes	Yes	Yes
Expert Analysis[a]	No	Yes	Yes	Yes	No
CVEs	No	No	Yes	Yes	Yes
Synthetic	No	No	No	Yes	Yes
Ockham Criteria	No	No	No	No	Yes
Random Sampling	No	Yes	Yes	Yes	Yes
Matching Method	Warning Names	Warning Names	Warning Names	CWE Groups	CWE Groups
Warning Rating (Manual Analysis)	True, False, Unknown	True, False, Insignificant, Unknown	Security, Quality, False, Insignificant, Unknown	Security, Quality, False, Insignificant, Unknown	Security, Quality, False[b], Insignificant, Unknown

[a]Time-limited analysis, including threat modeling and white box penetration testing, conducted by third-party experts

[b]In SATE V, we used a different method for calculating precision for production software: True (security + quality), False (false + insignificant), and Unknown. We called this "useful precision" in Sec. 3.1.3.2.

2. Overall Procedure

SATE follows the TREC model [4] and is divided into tracks. Early SATEs had only C/C++ and Java tracks. PHP track was introduced in SATE IV (but it had no participants) and its use was continued in SATE V. These three languages represented most of the marketspace in 2014, according to TIOBE Software[2] [29]. Each track contains a set of test cases of each type (Sec. 1.6), i.e., production software containing CVEs and synthetic test cases (except for PHP). Toolmakers are free to participate in any track and to analyze any test case.

2.1. Changes Since SATE IV

SATE V brings four significant changes compared to SATE IV.

2.1.1. Confidentiality

Some toolmakers shared concerns about publicly releasing the detailed analysis of their reports. We decided to accommodate their unease and keep the data confidential to encourage participation. Teams, however, are free to publish their own results. The data from previous SATEs [30–33] remain in the public domain. We exhort everyone to use them in their studies.

2.1.2. Environment

In the past SATEs, participants spent a substantial amount of resources compiling test cases. We addressed this issue in SATE IV by pre-compiling these test cases in a virtual

[2] C# and Objective-C were other candidates, but, unfortunately, potential test cases remain sparse.

machine (VM). Participants simply needed to run their tools inside the VM. For SATE V, we expanded the use of VMs by having VMs hosted by the Software Assurance Marketplace (SWAMP) [34], a cloud computing platform providing software security testing as a service. Toolmakers installed and ran their tools inside a private VM, containing pre-loaded test cases and hosted within the SWAMP cloud. This partnership was very successful. The SWAMP support team greatly facilitated the participants' tasks.

2.1.3. Fairness

Several metrics, such as recall, precision, and discrimination rate, can be used for measuring tool performance. However, it was unfair to rate tools on all weakness types if they only covered a few specific types. Therefore, we asked each participant to file a Coverage Claims Representation (CCR) [35] to identify the weakness classes his tool detected. We introduced a new metric, applicable recall (Sec. 1.5), that measures recall only on the weakness types supported by each tool. By removing the coverage factor from the metrics, we provide a fairer and more precise measure of each tool's ability to find code defects.

2.1.4. Soundness

Until now, we had not differentiated between static analyzers. There are, however, several approaches to tackling the static analysis problem. The large test cases we use tend to favor general-purpose tools that use heuristics, but are impractical for sound static analyzer tools, which, in theory, never report incorrect findings. We recognize the latter by introducing the Ockham Criteria [28], a list of requirements to validate tool soundness.

2.2. Steps / Organization

SATE follows a 6-step procedure (Fig. 2):

1. Preparation: We (NIST researchers) select the test data, while toolmakers are invited to sign up.
2. Kickoff: Test cases are released, and each team starts its analysis in SWAMP.
3. Submission: Each team sends its tool's findings back to us.
4. Analysis: We analyze tool reports, using methods specific to each test case type.
5. Workshop: Teams, NIST researchers, and others from industry and academia gather to share their experiences.
6. Publication: We release the SATE report, summarizing SATE V results.

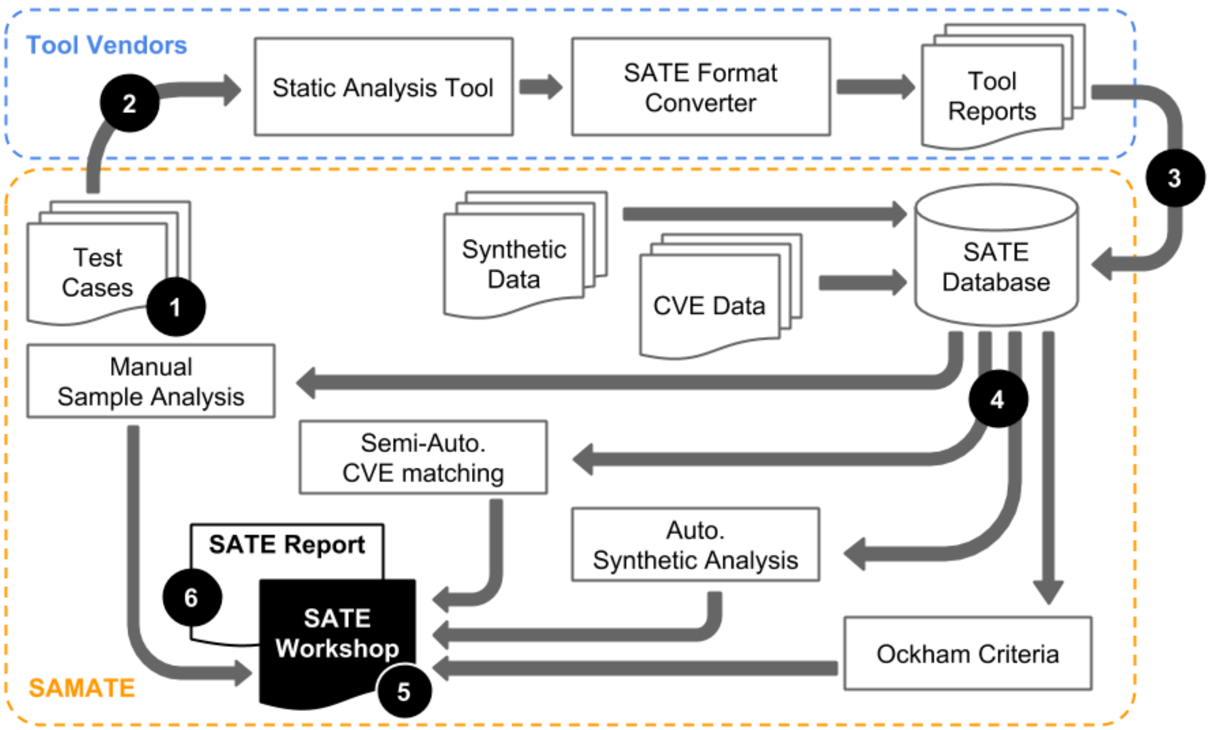

Figure 2. SATE Procedure.

2.3. Participation

Participation[3] in SATE V was the highest of all SATE events (2008, 2009, 2010, IV and V) with 14 unique participants (Table 4).

Table 4. Overall Participation per Track over SATEs.

SATE	C/C++	Java	PHP	Unique Participants
2008	4	7		9
2009	5	5		8
2010	8	4		10
IV	7	3	0	8
V	11	6	1	14

Our partnership with SWAMP generated additional interest from the toolmakers. Most teams took part in only one track. However, some toolmakers participated in two or three tracks (Tables 5 and 6).

[3] Certain commercial equipment, instruments, or materials are identified in this paper to foster understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

Table 5. Participation in the C/C++ Track over SATEs.

Tool	2008	2009	2010	IV	V
Clang					■
Coverity		■	■		■
Cppcheck			■	■	■
Flawfinder	■				
Fortify[a]	■				■
Frama-C					■
Grammatech	■	■	■	■	■
Klocwork		■			
LDRA		■	■	■	■
MARFCAT			■	■	
Monoidics				■	
Parasoft				■	■
Programing Research					■
Red Lizard			■	■	■
Sparrow			■		
Veracode	■	■	■		
Viva 64					■

[a] HP acquired Fortify in 2010.

Table 6. Participation in the Java Track over SATEs.

Tool	2008	2009	2010	IV	V
Armorize		■	■		
Aspect	■				
Buguroo				■	■
Checkmarx	■	■			
Coverity					■
FindBugs	■				■
Fortify[a]	■				■
HP DevInspect	■				
Klocwork		■			
MARFCAT			■	■	
Parasoft				■	■
PMD					■
SofCheck	■	■	■		
Veracode	■	■	■		

[a] HP acquired Fortify in 2010.

The PHP track was introduced in SATE IV, but it had no participants in SATE IV and only one participant in SATE V: HP Fortify.

The sound tool track (Ockham Criteria) also had a single participant: Frama-C. To differentiate from the sound tool track and to underline their historical precedence, the C/C++, Java, and PHP tracks are sometimes collectively called classic tracks in this report.

2.4. Data Anonymization

SATE is not a competition. To prevent endorsement and protect the intellectual property of toolmakers, aliases will be used to identify their products from this point on. Tools will be referred to as Tools A through R consistently throughout the report.

3. Procedure and Results for Classic Tracks

Our analysis used several dozen CWE categories (Appendix A). In this report, however, we present the results using the simpler Seven Pernicious Kingdoms (7PK) classification ("seven-plus-one", which includes Environment) [36]. Appendix B details the CWE distribution across the kingdoms. Note that both classifications contain overlap, i.e., CWEs can belong to several groups, and some categories contain many more CWEs than others.

Table 7 lists the original 7PK names and the abbreviated aliases we used in this report.

Table 7. Aliases for the Seven Pernicious Kingdoms Classes.

Original 7PK Names	Alias
Indicator of Poor Code Quality	Code Qual.
Improper Input Validation	Input Val.
Security Features	Sec. Feat.
Improper Fulfillment of API Contract ('API Abuse')	API
Time and State	T. & S.
Insufficient Encapsulation	Encap.
Error Handling	Error H.
Environment	Env.

3.1. Production Software

The original idea of SATE, as presented by Bill Pugh, was to run static analysis tools on large software[4] to observe their capabilities in conditions similar to real-world use. The

[4] Software with a large code base

toolmakers themselves would run their tools on code bases of our choice, in an approach combining expertise with impartiality.

We selected open source software for test cases. Selection was based upon their attack surface and their size, ranging from tens of thousands to several million lines of code.

Production software combines two of the three ideal test case characteristics: realism and statistical significance, due to the large number of warnings issued by tools. However, it lacks ground truth, since we do not know all of the bugs it contains. Precision, coverage, and overlap can be measured.

3.1.1. Test Sets

We carefully selected test cases, covering three different programming languages: C/C++, Java, and PHP (Table 8). We focused our attention on these test cases, because they are widely used, well maintained, and supported by a large open source community. They provide sufficient information to track down known vulnerabilities and perform our analyses.

For C/C++ and Java, we chose two common, open source programs per track. For the C/C++ track, we used Asterisk, an IP PBX platform[5], and Wireshark, a network traffic analyzer. Both of these programs were written in C. For the Java track, we used JSPWiki, a WikiWiki engine, and Openfire, a groupchat server. For the PHP track, we used WordPress, a blogging platform, which was an unused SATE IV test case. Each program included security-related aspects.

The test cases can be downloaded from the SARD [20].

Table 8. Test Sets.

Track	Test Case	Description	Version	Lines of Code	SARD Test Suite
C/C++	Asterisk[a]	IP PBX platform	10.2.0	> 500k	90
	Wireshark[b]	Network traffic analyzer	1.8.0	> 2M	94
Java	JSPWiki[c]	WikiWiki engine	2.5.124	> 60k	97
	Openfire[d]	Groupchat server	3.6.0	> 200k	98
PHP	WordPress[e]	Blogging platform	2.0	~ 24k	99

[a] http://www.asterisk.org/

[b] https://www.wireshark.org/

[c] https://jspwiki.apache.org/

[d] https://www.igniterealtime.org/projects/openfire/

[e] https://wordpress.com/

[5] IP PBX is a private branch exchange telephone switching system within an enterprise, which can be connected to traditional and voice over Internet protocol (VoIP) phones.

3.1.2. Procedure

Upon receipt of all SATE reports from the toolmakers, we randomly selected thirty warnings from each tool report, except for one tool report, which contained less than thirty warnings. We used the same sampling procedure as in earlier SATEs. It is described in detail in the SATE IV report [9, Sec. 2.8.1, Method 1]. Briefly, the selection was based on the types of weaknesses and severity ratings reported by each tool. Warnings of higher severity were selected more frequently than warnings of lower severity. Hence, the procedure produced a diverse sample that was heavy on more dangerous weaknesses.

We excluded from the selection process the warnings that referred exclusively to test code, parser generator code, and external header files.

In Sec. 2.3 we pointed out that there were eleven participants for the C/C++ track. However, one of them only submitted results for the synthetic test cases. In this section, we focus our analysis on ten reports for the C test cases, six for Java test cases, and one for PHP. In total, the reports contained about 500 000 warnings, of which we sampled 879 warnings for analysis. These numbers are detailed in Table 9 for C, Table 10 for Java, and Table 11 for PHP. When a tool was not run on a test case, the corresponding entry in Table 9 is 0. In particular, Tool F was run on synthetic test cases only.

Table 9. Warnings Reported by Tools per C Test Case.

Test Case	Asterisk	Wireshark
# Participants	8	9
Tool A	1482	13 829
Tool B	3283	1072
Tool C	63 288	76 360
Tool D	0	1729
Tool E	837	3362
Tool F	0	0
Tool G	2118	0
Tool H	12 357	14 739
Tool I	0	197 269
Tool J	2643	6873
Tool K	109	10
Total	**86 117**	**315 243**
# Selected	**240**	**249[a]**

[a] Tool K reported less than 30 warnings for Wireshark, nine of which were selected. One warning was omitted, because it was reported in a utility tool external to Wireshark.

Table 10. Warnings Reported by Tools per Java Test Case.

Test Case	Openfire	JSPWiki
# Participants	6	6
Tool L	13 568	2165
Tool M	87 631	19 734
Tool N	1144	753
Tool O	950	186
Tool P	1863	97
Tool Q	573	90
Total	**105 729**	**23 025**
# Selected	**180**	**180**

Table 11. Warnings Reported by Tools per PHP Test Case.

Test Case	WordPress
# Participants	1
Tool R	1321
Total	**1321**
# Selected	**30**

After sampling the 879 warnings, our team (NIST researchers) reviewed them manually for correctness. We rated each warning using the categories described in Table 12.

Table 12. Warning Rating Categories.

Label	Description
Security	A confirmed weakness related to security
Quality	A confirmed weakness unrelated to security, but requiring developers' attention
Insignificant	A true but insignificant claim
False	A false positive and invalid conclusion about the code
Unknown	The correctness of the claim could not be determined

Except for the *Unknown* category, the categories were ordered by relative importance (highest to lowest): *Security*, *Quality*, *Insignificant*, and *False*.

3.1.3. Results

3.1.3.1. Tool Warning Ratings

Table 13 presents the distribution of the sampled tool warnings across the evaluation categories. Overall, the C/C++ track appeared more difficult for tools to analyze than the Java and PHP tracks. The majority of analyzed warnings for the C/C++ track were *False* or *Insignificant* and only a minority of warnings were *Security*-rated. Several factors could be at play, including the fact that the test case size was significantly larger for the C/C++ track than for the other tracks. The number of tools in the exposition that analyzed C/C++ programs (sometimes referred to as C tools for brevity) was also greater, so the results from less advanced tools might have lowered the average ratings. Additionally, the Java and PHP test cases were all web applications, which tend to have a simpler architecture. On the Java and PHP tracks, over half the warnings were rated as *Security* or *Quality*.

Table 13. Analysis Results per Language.

Language	Security	Quality	Insignificant	False	Unknown
C	8 %	24 %	35 %	30 %	3 %
Java	23 %	37 %	17 %	22 %	1 %
PHP	30 %	20 %	17 %	33 %	0 %

On the C/C++ track, we discerned two main groups based on manual analysis (Table 14). The first group was comprised of Tools J, H, B, A, E, and G. Table 14 shows that these tools reported a significant proportion of *Security*- and *Quality*-rated warnings, but also a large number of *False* claims. The other group included Tools C, D and I. These tools did not report many, if any, *Security*-rated warnings. However, they reported a few *Quality*-rated warnings (17 % to 23 %). Most of their warnings were rated as *Insignificant* (67 % to 73 %). It should be noted, however, that Tools C, D, and I also reported a number of False claims (10 %, 3 %, and 13 %, respectively). In contrast, Tool K reported a significant number of *Quality*-rated warnings (74 %), with a few *Insignificant* and *False* claims (15 % and 10 %, respectively.) However, because the number of warnings produced by Tool K was very small, this result may be statistically insignificant.

Table 14. Analysis of Warning Results per Tool in the C/C++ Track.

Tool	Security	Quality	Insignificant	False	Unknown
Tool J	17 %	17 %	15 %	45 %	7 %
Tool H	15 %	10 %	25 %	48 %	2 %
Tool B	13 %	30 %	22 %	28 %	7 %
Tool A	8 %	28 %	20 %	42 %	2 %
Tool E	8 %	22 %	33 %	37 %	0 %
Tool G	7 %	3 %	30 %	47 %	13 %
Tool C	2 %	17 %	72 %	10 %	0 %
Tool D	0 %	23 %	73 %	3 %	0 %
Tool I	0 %	17 %	67 %	13 %	3 %
Tool K	0 %	74 %	15 %	10 %	0 %

On the Java track (Table 15), our analysis also revealed two main groups of tools. One group included Tools L and Q, which reported 58 % and 55 % *Security*-rated findings, respectively. The other group was comprised of Tools N, O and M. These tools mostly reported *Quality*-rated warnings (62 %, 65 %, and 79 %, respectively), with few (0 % to 5 %) *False* claims. One tool, Tool P, stood out with a large proportion of *False Positives* (70 %).

Table 15. Analysis of Warning Results per Tool in the Java Track.

Tool	Security	Quality	Insignificant	False	Unknown
Tool L	58 %	15 %	12 %	15 %	0 %
Tool Q	55 %	10 %	7 %	28 %	0 %
Tool N	13 %	62 %	12 %	8 %	5 %
Tool O	3 %	65 %	23 %	5 %	3 %
Tool M	0 %	79 %	14 %	7 %	0 %
Tool P	5 %	17 %	8 %	70 %	0 %

Only Tool R participated in the PHP track. Table 13 shows that 50 % of its claims were *Security*- or *Quality*-rated and 33 % were *False Positives*.

3.1.3.2. Useful Precision

We calculated the precision for each tool, using the formula in Sec.1.5, Eq. 3. We rated both *Quality*-rated and *Security*-rated warnings as true positives and both *Insignificant* and *False* warnings as false positives. Precision in this context strays from its original definition, because we counted *Insignificant* warnings as false positives, although they were true. We call it “useful precision”, as it is the number of “useful” warnings

(*Security*-rated and *Quality*-rated) divided by the total number of warnings, comprised of the sum of the "useful warnings" and "noise", i.e., *Insignificant* and *False* claims. These results are listed in Table 16.

On the C/C++ track, Tool K scored a significantly higher precision (68 %) than the next best tool (Tool B with 47 %). Interestingly, regarding "useful" warnings, Tool K reported only *Quality* issues (74 %) and no *Security* weaknesses (Table 14). Because we rated *Security* and *Quality* warnings equally for "useful precision", the metric ranks Tool K as the most precise tool in this context. The other tools reported less than 50 % precision, ranging from 47 % for Tool B down to 12 % for Tool G. The average precision for the tools in the C/C++ track was 31 %.

On the Java track, precision ranged from 79 % for Tool N down to 55 % for Tool M. Tool P reported a significantly lower precision of 22 %. Table 15 shows that Tool P had reported mostly *False* claims (70 %). The average precision for the tools in the Java track was 61 %.

On the PHP track, Tool R achieved 50 % precision.

Table 16. Useful Precision per Tool and per Track.

Track	Tool	Useful Precision
C/C++	Tool K	68 %
	Tool B	47 %
	Tool A	37 %
	Tool J	36 %
	Tool E	30 %
	Tool H	26 %
	Tool D	23 %
	Tool C	18 %
	Tool I	17 %
	Tool G	12 %
Java	Tool N	79 %
	Tool L	73 %
	Tool O	71 %
	Tool Q	65 %
	Tool M	55 %
	Tool P	22 %
PHP	Tool R	50 %

3.1.3.3. Covered Weakness Types

Tables 17 to 19 present the number of security-rated warnings reported by tools per kingdom in the Seven Pernicious Kingdoms (7PK) [36]. (Appendix B details the grouping of CWEs into kingdoms.)

Note that the 7PK classification has overlap, i.e., the same weakness can belong to several categories. For example, *CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer* [10] belongs to 7PK categories *Improper Input Validation* and *Indicator of Poor Code Quality*. The last column, *Unique Security Warnings*, is the total number of security-rated warnings per tool, where each warning is counted once even if it belongs to multiple 7PK categories. The bottom row, *Tool Count*, is the number of tools with security-rated warnings for the corresponding 7PK category. The columns are sorted in descending order by the tool count.

Table 17. Security-rated Warnings per 7PK for the C/C++ Track.

Tool	Code Qual.	Input Val.	API	T. & S.	Error H.	Sec. Feat.	Env.	Encap.	Unique Security Warnings
Tool J	9	5		1					11
Tool H	2	5	1	3	1				9
Tool B	7	3	1						8
Tool E	4	2	3						6
Tool A	5	4	1		1				5
Tool G	2								2
Tool C	1								1
Tool D									0
Tool I									0
Tool K									0
Tool Count	7	5	4	2	2	0	0	0	

Table 18. Security-rated Warnings per 7PK for the Java Track.

Tool	Input Val.	Sec. Feat.	T. & S.	Encap.	Code Qual.	API	Env.	Error H.	Unique Security Warnings
Tool L	16	7	5	14	4	3			36
Tool Q	24	10		19					35
Tool N	6		2						8
Tool P		3	1						3
Tool O	2								2
Tool M									0
Tool Count	4	3	3	2	1	1	0	0	

Table 19. Security-rated Warnings per 7PK for the PHP Track.

Tool	Input Val.	Encap.	Sec. Feat.	T. & S.	Code Qual.	API	Env.	Error H.	Unique Security Warnings
Tool R	5	4	3						12

Tables 17 to 19 reveal Input Validation as the best handled weakness class for the Java and PHP tracks and the second best for the C/C++ track. Code Quality issues were mostly found by tools in the C/C++ track. Our Java and PHP test cases were all web applications, so, unsurprisingly, tools reported predominantly the same weakness classes for both languages. However, because the PHP track had only one participant and one test case, the results may not generalize well.

Table 17 indicates two separate groups of tools in the C/C++ track:

- Tools J, H, B, E and A reported several *Security*-rated warnings in several 7PK categories.
- Tools G, C, D, I, and K reported few or no *Security*-rated warnings.

Table 18 shows two separate groups among the tools in the Java track:

- Tools L, Q and N reported several *Security*-rated warnings in several 7PK categories.
- Tools P, O and M reported few or no *Security*-rated warnings.

On average, Java tools seem more effective than C/C++ tools at reporting security weaknesses in real-world software. Hypothetically, the larger size and higher complexity of the C test cases made them harder to analyze than their Java counterparts. Java may also be an easier language to analyze.

3.2. CVEs

Paul Anderson, VP of engineering at Grammatech, insisted that tools should be studied on vulnerabilities that matter [37]. He proposed the use of Common Vulnerabilities and Exposures (CVEs) [18] as test cases to determine whether tools could identify vulnerabilities and prevent substantial defects. We have included CVEs in SATEs since SATE 2010 [8, Sec. 2.9].

Unfortunately, only a relatively small number of CVEs contain a sufficiently precise description to pinpoint the vulnerability in affected software. Our team browsed hundreds of entries and gathered information from bug tracking systems and other sources to turn these into usable test cases.

The work was well worth the effort. These test cases contain vulnerabilities found in the wild, thus, exhibiting two ideal qualities: ground truth and realism. They bear the certitude of exploitability and the complexity lacking in synthetic test cases. Therefore, recall, coverage, discrimination rate, and overlap can be measured (Sec. 1.5).

3.2.1. Test Sets

The test cases used in the warning subset analysis (Sec. 3.1.1) were selected because they contain numerous CVEs, allowing us to perform CVE-based analysis on the same test sets. Additionally, we asked participants to analyze a later version of the same test cases, which had the aforementioned CVEs fixed. We refer to these two versions as *flawed* and *fixed*. Table 20 lists the versions of the test sets used in SATE V.

Table 20. CVE-based Test Sets.

Track	Software	Description	Flawed Version	Fixed Version
C/C++	Asterisk[a]	IP PBX platform	10.2.0	10.12.2
	Wireshark[b]	Network traffic analyzer	1.8.0	1.8.7
Java	JSPWiki[c]	WikiWiki engine	2.5.124	2.5.139
	Openfire[d]	Groupchat server	3.6.0	3.6.4
PHP	WordPress[e]	Blogging platform	2	2.2.3

[a] http://www.asterisk.org/
[b] https://www.wireshark.org/
[c] https://jspwiki.apache.org/
[d] https://www.igniterealtime.org/projects/openfire/
[e] https://wordpress.com/

3.2.2. Procedure

The CVEs in the production software test cases are precisely characterized by metadata. We extracted the execution paths leading to the vulnerabilities, CWEs and other information useful for comparison against tool warnings. The metadata were rich enough to determine automatically whether tools found the CVEs. Due to the low number of entries, the results were also manually reviewed by experts to ensure accuracy.

For each CVE, we selected the tool findings reported at the corresponding lines of code. We only considered findings when their CWE and the CVE's CWE belonged to the same group. When this was the case, the expert was notified to review the suggested match. If the expert agreed with the automated analysis, the match was confirmed. If not, the suggestion was rejected. The experts also manually checked for matches that the algorithm might have missed.

Additionally, the experts rated the quality of the matches. Occasionally, a tool will precisely and completely report a CVE. Sometimes, tool warnings may be general, coincidental or only hint at a CVE. The experts rated a warning as equivalent or strongly-related if it precisely reported a CVE and as weakly-related if the warning only hinted at the vulnerability. In this section, we use the terms *Found* and *Hinted* to describe the two rating qualities.

Two versions of each test case were used: one containing the CVEs and one with the CVEs fixed. We refer to these variants as the flawed and fixed versions of the test case. If the expert validated a match in a flawed test case, it was rated a true positive. When no match for a CVE was detected in the flawed version, then it was rated a false negative. If the expert found a warning corresponding to the CVE in the fixed version, it was rated a false positive and a true negative otherwise.

3.2.3. Results

Before presenting the results, we would like to note a few changes to the CVE list that we made during the analysis phase of SATE V. A number of CVEs were removed from our analysis. Some others were merged into a single entry.

CVE-2006-7233 was removed from our CVE list, because the version of Openfire we used did not contain this vulnerability. Likewise, CVE-2004-1544 was removed, because it was fixed in our version of JSPWiki. We removed CVE-2007-1049 and CVE-2007-4893, because the flawed code was introduced in a slightly later release of WordPress than the version we had used. CVE-2013-4934 was not applicable to the version of Wireshark we had used and was, therefore, ignored.

CVE-2012-4294 and CVE-2012-4295 were duplicates and have been merged as CVE-2012-4294/4295. CVE-2007-5106 was a subset of CVE-2006-1263 and was also merged.

CVE-2013-3559, CVE-2013-3561, and CVE-2009-0496 each contain several unrelated vulnerabilities, which we separated and labelled as CVE-2013-3559 (1) and (2), CVE-2013-3561 (1) and (2), and CVE-2009-0496 (1) to (6), respectively.

3.2.3.1.Recall and Discrimination Rate

The C/C++ track proved difficult for tools (Table 21). In Asterisk, the best-performing tool found three CVEs out of fourteen, and half of the tools found none. With one exception, tools that found CVEs did not report false positives, i.e., weaknesses in the fixed version of the test case.

Most tools that analyzed Wireshark found CVEs, but only a fraction of the 84 were identified. The three best performers (Tools A, I, and C) each found 12 CVEs, yielding a recall of 14 %. Discrimination rate varied significantly across tools, regardless of their recall. For example, these three best performers with respect to recall had a discrimination rate of 83 %, 55 %, and 33 %, respectively.

Tools performed vastly better on the Java track (Table 22). Tool L found all of the CVEs, except one in both JSPWiki and Openfire. Tool Q found about half that number. Tool O found one in Openfire; the remaining tools missed the mark entirely. Discrimination rate was poor, regardless of the tool.

Tool R, which analyzed WordPress on the PHP track, performed remarkably well (Table 22). It found seven out of thirteen CVEs and reported only two false positives.

Applicable recall was nearly identical to recall, suggesting that most missed CVEs were of a type supported by the tools, which were unable to detect the vulnerabilities.

Regarding discrimination rate, CVEs that were found, i.e., that pointed directly to the vulnerability, were generally reported with a much higher discrimination rate than CVEs that were only hinted at (e.g., coincidental findings.)

Tables 21 and 22 summarize these results. Appendix C details how discrimination rate was calculated and Appendix D details how recall and applicable recall were obtained.

Table 21. Recall and Discrimination Rate on the CVE Test Cases (C/C++ Track).

Track	Test Case	Tool	Recall		App. Recall		Discrimination Rate		
			All	Found	All	Found	All	Found	Hinted
C/C++	Asterisk	Tool H	21 %	21 %	21 %	21 %	67 %	67 %	
		Tool J	14 %	14 %	18 %	18 %	100 %	100 %	
		Tool A	7 %	7 %	9 %	9 %	100 %	100 %	
		Tool B	7 %	7 %	8 %	8 %	100 %	100 %	
		Tool K	0 %	0 %	0 %	0 %			
		Tool G	0 %	0 %	0 %	0 %			
		Tool C	0 %	0 %	0 %	0 %			
		Tool E	0 %	0 %	0 %	0 %			
	Wireshark	Tool A	14 %	11 %	17 %	13 %	83 %	100 %	33 %
		Tool I	14 %	6 %	14 %	6 %	55 %	100 %	29 %
		Tool C	14 %	11 %	15 %	11 %	33 %	33 %	33 %
		Tool J	7 %	7 %	8 %	8 %	33 %	33 %	
		Tool H	5 %	5 %	6 %	6 %	25 %	25 %	
		Tool B	4 %	2 %	4 %	2 %	67 %	100 %	0 %
		Tool E	1 %	1 %	2 %	2 %	100 %	100 %	
		Tool K	0 %	0 %	0 %	0 %			
		Tool D	0 %	0 %	0 %	0 %			

Table 22. Recall and Discrimination Rate on the CVE Test Cases (Java and PHP Tracks).

Track	Test Case	Tool	Recall		App. Recall		Discrimination Rate		
			All	Found	All	Found	All	Found	Hinted
Java	JSPWiki	Tool L	100 %	100 %	100 %	100 %	0 %	0 %	
		Tool Q	100 %	0 %	100 %	0 %	0 %		0 %
		Tool N	0 %	0 %	0 %	0 %			
		Tool O	0 %	0 %	0 %	0 %			
		Tool P	0 %	0 %					
		Tool M	0 %	0 %					
	Openfire	Tool L	90 %	80 %	90 %	80 %	11 %	13 %	0 %
		Tool Q	60 %	60 %	67 %	67 %	33 %	33 %	
		Tool O	10 %	0 %	11 %	0 %	0 %		0 %
		Tool N	0 %	0 %	0 %	0 %			
		Tool M	0 %	0 %	0 %	0 %			
		Tool P	0 %	0 %	0 %	0 %			
PHP	WordPress	Tool R	54 %	54 %	54 %	54 %	67 %	67 %	

3.2.3.2. Coverage

The CVEs and tool warnings were associated with a large number of CWEs. To simplify the results' representation, we used the simpler Seven Pernicious Kingdoms (7PK) [36] classification instead of CWEs. Note that the 7PK contain overlap, i.e., the same weakness can belong to several groups.

Most CVEs from the C/C++ track (Wireshark and Asterisk) belonged to the Input Validation and Poor Code Quality categories, dominated by buffer overflows and pointer issues. Wireshark also presented a large number of Time and State-related CVEs, mainly infinite loops.

The Java and PHP test cases (Openfire, JSPWiki and WordPress) are all web applications, which contain CVEs related to Input Validation and Encapsulation issues, mostly cross-site scripting and path traversal.

Figure 3 displays a detailed distribution of CVE types per test case. No single test case contains all of the CVE types.

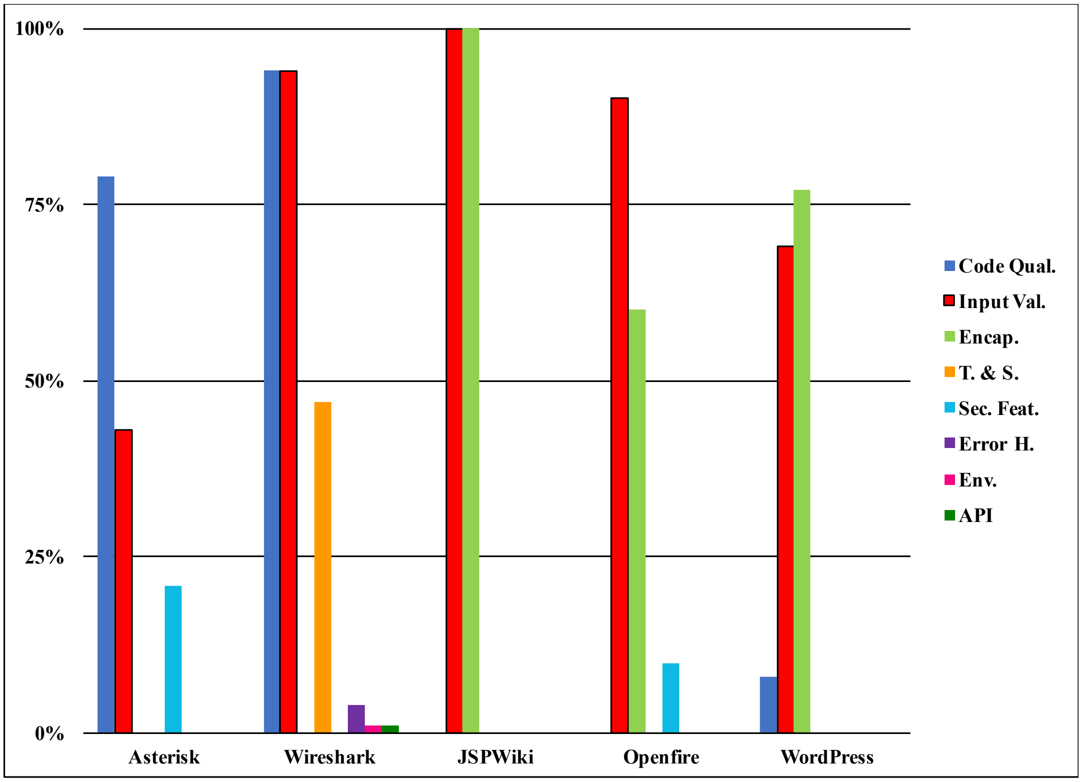

Figure 3. Distribution of CVE Types per Test Case.

Tables 23 to 27 summarize the types of CVEs tools detected in each test case. A weakness type was rated *Found* if, at least, one CVE of that type was directly reported by a tool. It was rated *Hinted* if a tool reported a coincidental warning that might lead a user to the discovery of the CVE. Note that if a tool missed all of vulnerabilities of a certain type and, therefore, scored a *Missed* rating for that category, it did not mean that the tool could not find that type of defect. Rather, that tool was unable to detect it in this particular context.

Table 23. CVEs' Weakness Categories Found by Tools in Asterisk.

Tool	Code Qual.	Input Val.	Sec. Feat.
Tool A	Found	Found	Missed
Tool B	Found	Found	Missed
Tool H	Found	Found	Missed
Tool J	Found	Found	Missed
Tool C	Missed	Missed	Missed
Tool E	Missed	Missed	Missed
Tool G	Missed	Missed	Missed
Tool K	Missed	Missed	Missed

Table 24. CVEs' Weakness Categories Found by Tools in Wireshark.

Tool	Code Qual.	Input Val.	T. & S.	API	Error H.	Env.
Tool J	Found	Found	Missed	Found	Found	Missed
Tool A	Found	Found	Found	Hinted	Hinted	Missed
Tool I	Found	Found	Found	Hinted	Hinted	Missed
Tool B	Found	Found	Missed	Hinted	Hinted	Missed
Tool C	Found	Found	Found	Missed	Missed	Missed
Tool H	Found	Found	Missed	Missed	Missed	Missed
Tool E	Found	Missed	Missed	Missed	Missed	Missed
Tool D	Missed	Missed	Missed	Missed	Missed	Missed
Tool K	Missed	Missed	Missed	Missed	Missed	Missed

Table 25. CVEs' Weakness Categories Found by Tools in JSPWiki.

Tool	Encap.	Input Val.
Tool L	Found	Found
Tool Q	Hinted	Hinted
Tool M	Missed	Missed
Tool N	Missed	Missed
Tool O	Missed	Missed
Tool P	Missed	Missed

Table 26. CVEs' Weakness Categories Found by Tools in Openfire.

Tool	Input Val.	Encap.	Sec. Feat.
Tool L	Found	Found	Missed
Tool Q	Found	Found	Missed
Tool O	Hinted	Missed	Missed
Tool M	Missed	Missed	Missed
Tool N	Missed	Missed	Missed
Tool P	Missed	Missed	Missed

Table 27. CVEs' Weakness Categories Found by Tools in WordPress.

Tool	Encap.	Input Val.	Code Qual.
Tool R	Found	Found	Missed

3.2.3.3. Unreported Vulnerabilities

In Ref. [38], Arthur Hicken, Chief Evangelist at Parasoft, expressed an interest in vulnerabilities that were not reported by tools. We refined this idea by rating CVEs to bring out the low-hanging fruits that we thought tools were capable of finding.

The CVEs were given a grade, ranging from *Simple* to *Extreme* (*Simple*, *Medium*, *Hard*, and *Extreme*). Considering the diversity of cases and the difficulty of the task, the ratings carry a subjective bias that we tried to mitigate using criteria [8, Sec. 3.6], such as control and data flow complexity and calculations. *Extreme* cases were usually out of the scope of static analysis, e.g., design problems.

Tables 29 to 36 list which CVEs were found by tools and which were not. Since CVEs are complex and a binary match/no match classification is insufficient, we used the following markings to classify tool findings:

- *Match* - a tool completely identified a CVE,
- *Partial* - a tool found one element of a CVE chain, such as an integer overflow in an integer overflow to buffer overflow vulnerability,
- *Hint* - a tool reported a coincidental warning that could lead a user to find a CVE,
- *Miss* - a tool did not find a CVE, but supported the same weakness type as the CVE,
- Blank cell - a tool did not support the weakness type of that CVE.

Support of weakness types was approximately determined by analyzing all warnings for each tool.

Table 28 displays types of weaknesses (both abbreviations and short descriptions) of the CVEs in the SATE V test cases.

Table 28. Weakness types of CVEs in SATE V.

Type	Description
ASRT	Reachable assertion
BOF	Buffer error
DIV	Division by zero
FREE	Memory freeing error
FSTR	Format string issue
IAC	Incorrect access control
IEX	Information exposure
INI	Initialization issue
LOOP	Loop issue
NPD	Null pointer dereference
PTR	Pointer issue
REX	Resource exhaustion
SQLI	SQL Injection
XSS	Cross-site scripting

In Asterisk (Table 29), Tools H, J, B, and A found only a few *Simple* CVEs. Most of the tools supported all of the vulnerability types in the *Simple* and *Medium* categories. For readability, Table 29 omits columns for tools G, C, E, and K, since they did not find any CVEs in Asterisk. A version of the table including columns for all tools is provided in Appendix H.

In Wireshark (Tables 30 to 33), we ranked match quality from high to low: *Match* > *Partial* > *Hint* > *Miss*. For readability, Tables 30 to 33 omit columns for tools D and K, since they did not find any CVEs in Asterisk. Versions of the tables including columns for all tools are provided in Appendix H.

Most of the *Match* findings were generated for the *Simple* CVEs. The tools reported fewer *Match* findings and more *Partial* and *Hint* findings for *Medium* CVEs. *Hard* CVEs also exhibited mostly *Partial* and *Hint* findings, but in fewer numbers. Only one *Match* finding was reported for *Extreme* cases. These tables demonstrate clearly that as the difficulty of the CVEs increased, tools reported fewer, lower quality findings.

On the Java track (Tables 34 and 35), Tool L found all of the *Simple* and *Medium* CVEs, while Tool P reported mostly *Simple* CVEs. Tools O and N, despite supporting the weakness classes for both *Simple* and *Medium* CVEs, did not report any of them, while Tools M and Q did not seem to support the most basic Java weakness classes.

On the PHP track (Table 36), Tool R found all of the *Simple* and *Medium* CVEs, but missed the more complex CVEs.

Overall, tools reported most low-hanging fruits in the Java and PHP test cases, whereas the C test cases proved significantly more difficult, even for simpler vulnerabilities. As a recommendation, we would suggest that participants identify which shortcomings cause their tools to miss Simple and Medium vulnerabilities. Detailed information about these CVEs is available in the SARD [20].

Table 29. CVEs Found and Missed on Asterisk.

Difficulty	CVE	Type	Tool H	Tool J	Tool B	Tool A
Simple	CVE-2012-1183	BOF	Match	Match	Match	Miss
	CVE-2013-2686	REX	Match	Match	Miss	Match
	CVE-2012-2415	BOF	Match	Miss	Miss	Miss
	CVE-2012-1184	BOF	Miss	Miss	Miss	Miss
	CVE-2012-2416	NPD	Miss	Miss	Miss	Miss
	CVE-2012-2947	NPD	Miss	Miss	Miss	Miss
	CVE-2012-3553	NPD	Miss	Miss	Miss	Miss
	CVE-2012-2948	NPD	Miss	Miss	Miss	Miss
Medium	CVE-2012-3812	FREE	Miss	Miss	Miss	Miss
Extreme	CVE-2012-5977	REX	Miss	Miss	Miss	Miss
	CVE-2012-4737	IAC	Miss		Miss	
	CVE-2012-3863	REX	Miss	Miss	Miss	Miss
	CVE-2012-2186	IAC	Miss			
	CVE-2012-2414	IAC	Miss			

Table 30. Simple-rated CVEs Found and Missed on Wireshark.

Difficulty	CVE	Type	Tool A	Tool C	Tool J	Tool I	Tool B	Tool H	Tool E
Simple	CVE-2012-5240	BOF	Match	Miss	Match	Miss	Match	Match	Miss
	CVE-2013-2475	NPD	Match	Miss	Match	Miss	Match	Miss	Match
	CVE-2013-2481	REX	Match	Miss	Miss	Hint	Miss	Miss	Miss
	CVE-2012-4285	DIV	Match	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-4286	DIV	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-4296	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1587	ASRT	Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2012-4293	ASRT	Miss	Miss	Miss	Miss	Miss		
	CVE-2012-5238	ASRT	Miss	Miss	Miss	Miss	Miss		

Table 31. Medium-rated CVEs Found and Missed on Wireshark.

Difficulty	CVE	Type	Tool A	Tool C	Tool J	Tool I	Tool B	Tool H	Tool E
Medium	CVE-2013-3559 (1)	BOF	Miss	Partial	Match	Partial	Miss	Miss	Miss
	CVE-2012-4298	BOF	Partial	Miss	Miss	Miss	Miss	Match	Miss
	CVE-2013-3559 (2)	BOF	Miss	Partial	Miss	Partial	Miss	Miss	Miss
	CVE-2013-4074	REX	Hint	Miss	Match	Hint	Hint	Miss	Miss
	CVE-2013-4082	BOF	Miss	Partial	Miss	Miss	Miss	Partial	Miss
	CVE-2013-3562	REX	Miss	Hint	Miss	Match	Miss	Miss	Miss
	CVE-2012-4294 / CVE-2012-4295	BOF	Match	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-2480	BOF	Miss	Hint	Miss	Hint	Miss	Miss	Miss
	CVE-2013-2487	LOOP	Miss	Hint	Miss	Hint	Miss	Miss	Miss
	CVE-2012-4048	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-4049	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-4297	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-6059	PTR	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1579	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1582	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1588	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1590	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-2483	DIV	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-2484	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-2488	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-3557	INI	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-4076	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-4935	INI	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-4081	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-3548	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2013-1575	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2013-2476	LOOP		Miss		Hint	Miss		
	CVE-2013-4933	REX	Miss		Miss	Miss	Miss		
	CVE-2012-5237	LOOP		Miss		Miss	Miss		
	CVE-2013-2485	LOOP		Miss		Miss	Miss		
	CVE-2013-4080	LOOP		Miss		Miss	Miss		

Table 32. Hard-rated CVEs Found and Missed on Wireshark.

Difficulty	CVE	Type	Tool A	Tool C	Tool J	Tool I	Tool B	Tool H	Tool E
	CVE-2012-6062	LOOP	Partial	Miss	Miss		Miss	Miss	Miss
	CVE-2013-1573	LOOP	Miss	Partial	Miss		Miss	Miss	Miss
	CVE-2013-4930	REX	Miss	Miss	Match	Miss	Miss	Miss	Miss
	CVE-2013-1585	BOF	Partial	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-2478	BOF		Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-6061	LOOP	Miss		Miss	Miss	Miss	Miss	Miss
	CVE-2013-1574	LOOP	Miss		Miss	Miss	Miss	Miss	Miss
	CVE-2013-1580	LOOP	Miss		Miss	Miss	Miss	Miss	Miss
	CVE-2013-1572	LOOP	Hint	Miss	Miss	Hint	Miss	Miss	Miss
	CVE-2013-2482	LOOP	Miss	Miss	Miss	Hint	Miss	Miss	Miss
	CVE-2012-4292	PTR	Miss	Miss	Miss	Miss	Miss	Miss	Miss
Hard	CVE-2012-6060	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1583	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1584	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1586	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-4075	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-4077	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-6056	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-6058	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-4287	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2012-6055	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2012-6053	LOOP		Miss		Miss	Miss		
	CVE-2013-2479	LOOP		Miss		Miss	Miss		

Table 33. Extreme-rated CVEs Found and Missed on Wireshark.

Difficulty	CVE	Type	Tool A	Tool C	Tool J	Tool I	Tool B	Tool H	Tool E
	CVE-2013-3558	BOF	Miss	Miss	Match	Miss	Miss	Miss	Miss
	CVE-2012-4288	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2012-4289	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	
Extreme	CVE-2012-4290	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2012-6054	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2013-3560	FSTR	Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2012-4291	REX	Miss		Miss	Miss	Miss		

Difficulty	CVE	Type	Tool A	Tool C	Tool J	Tool I	Tool B	Tool H	Tool E
	CVE-2013-4078	LOOP		Miss		Miss	Miss		
	CVE-2013-3561 (2)	LOOP	Miss		Miss	Miss	Miss	Miss	Miss
	CVE-2013-3561 (1)	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-4927	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2013-1581	LOOP		Miss		Miss	Miss		
	CVE-2013-4079	LOOP		Miss		Miss	Miss		
	CVE-2013-4929	LOOP		Miss		Miss	Miss		
	CVE-2012-6057	LOOP	Miss		Miss	Miss	Miss	Miss	Miss
	CVE-2013-4083	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-4931	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1577	LOOP	Miss	Miss	Miss	Miss	Miss		
	CVE-2013-1578	REX	Miss	Miss	Miss	Miss	Miss		
	CVE-2013-1576	LOOP		Miss		Miss	Miss		

Table 34. CVEs Found and Missed on JSPWiki.

Difficulty	CVE	Type	Tool L	Tool Q	Tool N	Tool O	Tool M	Tool P
Simple	CVE-2007-5120	XSS	Match	Hint	Miss	Miss		

Table 35. CVEs Found and Missed on Openfire.

Difficulty	CVE	Type	Tool L	Tool P	Tool O	Tool N	Tool M	Tool Q
Simple	CVE-2009-0496 (1)	IAC	Match	Match	Miss	Miss		
	CVE-2009-0496 (2)	SQLI	Match	Match	Miss	Miss		
	CVE-2009-0496 (3)	XSS	Match	Match	Miss	Miss		
	CVE-2009-0496 (4)	XSS	Match	Match	Miss	Miss		
	CVE-2009-0496 (5)	XSS	Match	Miss	Miss	Miss		
	CVE-2009-0496 (6)	XSS	Match	Match	Miss	Miss		
	CVE-2009-0497	XSS	Match	Match	Miss	Miss		
Medium	CVE-2008-6509	XSS	Match	Miss	Miss	Miss		Miss
Extreme	CVE-2008-6508	IAC	Hint	Miss	Hint	Miss		
	CVE-2009-1596	IAC	Miss			Miss	Miss	

Table 36. CVEs Found and Missed on WordPress.

Difficulty	CVE	Type	Tool R
Simple	CVE-2006-0985	XSS	Match
	CVE-2006-1263 / CVE-2007-5106	XSS	Match
	CVE-2006-1796	XSS	Match
	CVE-2006-6808	IEX	Match
	CVE-2007-5105	XSS	Match
Medium	CVE-2007-0233	IEX	Match
	CVE-2007-1894	SQLI	Match
Hard	CVE-2007-1622	REX	Miss
Extreme	CVE-2006-3389	IEX	Miss
	CVE-2007-0109	XSS	Miss
	CVE-2007-0540	XSS	Miss
	CVE-2007-0541	XSS	Miss
	CVE-2013-7233	IAC	Miss

3.2.3.4. Overlap

Overlap represents the number of CVEs found by more than one tool. This metric identifies which tools behave similarly and which vulnerabilities are easy or difficult for tools to find.

Tables 29 to 36 detail which CVEs were found and missed by each tool. On the other hand, overlap is the number of tools that found the same weakness. Table 37 summarizes the overlap for each test case. An overlap of zero means that no tools found the CVE. An overlap of one means that only one tool reported the CVE. An overlap of two means that two tools reported the CVE, and so forth. By definition, the overlap cannot be greater than the number of participants for each test case.

On the C/C++ track, most CVEs remained undetected, as was demonstrated in Sec. 3.2.3.3. However, when a CVE was found, it was usually detected by more than one tool. In Asterisk, two CVEs were found by three tools and only one by a single tool. In Wireshark, 16 % of the reported CVEs were detected by one tool, whereas 18 % were found by two to four tools.

On the Java track, most of the CVEs were discovered by two tools.

Because the PHP track had only one participant, there was no overlap.

Overall and despite a weak recall for C, the overlap on the CVEs was higher than in previous SATEs [9, Fig. 6-7]. Broader participation, different test cases, tool improvement or the use of CVEs as a benchmark might be factors in that increase.

Table 37. Overlap per CVE Test Case.

Track	Test Case	Participants	CVEs	Overlap				
				0	1	2	3	4
C/C++	Asterisk	8	14	79 %	7 %	0 %	14 %	0 %
	Wireshark	9	83	66 %	16 %	13 %	1 %	4 %
Java	JSPWiki	6	1	0 %	0 %	100 %	0 %	0 %
	Openfire	6	10	10 %	20 %	70 %	0 %	0 %
PHP	WordPress	1	13	46 %	54 %	N/A	N/A	N/A

3.3. Synthetic Test Suites

Synthetic test cases were introduced in SATE IV after the U.S. National Security Agency (NSA) Center for Assured Software (CAS) issued Juliet, an extensive test suite for C/C++ and Java [11]. This collection covered a vast number of weakness types embedded in different code constructs. It exhibited two desired qualities: statistical significance and ground truth, because the many planted weaknesses' locations were known. However, it lacked realism, because each program was computer-generated and served no other purpose than modeling a specific defect.

3.3.1. Test Sets

During SATE IV's preparation stage, CAS released Juliet 1.0 [39, 40], the first large-scale synthetic test suite. We seized the opportunity to study static analysis results in greater depth than in previous SATEs. In SATE V, participants ran their tools on Juliet 1.2 [41, 42], which had corrected several bugs and covered a wider range of CWEs and code constructs. (Since then Juliet 1.3 has been released. It has additional coverage and corrects many errors in version 1.2 [21].)

The Juliet 1.2 test suite is divided into a C/C++ and a Java component. Each component contains thousands of test cases comprised of matched functions with and without weaknesses. We refer to these as *bad* code and *good* code, respectively. The defect in bad code is marked with a CWE [10], so identifying the weakness was straightforward.

Table 38 summarizes a few statistics regarding the Juliet 1.2 test suite. The *CWEs* column contains the number of different CWE IDs covered by the test suite. The *Test cases* and *Files* columns are self-explanatory. The *LoC* column lists the number of non-blank, non-comment lines of code (LoC) for each language.

Table 38. Juliet 1.2 Statistics.

Track	CWEs	Test cases	Files	LoC [a]
C/C++	118	61 387	102 092	4 719 409 (C)
				3 882 727 (C++)
Java	112	25 477	41 170	4 565 713

[a]According to SLOCCount, which counts source lines of code (SLOC). *https://www.dwheeler.com/sloccount/*

To elucidate smart static analyzer functionality, Juliet's test cases were designed to incorporate specific flaws within a number of code constructs of diverse complexities. A basic case contained a straightforward weakness, whereas a more complex case harbored the same defect wrapped in intricate control or data flow code structures. An unsophisticated tool might find the weakness in the simple case, but it would miss the weakness embedded in a more complex structure. A more discerning tool would detect the second case, thus finding both vulnerabilities.

3.3.2. Procedure

The weaknesses in the synthetic test cases were precisely characterized by metadata. We extracted the different blocks of code (good and bad), the weakness locations, their associated CWEs and other information to compare tool warnings. The metadata were rich enough to allow automated assessment of tool outputs, enabling analysis of *all* tool warnings, in contrast to the sample analysis method used in the production test cases (Sec. 3.1).

For each synthetic test case, we selected the tool warnings reported in their associated files. We only considered a warning when its CWE and the test case's CWE belonged to the same CWE group, and the warning location was in an appropriate block of the test case, detailed as follows. When the tool reported a defect in good code, it was rated a f*alse positive (FP)*. When the tool reported a defect in bad code, we assumed that the tool correctly found the weakness and rated it as a *true positive (TP)*. If no warning was generated from bad code, it was rated a *false negative (FN)*, because the tool had missed the defect. Finally, an absence of warnings reported in good code resulted in a *true negative (TN)* rating. Figure 4 summarizes this evaluation process.

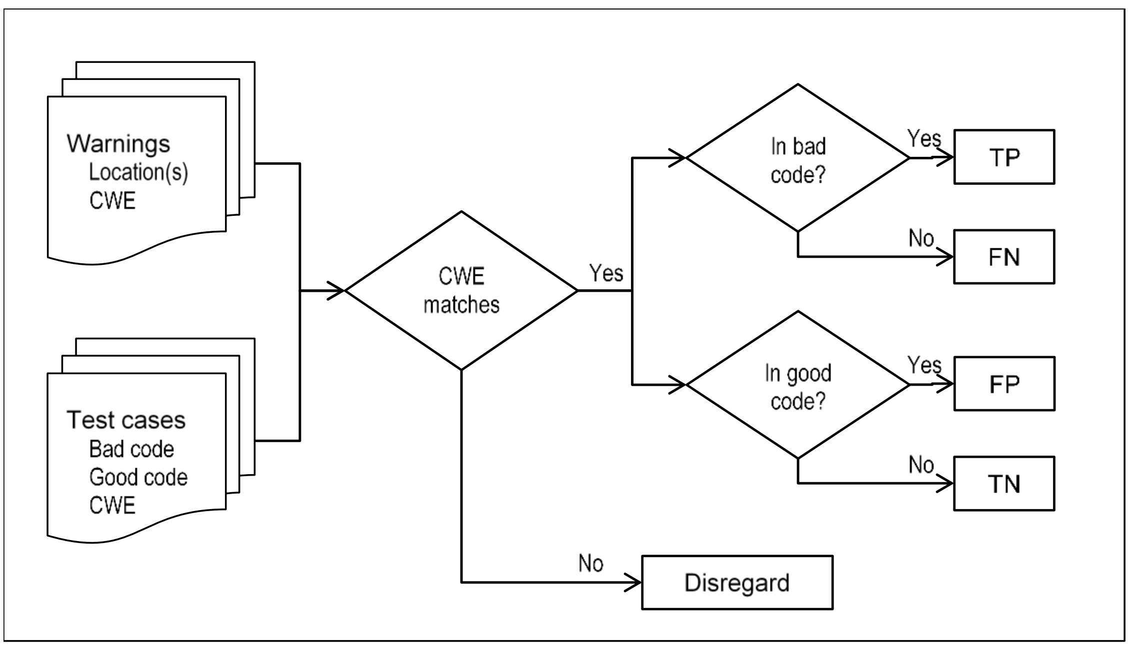

Figure 4. Evaluation Process for Synthetic Test Cases.

3.3.3. Analysis Cycles

Since we used an automated evaluation algorithm, we required an assessment and improvement process. This was achieved through review cycles. First, we ran the automated analysis, and then we sampled a subset of results. An expert reviewed the results, correcting the metadata (in particular, modifying CWE groups) or algorithm, as needed. For example, if a weakness was manifested in a specific function call, a tool warning location was matched to a specific line, instead of anywhere in the bad code. Then the process was repeated (Fig. 5).

At the end of each cycle, the expert also assessed the accuracy of the analysis. The process was repeated until the expert had obtained acceptable accuracy. Figures 6 and 7 show the improvement in accuracy over the review cycles of SATE V for both the C/C++ and Java tracks. Please note that the results for stage 6 are based on the samples taken during the previous five stages. After Stage 6, the results averaged 99 % accuracy, with a minimum of 98 %. The remaining discrepancies were mostly caused by defects in some test cases.

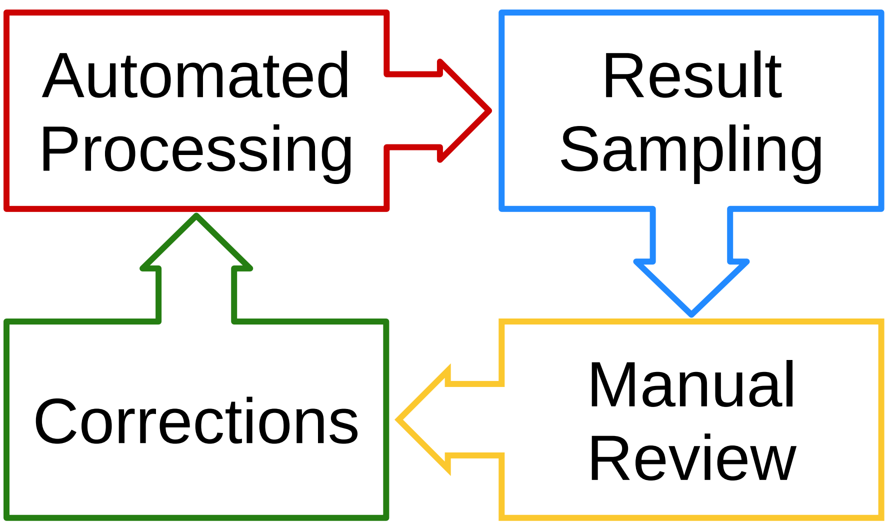

Figure 5. Synthetic Test Case Analysis Cycle.

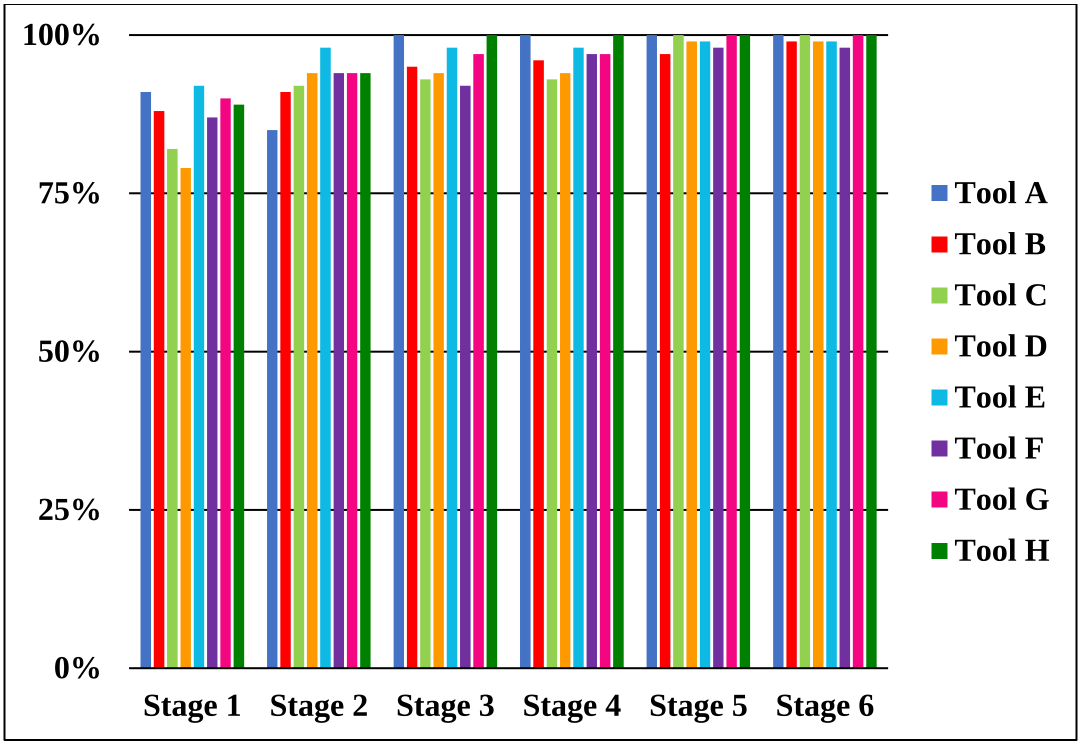

Figure 6. Improvement in the Evaluation Accuracy for C/C++.

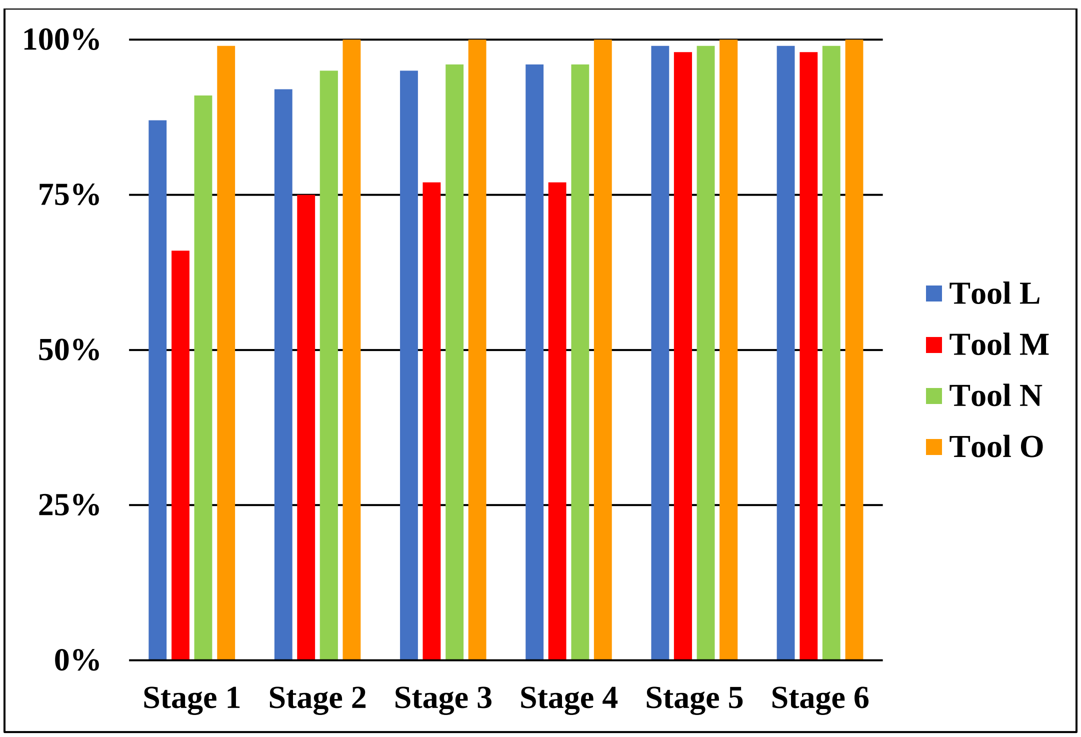

Figure 7. Improvement in the Evaluation Accuracy for Java.

3.3.4. Complexity

The Juliet test suites (C/C++ and Java) contained examples of most of the flaws detectable by a static analysis tool. The flaws, embedded in a broad range of code constructs, demonstrated the ability of a given tool to follow complex control and data flows. Each weakness existed in a simple form and, when possible, in a variety of more complex programs. Section 3.3 and Appendix C of both Ref. [43] and Ref. [44] describe the different constructs used in the Juliet 1.2 test suites for C/C++ and Java, respectively.

3.3.5. Results

Our automated analysis used the CWE categories described in Appendix A. As in the rest of this report, we represented the results using the simpler Seven Pernicious Kingdoms (7PK) [36], detailed in Appendix B. Again, both categorizations contain overlap, i.e., CWEs can belong to several groups. Figure 8 demonstrates that some categories contained many more CWEs than others, the largest categories being Code Quality and Input Validation.

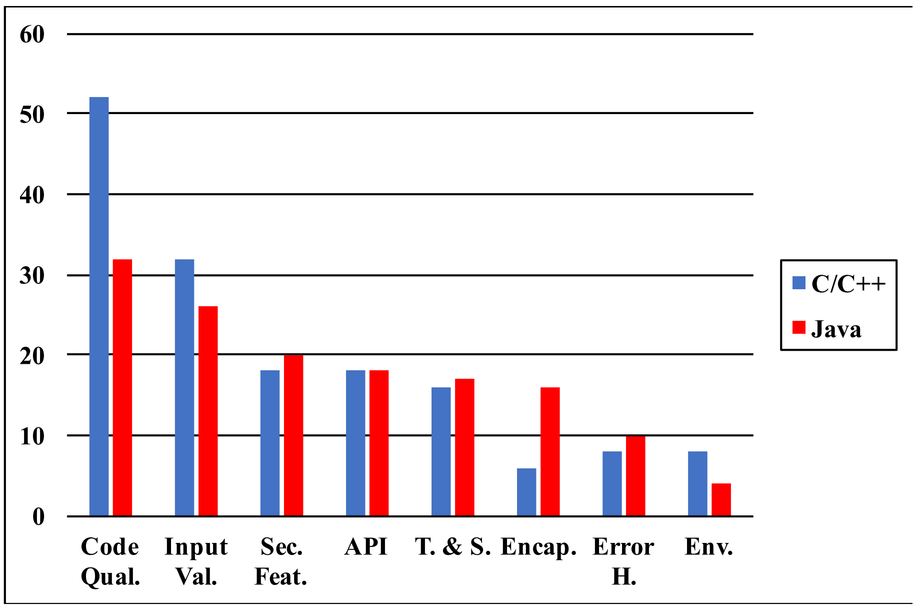

Figure 8. CWE Count per Category in Juliet 1.2 C/C++ and Java.

This imbalance was magnified by the dissymmetry in the number of test cases implementing each CWE. Indeed, some defect classes were represented by only a handful of test cases and others by several thousand test cases. Also, there were many more test cases in the C/C++ track than in the Java track. Figure 9 displays the test case distribution across the categories for C/C++ and Java. Input Validation and Code Quality were over-represented compared to the other 7PK, due to both having more CWEs and more test cases per CWE.

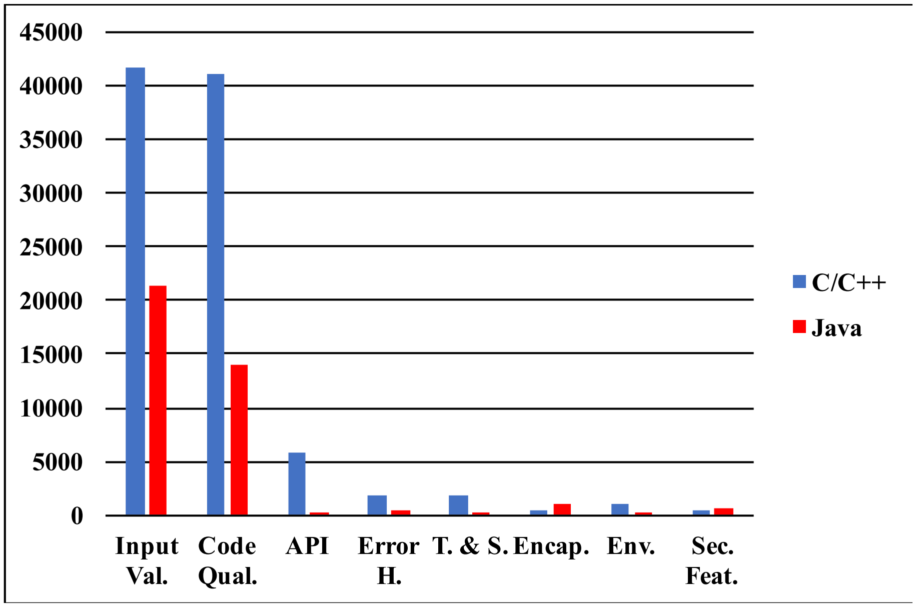

Figure 9. Test Case Count per Category in Juliet C/C++ and Java.

With these caveats in mind, the 7PK categorization offered a simple, semantically coherent way to present our results.

3.3.5.1. Coverage

As a reminder, *coverage* is determined by the type of weaknesses found by a tool. It is measured by the number of unique CWEs reported over the total number of CWEs tested (Sec. 1.5). These synthetic test suites provided a set of test cases for each CWE tested. If a tool reported a true positive on a given test case, then we assumed it was capable of detecting that type of CWE.

In the following tables, the CWEs were grouped by category according to the 7PK. Coverage represents the proportion of CWEs correctly identified in each group. For example, if a category contained four CWEs, of which a tool detected two, then the tool scored a coverage of 50 % for that category.

Tables 39 and 40 show significant variation in tools' coverage. Tool B from Table 39 reported far greater coverage than Tool F. Tool L from Table 40 detected more CWEs in most categories than other tools in the study. Please note that the average reported for Input Validation (33 %) was unexpectedly low due to the very low value for Tool M (4 %). If the value for Tool M were excluded, the average would be 43 %, ranking second in coverage per category.

Coverage of weakness categories varied by language. For C, Code Quality issues and Input Validation dominated, whereas for Java, Code Quality issues were predominant, Other categories in the C/C++ track exhibited less extensive coverage. In particular, tools found very few Security Features-related flaws. In the Java track, coverage was more uniform for Input Validation, Code Quality, Time and State, API Abuse, and Error Handling weaknesses. Other categories were covered by fewer tools or not as well.

Table 39. Coverage per Category for Synthetic C/C++.

Tool	Code Qual.	Input Val.	Error H.	Env.	T. & S.	API	Encap.	Sec. Feat.	Average
Tool B	65 %	53 %	50 %	38 %	38 %	33 %	17 %	0 %	37 %
Tool H	42 %	47 %	25 %	25 %	31 %	22 %	33 %	0 %	28 %
Tool G	65 %	44 %	13 %	38 %	13 %	28 %	0 %	6 %	26 %
Tool A	50 %	63 %	13 %	25 %	19 %	11 %	17 %	0 %	25 %
Tool C	23 %	28 %	50 %	0 %	13 %	33 %	17 %	6 %	21 %
Tool D	35 %	28 %	13 %	25 %	13 %	6 %	17 %	0 %	17 %
Tool E	31 %	16 %	0 %	13 %	19 %	17 %	0 %	0 %	12 %
Tool F	19 %	25 %	0 %	0 %	13 %	0 %	0 %	0 %	7 %
Average	41 %	38 %	21 %	21 %	20 %	19 %	13 %	2 %	

Table 40. Coverage per Category for Synthetic Java.

Tool	Code Qual.	T. & S.	API	Input Val.	Error H.	Sec. Feat.	Env.	Encap.	Average
Tool L	47 %	53 %	56 %	62 %	50 %	70 %	75 %	50 %	58 %
Tool N	53 %	41 %	39 %	31 %	20 %	5 %	0 %	6 %	24 %
Tool M	41 %	53 %	33 %	4 %	50 %	0 %	0 %	6 %	23 %
Tool O	47 %	24 %	22 %	35 %	10 %	25 %	0 %	6 %	21 %
Average	47 %	43 %	38 %	33 %	33 %	25 %	19 %	17 %	

Appendix E summarizes the coverage of the Juliet test suites for each tool. Note that coverage is only one aspect of tool effectiveness, so although some tools seemed to surpass others with respect to coverage, users should not select a tool based on coverage alone. In addition, a user's coverage requirements might be met by several tools. Other factors, e.g., recall and precision, should be examined to determine the most suitable tool for that user.

3.3.5.2. Recall

Recall is defined by the number of correct findings compared to the total number of defects present in the code (Sec. 1.5). The higher the recall, the more weaknesses the tool found.

Table 41 shows a greater propensity of tools finding the following C/C++ weakness categories: Time and State, Code Quality, API Abuse, and Input Validation. Arguably, these were the most prominent problems in C.

For Java, Table 42 indicates that only API Abuse and Time and State issues were found by all of the tools to a significant extent. Tool L detected nearly all of the Environment-related defects, but other tools found none. More surprisingly, Input Validation weaknesses were largely missed by the tools, although the low average was partly due to poor results by tool M.

Note that because the 7PK classification contains overlap, the numbers may add up to over 100 % in Tables 41 and 42.

Table 41. Recall per Category for Synthetic C/C++.

Tool	T. & S.	Code Qual.	API	Input Val.	Encap.	Error H.	Env.	Sec. Feat.
Tool B	33 %	21 %	38 %	11 %	23 %	11 %	14 %	0 %
Tool A	11 %	21 %	18 %	18 %	20 %	10 %	11 %	0 %
Tool H	12 %	18 %	2 %	19 %	12 %	25 %	6 %	0 %
Tool F	30 %	29 %	0 %	27 %	0 %	0 %	0 %	0 %
Tool E	23 %	12 %	28 %	1 %	0 %	0 %	1 %	0 %
Tool C	2 %	10 %	11 %	13 %	0 %	14 %	0 %	3 %
Tool D	8 %	1 %	1 %	4 %	14 %	1 %	9 %	0 %
Tool G	1 %	1 %	1 %	1 %	0 %	2 %	1 %	0 %
Avg. Recall	15 %	14 %	12 %	12 %	9 %	8 %	5 %	0 %

Table 42. Recall per Category for Synthetic Java.

Tool	API	Encap.	T. & S.	Sec. Feat.	Env.	Error H.	Input Val.	Code Qual.
Tool L	59 %	80 %	27 %	73 %	97 %	55 %	33 %	5 %
Tool O	26 %	35 %	18 %	25 %	0 %	4 %	17 %	2 %
Tool M	32 %	2 %	34 %	0 %	0 %	20 %	0 %	3 %
Tool N	33 %	2 %	21 %	1 %	0 %	8 %	11 %	2 %
Avg. Recall	38 %	30 %	25 %	25 %	24 %	22 %	15 %	3 %

Regarding the C/C++ test cases (Table 41), Tool B found significantly more weaknesses across all 7PK categories than other tools. Tools A and H detected many weaknesses in many different categories as well, while Tools F and E seemed to find more defects but in fewer categories.

Regarding Java test cases (Table 42), Tool L outperformed other tools, detecting about twice as many weaknesses as the other tools.

Except for Tool L, recall remained fairly low for both C/C++ and Java test cases. Tools struggled to find 25 % of defects in these test suites. The synthetic code used unusual constructs, possibly making weakness detection more difficult. However, this test case complexity was still fairly low compared to large software. This suggests that recall would be even lower on real-world software.

Appendix F summarizes the recall results from each tool, for all of the CWEs in the Juliet test suites.

3.3.5.3. Applicable Recall

The tools did not typically support all of the types of flaws contained in our test suites, and, therefore, they scored a null recall for those categories, lowering their average recall values. Consequently, tools focused on only a few weakness types were penalized. At the SATE V Workshop [45], we introduced the concept of *applicable recall*[6], i.e., recall calculated only for the weakness categories supported by each tool (Sec. 1.5). Combined with the coverage metric, applicable recall provided a better assessment of a tool's capabilities.

Tables 43 and 44 list the results for recall, applicable recall, and coverage. Tools with the lowest coverage produced the highest recall *increase* when calculated solely on the tool's supported weakness categories. That is, tools with the lowest coverage exhibited the highest positive differences between recall and applicable recall.

However, this did not mean that general tools performed worse than more specialized tools in the categories they both supported. For example, Tool E exhibited the second highest recall increase (from 8 % to 19 %) and Tool A the second lowest (from 17 % to 21 %) (Table 43). Yet, Tool A scored a higher applicable recall than Tool E on many CWEs, including *CWE-195: Signed to Unsigned Conversion Error* [10], where Tool A found 87 % of the flaws and Tool E only 11 %, as shown in Appendix G.

Applicable recall per CWE for each tool is detailed in Appendix G.

Note that these numbers do not match the results in Tables 41 and 42, which were calculated using the overlapping 7PK groups.

Table 43. Recall vs. Applicable Recall for Synthetic C/C++.

Tool	Recall	App. Recall	Coverage
Tool F	20 %	56 %	9 %
Tool H	18 %	25 %	31 %
Tool B	18 %	25 %	42 %
Tool A	17 %	21 %	29 %
Tool C	10 %	18 %	22 %
Tool E	8 %	19 %	15 %
Tool D	4 %	8 %	19 %
Tool G	1 %	2 %	35 %

[6] Labelled *condensed recall* at the time.

Table 44. Recall vs. Applicable Recall for Synthetic Java.

Tool	Recall	App. Recall	Coverage
Tool L	34 %	73 %	56 %
Tool O	16 %	52 %	29 %
Tool N	11 %	39 %	29 %
Tool M	2 %	78 %	25 %

3.3.5.4. Precision for 50 % Prevalence

Precision is the proportion of correct warnings produced by a tool (Sec. 1.5). The higher the precision, the less noise, i.e., false positives, a tool generates.

Precision depends on the prevalence of weaknesses in the software, where prevalence is the proportion of test cases with weaknesses. The higher the prevalence of weaknesses, the higher the precision [46]. In Juliet, 50 % of test cases have a weakness, in contrast with production software made by competent programmers, where a much smaller proportion of code is buggy. Accordingly, the precision for Synthetic test cases is based on 50 % prevalence, and it is not directly comparable with precision results for production software, Sec. 3.1.3.2.

In the rest of this paper, when we use term “precision” for Synthetic test cases, we mean “precision for 50 % prevalence.”

Tables 45 and 46 present precision for 50 % prevalence for each tool per 7PK category. Note that the blank cells in Tables 45 and 46 indicate that a given weakness category was not supported by that tool. Also, the *Average* columns in both tables contain the average precision values per category, which is not the same as the average precision over the entire C/C++ and Java tracks, since there is some overlap between categories.

Table 45. Precision for 50 % Prevalence per Category for Synthetic C/C++.

Tool	Encap.	API	Error H.	Env.	Code Qual.	Input Val.	T. & S.	Sec. Feat.	Average
Tool D	100 %	100 %	100 %	100 %	93 %	61 %	80 %		89 %
Tool B	100 %	95 %	89 %	94 %	88 %	80 %	70 %		86 %
Tool A	96 %	90 %	88 %	96 %	73 %	70 %	73 %		82 %
Tool H	100 %	63 %	81 %	100 %	83 %	72 %	66 %		78 %
Tool C	100 %	100 %	95 %		90 %	72 %	51 %	50 %	76 %
Tool G		87 %	92 %	73 %	72 %	52 %	74 %	100 %	79 %
Tool E		100 %		50 %	92 %	92 %	70 %		81 %
Tool F					94 %	93 %	100 %		96 %
Average	99 %	91 %	91 %	86 %	86 %	74 %	73 %	75 %	

In the C/C++ track (Table 45), tools achieved 84 % precision on average in all categories. Most warnings reported were correct. Interestingly, precision was rather uniform across tools for each category. This could indicate that some flaws are more prone to confuse tools than others. For example, Encapsulation weaknesses were correctly reported 99 % of the time, whereas Input Validation warnings were correct only 74 % of the time.

In the Java track (Table 46), the average precision reached 85 %. Claims about API Abuse were mostly correct for all of the tools, whereas tools' precision was disparate for other categories of warnings.

Table 46. Precision for 50 % Prevalence per Category for Synthetic Java.

Tool	API	Encap.	T. & S.	Code Qual.	Error H.	Input Val.	Sec. Feat.	Env.	Average
Tool N	96 %	100 %	77 %	96 %	68 %	93 %	100 %		90 %
Tool M	98 %	100 %	90 %	74 %	100 %	100 %			94 %
Tool O	100 %	59 %	91 %	98 %	100 %	53 %	62 %		80 %
Tool L	99 %	92 %	86 %	60 %	61 %	72 %	56 %	95 %	78 %
Average	98 %	88 %	86 %	82 %	82 %	80 %	73 %	95 %	

In Table 45, Tool F scored a high average precision of 96 % in the three categories it specialized in (Time and State, Code Quality, and Input Validation). Tool D was slightly less noisy (i.e., less precise) than Tool B (89 % vs. 86 %). Tool B was less noisy than Tool A (86 % vs. 82 %). Overall, all of the tools achieved a precision of 76 % to 96 %. Tools D, B, A, and H exhibited a similar profile, while the other tools presented different profiles.

For Java (Table 46), tools scored precision results ranging from 78 % to 94 %. Interestingly, Tool L generated the lowest average precision (78 %), but also by far the highest recall (34 %) and applicable recall (73 %) (Table 44). This suggests that toolmakers might have to consider a tradeoff between precision and recall.

3.3.5.5. Discrimination Rate

As noted in the previous section, one feature of the Juliet test suites is the near-symmetry between flawed (i.e., bad) and fixed (i.e., good) test cases (Sec. 3.3.1). The ratio of bad to good sites in production software is much lower than Juliet's approximately 1:1 ratio.

On real-world code, a tool that blindly reports every site, whether good or not, would score a low precision value, because good sites are preponderant. For code containing 19 good sites per bad site, precision would be 5 % (1 / (1 + 19) = 5 %).

On the Juliet test suite, however, the same tool would have a precision of about 50 % due to the near-parity between flawed and fixed code (i.e., 1:1 ratio). CAS mitigated this bias by introducing the discrimination rate metric [11, Sec. 2.3.2], which reported a true positive for a flawed test case only if a true negative was reported on the associated fixed test case.

Note that the blank cells in Table 47 indicate that a given weakness category was not supported by that tool. Also, the *Average* columns in Tables 47 and 48 contain the average discrimination rates per category. This is not the same as the average discrimination rate over the entire C/C++ track, which is summarized in Table 61 in Sec. 4. Recall that there is some overlap between categories.

Table 47 shows how well tools discriminated between good and bad test cases on the C/C++ test suite. For example, Tool F was vastly "smarter" than Tool D when analyzing Input Validation test cases (93 % vs. 36 %). Interestingly, Tools D, B, A, and H exhibited a similar profile, while the other tools were different. Note that the Environment category value determined for Tool E (0 %) was excluded from the overall average discrimination rate for that category, because Tool E had a very low recall for this category (1 %) (Table 41). The results for the Security Features category are irrelevant, because recall was very low (0 % to 3 %).

In Java (Table 48), Tools N and M surpassed the overall discrimination rate of the other participating tools. They had reported few false positives, although their average recall values had been lower (11% and 2 %, respectively). The other tools exhibited different profiles. The API Abuse category once again appeared easier for tools to detect. All tools performed similarly well for Time and State issues, with discrimination rate ranging from 72 % to 90 %.

Table 47. Discrimination Rate per Category for Synthetic C/C++.

Tool	Encap.	Env.	API	Error H.	Code Qual.	T. & S.	Input Val.	Sec. Feat.	Average
Tool F					93 %	100 %	93 %		95 %
Tool D	100 %	100 %	100 %	100 %	93 %	75 %	36 %		86 %
Tool B		93 %	95 %	88 %	88 %	64 %	75 %		84 %
Tool A	100 %	100 %	97 %	86 %	70 %	63 %	66 %		83 %
Tool H	100 %	100 %	84 %	76 %	79 %	71 %	61 %		82 %
Tool G		100 %	94 %	91 %	66 %	65 %	14 %	100 %	76 %
Tool E		0 %	100 %		93 %	67 %	91 %		70 %
Tool C	100 %		100 %	95 %	99 %	5 %	69 %	0 %	67 %
Average	100 %	99 %	96 %	89 %	85 %	64 %	63 %	50 %	

Table 48. Discrimination Rate per Category for Synthetic Java.

Tool	API	Error H.	T. & S.	Encap.	Code Qual.	Input Val.	Sec. Feat.	Env.	Average
Tool N	96 %	100 %	72 %	100 %	96 %	93 %	100 %	0 %	82 %
Tool M	98 %	100 %	88 %	100 %	66 %	100 %	0 %	0 %	69 %
Tool L	99 %	50 %	84 %	92 %	37 %	62 %	24 %	95 %	68 %
Tool O	100 %	100 %	90 %	30 %	100 %	10 %	39 %	0 %	59 %
Average	98 %	88 %	84 %	81 %	75 %	66 %	41 %	24 %	

3.3.5.6. Precision for 50 % Prevalence vs. Discrimination Rate

On the Juliet test suites, precision results were similar across all of the tools for both the C/C++ and Java tracks, whereas the discrimination rate results were not (Fig. 10 and 11). As discussed earlier, Juliet test cases were designed to have a similar number of flawed and fixed sites. Thus, discrimination rate is a better metric to differentiate tools. Note that for real-world software, most of the sites are fixed and only a small proportion of the sites are flawed, so reported precision would be very low for a tool that reports a warning for every site, flawed or not.

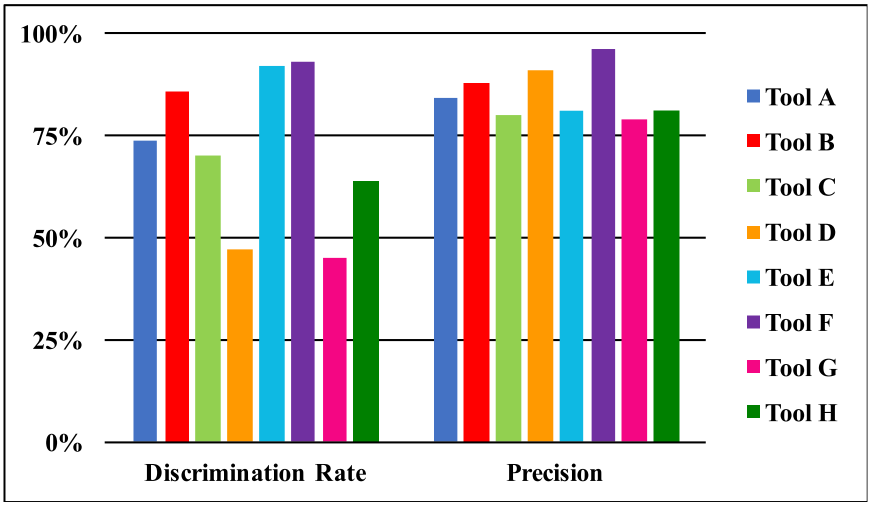

Figure 10. Precision for 50 % Prevalence vs. Discrimination Rate for Synthetic C/C++.

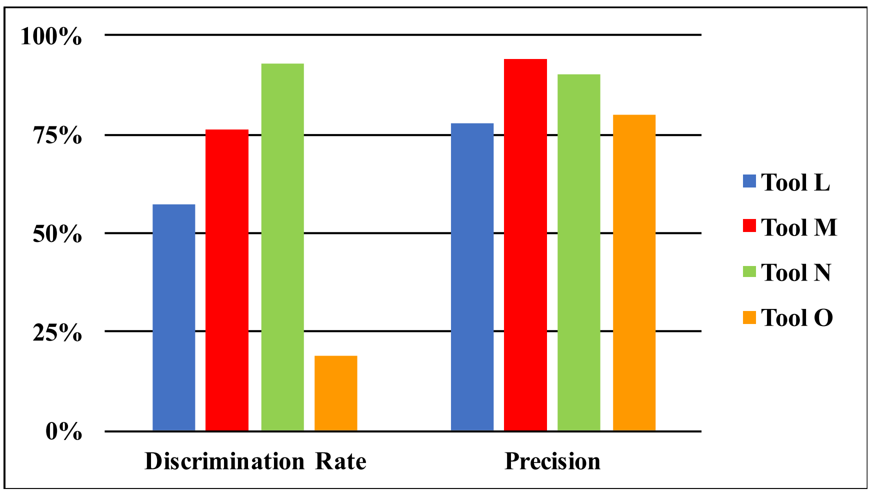

Figure 11. Precision for 50 % Prevalence vs. Discrimination Rate for Synthetic Java.

3.3.5.7. Combination of Metrics

Using a combination of metrics helps demonstrate tool efficiency. Tables 49 and 50 combine the three most significant metric results when analyzing the Juliet test suites: applicable recall, coverage and discrimination rate. These are with respect to the entire C/C++ and Java tracks.

For C/C++, Table 49 shows that Tool F exhibited the highest applicable recall and discrimination rate (56 % and 93 %, respectively), but the lowest coverage (9 %). Tool B, on the other hand, demonstrated the broadest coverage (42 %) and lower discrimination rate than that of Tool F (86 % vs. 93 %). Based upon these results, Tool B would be an effective general tool. On the other hand, Tool F emerged as an excellent specialized tool, with the best applicable recall and prime discrimination rate on a narrow band of weaknesses, as indicated by its low coverage.

For Java, Table 50 shows that Tool L reported higher values for all three metrics: applicable recall, coverage, and discrimination rate (73 %, 56 %, and 57 %, respectively). Tool M reported higher values for applicable recall and coverage (78 % and 76 %, respectively), but it demonstrated much lower coverage (25 %). Like Tool B, Tool L would be an effective general tool.

Tool N reported lower applicable recall and coverage than Tools L and M. However, because it exhibited the highest discrimination rate (93 %), Tool N would be a candidate for testing code, where noise is a significant factor.

As demonstrated, tools each have strengths and weaknesses. Using these metrics, which cover only some technical aspects of tool effectiveness, users can assess tools more objectively against their requirements and make more informed decisions.

Table 49. Applicable Recall, Coverage, and Discrimination Rate for Synthetic C/C++.

Tool	App. Recall	Coverage	Discrimination Rate
Tool A	21 %	29 %	74 %
Tool B	25 %	42 %	86 %
Tool C	18 %	22 %	70 %
Tool D	8 %	19 %	47 %
Tool E	19 %	15 %	92 %
Tool F	56 %	9 %	93 %
Tool G	2 %	35 %	45 %
Tool H	25 %	31 %	64 %

Table 50. Applicable Recall, Coverage, and Discrimination Rate for Synthetic Java.

Tool	App. Recall	Coverage	Discrimination Rate
Tool L	73 %	56 %	57 %
Tool M	78 %	25 %	76 %
Tool N	39 %	29 %	93 %
Tool O	52 %	29 %	19 %

3.3.5.8. Unreported Weaknesses

As discussed in Sec. 3.2.3.3, Arthur Hicken expressed an interest in vulnerabilities that were not reported by tools [38]. The SATE team reviewed the CWEs and divided them into two categories: those that at least one tool had found and those that remained completely unreported.

We classified a tool as supporting a particular CWE if it scored at least one true positive on the test cases for that CWE. Conversely, if a tool did not report a true positive on the test cases for that CWE, we classified it as not supporting that CWE.

Appendix E details the support of each tool for all of Juliet's CWEs. Tables 64 and 65 are divided in three sections: CWEs supported by all tools, CWEs supported by some tools and CWEs that are completely unsupported.

In C/C++, only two CWEs were reported by all eight tools: *CWE-121: Stack-based Buffer Overflow* and *CWE-457: Use of Uninitialized Variable* [10]. In Java, eleven CWEs were found by all four tools. Considering the difference in participation on both of the C/C++ and Java tracks, the diversity of the tools and the difference in the two test suites, we cannot draw any direct comparison between these two results.

The central sections of Tables 64 and 65 list CWEs that are supported by at least one tool, demonstrating that these weakness classes are within reach of static analysis. This could be an area of improvement for the tools that did not report these CWEs.

The last sections of Tables 64 and 65 contain CWEs that remained completely inscrutable for tools. Some, such as *CWE-835: Loop with Unreachable Exit Condition ('Infinite Loop')* [10], seem technically manageable and could be supported in the future. Others, like *CWE-15: External Control of System or Configuration Setting* [10], would require the user to provide context or specifications, so a tool could determine what is proper behavior and what is not.

3.3.5.9. Overlap

Overlap demonstrates how similar tools are. There was overlap when more than one tool correctly reported a weakness in a given test case. For example, if a weakness was reported by three tools, it was listed under the "3 tools" category in Table 51.

The *Test Cases Found* column provides the number of test cases found by the corresponding number of tools. The case of 0 tools gives the number of test cases missed by all tools. The *Overlap* column contains the proportions of test cases found by the corresponding number of tools. In the case of 0 tools, the *Overlap* column contains the proportion of test cases missed by all tools. The *Overlap* column demonstrates that 49 % of the C/C++ test cases and 63 % of the Java test cases went unreported by tools. Furthermore, the *Proportion Found* column, which contains the proportion of test cases that were correctly reported by the corresponding number of tools, shows that about half of the identified test cases for both C/C++ and Java test cases were reported by only one tool (50 % and 49 %, respectively). Less than a third of the findings were reported by two tools. Test cases correctly identified by more than two tools made up less than a quarter of the findings.

Considering that there was no overlap for nearly half of the findings, using multiple tools on target software can significantly increase recall. Additionally, warnings reported by two or more independent tools are more likely to be true positives.

Table 51. Overlap per Track for the Synthetic Test Cases.

Track	Participants	Number of Tools	Test Cases Found	Overlap	Proportion Found
C/C++	8	0	30 160	49 %	N/A
		1	15 663	26 %	50 %
		2	8006	13 %	26 %
		3	4279	7 %	14 %
		4	2479	4 %	8 %
		5	593	1 %	2 %
		6	191	0 %	1 %
		7	16	0 %	0 %
		8	0	0 %	0 %
Java	4	0	16 052	63 %	N/A
		1	4659	18 %	49 %
		2	2944	12 %	31 %
		3	1747	7 %	19 %
		4	75	0 %	1 %

Tables 52 and 53 detail the overlap between tool pairs. The entry in a row for tool X and column for tool Y is the proportion of weaknesses found by tool Y that is also found by tool X. Note that the tables are not symmetric, because the overlap depends on tool recall. For example, in Table 52, Tool A overlaps at 47 % with Tool B, but Tool B overlaps at 51 % with Tool A, because Tool A had a lower average recall than Tool B (17 % vs. 18 %) (Table 43). That is, Tool A found fewer defects.

On the C/C++ track, Tool B overlapped at 68 % with Tool E, indicating that about two thirds of the defects reported by Tool E were found by Tool B. Because Tool B reported higher recall than Tool E, only 30 % of its warnings overlapped with Tool E's. Tool B almost superseded Tool E with respect to recall. Moreover, Tool E could be considered a good companion to Tool B, if the goal was to increase confidence in Tool B's results by supporting its claims with Tool E's.

Tools B, H and A have similar overlap with each other of about 50 % and similar recall rates (Tables 43 and 52). Tool F, on the other hand, has little overlap with other tools despite its high recall and it stands out as an independent tool.

Table 52. Overlap between tool pairs for Synthetic C/C++.

	Tool F	Tool B	Tool H	Tool A	Tool C	Tool E	Tool D	Tool G	Recall
Tool F		16 %	18 %	25 %	29 %	15 %	13 %	15 %	20 %
Tool B	15 %		48 %	51 %	38 %	68 %	36 %	38 %	18 %
Tool H	17 %	48 %		47 %	51 %	35 %	45 %	26 %	18 %
Tool A	21 %	47 %	44 %		31 %	47 %	31 %	37 %	17 %
Tool C	15 %	22 %	30 %	19 %		13 %	32 %	10 %	10 %
Tool E	6 %	30 %	16 %	22 %	10 %		11 %	12 %	8 %
Tool D	2 %	8 %	10 %	7 %	12 %	5 %		16 %	4 %
Tool G	1 %	2 %	1 %	2 %	1 %	1 %	4 %		1 %
Recall	20 %	18 %	18 %	17 %	10 %	8 %	4 %	1 %	

On the Java track, more extreme imbalances appeared (Table 53). Tool L outperformed Tools O and N almost entirely. However, recall that Tool N had reported the highest discrimination rate (93 %) (Table 50), so one should not judge a tool solely on a single metric.

Table 53. Overlap between tool pairs for Synthetic Java.

	Tool L	Tool O	Tool N	Tool M	Recall
Tool L		94 %	87 %	43 %	34 %
Tool O	45 %		68 %	24 %	16 %
Tool N	28 %	45 %		25 %	11 %
Tool M	3 %	3 %	5 %		2 %
Recall	34 %	16 %	11 %	2 %	

3.3.5.10. Code Complexity

As one would expect, the less complex the test cases, the easier it was for tools to correctly assess them. The Juliet test suite contains four broad complexity categories. First, baseline test cases comprise the simplest weakness instances without added control or data flow complexity. Second, control flow test cases cover various control flow constructs. Third, data flow test cases cover various types of data flow constructs. Finally, data/control flow test cases combine control and data flow constructs. Note that there were a small number of data/control flow test cases in the C/C++ track and no data/control flow test cases in the Java track.

Tables 54 and 55 present, for each complexity category, the percentage of test cases found by at least one tool, as well as averages of tool recall, precision for 50 % prevalence and discrimination rate. On the C/C++ track (Table 54), tools correctly identified flaws in 67 % of the simple (i.e., non-complex) test cases. This number dropped to 58 % when control flow complexity was introduced. It was 50 % or less when data flow complexity was introduced in combination with control flow or separately.

Average recall, precision for 50 % prevalence, and discrimination rate followed the same general pattern. For each metric, the numbers were significantly lower when the test cases included data complexity.

Table 54. Effect of Code Complexity on Tool Metrics for C/C++.

Complexity	Test Cases Found	Average Recall	Average Precision	Average Discrimination Rate
None	67 %	19 %	88 %	82 %
Control	58 %	15 %	89 %	83 %
Data	44 %	8 %	78 %	64 %
Data/Control	50 %	9 %	79 %	68 %

On the Java track (Table 55), tools correctly identified 41 % of simple test cases. This number dropped slightly when control or data flow complexities were introduced. Average recall and precision for 50 % prevalence followed the same pattern, but average discrimination rate dropped significantly when data flow complexity was introduced separately.

Table 55. Effect of Code Complexity on Tool Metrics for Java.

Complexity	Test Cases Found	Average Recall	Average Precision	Average Discrimination Rate
None	41 %	9 %	79 %	68 %
Control	39 %	8 %	74 %	61 %
Data	35 %	7 %	69 %	41 %

But do some individual tools exhibit resistance to complexity? Table 56 demonstrates that most C/C++ tools found fewer defects as complexity increased. Tools C and F, however, performed consistently regardless of complexity.

Table 56. Effect of Complexity on Recall for C/C++

	None	Control	Data	Data/Control
Tool A	30 %	26 %	8 %	5 %
Tool B	27 %	26 %	11 %	3 %
Tool C	11 %	11 %	10 %	10 %
Tool D	8 %	4 %	3 %	3 %
Tool E	14 %	13 %	8 %	12 %
Tool F	21 %	22 %	26 %	27 %
Tool G	23 %	0 %	0 %	0 %
Tool H	21 %	21 %	15 %	16 %

Tools C and F indicated the same resistance with respect to discrimination rate, whereas the other tools were affected by the level of complexity (Table 57). Note that Tool E performed better when data and control flow complexities were combined. However, this could be specific to the Juliet test suite, because there were fewer test cases in this category.

Table 57. Effect of Complexity on Discrimination Rate for C/C++.

	None	Control	Data	Data/Control
Tool A	94 %	82 %	31 %	0 %
Tool B	92 %	86 %	67 %	49 %
Tool C	74 %	72 %	67 %	69 %
Tool D	80 %	54 %	34 %	32 %
Tool E	100 %	91 %	72 %	100 %
Tool F	94 %	93 %	93 %	93 %
Tool G	38 %	61 %	15 %	0 %
Tool H	81 %	77 %	43 %	45 %

On the Java track, tools performed more consistently with respect to recall (Table 58). Discrimination rate, however, was significantly impacted by the level of complexity. Tool N performed significantly better than the other tools for both types of complexities (Table 59). Since there were no true positives from Tool M on the data flow complexity test cases, the corresponding entry in Table 59 is N/A. Note that there were no data/control flow complexity test cases on the Java track, so there is no corresponding column in Tables 58 and 59.

Table 58. Effect of Complexity on Recall for Java.

	None	Control	Data
Tool L	36 %	35 %	33 %
Tool M	6 %	4 %	0 %
Tool N	13 %	12 %	11 %
Tool O	17 %	17 %	16 %

Table 59. Effect of Complexity on Discrimination Rate for Java.

	None	Control	Data
Tool L	69 %	62 %	48 %
Tool M	82 %	76 %	N/A
Tool N	88 %	99 %	82 %
Tool O	43 %	29 %	3 %

Finally, Table 60 shows the number of weaknesses initially contained in the C/C++ and Java test suites and the number remaining after all tools were run on the test cases and the reported weaknesses in the test cases were fixed. In addition, the resulting percent reduction in the number of weaknesses is displayed. In other words, the last column lists the percentage of test cases found by at least one tool, the same numbers as in the corresponding columns in Tables 54 and 55. For C/C++, static analyzers had more difficulty identifying weaknesses with respect to data flow complexity than control flow complexity.

Table 60. Reduction in the Number of Weaknesses per Complexity.

Track	Complexity	Before	After	Reduction
C/C++	None	1617	529	67 %
	Control	27 983	11 866	58 %
	Data	29 453	16 599	44 %
	Data/Control	2334	1166	50 %
Java	None	840	495	41 %
	Control	13 199	8078	39 %
	Data	11 437	7478	35 %

4. Analysis Result Summary for Classic Tracks

To summarize the results on the three types of test cases (Production Software, Software with CVEs, and Synthetic Test Cases), we compiled their metrics in Table 61. If a tool did not analyze all the test cases, the corresponding cells in Table 61 were left blank. The table was sorted by tool name, because we did not want to indicate a preference for one metric over another. We gathered coverage results only from the Synthetic test cases, because the results were not directly relevant to the other test sets.

We did not include precision for 50 % prevalence for the Synthetic test cases, since, as discussed in Sec. 3.3.5.6, discrimination rate better explains tool performance on Synthetic test cases.

Please note that grouping all results, regardless of weakness types, offers an imprecise overview of the tools' effectiveness. Ideally, we would need to use groups more granular than the 7PK to properly depict tool profiles, but at the cost of losing the bird's eye view.

On the C/C++ track, Tools A and B scored above average useful precision, coverage, and applicable recall. Tool H generated similar applicable recall and coverage results, but it reported a lower discrimination rate. Tool F achieved the best applicable recall and discrimination rate, but it had lower coverage.

On the Java track, Tool L exhibited above average coverage, applicable recall, and useful precision, but a lower discrimination rate.

On the PHP track, Tool R performed well, finding more than half the CVEs.

Table 61. Metrics per Tool in SATE V.

Track	Tool	Production	CVEs		Synthetic		
		Useful Precision	Applicable Recall	Discrimination Rate	Applicable Recall	Discrimination Rate	Coverage
C/C++	Tool A	37 %	13 %	92 %	21 %	74 %	29 %
	Tool B	47 %	6 %	83 %	25 %	86 %	42 %
	Tool C	18 %	7 %	33 %	18 %	70 %	22 %
	Tool D	23 %	0 %		8 %	47 %	19 %
	Tool E	30 %	1 %	100 %	19 %	92 %	15 %
	Tool F				56 %	93 %	9 %
	Tool G	12 %	0 %		2 %	45 %	35 %
	Tool H	26 %	14 %	46 %	25 %	64 %	31 %
	Tool I	17 %	14 %	55 %			
	Tool J	36 %	13 %	67 %			
	Tool K	68 %	0 %				
Java	Tool L	73 %	95 %	6 %	73 %	57 %	56 %
	Tool M	55 %	0 %		78 %	76 %	25 %
	Tool N	79 %	0 %		39 %	93 %	29 %
	Tool O	71 %	6 %	0 %	52 %	19 %	29 %
	Tool P	22 %	0 %				
	Tool Q	65 %	83 %	17 %			
PHP	Tool R	50 %	54 %	67 %			

Table 61 demonstrates differences between the three types of test cases, which makes generalization difficult. Later in this section, we explore this issue in more detail by considering results for groups of CWEs.

Figure 12 shows overlap distribution for the Synthetic C/C++ test cases. The figure indicates that there was very little overlap between tools, that is, the tools mostly did not report the same weaknesses.

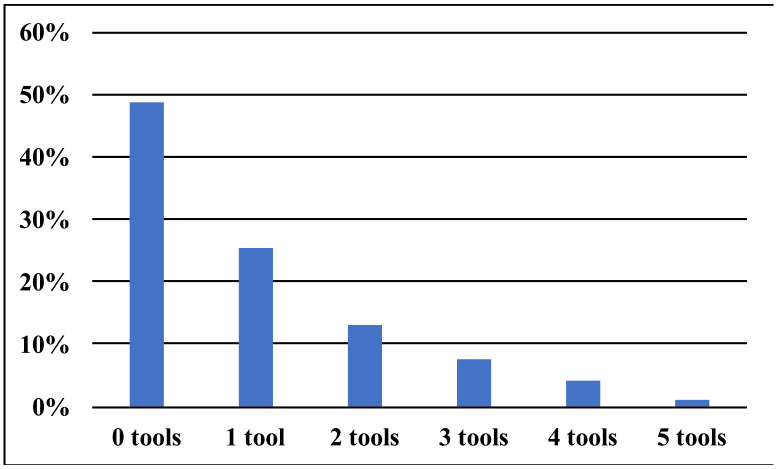

Figure 12. Overlap Distribution for Synthetic C/C++ Test Cases.

The SATE team had grouped CWEs to facilitate the analysis of the SATE V results reported by toolmakers. These varied in number and range. Table 62 lists the nine CWE groups most represented in the Synthetic and CVE-selected test cases in the C/C++ track. Most of the results were associated with buffer operations, input validation, and numeric errors. Some CWEs under the loop and recursion CWE group were easy to detect, whereas others were very difficult to detect. Consequently, these results were lower for these Synthetic test cases compared to other CWE groups.

Table 62. CWE Groups Most Represented in the CVE and Synthetic Test Cases in the C/C++ Track.

CWE Group	CVE Count	Synthetic Count
Loop and recursion	42	488
Post buffer operation	39	13 170
Numeric errors	27	7992
Ante buffer operation	21	4276
Input validation	11	9216
Invalid pointer	8	1406
Type-related	8	1384
Initialization	6	1141
Memory allocation	6	960

Figures 13 to 15 display the results for a subset of tools, which had reported results from the C/C++ track. We selected Tools B, H, and A as examples to demonstrate the differences between the recall results for the Synthetic and CVE-selected test cases. Note that the horizontal axis ends at 60 %.

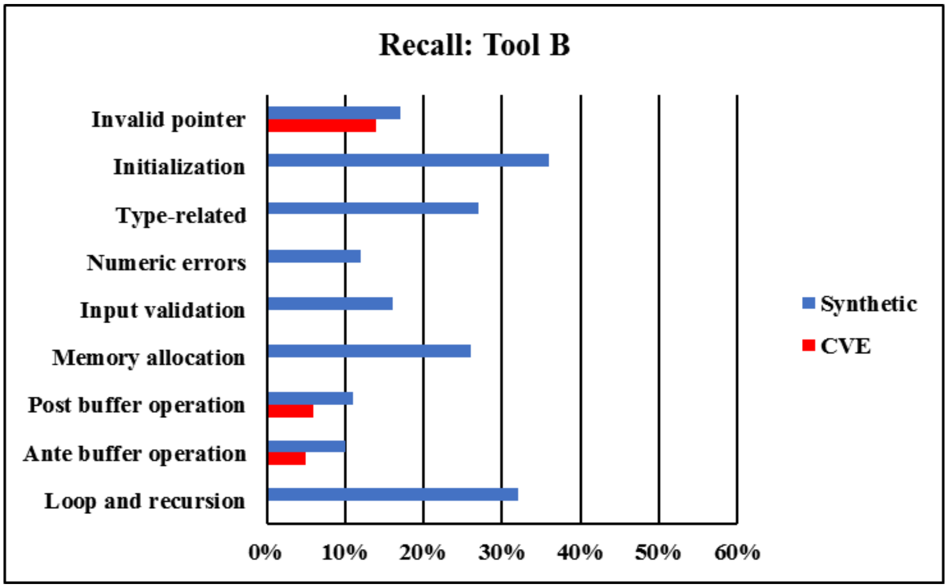

Figure 13. Recall for Synthetic vs. CVE Test Cases for Tool B in the C/C++ Track.

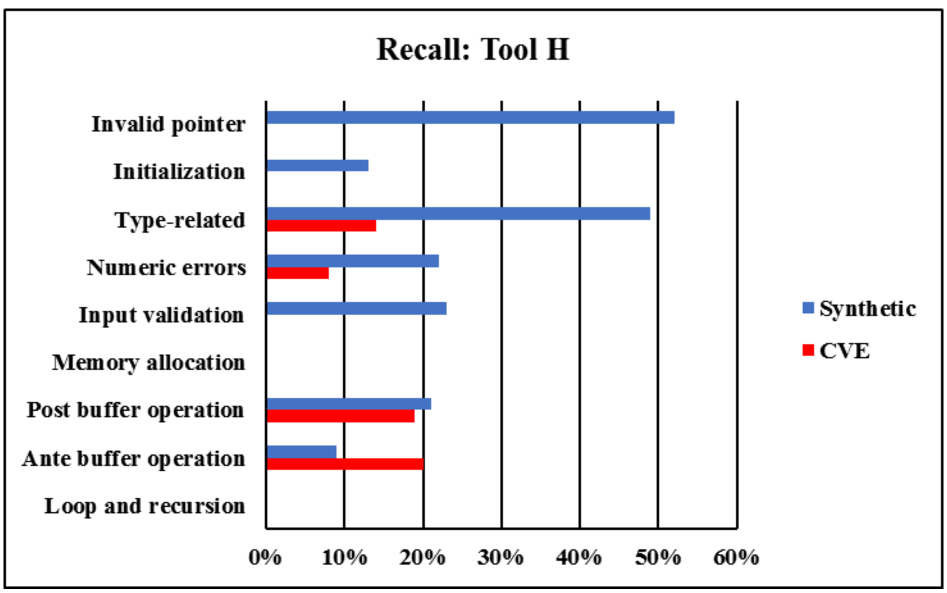

Figure 14. Recall for Synthetic vs. CVE Test Cases for Tool H in the C/C++ Track.

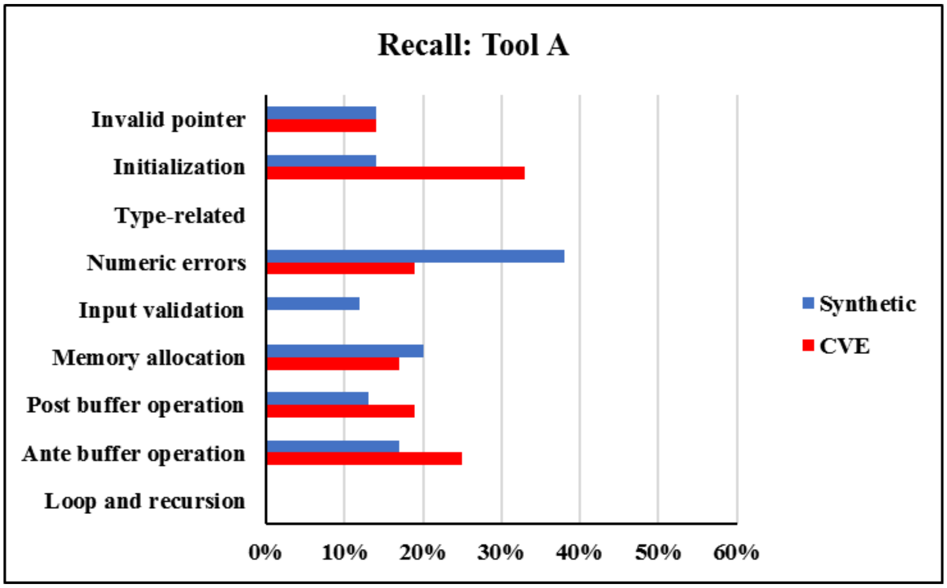

Figure 15. Recall for Synthetic vs. CVE Test Cases for Tool A in the C/C++ Track.

For the most part, recall was higher for the Synthetic test cases than for the CVE-selected test cases, probably because of the lower complexity of the Synthetic test cases. For the Production test cases, recall could not be determined due to lack of ground truth.

In summary, the differences between the three types of test cases make generalization challenging. We discussed this issue and a different approach, bug injection, that we plan to use for SATE VI, in more detail in Ref. [47].

5. Ockham Criteria

This section explains some details of SATE V Ockham Sound Analysis Criteria. The complete report is NIST-IR 8113 [48]. We introduced the Criteria in SATE V to recognize static analyzers whose findings were always correct.

Only one tool's results were submitted to be reviewed. Pascal Cuoq, Chief Scientist at Trust-in-Soft, and Florent Kirchner, Head of Laboratory at CEA, ran the August 2013 development version of Frama-C on pertinent parts of the Juliet 1.2 test suite. This section details some of the technical and theoretical challenges we addressed to evaluate Frama-C's results against the Criteria. It also describes anomalies, our observations, and interpretations.

Frama-C reports led us to discover three unintentional, systematic flaws in the Juliet 1.2 test suite, involving 416 test cases. Our conclusion is that Frama-C satisfied the SATE V Ockham Sound Analysis Criteria.

5.1. The Criteria

The Criteria is named for William of Ockham, best known for Ockham's Razor. Since the details of the Criteria will likely change in the future, the Criteria name always includes a time reference: SATE V Ockham Sound Analysis Criteria.

The value of a sound analyzer is that every one of its findings can be assumed to be correct, even if it cannot handle enormous pieces of software or does not handle dozens of weakness classes. In brief, the Criteria are:

1. The tool is claimed to be sound.
2. For at least one weakness class and one test case, the tool produces findings for a minimum of 60 % of buggy sites OR of non-buggy sites.
3. Even one incorrect finding disqualifies a tool.

An implicit criterion is that the tool is useful, not merely a toy.

We use the term *warning* to mean a single report produced by a tool. For example, `integer overflow at line 14` is a warning. A *finding* may be a warning or it may be a site with no warning. For example, a tool may be implemented to overapproximate and sometimes produce warnings about (possible) bugs at sites that are actually bug free. If it never misses a bug, then any site without a warning is sure to be correct. Toolmakers may declare that sites without warnings are findings, and that all findings are correct.

5.1.1. Details

This subsection covers the details of the Criteria. First, we give the three formal Criteria, then we follow with definitions, informative statements, and discussion.

We set requirements that communicated our intent, ruled out trivial satisfaction, and were understandable.

No manual editing of the tool output was allowed. No automated filtering specialized to a test case or to SATE V was allowed.

Criterion 1 stated, “The tool is claimed to be sound.” We used the term *sound* to mean that every finding was correct. The tool need not produce a finding for every site; that is completeness. Section 5.1.3 discusses our use of the terms “sound” and “complete.”

A tool may have settings that allow unsound analysis. The tool still qualified if it had clearly sound settings. A more inclusive statement of Criterion 1 is: “The tool is claimed to be sound or has a mode, in which analysis is sound.”

Criterion 2 deals with the number of findings produced: the tool produces findings for a minimum of 60 % of sites.

After consultation with the SATE program committee, we chose this as a level that is useful, yet achievable by current tools.

A *site* is a location in code where a weakness might occur. For example, *every buffer access in a C program is a site where buffer overflow might occur if the code is buggy. In other words, sites for a weakness are places that must be checked for that weakness.* Section 5.1.2 provides more details regarding what constitutes a site.

A *buggy site* is one that has an instance of the weakness. That is, there is some input that will cause a violation. A *non-buggy site* is one that does not have an instance of the weakness.

A *finding* is a definitive report about a site. In other words, the site has a specific weakness (is buggy) or the site does not have a specific weakness (is not buggy).

We offered SATE V test cases as Ockham test cases. Participants designated weaknesses that their tool could find and chose the test cases to use.

5.1.2. Definition of "Site"

As stated above, a *site* is a location in code where a weakness might occur. In other words, sites are places that must be checked. The determination of a site depends on local information. That is, global or flow-sensitive information is not required for determining where sites are in code.

For example, the following code comes from SARD Test Case 62 804 [20]. It has one site of writing to a buffer, `data[i] =`, which needs to be checked for a write-outside-buffer bug. There is also one site of reading from a buffer, `source[i]`, where the program might read outside the buffer if there is a bug.

```
for (i = 0; i < 10; i++)
{
     data[i] = source[i];
}
```

In addition, the code has sites of uninitialized variable, i.e., every place that `i` is used, and an integer overflow site, i.e., `i++`. Thus, the assignment statement in the body of the loop has several sites: a write buffer site, a read buffer site, and sites where variables are used.

Locations in code may be excluded as sites because of local information. For example, for the weakness class *CWE-369: Divide By Zero* [10], a simple definition of site is every occurrence of a division operator (`/`). Consider the following code fragment: `mid = height/2`. Since division by a constant other than zero is never a divide by zero and this situation can be detected easily, we may exclude division by a non-zero constant as a site for divide by zero.

5.1.3. About "Sound" and "Complete" Analysis

The terms *sound* and *complete* are used differently by different communities. The two different pairs of meanings both have valid reasons.

Most of the theorem proving, formal methods, and static analysis communities use "sound" to mean that all bugs are reported and "complete" to mean that every bug report is a correct report. In other words, sound analysis in this sense may produce false alarms (false positives), but never misses a possible problem (no false negatives). By analogous argument, complete analysis never produces false alarms, but it may miss some problems.

For the Ockham Criteria, we used "sound" to mean that every finding[7] was correct. We used "complete" to convey the meaning of a finding for every site.

5.2. Frama-C Evaluation

There was only one participant in SATE V Ockham Sound Analysis Criteria: Frama-C. Pascal Cuoq and Florent Kirchner ran the August 2013 development version. (Changes were released to the open source engine in version 20140301 "Neon.")

Frama-C is a suite of tools for analyzing software written in C [49]. It is free software licensed under the GNU Lesser General Public License (LGPL) v2.1 license[8].

By its own definition, Frama-C claimed to be sound: "it aims at being *correct*, that is, never to remain silent for a location in the source code where an error can happen at run-time" [49].

This satisfies Criterion 1.

The following general procedure was used to evaluate a tool for Criteria 2 and 3. This procedure was repeated for each weakness.

1. Decide what constitutes a site.

2. Determine the list of sites

   U = the set of all sites

3. Determine the list of findings

   F = the set of all findings

4. Check that all findings are at sites

   $$F \subseteq U \tag{4}$$

5. Determine which sites are buggy or non-buggy

   B = the set of all buggy (bad) sites

   G = the set of all non-buggy (good) sites

6. Check that

   $$|F| \geq 0.6 \times |G| \tag{5}$$

   where $|F|$ is the number of items in set F, i.e., the number of findings, and $|G|$ is the number of good sites. If that is true, Criterion 2 is satisfied.

7. Check that

[7] For Frama-C, a finding is a site that does not have a bug report. That is, it is sure that it is not buggy.

[8] http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html

$$F \cap B = \varnothing \tag{6}$$

If that is true, Criterion 3 is satisfied.

When problems or mismatches were found, we reviewed and compared the definitions of site or warning and checked for errors in our programs.

These general procedures were instantiated for Frama-C.

5.2.1. Undefined Behavior Stops Analysis

The first elaboration is for undefined states. Some situations in the C programming language have “undefined behavior,” which is more drastic than “the result may be any number.” In fact, no further analysis is reasonable. Section 5.2.5 provides more details about undefined behavior.

Frama-C issues a warning and terminates analysis when it detects that the resulting state may be undefined. Consequently, sites following a terminating failure (*T*) have no judgments made at all, neither buggy nor non-buggy. The universe of sites is, therefore, syntactic sites (*S*) **U**ntil (**U**) a terminating failure.

$$U = S\ \mathbf{U}\ T \tag{7}$$

5.2.2. Warnings Are Union of Two Runs

Pascal Cuoq and Florent Kirchner sent two files of warnings each from a different set of runs of Frama-C. One set of runs modeled that every allocation failed, and the other set of runs modeled that every allocation succeeded. Frama-C must assume allocation failure to catch a possible NULL pointer dereference, e.g., in the following code, which comes from SARD Test Case 74 328 [20]:

```
char * dataBuffer = (char *) malloc(100*sizeof(char));
            memset(dataBuffer, 'A', 100-1);
```

Because Frama-C could not model both allocation failure and allocation success in one run, *Warnings* are the union of warnings from both files.

5.2.3. Frama-C Gives Findings for Good Sites

Frama-C always warns about a bug at a site when there is a bug, i.e., there are no false negatives. Note that because of the limitations of Frama-C’s models, it may report a bug when there is no bug, i.e., there may be false alarms. Such false alarms are allowed, because for Frama-C, a finding is that a site is not buggy. If Frama-C does not produce a warning for a site, then that site is definitely not buggy. In other words, given that *W* is the set of all warnings, the set of all findings is the difference of the set of all sites and the set of all warnings:

$$F = U - W \tag{8}$$

By definition, the consistency check in Step 4, $F \subseteq U$, was trivially satisfied. However, we gained confidence by checking that all warnings are sites. Therefore, we replaced the consistency check from Step 4:

4. Check that

$$W \subseteq U \tag{9}$$

To determine buggy sites, we developed a "master list" from the comments and repeated structures in the Juliet code. This master list was produced by converters and extractors. When we found inconsistencies, we investigated and resolved them, improving the code as needed. Since findings were good sites for Frama-C, the Criteria checks were Eq. (5) and Eq. (6) from Steps 6 to 7:

6. Check that:

$$|F| \geq 0.6 \times |G|$$

7. Check that:

$$F \cap B = \varnothing$$

If that was not true, the reasons, including the definition of the site and the assignment of the warning, were investigated. Since $G = U - B$ (*and* $B \subseteq U$) [9], we rewrote Step 6, so only buggy (*B*) sites were used:

6. Check that:

$$|F| \geq 0.6 \times (|U| - |B|) \tag{10}$$

5.2.4. Implementation

We performed the bulk of the analysis with automated scripts and custom programs. The general flow was to:

1. Extract appropriate sites from the Juliet tests
2. Extract and interpret appropriate warnings from the Frama-C report
3. Match and process the two extracts in various ways

Automated scripts allowed us to rerun them with relative ease, as needed.

Some exclusions and special handling were built into the code. These are mentioned where we discuss the exclusions or special handling, e.g., Sec. 5.3.1, 5.3.5, and 5.3.7.

All of the scripts and files are available in a TAR file with XZ compression [50] at *https://s3. amazonaws.com/nist-ockham-criteria-satevdata /ockhamCriteriaSATEVdata.tar.xz*

5.2.5. Analysis Termination after `RAND32()` macro

The Juliet 1.2 test suite uses a macro, `RAND32()`, defined as follows:

[9] We need to know that $B \subseteq U$, because, in general, $|U - B| = |U| - |B| + |B - U|$. Since $B \subseteq U$, then $|B - U| = 0$ and, therefore, $|U - B| = |U| - |B|$.

```
#define RAND32() \
     ((rand()<<30) ^ (rand()<<15) ^ rand())
```

The International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC) C 2011 standard Sec. 6.5.7 Bitwise shift operators states, "If the value of the right operand is negative or is greater than or equal to the width in bits of the promoted left operand, the behavior is undefined." [51]

Frama-C models `rand()` as returning a type that is less than 30 bits. According to the standard, the result of executing a statement with `RAND32()` is undefined. Frama-C stops analyzing the code after an undefined state is encountered.

Our site extraction is largely syntactic or local, so it was difficult to exclude sites that followed undefined behavior. Given this limitation to our analysis, we completely excluded the 76 test cases, comprising a total of 112 files, that use `RAND32()`.

Frama-C produced 2101 warnings about integer overflow for many uses of left shift (<<) in `RAND32()` and `RAND64()`. These are legitimate warnings, but since they do not correspond to our weakness classes, we excluded them.

5.2.6. Cases Under *CWE-191* Not Processed

During our evaluation, we observed that there were no warnings for *CWE-191: Integer Underflow (Wrap or Wraparound)* [10] in the test cases. Upon inquiry, we learned that because of a simple human mistake, Frama-C was not run on any cases under CWE-191.

Consequently, we excluded all sites under the CWE-191 subdirectory from our analysis to avoid misinterpretations in the final results.

The developers later submitted files with the warnings. However, we did not evaluate them, since they were obtained, using a later version of Frama-C.

5.3. Evaluation by Weakness Classes

We sent a set of Juliet 1.2 test cases, containing the following CWEs to those running Frama-C:

- *CWE-121: Stack-based Buffer Overflow*
- *CWE-122: Heap-based Buffer Overflow*
- *CWE-123: Write-what-where Condition*
- *CWE-124: Buffer Underwrite ('Buffer Underflow')*
- *CWE-126: Buffer Over-read*
- *CWE-127: Buffer Under-read*
- *CWE-190: Integer Overflow or Wraparound*
- *CWE-191: Integer Underflow (Wrap or Wraparound)*

- *CWE-369: Divide by Zero*
- *CWE-457: Use of Uninitialized Variable*
- *CWE-476: NULL Pointer Dereference*
- *CWE-562: Return of Stack Variable Address*

The result we received from them contained the following nine warnings:

- division by zero
- floating-point NaN[10] or infinity
- invalid arguments to library function
- invalid memory access
- making use of address of object past its lifetime
- overflow in conversion
- passing INT_MIN to standard function abs()
- reading from uninitialized `lvalue`
- undefined arithmetic overflow

The warnings did not match simply to CWE classes, so we created nine classes of weaknesses. By examining verbose information that Frama-C supplied with each warning, we matched most warnings to one of the weakness classes. Some warnings did not fit into these classes or were not readily handled by our automatic processing. We explain some of these in Sec. 5.4.1.

Sections 5.3.1 through 5.3.8 describe each weakness class and its evaluation with respect to Criteria 2 and 3. The results for these weakness classes are summarized in Sec. 5.3.9, Table 63. Note that only eight weakness classes are discussed below, because *CWE-191* was excluded, as explained in Sec. 5.2.6.

5.3.1. Write Outside Buffer

The Write Outside Buffer weakness class includes *CWE-121: Stack-based Buffer Overflow*, *CWE-122: Heap-based Buffer Overflow*, and *CWE-124: Buffer Underwrite ('Buffer Underflow')* [10]. Frama-C did not distinguish between stack-based and heap-based buffers. For the Ockham Criteria, the distinction between stack-based and heap-based or between underflow and overflow is not important.

[10] NaN = not a number

5.3.1.1. Site Definition

This site is defined by a write to an array (buffer), either by `[]` or unary `*` operation, specifically by array access on the left-hand side of an assignment or used as a destination in a standard library function. The exception is that `memcpy()` or `memmove()` into a structure is not a site.

5.3.1.2. Anomalies, Observations, and Interpretations

The version of Frama-C that was used for the Ockham Criteria, the August 2013 development version, did not support wide string literals, e.g., `L"Good"`, nor the format specifier for wide string (`%ls`). Consequently, we excluded sites with wide string literals, the wide string format specifier, or wide character arrays passed to `printWLine()`.

5.3.1.3. Results

The results for this weakness class were: 97 678 sites (|*U*|), 18 767 warnings (|*W*|), 78 911 findings (|*F*|), and 7400 buggy sites (|*B*|).

For Write Outside Buffer, which includes *CWE-121*, *CWE-122* and *CWE-124*, Frama-C satisfied the Criteria.

5.3.2. *CWE-123: Write-what-where Condition*

The *CWE-123: Write-what-where Condition* weakness class describes the condition whereby code can be written at any location.

5.3.2.1. Site Definition

This site is defined by the use of `*`, `->`, or `[]` operators.

5.3.2.2. Results

The results for this weakness class were: 72 084 sites (|*U*|), 791 warnings (|*W*|), 71 293 findings (|*F*|), and 228 buggy sites (|*B*|).

For *CWE-123: Write-what-where Condition* [10], Frama-C satisfied the Criteria.

5.3.3. Read Outside Buffer

The Read Outside Buffer weakness class includes *CWE-126: Buffer Over-read* and *CWE-127: Buffer Under-read* [10]. Frama-C did not distinguish between read before the beginning of buffer and read after the end of buffer. For the Ockham Criteria, the difference is not important.

5.3.3.1. Site Definition

This site is defined by a read from an array (buffer), either by `[]` or unary `*`. The access could be in an expression or it could be embedded in the left-hand side of an assignment. For example, `a[b[i]] =` … reads buffer `b`.

5.3.3.2. Anomalies, Observations, and Interpretations

Some warnings dealt with an invalid argument to `printf(): invalid arguments to library function for printf`. We assigned them as Read Outside Buffer warnings, since they only happened to strings that were not null terminated that could lead `printf()` to an overread.

5.3.3.3. Results

The results for this weakness class were: 65 615 sites (|*U*|), 3396 warnings (|*W*|), 62 219 findings (|*F*|), and 2168 buggy sites (|*B*|).

For Read Outside Buffer, which includes *CWE-126* and *CWE-127* [10], Frama-C satisfied the Criteria.

5.3.4. *CWE-476: NULL Pointer Dereference*

The *CWE-476: NULL Pointer Dereference* weakness class covers NULL pointer dereference warnings.

5.3.4.1. Site Definition

This site is defined by the use of `*`, `->`, or `[]` operators.

5.3.4.2. Anomalies, Observations, and Interpretations

It was very difficult to distinguish the Frama-C warnings for this class from those for array access out-of-bounds. Therefore, we only included "invalid memory access" warnings for test cases in the *CWE-476* subdirectory.

5.3.4.3. Results

The results for this weakness class were: 72 084 sites (|*U*|), 303 warnings (|*W*|), 71 781 findings (|*F*|), and 271 buggy sites (|*B*|).

For *CWE-476: NULL Pointer Dereference* [10], Frama-C satisfied the criteria.

5.3.5. *CWE-190: Integer Overflow or Wraparound*

The *CWE-190: Integer Overflow or Wraparound* weakness class covers integer overflow warnings.

5.3.5.1.Site Definition

This site is defined by the use of `+`, `++`, `*` (multiplication), `+=`, and `*=`. This includes array indexing (and array index scaling), hence the use of `[]` is included, too. The version of Frama-C used in the Ockham Criteria only identified signed arithmetic overflows, involving types of width `int` or greater. We excluded sites from files with `_char_`, `_short_`, or `_unsigned_` in the file name. This excluded 7113 files in 4876 test cases.

5.3.5.2. Results

The results for this weakness class were: 40 570 sites (|*U*|), 1356 warnings (|*W*|), 39 214 findings (|*F*|), and 1026 buggy sites (|*B*|).

For *CWE-190: Integer Overflow or Wraparound* [10], Frama-C satisfied the Criteria.

5.3.6. *CWE-369: Divide By Zero*

The *CWE-369: Divide By Zero* weakness class is characterized by variables divided by zero.

5.3.6.1. Site Definition

This site is defined by the use of `/`, `%`, `/=`, and `%=`[11]. This includes all arithmetic types, including float and double computations. However, this does not include cases in which the right-hand side is a constant, e.g., `height/2`.

5.3.6.2. Anomalies, Observations, and Interpretations

Frama-C's implementation of abstract interpretation could not handle a range with an "omitted middle." For example, consider checking for a divide-by-zero failure in the following code fragment:

```
int x = readInput();
if (x != 0) {
     x = 1776/x;
}
```

After the first line, `x` can have any `int` value. This can be represented exactly as a range from the minimum `int` to the maximum `int`. Immediately after the `if` conditional, the possible values of `x`, that is, all values except zero, cannot be represented. One solution is to represent the possible values as the entire range. When analysis checks the next line, zero is found to be a possible value. Analysis reports a (possible) divide-by-zero, even though it is properly guarded.

The incorrect warnings and, therefore, the relatively low number of findings, were attributed to this implementation.

5.3.6.3. Results

The results for this weakness class were: 3018 sites ($|U|$), 1399 warnings ($|W|$), 1619 findings ($|F|$), and 684 buggy sites ($|B|$).

For *CWE-369: Divide By Zero* [10], Frama-C satisfied the Criteria.

5.3.7. *CWE-457: Use of Uninitialized Variable*

The *CWE-457: Use of Uninitialized Variable* weakness class covers warnings where a variable is uninitialized.

5.3.7.1. Site definition

The site is defined when the value of a variable is used. In some instances after an uninitialized variable was reported, Frama-C did not produce additional warnings. We could not determine whether this was due to an undefined program state, as explained in Sec. 5.2.5, a cleanup to avoid repeated warnings about essentially the same problem, or something else.

We handled this by only including the first buggy site in a file. That is, the first buggy site is included, and subsequent buggy sites in the same file were excluded.

[11] Juliet includes the modulo (%) operator in divide by zero.

5.3.7.2. Results

The results for this weakness class were: 263 520 sites (|*U*|), 770 warnings (|*W*|), 262 750 findings (|*F*|), and 560 buggy sites (|*B*|).

For *CWE-457: Use of Uninitialized Variable* [10], Frama-C satisfied the Criteria.

5.3.8. *CWE-562: Return of Stack Variable Address*

The *CWE-562: Return of Stack Variable Address* weakness class covers warnings of the use of stack memory after its lifetime.

5.3.8.1. Site definition

The site is defined when return statements return an expression. Return of a constant, e.g., `return 0;`, is not a site.

5.3.8.2. Anomalies, Observations, and Interpretations

There was significant mismatch between our site definition and Frama-C's warning. Our site definition was in the statement where a stack address is returned. Frama-C reported the statement where an expired address was used. Consider the following code from *CWE562_Return_of_Stack_Variable_Address__return_buf_01.c* in SARD Test Case 105 491 [20]:

```
static char *helperBad() {
     char charString[] = "helperBad string";
     return charString;
}
{
     …
     printLine(helperBad());
     …
}
```

Our extractor reported a site in the return statement, while Frama-C reported the printLine(), where the invalid address is used. Both make sense. Since only two test cases had examples of this condition, we checked them manually.

5.3.8.3. Results

The results for this weakness type were: 1838 sites (|*U*|), 2 warnings (|*W*|), 1836 findings (|*F*|), and 2 buggy sites (|*B*|).

For *CWE-562: Return of Stack Variable Address* [10], Frama-C satisfied the criteria.

5.3.9. Summary of the Evaluation by Weakness Classes

The number of sites, warnings, findings, and buggy sites for each class is given in Table 63. In the test cases selected from the Juliet 1.2 test suite, we considered a total of 616 407 sites in eight classes of weaknesses. There were a total of 12 339 buggy sites. Counting the excluded and the unclassified warnings, which are not listed above, we processed a total of 31 955 unique Frama-C warnings. Frama-C satisfied the SATE V Ockham Sound Analysis Criteria.

Table 63. Number of Sites, Warnings, Findings, and Buggy Sites for Each Weakness Class.

Class (Related CWE)	Sites (|*U*|)	Warnings (|*W*|)	Findings (|*F*|)	Buggy Sites (|*B*|)
Write Outside Buffer Condition (121, 122, 124)	97 678	18 767	78 911	7400
Write-what-where Condition (123)	72 084	791	71 293	228
Read Outside Buffer (126, 127)	65 615	3396	62 219	2168
NULL Pointer Dereference (476)	72 084	303	71 781	271
Integer Overflow (190)	40 570	1356	39 214	1026
Divide by Zero (369)	3018	1399	1619	684
Use of Uninitialized Variable (457)	263 520	770	262 750	560
Return Stack Variable Address (562)	1838	2	1836	2

5.4. General Observations

This section reports on other general observations we made while evaluating the warnings.

5.4.1. Warnings Handled as Exceptions

Frama-C produced 152 "invalid memory access" warnings, specifically invalid write, for `calloc()` when the allocation fails. We doubt that actual library code tries to zero memory if allocation fails, so we considered these warnings to be model artifacts.

Frama-C warned about constructs that occurred in four test cases. The following is the pertinent code from file *CWE476_NULL_Pointer_Dereference__int_34.c* in SARD Test Case 104 717 [20]:

```
typedef union {
     int * unionFirst;
     int * unionSecond;
} CWE476_…int_34_unionType;
…
CWE476_…int_34_unionType myUnion;
{
     int tmpData = 5;
     data = &tmpData;
}
myUnion.unionFirst = data;
{
     int *data = myUnion.unionSecond;
     printIntLine(*data);
}
```

The ISO/IEC C 2011 standard 6.5.2.3 Structure and union members, footnote 95 says, "If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type… (a process sometimes called "type punning")." [51]

This construct is well defined in the C 2011 standard. However, since other versions of the standard are not clear about how it should be treated, we believe that Frama-C was reasonable to model this as incompatible access type.

In addition to this example, Frama-C's warnings led us to discover three previously unknown, systematic errors in Juliet 1.2. These are detailed in the Ockham report [47] and the report on Juliet version 1.3 [21].

5.5. Ockham Criteria Summary

We processed a total of 31 955 unique warnings from Frama-C, covering over half a million sites in the Juliet 1.2 test suite.

The version of Frama-C that was used, the August 2013 development version, did not support wide string literals, e.g. `L"Good"`, nor the format specifier for wide string (`%ls`).

Frama-C satisfied the SATE V Ockham Sound Analysis Criteria.

5.6. Future Plans for Ockham Criteria

This section suggests changes for future Ockham Criteria.

5.6.1. Weakness Classes

Although the Ockham Criteria used the term "weakness classes," the classes are not specified. We had CWE classes in mind. In most cases Frama-C used classes of warnings that did not correspond well to CWEs. For instance, Frama-C did not distinguish between *CWE-121: Stack-based Buffer Overflow*, *CWE-122: Heap-based Buffer Overflow*, and *CWE-124: Buffer Underwrite ('Buffer Underflow')*. In general, weakness classes that tools use only approximately match CWE classes [9, Sec. 2.4].

In the future, we plan to use the weakness classes that the tools use.

For ease of information sharing, we are researching a more universal approach to characterizing weakness classes.

5.6.2. Definition of "Site"

As mentioned in Sec. 5.1.2, it is not always clear what location in a flow of execution should be a site. For instance, a function may have a few lines of code to copy a string, which have sites of read buffer and write buffer. If the code instead calls the standard library function `strcpy()`, the situation changes. If we consider sites to be within the body of `strcpy()`, then thousands of invocations throughout the code base appear to condense into a few places. In addition, the source code is probably not available.

A better definition may be that a site is the final place that the programmer can make any checks that are necessary or arrange the state properly. When the programmer invokes a library function or uses a built-in operator, the programmer must satisfy their preconditions. This may justify declaring that sites are in the main line code.

This does not address the question of what should be declared as the site of missing code, such as failure to check user input.

5.6.3. Use of the Term "Sound"

As explained in Sec. 5.1.3, the SATE V Ockham Criteria used the term "sound" and "complete" in almost the reverse sense of that used by a large, well-established formal methods community and their considerable body of published work. Although Ockham's use may have been reasonable, it would cause unnecessary and unproductive confusion for the terms to be used very differently in similar contexts. Trying to change the community's use would require a huge effort for a relatively small gain. Future Ockham Criteria should adopt a term other than "sound." Some possibilities are "correct," "flawless," "reliable," "faithful," "faultless," or "exact."

6. Workshop Outcome

On March 14, 2014, NIST welcomed participants, tool users and members of academia to the SATE V Workshop. While the organizers presented initial results, toolmakers shared their experiences in participating in SATE and tool users their practical tool use.

A few toolmakers disagreed with the rating our experts gave to some tool warnings during manual analysis. The SAMATE team concluded that the execution path leading to these weaknesses was infeasible, rating the warnings as false positives per our guidelines [8, Sec. 2.7]. Although these weaknesses were unreachable, developers had the option to fix them or not. Arguably, the warnings could have been rated as insignificant or quality-related, but the outcome would have been more subjective.

Some toolmakers reported improving their tools in the process, fulfilling one of SATE's goals. For example, Franck Cassez mentioned that Goanna improved its CWE mapping and refined its checkers.

SATE also increased the adoption of the Juliet test suite for tool assessment. The test suite offers much value, but has shortcomings. For example, Arthur Hicken mentioned an inconsistent use of memory allocation functions and an untypical amount of dead code. Pascal Cuoq also reported several bugs in the test cases. Peter Henriksen noted that the test suite did not compile out-of-the-box and argued that some test cases were too simple.

Some toolmakers expressed interest in having more tracks, such as C#, .NET, and Android.

The use of CWEs elicited some cautionary comments from some toolmakers, because many CWEs were too broad and ambiguous, and frequently misaligned with reported tool warnings. SATE's analysis automation is largely based on CWEs, but the aforementioned issues were mitigated by the use of CWE groups for matching warnings and weaknesses. Furthermore, the SAMATE team is developing the Bugs Framework, an effort to formally define weakness classes and address some of these issues [52].

An interesting point was made by Arthur Hicken about code coverage. In large software, it is not clear what code has been analyzed by tools and what has been overlooked. We witnessed this behavior in Wireshark, where some tools did not produce warnings for some dissectors. Retrospectively, it appears essential to know which parts of the code have been analyzed and which have not.

The Common Weakness Scoring System (CWSS) [53] was mentioned as a useful mechanism for tools, offering a risk-based approach to prioritizing warnings.

Arthur Hicken asserted that data integration is key to leverage the different software assurance sources (e.g., static analysis, pen-test, bug tracking, and unit tests). This means centralizing information throughout the entire software development lifecycle (SDLC), since these data sources activate at different points in time. Peter Henriksen observed that static analysis is being introduced earlier in the development process and now covers most of the SDLC, including review, testing, and actual development.

The use of the Common Coverage Representation (CCR) [35] in SATE IV and V was not met with enthusiasm by all tool makers. Some participants put serious effort in providing a CCR, but others provided a document that was incomplete, incorrect or otherwise nonexistent. CCR was judged by some as poorly designed and posing several questions. From a SATE perspective, it helped the team map tool warnings to CWEs.

The Software Assurance Marketplace (SWAMP) [34], on the other hand, was praised for its excellent work and support. The virtual machines (VMs) SWAMP had provided made test case compilation and analysis much smoother for the participants. James Kupsch presented SWAMP's role as an online laboratory for software assessment. Its centralized cloud computing platform offers a no-cost, high-performance array of open source and commercial software security testing tools, as well as a comprehensive results viewer to simplify vulnerability remediation.

The use of CVEs also brought positive feedback, although providing the details upfront (the weakness location, in particular) would have helped the toolmakers improve their analysis by checking whether their tool found the CVE and determining the cause of the shortcoming.

John Keane shared the experience with static analysis at the Department of Defense, finding that the use of automated tools by committed developers systematically leads to a reduction in security vulnerabilities and directly results in code quality improvement. He also observed that high failure rates during operational testing correlate to high security defect density and high code quality technical debt.

Nathan Ryan offered some answers as to why static analysis did not fulfill some of its past promises. Performance-wise, more complex software offset gains brought by more powerful hardware. From a technical perspective, the focus had shifted, making past expectations irrelevant. Ryan advocated that software should provide richer information to facilitate analysis and also proposed reducing computational cost by limiting inter-process analysis to where it is necessary and by favoring partial and incremental analyses. Ryan recommended pre-processing prior to analysis, enabling querying and reuse of results.

The workshop information and presentations are listed in Ref. [45].

7. Conclusion

In SATE V, we used three types of test cases to measure tool effectiveness: Production Software, CVE-selected Test Cases and a Synthetic test suite. Each type of test case offered two of three sought-after characteristics: ground truth, realism, and statistical significance. Different types of test cases enabled measurement of different metrics.

Overall results showed several ways to describe, or separate, tools: sound vs. unsound, basic vs. advanced, general vs. specialized, and security vs. quality. These dimensions help narrow down the type of tool that might fit a user's needs.

However, certain tools perform better than others of the same type. Tool effectiveness also significantly depends on the codebase, on which the tools are tested. Users can assess their candidate tools using the metrics presented in this paper and, therefore, determine the tool or tools best fitting their requirements and codebases.

Results also showed limited overlap between tool reports. The use of multiple tools can increase overall recall and boost confidence in overlapping results.

Code complexity appeared to pose the greatest challenge for advanced tools. Tools performed better on the simpler test cases of the Java and PHP tracks, as compared to the more complex test cases of the C/C++ track. Simpler CVEs were found in significant numbers, however, as complexity increased, fewer and fewer were reported. Even the Synthetic test cases showed diminishing effectiveness as code complexity increased.

Tools tended to perform better on more technical weaknesses, such as input validation and code quality. Higher level weaknesses inherent to design, such as security features, were seldom reported.

Altogether, the metrics we calculated on the three types of test cases in SATE V produced three perspectives on tool effectiveness, which could not be generalized well. Consequently, test suites, offering all three sought-after characteristics, are required. Instead of having three disparate perspectives, a unified view of tools' performance is required.

7.1 Future Plans

In SATE VI, we plan to combine the three characteristics into one test suite by exploring bug-injection. Injecting a sufficient number of realistic bugs into production software should provide ground-truth, statistical significance, and realism. We are open to using manual, assisted, and automated injection to achieve our goal.

SATE VI will be structured differently to accommodate its growth. We will combine SATE V's C/C++ and Java tracks into a new Classic track, keep Ockham as its own track, and add a Mobile track for Android applications. The PHP track will likely be abandoned, due to limited participation.

8. Acknowledgments

SATE V owes its success to many contributors. We would like to thank the Software Assurance Marketplace (SWAMP), which provided the support and infrastructure to host the SATE V virtual machines, used by the participants to analyze the test cases. SWAMP also ran a set of open source tools (Clang, PMD, and FindBugs) on our test cases to broaden our study.

Paul Anderson suggested the use of CVEs as vulnerabilities that matter for assessing the tools' capabilities in detecting and reporting real, effectual vulnerabilities.

Bill Pugh shared his vision of how a tool evaluation should be conducted without hindering innovation. He suggested the adoption of the NIST TREC model that has since been used in SATE.

Arthur Hicken expressed interest in vulnerabilities that were not reported by tools. Based on his idea, we added a new axis of research to SATE V, summarized in Sec. 3.2.3.3 and 3.3.5.8.

The Center for Assured Software (CAS) provided the community with the Juliet test suite, the largest synthetic benchmark for static analyzers. SATE IV and V used the suite extensively.

The entire analysis in SATE V was performed by the NIST SAMATE team, including Charles de Oliveira, Kamilla Holanda Crozara, and Yan Wu.

Most importantly, we want to thank the SATE V participants, some of whom have been stepping up for SATE since 2008. We recognize and appreciate their contributions in the ongoing efforts to improve software assurance.

8.1 Ockham Criteria Acknowledgements

We thank Yaacov Yesha and Irena Bojanova for their extensive comments, which greatly improved the Ockham Criteria sections. We also thank Charles de Oliveira and Christopher Long for their work in analysis. We are particularly indebted to Pascal Cuoq and Florent Kirchner, who ran Frama-C and answered many questions about interpreting the results.

9. References

[1] Larsen, G., Fong, E. K. H., Wheeler, D. A., & Moorthy, R. S. (2014, July). State-of-the-art resources (SOAR) for software vulnerability detection, test, and evaluation. Institute for Defense Analyses IDA Paper P-5061. Available: http://www.dtic.mil/dtic/tr/fulltext/u2/a607954.pdf Accessed 19 July 2018.

[2] SAMATE. (2017). Source code security analyzers (SAMATE list of static analysis tools). Available: https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html

[3] Paul E. Black, Michael Kass, Michael Koo, Elizabeth Fong, "Source Code Security Analysis Tool Functional Specification Version 1.1", NIST Special Publication 500-268 v1.1, February 2011,

https://samate.nist.gov/docs/source_code_security_analysis_spec_SP500-268_v1.1.pdf
[4] Text REtrieval Conference (TREC), http://trec.nist.gov
[5] Bill Pugh, "Judging the Value of Static Analysis", 2007, https://www.cs.umd.edu/~pugh/JudgingStaticAnalysis.pdf
[6] Okun, Vadim, Romain Gaucher, and Paul E. Black, editors, "Static Analysis Tool Exposition (SATE) 2008", NIST Special Publication 500-279, June 2009, https://samate.nist.gov/docs/NIST_Special_Publication_500-279.pdf
[7] Okun, Vadim, Aurelien Delaitre, and Paul E. Black, "The Second Static Analysis Tool Exposition (SATE) 2009", NIST Special Publication 500-287, June 2010, https://samate.nist.gov/docs/NIST_Special_Publication_500-287.pdf
[8] Okun, Vadim, Aurelien Delaitre, and Paul E. Black, editors, "Report on the Third Static Analysis Tool Exposition (SATE) 2010", NIST Special Publication 500-283, October 2011, https://dx.doi.org/10.6028/NIST.SP.500-283.
[9] Okun, Vadim, Aurelien Delaitre, and Paul E. Black, "Report on the Static Analysis Tool Exposition (SATE) IV", NIST Special Publication 500-297, January 2013, https://dx.doi.org/10.6028/NIST.SP.500-297.
[10] MITRE, "Common weakness enumeration (CWE)," https://cwe.mitre.org
[11] Center for Assured Software, U.S. National Security Agency, "CAS Static Analysis Tool Study – Methodology", December 2011, http://samate.nist.gov/docs/CAS_2011_SA_Tool_Method.pdf
[12] Kratkiewicz, K., and Lippmann, R., "Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools", Workshop on the Evaluation of Software Defect Tools, 2005
[13] Zitser, M., Lippmann, R., Leek, T., "Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code", IGSOGT Software Engineering Notes, 29(6):97-106, ACM Press, New York (2004), http://dx.doi.org/10.1145/1041685.1029911
[14] Emanuelsson, Par, and Ulf Nilsson, "A Comparative Study of Industrial Static Analysis Tools (Extended Version)", Linkoping University, Technical report 2008:3, 2008
[15] Johns, Martin and Moritz Jodeit, "Scanstud: A Methodology for Systematic, Fine-grained Evaluation of Static Analysis Tools", in Second International Workshop on Security Testing (SECTEST'11), March 2011
[16] Michaud, Frédéric and Richard Carbone, "Practical verification & safeguard tools for C/C++", DRDC Canada – Valcartier, TR 2006-735, 2007
[17] Rutar, Nick, Christian B. Almazan, and Jeffrey. S. Foster, "A Comparison of Bug Finding Tools for Java", 15th IEEE Int. Symp. on Software Reliability Eng. (ISSRE'04), France, November 2004, http://dx.doi.org/10.1109/ISSRE.2004.1
[18] MITRE, "Common Vulnerabilities and Exposures" (CVE), https://cve.mitre.org
[19] Stoneburner, G., Hayden, C., Feringa, A. (2004, June) "Engineering principles for information technology security (a baseline for achieving security), revision A: recommendations of the National Institute of Standards and Technology", National Institute of Standards and Technology Special Publication 800-27 Rev A, https://dx.doi.org/10.6028/NIST.SP.800-27rA

[20] SAMATE, "Software Assurance Reference Dataset (SARD)," https://samate.nist.gov/SARD/

[21] Black, Paul E., "Juliet 1.3 Test Suite: Changes From 1.2", June 2018, NIST Technical Note (TN) 1995, https://dx.doi.org/10.6028/NIST.TN.1995

[22] A. Wagner and J. Sametinger, "Using the Juliet Test Suite to compare static security scanners," 2014 11th International Conference on Security and Cryptography (SECRYPT), Vienna, 2014, pp. 1-9.

[23] Walden, James, Adam Messer, and Alex Kuhl, Measuring the Effect of Code Complexity on Static Analysis, International Symposium on Engineering Secure Software and Systems (ESSoS), Leuven, Belgium, February 4-6, 2009.

[24] Kupsch, J. A., & Miller, B. P. (2009). Manual vs. automated vulnerability assessment: A case study. In Proceedings of the 1st International Workshop on Managing Insider Security Threats (MIST-2009), Purdue University, West Lafayette, IN, June 15-19, 2009.

[25] De Oliveira, C., & Boland, F. (2015). Real world software assurance test suite: STONESOUP (Presentation). IEEE 27th Software Technology Conference (STC'2015) October 12-15, 2015.

[26] De Oliveira, C. D., Fong, E., & Black, P. E. (2017, February). Impact of code complexity on software analysis. NISTIR 8165. https://dx.doi.org/10.6028/NIST.IR.8165.

[27] Zheng, Jiang, Laurie Williams, Nachiappan Nagappan, Will Snipes, John. P. Hudepohl, and Mladen A. Vouk, On the Value of Static Analysis for Fault Detection in Software, IEEE Trans. on Software Engineering, v. 32, n. 4, Apr. 2006, http://dx.doi.org/10.1109/TSE.2006.38.

[28] SAMATE, "SATE V Ockham Sound Analysis Criteria", 2013, https://samate.nist.gov/SATE5OckhamCriteria.html

[29] TIOBE Software, "TIOBE Index for April 2016", 2016, http://www.tiobe.com/tiobe_index?page=index

[30] SAMATE, "SATE 2008 Data", 2008, https://samate.nist.gov/SATE2008.html

[31] SAMATE, "SATE 2009 Data", 2009, https://samate.nist.gov/SATE2009.html

[32] SAMATE, "SATE 2010 Data", 2010, https://samate.nist.gov/SATE2010.html

[33] SAMATE, "SATE IV Data", 2012, https://samate.nist.gov/SATE4.html

[34] U.S. Department of Homeland Security (DHS), "Software Assurance Marketplace" (SWAMP), 2016, https://continuousassurance.org

[35] MITRE, "Coverage Claims Representation", 2011, http://cwe.mitre.org/compatible/ccr.html

[36] Katrina Tsipenyuk, Brian Chess, and Gary McGraw, "The Seven Pernicious Kingdoms", December 2005, https://cwe.mitre.org/documents/sources/SevenPerniciousKingdoms.pdf

[37] Paul Anderson, "Truth is Subjective", 2009, SATE 2009 Workshop, https://samate.nist.gov/docs/SATE2009/SATE09%2005%20Anderson.pdf

[38] Arthur Hicken, "What We've Learned from SATE", March 2014, https://samate.nist.gov/docs/SATE5/SATE%20V%2004%20Parasoft%20Hicken.pdf

[39] Center for Assured Software, U.S. National Security Agency, "Juliet Test Suite for C/C++ v1.0", December 2010, https://samate.nist.gov/SARD/testsuites/juliet/Juliet-2010-12.c.cpp.zip

[40] Center for Assured Software, U.S. National Security Agency, "Juliet Test Suite for Java v1.0", December 2010, https://samate.nist.gov/SARD/testsuites/juliet/Juliet-2010-12.java.zip
[41] Center for Assured Software, U.S. National Security Agency, "Juliet Test Suite for C/C++ v1.2", May 2013, https://samate.nist.gov/SARD/testsuites/juliet/Juliet_Test_Suite_v1.2_for_C_Cpp.zip
[42] Center for Assured Software, U.S. National Security Agency, "Juliet Test Suite for Java v1.2", May 2013, https://samate.nist.gov/SARD/testsuites/juliet/Juliet_Test_Suite_v1.2_for_Java.zip
[43] Center for Assured Software, U.S. National Security Agency, "Juliet Test Suite v1.2 for C/C++ User Guide", December 2012, https://samate.nist.gov/SARD/resources/Juliet_Test_Suite_v1.2_for_C_Cpp_-_User_Guide.pdf
[44] Center for Assured Software, U.S. National Security Agency, "Juliet Test Suite v1.2 for Java User Guide", December 2012, https://samate.nist.gov/SARD/resources/Juliet_Test_Suite_v1.2_for_Java_-_User_Guide.pdf
[45] SAMATE, "SATE V Workshop information", 2014, https://samate.nist.gov/SATE5Workshop.html
[46] "Positive and Negative Predictive Values", Wikipedia. https://en.wikipedia.org/wiki/Positive_and_negative_predictive_values.
[47] Terry S. Cohen, Damien Cupif, Aurelien Delaitre, Charles D. De Oliveira, Elizabeth Fong, and Vadim Okun, "Improving Software Assurance through Static Analysis Tool Expositions," Journal of Cyber Security and Information Systems - Tools & Testing Techniques for Assured Software - DoD Software Assurance Community of Practice: Volume 2, 5(3):14-22, October 2017.
[48] Black, Paul E. and Athos Ribeiro, "SATE V Ockham Sound Analysis Criteria", NIST Internal Report 8113, March 2016, https://dx.doi.org/10.6028/NIST.IR.8113.
[49] What is Frama-C. Available: http://frama-c.com/what_is.html Accessed 13 July 2018.
[50] XZ Utils. Available: http://tukaani.org/xz/ Accessed 13 July 2018.
[51] "ISO/IEC 9899:2011 programming languages - C, Committee Draft - April 12, 2011 N1570)," The International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC) Joint Technical Committee JTC 1, Information technology, Subcommittee SC 22, Programming languages, their environments and system interfaces, Working Group WG 14 - C, Tech. Rep., 2011.
[52] Black, Paul E., Irena Bojanova, Yaacov Yesha, and Yan Wu (2015) "Towards a Periodic Table of Bugs", 15th High Confidence Software and Systems Conference (HCSS), May 2015, Available: http://cps-vo.org/node/19235 Accessed 13 July 2018.
[53] MITRE, "Common Weakness Scoring System", 2014, Available: http://cwe.mitre.org/cwss/ Accessed 13 July 2018.

Appendix A[12]: CWE Groups

Appendix A subdivides CWEs into 43 different CWE groups.

CWE Group	CWE #	Description
Access control	15	External Control of System or Configuration Setting
	264	Permissions, Privileges, and Access Controls
	284	Improper Access Control
	285	Improper Authorization
	377	Insecure Temporary File
	378	Creation of Temporary File With Insecure Permissions
	379	Creation of Temporary File in Directory with Incorrect Permissions
	402	Transmission of Private Resources into a New Sphere ('Resource Leak')
	403	Exposure of File Descriptor to Unintended Control Sphere ('File Descriptor Leak')
	552	Files or Directories Accessible to External Parties
	566	Authorization Bypass Through User-Controlled SQL Primary Key
	582	Array Declared Public, Final, and Static
	591	Sensitive Data Storage in Improperly Locked Memory
	607	Public Static Final Field References Mutable Object
	639	Authorization Bypass Through User-Controlled Key
	642	External Control of Critical State Data
	653	Insufficient Compartmentalization
	668	Exposure of Resource to Wrong Sphere
	732	Incorrect Permission Assignment for Critical Resource
Ante buffer operation	118	Improper Access of Indexable Resource ('Range Error')
	119	Improper Restriction of Operations within the Bounds of a Memory Buffer
	123	Write-what-where Condition
	124	Buffer Underwrite ('Buffer Underflow')
	125	Out-of-bounds Read
	127	Buffer Under-read

[12] Certain commercial equipment, instruments, or materials are identified in this paper to foster understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

CWE Group	CWE #	Description
Ante buffer operation	129	Improper Validation of Array Index
	188	Reliance on Data/Memory Layout
	466	Return of Pointer Value Outside of Expected Range
	740	CERT C Secure Coding Section 06 - Arrays (ARR)
	786	Access of Memory Location Before Start of Buffer
	787	Out-of-bounds Write
	805	Buffer Access with Incorrect Length Value
	823	Use of Out-of-range Pointer Offset
API	18	Source Code
	227	Improper Fulfillment of API Contract ('API Abuse')
	242	Use of Inherently Dangerous Function
	245	J2EE Bad Practices: Direct Management of Connections
	246	J2EE Bad Practices: Direct Use of Sockets
	249	DEPRECATED: Often Misused: Path Manipulation
	382	J2EE Bad Practices: Use of System.exit()
	383	J2EE Bad Practices: Direct Use of Threads
	440	Expected Behavior Violation
	474	Use of Function with Inconsistent Implementations
	475	Undefined Behavior for Input to API
	477	Use of Obsolete Functions
	558	Use of getlogin() in Multithreaded Application
	560	Use of umask() with chmod-style Argument
	568	finalize() Method Without super.finalize()
	572	Call to Thread run() instead of start()
	573	Improper Following of Specification by Caller
	579	J2EE Bad Practices: Non-serializable Object Stored in Session
	580	clone() Method Without super.clone()
	581	Object Model Violation: Just One of Equals and Hashcode Defined
	586	Explicit Call to Finalize()
	605	Multiple Binds to the Same Port
	676	Use of Potentially Dangerous Function
	710	Coding Standards Violation
	785	Use of Path Manipulation Function without Maximum-sized Buffer

CWE Group	CWE #	Description
Authentication	247	DEPRECATED (Duplicate): Reliance on DNS Lookups in a Security Decision
	292	DEPRECATED (Duplicate): Trusting Self-reported DNS Name
	293	Using Referer Field for Authentication
	300	Channel Accessible by Non-Endpoint ('Man-in-the-Middle')
	346	Origin Validation Error
	350	Reliance on Reverse DNS Resolution for a Security-Critical Action
	565	Reliance on Cookies without Validation and Integrity Checking
	603	Use of Client-Side Authentication
	613	Insufficient Session Expiration
	807	Reliance on Untrusted Inputs in a Security Decision
Calculation	131	Incorrect Calculation of Buffer Size
	135	Incorrect Calculation of Multi-Byte String Length
	193	Off-by-one Error
	369	Divide By Zero
	467	Use of sizeof() on a Pointer Type
	468	Incorrect Pointer Scaling
	469	Use of Pointer Subtraction to Determine Size
	682	Incorrect Calculation
	737	CERT C Secure Coding Section 03 - Expressions (EXP)
	738	CERT C Secure Coding Section 04 - Integers (INT)
	739	CERT C Secure Coding Section 05 - Floating Point (FLP)
	740	CERT C Secure Coding Section 06 - Arrays (ARR)
Cleanup	404	Improper Resource Shutdown or Release
	459	Incomplete Cleanup
	460	Improper Cleanup on Thrown Exception
	568	finalize() Method Without super.finalize()
	586	Explicit Call to Finalize()
Code quality	18	Source Code
	245	J2EE Bad Practices: Direct Management of Connections
	246	J2EE Bad Practices: Direct Use of Sockets
	382	J2EE Bad Practices: Use of System.exit()
	383	J2EE Bad Practices: Direct Use of Threads

CWE Group	CWE #	Description
Code quality	395	Use of NullPointerException Catch to Detect NULL Pointer Dereference
	396	Declaration of Catch for Generic Exception
	397	Declaration of Throws for Generic Exception
	398	Indicator of Poor Code Quality
	407	Algorithmic Complexity
	484	Omitted Break Statement in Switch
	489	Leftover Debug Code
	546	Suspicious Comment
	561	Dead Code
	563	Unused Variable
	568	finalize() Method Without super.finalize()
	570	Expression is Always False
	571	Expression is Always True
	572	Call to Thread run() instead of start()
	579	J2EE Bad Practices: Non-serializable Object Stored in Session
	580	clone() Method Without super.clone()
	581	Object Model Violation: Just One of Equals and Hashcode Defined
	585	Empty Synchronized Block
	710	Coding Standards Violation
	747	CERT C Secure Coding Section 49 - Miscellaneous (MSC)
Comparison	41	Improper Resolution of Path Equivalence
	185	Incorrect Regular Expression
	187	Partial Comparison
	478	Missing Default Case in Switch Statement
	481	Assigning instead of Comparing
	482	Comparing instead of Assigning
	486	Comparison of Classes by Name
	595	Comparison of Object References Instead of Object Contents
	596	Incorrect Semantic Object Comparison
	597	Use of Wrong Operator in String Comparison
	697	Insufficient Comparison
	747	CERT C Secure Coding Section 49 - Miscellaneous (MSC)
	768	Incorrect Short Circuit Evaluation

CWE Group	CWE #	Description
Comparison	839	Numeric Range Comparison Without Minimum Check
Concurrency	362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
	363	Race Condition Enabling Link Following
	364	Signal Handler Race Condition
	365	Race Condition in Switch
	366	Race Condition within a Thread
	367	Time-of-check Time-of-use (TOCTOU) Race Condition
	368	Context Switching Race Condition
	373	DEPRECATED: State Synchronization Error
	383	J2EE Bad Practices: Direct Use of Threads
	411	Resource Locking Problems
	413	Improper Resource Locking
	479	Signal Handler Use of a Non-reentrant Function
	543	Use of Singleton Pattern Without Synchronization in a Multithreaded Context
	557	Concurrency Issues
	558	Use of getlogin() in Multithreaded Application
	567	Unsynchronized Access to Shared Data in a Multithreaded Context
	572	Call to Thread run() instead of start()
	585	Empty Synchronized Block
	609	Double-Checked Locking
	662	Improper Synchronization
	663	Use of a Non-reentrant Function in a Concurrent Context
	667	Improper Locking
	764	Multiple Locks of a Critical Resource
	765	Multiple Unlocks of a Critical Resource
	820	Missing Synchronization
	821	Incorrect Synchronization
	832	Unlock of a Resource that is not Locked
	833	Deadlock
Confidentiality	200	Information Exposure
	204	Response Discrepancy Information Exposure
	209	Information Exposure Through an Error Message
	226	Sensitive Information Uncleared Before Release

CWE Group	CWE #	Description
Confidentiality	244	Improper Clearing of Heap Memory Before Release ('Heap Inspection')
	256	Plaintext Storage of a Password
	257	Storing Passwords in a Recoverable Format
	261	Weak Cryptography for Passwords
	300	Channel Accessible by Non-Endpoint ('Man-in-the-Middle')
	310	Cryptographic Issues
	311	Missing Encryption of Sensitive Data
	315	Cleartext Storage of Sensitive Information in a Cookie
	319	Cleartext Transmission of Sensitive Information
	325	Missing Required Cryptographic Step
	326	Inadequate Encryption Strength
	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
	359	Privacy Violation
	402	Transmission of Private Resources into a New Sphere ('Resource Leak')
	403	Exposure of File Descriptor to Unintended Control Sphere ('File Descriptor Leak')
	488	Exposure of Data Element to Wrong Session
	497	Exposure of System Data to an Unauthorized Control Sphere
	499	Serializable Class Containing Sensitive Data
	501	Trust Boundary Violation
	523	Unprotected Transport of Credentials
	525	Information Exposure Through Browser Caching
	526	Information Exposure Through Environmental Variables
	533	Information Exposure Through Server Log Files
	534	Information Exposure Through Debug Log Files
	535	Information Exposure Through Shell Error Message
	539	Information Exposure Through Persistent Cookies
	549	Missing Password Field Masking
	552	Files or Directories Accessible to External Parties

CWE Group	CWE #	Description
Confidentiality	566	Authorization Bypass Through User-Controlled SQL Primary Key
	591	Sensitive Data Storage in Improperly Locked Memory
	598	Information Exposure Through Query Strings in GET Request
	614	Sensitive Cookie in HTTPS Session Without 'Secure' Attribute
	615	Information Exposure Through Comments
	642	External Control of Critical State Data
	668	Exposure of Resource to Wrong Sphere
	756	Missing Custom Error Page
	759	Use of a One-Way Hash without a Salt
	760	Use of a One-Way Hash with a Predictable Salt
Control flow	179	Incorrect Behavior Order: Early Validation
	181	Incorrect Behavior Order: Validate Before Filter
	382	J2EE Bad Practices: Use of System.exit()
	480	Use of Incorrect Operator
	481	Assigning instead of Comparing
	482	Comparing instead of Assigning
	483	Incorrect Block Delimitation
	484	Omitted Break Statement in Switch
	583	finalize() Method Declared Public
	584	Return Inside Finally Block
	617	Reachable Assertion
	670	Always-Incorrect Control Flow Implementation
	691	Insufficient Control Flow Management
	696	Incorrect Behavior Order
	698	Execution After Redirect (EAR)
	705	Incorrect Control Flow Scoping
	768	Incorrect Short Circuit Evaluation
Credentials management	13	ASP.NET Misconfiguration: Password in Configuration File
	255	Credentials Management
	256	Plaintext Storage of a Password
	257	Storing Passwords in a Recoverable Format
	259	Use of Hard-coded Password
	260	Password in Configuration File
	261	Weak Cryptography for Passwords

CWE Group	CWE #	Description
Credentials management	523	Unprotected Transport of Credentials
	547	Use of Hard-coded, Security-relevant Constants
	555	J2EE Misconfiguration: Plaintext Password in Configuration File
	613	Insufficient Session Expiration
	620	Unverified Password Change
	798	Use of Hard-coded Credentials
Data structure	130	Improper Handling of Length Parameter Inconsistency
	137	Representation Errors
	138	Improper Neutralization of Special Elements
	170	Improper Null Termination
	188	Reliance on Data/Memory Layout
	228	Improper Handling of Syntactically Invalid Structure
	234	Failure to Handle Missing Parameter
	237	Improper Handling of Structural Elements
	238	Improper Handling of Incomplete Structural Elements
	239	Failure to Handle Incomplete Element
	240	Improper Handling of Inconsistent Structural Elements
	463	Deletion of Data Structure Sentinel
	464	Addition of Data Structure Sentinel
	588	Attempt to Access Child of a Non-structure Pointer
	707	Improper Enforcement of Message or Data Structure
Denial of Service	400	Uncontrolled Resource Consumption ('Resource Exhaustion')
	401	Improper Release of Memory Before Removing Last Reference ('Memory Leak')
	404	Improper Resource Shutdown or Release
	405	Asymmetric Resource Consumption (Amplification)
	674	Uncontrolled Recursion
	730	OWASP Top Ten 2004 Category A9 - Denial of Service
	770	Allocation of Resources Without Limits or Throttling
	776	Improper Restriction of Recursive Entity References in DTDs ('XML Entity Expansion')
Design and implementation	358	Improperly Implemented Security Check for Standard
	573	Improper Following of Specification by Caller

CWE Group	CWE #	Description
Design and implementation	657	Violation of Secure Design Principles
	693	Protection Mechanism Failure
	701	Weaknesses Introduced During Design
	710	Coding Standards Violation
Dynamic code	94	Improper Control of Generation of Code ('Code Injection')
	95	Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')
	96	Improper Neutralization of Directives in Statically Saved Code ('Static Code Injection')
	98	Improper Control of Filename for Include/Require Statement in PHP Program ('PHP Remote File Inclusion')
	434	Unrestricted Upload of File with Dangerous Type
	470	Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')
	545	Use of Dynamic Class Loading
	578	EJB Bad Practices: Use of Class Loader
	913	Improper Control of Dynamically-Managed Code Resources
Encapsulation	18	Source Code
	374	Passing Mutable Objects to an Untrusted Method
	375	Returning a Mutable Object to an Untrusted Caller
	485	Insufficient Encapsulation
	486	Comparison of Classes by Name
	488	Exposure of Data Element to Wrong Session
	489	Leftover Debug Code
	491	Public cloneable() Method Without Final ('Object Hijack')
	493	Critical Public Variable Without Final Modifier
	497	Exposure of System Data to an Unauthorized Control Sphere
	499	Serializable Class Containing Sensitive Data
	500	Public Static Field Not Marked Final
	501	Trust Boundary Violation
	545	Use of Dynamic Class Loading
	580	clone() Method Without super.clone()
	583	finalize() Method Declared Public

CWE Group	CWE #	Description
Encapsulation	607	Public Static Final Field References Mutable Object
	766	Critical Variable Declared Public
Environment induced	2	Environment
	14	Compiler Removal of Code to Clear Buffers
	15	External Control of System or Configuration Setting
	16	Configuration
	114	Process Control
	426	Untrusted Search Path
	435	Interaction Error
	436	Interpretation Conflict
	733	Compiler Optimization Removal or Modification of Security-critical Code
Error condition	18	Source Code
	388	Error Handling
	389	Error Conditions, Return Values, Status Codes
	395	Use of NullPointerException Catch to Detect NULL Pointer Dereference
	396	Declaration of Catch for Generic Exception
	397	Declaration of Throws for Generic Exception
	460	Improper Cleanup on Thrown Exception
	584	Return Inside Finally Block
	617	Reachable Assertion
	705	Incorrect Control Flow Scoping
Expired memory	415	Double Free
	416	Use After Free
	562	Return of Stack Variable Address
	742	CERT C Secure Coding Section 08 - Memory Management (MEM)
	825	Expired Pointer Dereference
Expression	480	Use of Incorrect Operator
	481	Assigning instead of Comparing
	482	Comparing instead of Assigning
	569	Expression Issues
	570	Expression is Always False
	571	Expression is Always True
	737	CERT C Secure Coding Section 03 - Expressions (EXP)

CWE Group	CWE #	Description
Expression	747	CERT C Secure Coding Section 49 - Miscellaneous (MSC)
	768	Incorrect Short Circuit Evaluation
	783	Operator Precedence Logic Error
Free of stack memory	590	Free of Memory not on the Heap
	742	CERT C Secure Coding Section 08 - Memory Management (MEM)
	825	Expired Pointer Dereference
Function call	227	Improper Fulfillment of API Contract ('API Abuse')
	573	Improper Following of Specification by Caller
	628	Function Call with Incorrectly Specified Arguments
	685	Function Call With Incorrect Number of Arguments
	686	Function Call With Incorrect Argument Type
	687	Function Call With Incorrectly Specified Argument Value
	688	Function Call With Incorrect Variable or Reference as Argument
Information loss	221	Information Loss or Omission
	222	Truncation of Security-relevant Information
	223	Omission of Security-relevant Information
	778	Insufficient Logging
Initialization	18	Source Code
	456	Missing Initialization of a Variable
	457	Use of Uninitialized Variable
	665	Improper Initialization
	736	CERT C Secure Coding Section 02 - Declarations and Initialization (DCL)
	824	Access of Uninitialized Pointer
	908	Use of Uninitialized Resource
	909	Missing Initialization of Resource
Input validation	20	Improper Input Validation
	73	External Control of File Name or Path
	74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')
	75	Failure to Sanitize Special Elements into a Different Plane (Special Element Injection)

CWE Group	CWE #	Description
Input validation	76	Improper Neutralization of Equivalent Special Elements
	77	Improper Neutralization of Special Elements used in a Command ('Command Injection')
	78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
	88	Argument Injection or Modification
	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')
	91	XML Injection (aka Blind XPath Injection)
	94	Improper Control of Generation of Code ('Code Injection')
	95	Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')
	96	Improper Neutralization of Directives in Statically Saved Code ('Static Code Injection')
	98	Improper Control of Filename for Include/Require Statement in PHP Program ('PHP Remote File Inclusion')
	99	Improper Control of Resource Identifiers ('Resource Injection')
	111	Direct Use of Unsafe JNI
	112	Missing XML Validation
	114	Process Control
	116	Improper Encoding or Escaping of Output
	117	Improper Output Neutralization for Logs
	134	Uncontrolled Format String
	138	Improper Neutralization of Special Elements
	140	Improper Neutralization of Delimiters
	141	Improper Neutralization of Parameter/Argument Delimiters
	142	Improper Neutralization of Value Delimiters
	143	Improper Neutralization of Record Delimiters
	144	Improper Neutralization of Line Delimiters
	145	Improper Neutralization of Section Delimiters
	146	Improper Neutralization of Expression/Command Delimiters

CWE Group	CWE #	Description
Input validation	147	Improper Neutralization of Input Terminators
	148	Improper Neutralization of Input Leaders
	149	Improper Neutralization of Quoting Syntax
	150	Improper Neutralization of Escape, Meta, or Control Sequences
	151	Improper Neutralization of Comment Delimiters
	152	Improper Neutralization of Macro Symbols
	153	Improper Neutralization of Substitution Characters
	154	Improper Neutralization of Variable Name Delimiters
	155	Improper Neutralization of Wildcards or Matching Symbols
	156	Improper Neutralization of Whitespace
	157	Failure to Sanitize Paired Delimiters
	158	Improper Neutralization of Null Byte or NUL Character
	159	Failure to Sanitize Special Element
	160	Improper Neutralization of Leading Special Elements
	180	Incorrect Behavior Order: Validate Before Canonicalize
	182	Collapse of Data into Unsafe Value
	228	Improper Handling of Syntactically Invalid Structure
	249	DEPRECATED: Often Misused: Path Manipulation
	470	Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')
	606	Unchecked Input for Loop Condition
	610	Externally Controlled Reference to a Resource in Another Sphere
	611	Improper Restriction of XML External Entity Reference ('XXE')
	641	Improper Restriction of Names for Files and Other Resources
	643	Improper Neutralization of Data within XPath Expressions ('XPath Injection')
	707	Improper Enforcement of Message or Data Structure
	743	CERT C Secure Coding Section 09 - Input Output (FIO)
	896	SFP Cluster: Tainted Input
	917	Improper Neutralization of Special Elements used in an Expression Language Statement ('Expression Language Injection')

CWE Group	CWE #	Description
Invalid pointer	18	Source Code
	395	Use of NullPointerException Catch to Detect NULL Pointer Dereference
	465	Pointer Issues
	466	Return of Pointer Value Outside of Expected Range
	476	NULL Pointer Dereference
	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
	763	Release of Invalid Pointer or Reference
	823	Use of Out-of-range Pointer Offset
	824	Access of Uninitialized Pointer
Loop and recursion	606	Unchecked Input for Loop Condition
	674	Uncontrolled Recursion
	776	Improper Restriction of Recursive Entity References in DTDs ('XML Entity Expansion')
	835	Loop with Unreachable Exit Condition ('Infinite Loop')
Malware-related	506	Embedded Malicious Code
	510	Trapdoor
	511	Logic/Time Bomb
	912	Hidden Functionality
Memory allocation	742	CERT C Secure Coding Section 08 - Memory Management (MEM)
	789	Uncontrolled Memory Allocation
Memory leak	401	Improper Release of Memory Before Removing Last Reference ('Memory Leak')
	742	CERT C Secure Coding Section 08 - Memory Management (MEM)
Memory release	590	Free of Memory not on the Heap
	742	CERT C Secure Coding Section 08 - Memory Management (MEM)
	761	Free of Pointer not at Start of Buffer
	762	Mismatched Memory Management Routines
	763	Release of Invalid Pointer or Reference
	891	SFP Cluster: Memory Management

CWE Group	CWE #	Description
Numeric errors	128	Wrap-around Error
	189	Numeric Errors
	190	Integer Overflow or Wraparound
	191	Integer Underflow (Wrap or Wraparound)
	192	Integer Coercion Error
	194	Unexpected Sign Extension
	195	Signed to Unsigned Conversion Error
	196	Unsigned to Signed Conversion Error
	197	Numeric Truncation Error
	680	Integer Overflow to Buffer Overflow
	681	Incorrect Conversion between Numeric Types
	682	Incorrect Calculation
	738	CERT C Secure Coding Section 04 - Integers (INT)
	739	CERT C Secure Coding Section 05 - Floating Point (FLP)
Path-related	18	Source Code
	20	Improper Input Validation
	22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
	23	Relative Path Traversal
	24	Path Traversal: '../filedir'
	25	Path Traversal: '/../filedir'
	26	Path Traversal: '/dir/../filename'
	27	Path Traversal: 'dir/../../filename'
	28	Path Traversal: '..\\filedir'
	29	Path Traversal: '\\..\\filename'
	30	Path Traversal: '\\dir\\..\\filename'
	31	Path Traversal: 'dir\\..\\..\\filename'
	32	Path Traversal: '...' (Triple Dot)
	33	Path Traversal: '....' (Multiple Dot)
	34	Path Traversal: '....//'
	35	Path Traversal: '.../...//'
	36	Absolute Path Traversal
	37	Path Traversal: '/absolute/pathname/here'
	38	Path Traversal: '\\absolute\\pathname\\here'
	39	Path Traversal: 'C:dirname'
	40	Path Traversal: '\\\\UNC\\share\\name\\' (Windows UNC Share)

CWE Group	CWE #	Description
Path-related	41	Improper Resolution of Path Equivalence
	59	Improper Link Resolution Before File Access ('Link Following')
	73	External Control of File Name or Path
	182	Collapse of Data into Unsafe Value
	249	DEPRECATED: Often Misused: Path Manipulation
	426	Untrusted Search Path
	427	Uncontrolled Search Path Element
	610	Externally Controlled Reference to a Resource in Another Sphere
	641	Improper Restriction of Names for Files and Other Resources
	706	Use of Incorrectly-Resolved Name or Reference
Post buffer operation	118	Improper Access of Indexable Resource ('Range Error')
	119	Improper Restriction of Operations within the Bounds of a Memory Buffer
	120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
	121	Stack-based Buffer Overflow
	122	Heap-based Buffer Overflow
	123	Write-what-where Condition
	125	Out-of-bounds Read
	126	Buffer Over-read
	129	Improper Validation of Array Index
	130	Improper Handling of Length Parameter Inconsistency
	135	Incorrect Calculation of Multi-Byte String Length
	170	Improper Null Termination
	188	Reliance on Data/Memory Layout
	249	DEPRECATED: Often Misused: Path Manipulation
	466	Return of Pointer Value Outside of Expected Range
	467	Use of sizeof() on a Pointer Type
	680	Integer Overflow to Buffer Overflow
	740	CERT C Secure Coding Section 06 - Arrays (ARR)
	741	CERT C Secure Coding Section 07 - Characters and Strings (STR)
	785	Use of Path Manipulation Function without Maximum-sized Buffer
	787	Out-of-bounds Write

CWE Group	CWE #	Description
Post buffer operation	788	Access of Memory Location After End of Buffer
	805	Buffer Access with Incorrect Length Value
	823	Use of Out-of-range Pointer Offset
Privileges	250	Execution with Unnecessary Privileges
	265	Privilege / Sandbox Issues
	269	Improper Privilege Management
	271	Privilege Dropping / Lowering Errors
	272	Least Privilege Violation
	273	Improper Check for Dropped Privileges
	653	Insufficient Compartmentalization
Resource management	99	Improper Control of Resource Identifiers ('Resource Injection')
	399	Resource Management Errors
	400	Uncontrolled Resource Consumption ('Resource Exhaustion')
	404	Improper Resource Shutdown or Release
	405	Asymmetric Resource Consumption (Amplification)
	413	Improper Resource Locking
	459	Incomplete Cleanup
	460	Improper Cleanup on Thrown Exception
	568	finalize() Method Without super.finalize()
	605	Multiple Binds to the Same Port
	610	Externally Controlled Reference to a Resource in Another Sphere
	664	Improper Control of a Resource Through its Lifetime
	666	Operation on Resource in Wrong Phase of Lifetime
	672	Operation on a Resource after Expiration or Release
	675	Duplicate Operations on Resource
	770	Allocation of Resources Without Limits or Throttling
	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
	826	Premature Release of Resource During Expected Lifetime
	908	Use of Uninitialized Resource
	909	Missing Initialization of Resource
Return value	252	Unchecked Return Value

CWE Group	CWE #	Description
Return value	253	Incorrect Check of Function Return Value
	273	Improper Check for Dropped Privileges
	389	Error Conditions, Return Values, Status Codes
	394	Unexpected Status Code or Return Value
	690	Unchecked Return Value to NULL Pointer Dereference
Strings	133	String Errors
	134	Uncontrolled Format String
	135	Incorrect Calculation of Multi-Byte String Length
	251	Often Misused: String Management
	597	Use of Wrong Operator in String Comparison
	741	CERT C Secure Coding Section 07 - Characters and Strings (STR)
Type-related	136	Type Errors
	195	Signed to Unsigned Conversion Error
	196	Unsigned to Signed Conversion Error
	588	Attempt to Access Child of a Non-structure Pointer
	681	Incorrect Conversion between Numeric Types
	686	Function Call With Incorrect Argument Type
	704	Incorrect Type Conversion or Cast
	747	CERT C Secure Coding Section 49 - Miscellaneous (MSC)
	843	Access of Resource Using Incompatible Type ('Type Confusion')
Undefined behavior	188	Reliance on Data/Memory Layout
	234	Failure to Handle Missing Parameter
	374	Passing Mutable Objects to an Untrusted Method
	375	Returning a Mutable Object to an Untrusted Caller
	587	Assignment of a Fixed Address to a Pointer
	588	Attempt to Access Child of a Non-structure Pointer
	758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior
Unhandled errors	248	Uncaught Exception
	273	Improper Check for Dropped Privileges
	390	Detection of Error Condition Without Action
	391	Unchecked Error Condition
	392	Missing Report of Error Condition

CWE Group	CWE #	Description
Unhandled errors	431	Missing Handler
	600	Uncaught Exception in Servlet
	703	Improper Check or Handling of Exceptional Conditions
	754	Improper Check for Unusual or Exceptional Conditions
	755	Improper Handling of Exceptional Conditions
	756	Missing Custom Error Page
Web	18	Source Code
	20	Improper Input Validation
	79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
	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
	82	Improper Neutralization of Script in Attributes of IMG Tags in a Web Page
	83	Improper Neutralization of Script in Attributes in a Web Page
	84	Improper Neutralization of Encoded URI Schemes in a Web Page
	85	Doubled Character XSS Manipulations
	86	Improper Neutralization of Invalid Characters in Identifiers in Web Pages
	87	Improper Neutralization of Alternate XSS Syntax
	113	Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')
	158	Improper Neutralization of Null Byte or NUL Character
	346	Origin Validation Error
	352	Cross-Site Request Forgery (CSRF)
	384	Session Fixation
	436	Interpretation Conflict
	472	External Control of Assumed-Immutable Web Parameter
	473	PHP External Variable Modification
	601	URL Redirection to Untrusted Site ('Open Redirect')
	611	Improper Restriction of XML External Entity Reference ('XXE')

CWE Group	CWE #	Description
Web	642	External Control of Critical State Data
	692	Incomplete Blacklist to Cross-Site Scripting
	776	Improper Restriction of Recursive Entity References in DTDs ('XML Entity Expansion')
	896	SFP Cluster: Tainted Input

Appendix B: Seven Pernicious Kingdoms

Appendix B subdivides CWEs into eight different kingdoms, using the simpler Seven Pernicious Kingdoms (7PK) classification ("seven-plus-one," which includes Environment) [36].

Kingdom	CWE #	Description
Environment	2	Environment
	3	Technology-specific Environment Issues
	4	J2EE Environment Issues
	5	J2EE Misconfiguration: Data Transmission Without Encryption
	6	J2EE Misconfiguration: Insufficient Session-ID Length
	7	J2EE Misconfiguration: Missing Custom Error Page
	8	J2EE Misconfiguration: Entity Bean Declared Remote
	9	J2EE Misconfiguration: Weak Access Permissions for EJB Methods
	10	ASP.NET Environment Issues
	11	ASP.NET Misconfiguration: Creating Debug Binary
	12	ASP.NET Misconfiguration: Missing Custom Error Page
	13	ASP.NET Misconfiguration: Password in Configuration File
	14	Compiler Removal of Code to Clear Buffers
	15	External Control of System or Configuration Setting
	188	Reliance on Data/Memory Layout
	198	Use of Incorrect Byte Ordering
	260	Password in Configuration File
	427	Uncontrolled Search Path Element
	428	Unquoted Search Path or Element
	434	Unrestricted Upload of File with Dangerous Type
	435	Interaction Error
	436	Interpretation Conflict
	437	Incomplete Model of Endpoint Features
	439	Behavioral Change in New Version or Environment
	444	Inconsistent Interpretation of HTTP Requests ('HTTP Request Smuggling')
	519	.NET Environment Issues
	520	.NET Misconfiguration: Use of Impersonation
	527	Exposure of CVS Repository to an Unauthorized Control Sphere

Kingdom	CWE #	Description
Environment	528	Exposure of Core Dump File to an Unauthorized Control Sphere
	529	Exposure of Access Control List Files to an Unauthorized Control Sphere
	530	Exposure of Backup File to an Unauthorized Control Sphere
	532	Information Exposure Through Log Files
	533	Information Exposure Through Server Log Files
	534	Information Exposure Through Debug Log Files
	538	File and Directory Information Exposure
	540	Information Exposure Through Source Code
	541	Information Exposure Through Include Source Code
	542	Information Exposure Through Cleanup Log Files
	548	Information Exposure Through Directory Listing
	552	Files or Directories Accessible to External Parties
	553	Command Shell in Externally Accessible Directory
	554	ASP.NET Misconfiguration: Not Using Input Validation Framework
	555	J2EE Misconfiguration: Plaintext Password in Configuration File
	556	ASP.NET Misconfiguration: Use of Identity Impersonation
	587	Assignment of a Fixed Address to a Pointer
	588	Attempt to Access Child of a Non-structure Pointer
	589	Call to Non-ubiquitous API
	615	Information Exposure Through Comments
	626	Null Byte Interaction Error (Poison Null Byte)
	650	Trusting HTTP Permission Methods on the Server Side
	733	Compiler Optimization Removal or Modification of Security-critical Code
	758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior
	920	Improper Restriction of Power Consumption
Error Handling	7	J2EE Misconfiguration: Missing Custom Error Page
	12	ASP.NET Misconfiguration: Missing Custom Error Page
	248	Uncaught Exception
	252	Unchecked Return Value
	253	Incorrect Check of Function Return Value
	273	Improper Check for Dropped Privileges

Kingdom	CWE #	Description
Error Handling	388	Error Handling
	389	Error Conditions, Return Values, Status Codes
	390	Detection of Error Condition Without Action
	391	Unchecked Error Condition
	392	Missing Report of Error Condition
	393	Return of Wrong Status Code
	394	Unexpected Status Code or Return Value
	395	Use of NullPointerException Catch to Detect NULL Pointer Dereference
	396	Declaration of Catch for Generic Exception
	397	Declaration of Throws for Generic Exception
	455	Non-exit on Failed Initialization
	460	Improper Cleanup on Thrown Exception
	537	Information Exposure Through Java Runtime Error Message
	544	Missing Standardized Error Handling Mechanism
	550	Information Exposure Through Server Error Message
	584	Return Inside Finally Block
	600	Uncaught Exception in Servlet
	636	Not Failing Securely ('Failing Open')
	690	Unchecked Return Value to NULL Pointer Dereference
	703	Improper Check or Handling of Exceptional Conditions
	705	Incorrect Control Flow Scoping
	754	Improper Check for Unusual or Exceptional Conditions
	755	Improper Handling of Exceptional Conditions
	756	Missing Custom Error Page
Improper Fulfillment of API Contract ('API Abuse')	102	Struts: Duplicate Validation Forms
	103	Struts: Incomplete validate() Method Definition
	104	Struts: Form Bean Does Not Extend Validation Class
	111	Direct Use of Unsafe JNI
	174	Double Decoding of the Same Data
	227	Improper Fulfillment of API Contract ('API Abuse')
	234	Failure to Handle Missing Parameter
	242	Use of Inherently Dangerous Function

Kingdom	CWE #	Description
Improper Fulfillment of API Contract ('API Abuse')	243	Creation of chroot Jail Without Changing Working Directory
	244	Improper Clearing of Heap Memory Before Release ('Heap Inspection')
	245	J2EE Bad Practices: Direct Management of Connections
	246	J2EE Bad Practices: Direct Use of Sockets
	247	DEPRECATED (Duplicate): Reliance on DNS Lookups in a Security Decision
	248	Uncaught Exception
	250	Execution with Unnecessary Privileges
	251	Often Misused: String Management
	252	Unchecked Return Value
	253	Incorrect Check of Function Return Value
	273	Improper Check for Dropped Privileges
	296	Improper Following of a Certificate's Chain of Trust
	304	Missing Critical Step in Authentication
	325	Missing Required Cryptographic Step
	329	Not Using a Random IV with CBC Mode
	350	Reliance on Reverse DNS Resolution for a Security-Critical Action
	358	Improperly Implemented Security Check for Standard
	370	Missing Check for Certificate Revocation after Initial Check
	382	J2EE Bad Practices: Use of System.exit()
	383	J2EE Bad Practices: Direct Use of Threads
	440	Expected Behavior Violation
	446	UI Discrepancy for Security Feature
	447	Unimplemented or Unsupported Feature in UI
	448	Obsolete Feature in UI
	449	The UI Performs the Wrong Action
	450	Multiple Interpretations of UI Input
	451	UI Misrepresentation of Critical Information
	462	Duplicate Key in Associative List (Alist)
	474	Use of Function with Inconsistent Implementations
	475	Undefined Behavior for Input to API
	477	Use of Obsolete Functions
	558	Use of getlogin() in Multithreaded Application

Kingdom	CWE #	Description
Improper Fulfillment of API Contract ('API Abuse')	559	Often Misused: Arguments and Parameters
	560	Use of umask() with chmod-style Argument
	568	finalize() Method Without super.finalize()
	572	Call to Thread run() instead of start()
	573	Improper Following of Specification by Caller
	574	EJB Bad Practices: Use of Synchronization Primitives
	575	EJB Bad Practices: Use of AWT Swing
	576	EJB Bad Practices: Use of Java I/O
	577	EJB Bad Practices: Use of Sockets
	578	EJB Bad Practices: Use of Class Loader
	579	J2EE Bad Practices: Non-serializable Object Stored in Session
	580	clone() Method Without super.clone()
	581	Object Model Violation: Just One of Equals and Hashcode Defined
	586	Explicit Call to Finalize()
	589	Call to Non-ubiquitous API
	605	Multiple Binds to the Same Port
	628	Function Call with Incorrectly Specified Arguments
	648	Incorrect Use of Privileged APIs
	650	Trusting HTTP Permission Methods on the Server Side
	675	Duplicate Operations on Resource
	676	Use of Potentially Dangerous Function
	683	Function Call With Incorrect Order of Arguments
	684	Incorrect Provision of Specified Functionality
	685	Function Call With Incorrect Number of Arguments
	686	Function Call With Incorrect Argument Type
	687	Function Call With Incorrectly Specified Argument Value
	688	Function Call With Incorrect Variable or Reference as Argument
	694	Use of Multiple Resources with Duplicate Identifier
	695	Use of Low-Level Functionality
	710	Coding Standards Violation
	736	CERT C Secure Coding Section 02 - Declarations and Initialization (DCL)
	742	CERT C Secure Coding Section 08 - Memory Management (MEM)

Kingdom	CWE #	Description
Improper Fulfillment of API Contract ('API Abuse')	758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior
	761	Free of Pointer not at Start of Buffer
	762	Mismatched Memory Management Routines
	763	Release of Invalid Pointer or Reference
	764	Multiple Locks of a Critical Resource
	765	Multiple Unlocks of a Critical Resource
	785	Use of Path Manipulation Function without Maximum-sized Buffer
Improper Input Validation	15	External Control of System or Configuration Setting
	20	Improper Input Validation
	21	Pathname Traversal and Equivalence Errors
	22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
	23	Relative Path Traversal
	24	Path Traversal: '../filedir'
	25	Path Traversal: '/../filedir'
	26	Path Traversal: '/dir/../filename'
	27	Path Traversal: 'dir/../../filename'
	28	Path Traversal: '..\\filedir'
	29	Path Traversal: '\\..\\filename'
	30	Path Traversal: '\\dir\\..\\filename'
	31	Path Traversal: 'dir\\..\\..\\filename'
	32	Path Traversal: '...' (Triple Dot)
	33	Path Traversal: '....' (Multiple Dot)
	34	Path Traversal: '....//'
	35	Path Traversal: '.../...//'
	36	Absolute Path Traversal
	37	Path Traversal: '/absolute/pathname/here'
	38	Path Traversal: '\\absolute\\pathname\\here'
	39	Path Traversal: 'C:dirname'
	40	Path Traversal: '\\\\UNC\\share\\name\\' (Windows UNC Share)
	41	Improper Resolution of Path Equivalence
	42	Path Equivalence: 'filename.' (Trailing Dot)

Kingdom	CWE #	Description
Improper Input Validation	43	Path Equivalence: 'filename....' (Multiple Trailing Dot)
	44	Path Equivalence: 'file.name' (Internal Dot)
	45	Path Equivalence: 'file...name' (Multiple Internal Dot)
	46	Path Equivalence: 'filename ' (Trailing Space)
	47	Path Equivalence: ' filename' (Leading Space)
	48	Path Equivalence: 'file name' (Internal Whitespace)
	49	Path Equivalence: 'filename/' (Trailing Slash)
	50	Path Equivalence: '//multiple/leading/slash'
	51	Path Equivalence: '/multiple//internal/slash'
	52	Path Equivalence: '/multiple/trailing/slash//'
	53	Path Equivalence: '\\multiple\\\\internal\\backslash'
	54	Path Equivalence: 'filedir\\' (Trailing Backslash)
	55	Path Equivalence: '/./' (Single Dot Directory)
	56	Path Equivalence: 'filedir*' (Wildcard)
	57	Path Equivalence: 'fakedir/../realdir/filename'
	58	Path Equivalence: Windows 8.3 Filename
	59	Improper Link Resolution Before File Access ('Link Following')
	60	UNIX Path Link Problems
	62	UNIX Hard Link
	63	Windows Path Link Problems
	64	Windows Shortcut Following (.LNK)
	65	Windows Hard Link
	66	Improper Handling of File Names that Identify Virtual Resources
	67	Improper Handling of Windows Device Names
	68	Windows Virtual File Problems
	69	Improper Handling of Windows ::DATA Alternate Data Stream
	70	Mac Virtual File Problems
	71	Apple '.DS_Store'
	72	Improper Handling of Apple HFS+ Alternate Data Stream Path
	73	External Control of File Name or Path
	74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')

Kingdom	CWE #	Description
Improper Input Validation	75	Failure to Sanitize Special Elements into a Different Plane (Special Element Injection)
	76	Improper Neutralization of Equivalent Special Elements
	77	Improper Neutralization of Special Elements used in a Command ('Command Injection')
	78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
	79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
	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
	82	Improper Neutralization of Script in Attributes of IMG Tags in a Web Page
	83	Improper Neutralization of Script in Attributes in a Web Page
	84	Improper Neutralization of Encoded URI Schemes in a Web Page
	85	Doubled Character XSS Manipulations
	86	Improper Neutralization of Invalid Characters in Identifiers in Web Pages
	87	Improper Neutralization of Alternate XSS Syntax
	88	Argument Injection or Modification
	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')
	91	XML Injection (aka Blind XPath Injection)
	93	Improper Neutralization of CRLF Sequences ('CRLF Injection')
	94	Improper Control of Generation of Code ('Code Injection')
	95	Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')
	96	Improper Neutralization of Directives in Statically Saved Code ('Static Code Injection')
	97	Improper Neutralization of Server-Side Includes (SSI) Within a Web Page

Kingdom	CWE #	Description
Improper Input Validation	98	Improper Control of Filename for Include/Require Statement in PHP Program ('PHP Remote File Inclusion')
	99	Improper Control of Resource Identifiers ('Resource Injection')
	100	Technology-Specific Input Validation Problems
	101	Struts Validation Problems
	102	Struts: Duplicate Validation Forms
	103	Struts: Incomplete validate() Method Definition
	104	Struts: Form Bean Does Not Extend Validation Class
	105	Struts: Form Field Without Validator
	106	Struts: Plug-in Framework not in Use
	107	Struts: Unused Validation Form
	108	Struts: Unvalidated Action Form
	109	Struts: Validator Turned Off
	110	Struts: Validator Without Form Field
	111	Direct Use of Unsafe JNI
	112	Missing XML Validation
	113	Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')
	114	Process Control
	115	Misinterpretation of Input
	116	Improper Encoding or Escaping of Output
	117	Improper Output Neutralization for Logs
	118	Improper Access of Indexable Resource ('Range Error')
	119	Improper Restriction of Operations within the Bounds of a Memory Buffer
	120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
	121	Stack-based Buffer Overflow
	122	Heap-based Buffer Overflow
	123	Write-what-where Condition
	124	Buffer Underwrite ('Buffer Underflow')
	125	Out-of-bounds Read
	126	Buffer Over-read
	127	Buffer Under-read
	128	Wrap-around Error
	129	Improper Validation of Array Index
	130	Improper Handling of Length Parameter Inconsistency

Kingdom	CWE #	Description
Improper Input Validation	131	Incorrect Calculation of Buffer Size
	133	String Errors
	134	Uncontrolled Format String
	135	Incorrect Calculation of Multi-Byte String Length
	136	Type Errors
	137	Representation Errors
	138	Improper Neutralization of Special Elements
	140	Improper Neutralization of Delimiters
	141	Improper Neutralization of Parameter/Argument Delimiters
	142	Improper Neutralization of Value Delimiters
	143	Improper Neutralization of Record Delimiters
	144	Improper Neutralization of Line Delimiters
	145	Improper Neutralization of Section Delimiters
	146	Improper Neutralization of Expression/Command Delimiters
	147	Improper Neutralization of Input Terminators
	148	Improper Neutralization of Input Leaders
	149	Improper Neutralization of Quoting Syntax
	150	Improper Neutralization of Escape, Meta, or Control Sequences
	151	Improper Neutralization of Comment Delimiters
	152	Improper Neutralization of Macro Symbols
	153	Improper Neutralization of Substitution Characters
	154	Improper Neutralization of Variable Name Delimiters
	155	Improper Neutralization of Wildcards or Matching Symbols
	156	Improper Neutralization of Whitespace
	157	Failure to Sanitize Paired Delimiters
	158	Improper Neutralization of Null Byte or NUL Character
	159	Failure to Sanitize Special Element
	160	Improper Neutralization of Leading Special Elements
	161	Improper Neutralization of Multiple Leading Special Elements
	162	Improper Neutralization of Trailing Special Elements
	163	Improper Neutralization of Multiple Trailing Special Elements
	164	Improper Neutralization of Internal Special Elements

Kingdom	CWE #	Description
Improper Input Validation	165	Improper Neutralization of Multiple Internal Special Elements
	166	Improper Handling of Missing Special Element
	167	Improper Handling of Additional Special Element
	168	Improper Handling of Inconsistent Special Elements
	169	Technology-Specific Special Elements
	170	Improper Null Termination
	171	Cleansing, Canonicalization, and Comparison Errors
	172	Encoding Error
	173	Improper Handling of Alternate Encoding
	174	Double Decoding of the Same Data
	175	Improper Handling of Mixed Encoding
	176	Improper Handling of Unicode Encoding
	177	Improper Handling of URL Encoding (Hex Encoding)
	178	Improper Handling of Case Sensitivity
	179	Incorrect Behavior Order: Early Validation
	180	Incorrect Behavior Order: Validate Before Canonicalize
	181	Incorrect Behavior Order: Validate Before Filter
	182	Collapse of Data into Unsafe Value
	183	Permissive Whitelist
	184	Incomplete Blacklist
	185	Incorrect Regular Expression
	186	Overly Restrictive Regular Expression
	187	Partial Comparison
	188	Reliance on Data/Memory Layout
	189	Numeric Errors
	190	Integer Overflow or Wraparound
	191	Integer Underflow (Wrap or Wraparound)
	192	Integer Coercion Error
	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
	198	Use of Incorrect Byte Ordering
	228	Improper Handling of Syntactically Invalid Structure
	231	Improper Handling of Extra Values

Kingdom	CWE #	Description
Improper Input Validation	234	Failure to Handle Missing Parameter
	237	Improper Handling of Structural Elements
	239	Failure to Handle Incomplete Element
	240	Improper Handling of Inconsistent Structural Elements
	351	Insufficient Type Distinction
	352	Cross-Site Request Forgery (CSRF)
	369	Divide By Zero
	386	Symbolic Name not Mapping to Correct Object
	428	Unquoted Search Path or Element
	434	Unrestricted Upload of File with Dangerous Type
	444	Inconsistent Interpretation of HTTP Requests ('HTTP Request Smuggling')
	454	External Initialization of Trusted Variables or Data Stores
	463	Deletion of Data Structure Sentinel
	464	Addition of Data Structure Sentinel
	465	Pointer Issues
	466	Return of Pointer Value Outside of Expected Range
	467	Use of sizeof() on a Pointer Type
	468	Incorrect Pointer Scaling
	469	Use of Pointer Subtraction to Determine Size
	470	Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')
	494	Download of Code Without Integrity Check
	502	Deserialization of Untrusted Data
	551	Incorrect Behavior Order: Authorization Before Parsing and Canonicalization
	554	ASP.NET Misconfiguration: Not Using Input Validation Framework
	564	SQL Injection: Hibernate
	601	URL Redirection to Untrusted Site ('Open Redirect')
	606	Unchecked Input for Loop Condition
	616	Incomplete Identification of Uploaded File Variables (PHP)
	618	Exposed Unsafe ActiveX Method
	621	Variable Extraction Error
	622	Improper Validation of Function Hook Arguments
	624	Executable Regular Expression Error

Kingdom	CWE #	Description
Improper Input Validation	625	Permissive Regular Expression
	626	Null Byte Interaction Error (Poison Null Byte)
	627	Dynamic Variable Evaluation
	641	Improper Restriction of Names for Files and Other Resources
	643	Improper Neutralization of Data within XPath Expressions ('XPath Injection')
	644	Improper Neutralization of HTTP Headers for Scripting Syntax
	646	Reliance on File Name or Extension of Externally-Supplied File
	652	Improper Neutralization of Data within XQuery Expressions ('XQuery Injection')
	680	Integer Overflow to Buffer Overflow
	681	Incorrect Conversion between Numeric Types
	682	Incorrect Calculation
	690	Unchecked Return Value to NULL Pointer Dereference
	692	Incomplete Blacklist to Cross-Site Scripting
	706	Use of Incorrectly-Resolved Name or Reference
	707	Improper Enforcement of Message or Data Structure
	738	CERT C Secure Coding Section 04 - Integers (INT)
	739	CERT C Secure Coding Section 05 - Floating Point (FLP)
	740	CERT C Secure Coding Section 06 - Arrays (ARR)
	741	CERT C Secure Coding Section 07 - Characters and Strings (STR)
	742	CERT C Secure Coding Section 08 - Memory Management (MEM)
	743	CERT C Secure Coding Section 09 - Input Output (FIO)
	747	CERT C Secure Coding Section 49 - Miscellaneous (MSC)
	777	Regular Expression without Anchors
	781	Improper Address Validation in IOCTL with METHOD_NEITHER I/O Control Code
	785	Use of Path Manipulation Function without Maximum-sized Buffer
	786	Access of Memory Location Before Start of Buffer
	787	Out-of-bounds Write
	788	Access of Memory Location After End of Buffer

Kingdom	CWE #	Description
Improper Input Validation	789	Uncontrolled Memory Allocation
	790	Improper Filtering of Special Elements
	791	Incomplete Filtering of Special Elements
	792	Incomplete Filtering of One or More Instances of Special Elements
	793	Only Filtering One Instance of a Special Element
	794	Incomplete Filtering of Multiple Instances of Special Elements
	795	Only Filtering Special Elements at a Specified Location
	796	Only Filtering Special Elements Relative to a Marker
	797	Only Filtering Special Elements at an Absolute Position
	805	Buffer Access with Incorrect Length Value
	806	Buffer Access Using Size of Source Buffer
	822	Untrusted Pointer Dereference
	823	Use of Out-of-range Pointer Offset
	838	Inappropriate Encoding for Output Context
	839	Numeric Range Comparison Without Minimum Check
	896	SFP Cluster: Tainted Input
	917	Improper Neutralization of Special Elements used in an Expression Language Statement ('Expression Language Injection')
Indicator of Poor Code Quality	107	Struts: Unused Validation Form
	110	Struts: Validator Without Form Field
	118	Improper Access of Indexable Resource ('Range Error')
	119	Improper Restriction of Operations within the Bounds of a Memory Buffer
	121	Stack-based Buffer Overflow
	122	Heap-based Buffer Overflow
	124	Buffer Underwrite ('Buffer Underflow')
	125	Out-of-bounds Read
	126	Buffer Over-read
	127	Buffer Under-read
	128	Wrap-around Error
	129	Improper Validation of Array Index
	130	Improper Handling of Length Parameter Inconsistency
	131	Incorrect Calculation of Buffer Size

Kingdom	CWE #	Description
Indicator of Poor Code Quality	136	Type Errors
	189	Numeric Errors
	190	Integer Overflow or Wraparound
	191	Integer Underflow (Wrap or Wraparound)
	192	Integer Coercion Error
	194	Unexpected Sign Extension
	195	Signed to Unsigned Conversion Error
	196	Unsigned to Signed Conversion Error
	197	Numeric Truncation Error
	252	Unchecked Return Value
	369	Divide By Zero
	398	Indicator of Poor Code Quality
	399	Resource Management Errors
	400	Uncontrolled Resource Consumption ('Resource Exhaustion')
	401	Improper Release of Memory Before Removing Last Reference ('Memory Leak')
	402	Transmission of Private Resources into a New Sphere ('Resource Leak')
	403	Exposure of File Descriptor to Unintended Control Sphere ('File Descriptor Leak')
	404	Improper Resource Shutdown or Release
	405	Asymmetric Resource Consumption (Amplification)
	406	Insufficient Control of Network Message Volume (Network Amplification)
	407	Algorithmic Complexity
	408	Incorrect Behavior Order: Early Amplification
	409	Improper Handling of Highly Compressed Data (Data Amplification)
	410	Insufficient Resource Pool
	411	Resource Locking Problems
	412	Unrestricted Externally Accessible Lock
	413	Improper Resource Locking
	414	Missing Lock Check
	415	Double Free
	416	Use After Free
	417	Channel and Path Errors

Kingdom	CWE #	Description
Indicator of Poor Code Quality	418	Channel Errors
	454	External Initialization of Trusted Variables or Data Stores
	456	Missing Initialization of a Variable
	457	Use of Uninitialized Variable
	459	Incomplete Cleanup
	460	Improper Cleanup on Thrown Exception
	465	Pointer Issues
	466	Return of Pointer Value Outside of Expected Range
	467	Use of sizeof() on a Pointer Type
	468	Incorrect Pointer Scaling
	469	Use of Pointer Subtraction to Determine Size
	474	Use of Function with Inconsistent Implementations
	475	Undefined Behavior for Input to API
	476	NULL Pointer Dereference
	477	Use of Obsolete Functions
	478	Missing Default Case in Switch Statement
	480	Use of Incorrect Operator
	481	Assigning instead of Comparing
	482	Comparing instead of Assigning
	483	Incorrect Block Delimitation
	484	Omitted Break Statement in Switch
	489	Leftover Debug Code
	546	Suspicious Comment
	547	Use of Hard-coded, Security-relevant Constants
	561	Dead Code
	562	Return of Stack Variable Address
	563	Unused Variable
	568	finalize() Method Without super.finalize()
	569	Expression Issues
	570	Expression is Always False
	571	Expression is Always True
	585	Empty Synchronized Block
	586	Explicit Call to Finalize()
	587	Assignment of a Fixed Address to a Pointer
	588	Attempt to Access Child of a Non-structure Pointer
	590	Free of Memory not on the Heap

Kingdom	CWE #	Description
Indicator of Poor Code Quality	595	Comparison of Object References Instead of Object Contents
	596	Incorrect Semantic Object Comparison
	597	Use of Wrong Operator in String Comparison
	617	Reachable Assertion
	670	Always-Incorrect Control Flow Implementation
	674	Uncontrolled Recursion
	676	Use of Potentially Dangerous Function
	681	Incorrect Conversion between Numeric Types
	682	Incorrect Calculation
	701	Weaknesses Introduced During Design
	704	Incorrect Type Conversion or Cast
	710	Coding Standards Violation
	730	OWASP Top Ten 2004 Category A9 - Denial of Service
	737	CERT C Secure Coding Section 03 - Expressions (EXP)
	738	CERT C Secure Coding Section 04 - Integers (INT)
	739	CERT C Secure Coding Section 05 - Floating Point (FLP)
	740	CERT C Secure Coding Section 06 - Arrays (ARR)
	741	CERT C Secure Coding Section 07 - Characters and Strings (STR)
	742	CERT C Secure Coding Section 08 - Memory Management (MEM)
	747	CERT C Secure Coding Section 49 - Miscellaneous (MSC)
	758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior
	761	Free of Pointer not at Start of Buffer
	762	Mismatched Memory Management Routines
	763	Release of Invalid Pointer or Reference
	769	File Descriptor Exhaustion
	770	Allocation of Resources Without Limits or Throttling
	771	Missing Reference to Active Allocated Resource
	772	Missing Release of Resource after Effective Lifetime
	773	Missing Reference to Active File Descriptor or Handle
	774	Allocation of File Descriptors or Handles Without Limits or Throttling
	775	Missing Release of File Descriptor or Handle after Effective Lifetime

Kingdom	CWE #	Description
Indicator of Poor Code Quality	776	Improper Restriction of Recursive Entity References in DTDs ('XML Entity Expansion')
	783	Operator Precedence Logic Error
	786	Access of Memory Location Before Start of Buffer
	787	Out-of-bounds Write
	788	Access of Memory Location After End of Buffer
	789	Uncontrolled Memory Allocation
	805	Buffer Access with Incorrect Length Value
	806	Buffer Access Using Size of Source Buffer
	823	Use of Out-of-range Pointer Offset
	824	Access of Uninitialized Pointer
	825	Expired Pointer Dereference
	839	Numeric Range Comparison Without Minimum Check
	843	Access of Resource Using Incompatible Type ('Type Confusion')
	891	SFP Cluster: Memory Management
	911	Improper Update of Reference Count
	912	Hidden Functionality
Insufficient Encapsulation	73	External Control of File Name or Path
	79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
	98	Improper Control of Filename for Include/Require Statement in PHP Program ('PHP Remote File Inclusion')
	200	Information Exposure
	201	Information Exposure Through Sent Data
	202	Exposure of Sensitive Data Through Data Queries
	203	Information Exposure Through Discrepancy
	204	Response Discrepancy Information Exposure
	205	Information Exposure Through Behavioral Discrepancy
	206	Information Exposure of Internal State Through Behavioral Inconsistency
	207	Information Exposure Through an External Behavioral Inconsistency
	208	Information Exposure Through Timing Discrepancy
	209	Information Exposure Through an Error Message
	210	Information Exposure Through Self-generated Error Message

Kingdom	CWE #	Description
Insufficient Encapsulation	211	Information Exposure Through Externally-generated Error Message
	212	Improper Cross-boundary Removal of Sensitive Data
	213	Intentional Information Exposure
	214	Information Exposure Through Process Environment
	215	Information Exposure Through Debug Information
	216	Containment Errors (Container Errors)
	219	Sensitive Data Under Web Root
	220	Sensitive Data Under FTP Root
	288	Authentication Bypass Using an Alternate Path or Channel
	374	Passing Mutable Objects to an Untrusted Method
	375	Returning a Mutable Object to an Untrusted Caller
	385	Covert Timing Channel
	386	Symbolic Name not Mapping to Correct Object
	402	Transmission of Private Resources into a New Sphere ('Resource Leak')
	403	Exposure of File Descriptor to Unintended Control Sphere ('File Descriptor Leak')
	417	Channel and Path Errors
	418	Channel Errors
	425	Direct Request ('Forced Browsing')
	427	Uncontrolled Search Path Element
	428	Unquoted Search Path or Element
	430	Deployment of Wrong Handler
	431	Missing Handler
	433	Unparsed Raw Web Content Delivery
	434	Unrestricted Upload of File with Dangerous Type
	441	Unintended Proxy or Intermediary ('Confused Deputy')
	454	External Initialization of Trusted Variables or Data Stores
	470	Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')
	471	Modification of Assumed-Immutable Data (MAID)
	472	External Control of Assumed-Immutable Web Parameter
	473	PHP External Variable Modification
	485	Insufficient Encapsulation
	486	Comparison of Classes by Name
	487	Reliance on Package-level Scope
	488	Exposure of Data Element to Wrong Session

Kingdom	CWE #	Description
Insufficient Encapsulation	489	Leftover Debug Code
	490	Mobile Code Issues
	491	Public cloneable() Method Without Final ('Object Hijack')
	492	Use of Inner Class Containing Sensitive Data
	493	Critical Public Variable Without Final Modifier
	494	Download of Code Without Integrity Check
	495	Private Array-Typed Field Returned From A Public Method
	496	Public Data Assigned to Private Array-Typed Field
	497	Exposure of System Data to an Unauthorized Control Sphere
	498	Cloneable Class Containing Sensitive Information
	499	Serializable Class Containing Sensitive Data
	500	Public Static Field Not Marked Final
	501	Trust Boundary Violation
	514	Covert Channel
	515	Covert Storage Channel
	524	Information Exposure Through Caching
	525	Information Exposure Through Browser Caching
	526	Information Exposure Through Environmental Variables
	527	Exposure of CVS Repository to an Unauthorized Control Sphere
	528	Exposure of Core Dump File to an Unauthorized Control Sphere
	529	Exposure of Access Control List Files to an Unauthorized Control Sphere
	530	Exposure of Backup File to an Unauthorized Control Sphere
	531	Information Exposure Through Test Code
	532	Information Exposure Through Log Files
	533	Information Exposure Through Server Log Files
	534	Information Exposure Through Debug Log Files
	535	Information Exposure Through Shell Error Message
	536	Information Exposure Through Servlet Runtime Error Message
	537	Information Exposure Through Java Runtime Error Message
	538	File and Directory Information Exposure

Kingdom	CWE #	Description
Insufficient Encapsulation	539	Information Exposure Through Persistent Cookies
	540	Information Exposure Through Source Code
	541	Information Exposure Through Include Source Code
	542	Information Exposure Through Cleanup Log Files
	545	Use of Dynamic Class Loading
	548	Information Exposure Through Directory Listing
	550	Information Exposure Through Server Error Message
	552	Files or Directories Accessible to External Parties
	553	Command Shell in Externally Accessible Directory
	567	Unsynchronized Access to Shared Data in a Multithreaded Context
	580	clone() Method Without super.clone()
	582	Array Declared Public, Final, and Static
	583	finalize() Method Declared Public
	591	Sensitive Data Storage in Improperly Locked Memory
	594	J2EE Framework: Saving Unserializable Objects to Disk
	598	Information Exposure Through Query Strings in GET Request
	601	URL Redirection to Untrusted Site ('Open Redirect')
	602	Client-Side Enforcement of Server-Side Security
	603	Use of Client-Side Authentication
	607	Public Static Final Field References Mutable Object
	608	Struts: Non-private Field in ActionForm Class
	610	Externally Controlled Reference to a Resource in Another Sphere
	611	Improper Restriction of XML External Entity Reference ('XXE')
	612	Information Exposure Through Indexing of Private Data
	618	Exposed Unsafe ActiveX Method
	619	Dangling Database Cursor ('Cursor Injection')
	621	Variable Extraction Error
	623	Unsafe ActiveX Control Marked Safe For Scripting
	627	Dynamic Variable Evaluation
	651	Information Exposure Through WSDL File
	653	Insufficient Compartmentalization
	668	Exposure of Resource to Wrong Sphere
	669	Incorrect Resource Transfer Between Spheres

Kingdom	CWE #	Description
Insufficient Encapsulation	673	External Influence of Sphere Definition
	706	Use of Incorrectly-Resolved Name or Reference
	732	Incorrect Permission Assignment for Critical Resource
	749	Exposed Dangerous Method or Function
	766	Critical Variable Declared Public
	767	Access to Critical Private Variable via Public Method
	782	Exposed IOCTL with Insufficient Access Control
	827	Improper Control of Document Type Definition
	829	Inclusion of Functionality from Untrusted Control Sphere
	830	Inclusion of Web Functionality from an Untrusted Source
	913	Improper Control of Dynamically-Managed Code Resources
	914	Improper Control of Dynamically-Identified Variables
	915	Improperly Controlled Modification of Dynamically-Determined Object Attributes
	918	Server-Side Request Forgery (SSRF)
Security Features	5	J2EE Misconfiguration: Data Transmission Without Encryption
	6	J2EE Misconfiguration: Insufficient Session-ID Length
	9	J2EE Misconfiguration: Weak Access Permissions for EJB Methods
	13	ASP.NET Misconfiguration: Password in Configuration File
	171	Cleansing, Canonicalization, and Comparison Errors
	183	Permissive Whitelist
	184	Incomplete Blacklist
	221	Information Loss or Omission
	222	Truncation of Security-relevant Information
	223	Omission of Security-relevant Information
	224	Obscured Security-relevant Information by Alternate Name
	226	Sensitive Information Uncleared Before Release
	247	DEPRECATED (Duplicate): Reliance on DNS Lookups in a Security Decision
	250	Execution with Unnecessary Privileges
	254	Security Features
	255	Credentials Management
	256	Plaintext Storage of a Password

Kingdom	CWE #	Description
Security Features	257	Storing Passwords in a Recoverable Format
	258	Empty Password in Configuration File
	259	Use of Hard-coded Password
	260	Password in Configuration File
	261	Weak Cryptography for Passwords
	262	Not Using Password Aging
	263	Password Aging with Long Expiration
	264	Permissions, Privileges, and Access Controls
	265	Privilege / Sandbox Issues
	266	Incorrect Privilege Assignment
	267	Privilege Defined With Unsafe Actions
	268	Privilege Chaining
	269	Improper Privilege Management
	270	Privilege Context Switching Error
	271	Privilege Dropping / Lowering Errors
	272	Least Privilege Violation
	273	Improper Check for Dropped Privileges
	274	Improper Handling of Insufficient Privileges
	275	Permission Issues
	276	Incorrect Default Permissions
	277	Insecure Inherited Permissions
	278	Insecure Preserved Inherited Permissions
	279	Incorrect Execution-Assigned Permissions
	280	Improper Handling of Insufficient Permissions or Privileges
	281	Improper Preservation of Permissions
	282	Improper Ownership Management
	283	Unverified Ownership
	284	Improper Access Control
	285	Improper Authorization
	287	Improper Authentication
	288	Authentication Bypass Using an Alternate Path or Channel
	289	Authentication Bypass by Alternate Name
	290	Authentication Bypass by Spoofing
	291	Reliance on IP Address for Authentication
	293	Using Referer Field for Authentication
	294	Authentication Bypass by Capture-replay

Kingdom	CWE #	Description
Security Features	295	Improper Certificate Validation
	296	Improper Following of a Certificate's Chain of Trust
	297	Improper Validation of Certificate with Host Mismatch
	298	Improper Validation of Certificate Expiration
	299	Improper Check for Certificate Revocation
	300	Channel Accessible by Non-Endpoint ('Man-in-the-Middle')
	301	Reflection Attack in an Authentication Protocol
	302	Authentication Bypass by Assumed-Immutable Data
	303	Incorrect Implementation of Authentication Algorithm
	304	Missing Critical Step in Authentication
	305	Authentication Bypass by Primary Weakness
	306	Missing Authentication for Critical Function
	307	Improper Restriction of Excessive Authentication Attempts
	308	Use of Single-factor Authentication
	309	Use of Password System for Primary Authentication
	310	Cryptographic Issues
	311	Missing Encryption of Sensitive Data
	312	Cleartext Storage of Sensitive Information
	313	Cleartext Storage in a File or on Disk
	314	Cleartext Storage in the Registry
	315	Cleartext Storage of Sensitive Information in a Cookie
	316	Cleartext Storage of Sensitive Information in Memory
	317	Cleartext Storage of Sensitive Information in GUI
	318	Cleartext Storage of Sensitive Information in Executable
	319	Cleartext Transmission of Sensitive Information
	320	Key Management Errors
	321	Use of Hard-coded Cryptographic Key
	322	Key Exchange without Entity Authentication
	323	Reusing a Nonce, Key Pair in Encryption
	324	Use of a Key Past its Expiration Date
	325	Missing Required Cryptographic Step
	326	Inadequate Encryption Strength
	327	Use of a Broken or Risky Cryptographic Algorithm
	328	Reversible One-Way Hash
	329	Not Using a Random IV with CBC Mode

Kingdom	CWE #	Description
Security Features	330	Use of Insufficiently Random Values
	331	Insufficient Entropy
	332	Insufficient Entropy in PRNG
	333	Improper Handling of Insufficient Entropy in TRNG
	334	Small Space of Random Values
	335	PRNG Seed Error
	336	Same Seed in PRNG
	337	Predictable Seed in PRNG
	338	Use of Cryptographically Weak PRNG
	339	Small Seed Space in PRNG
	340	Predictability Problems
	341	Predictable from Observable State
	342	Predictable Exact Value from Previous Values
	343	Predictable Value Range from Previous Values
	344	Use of Invariant Value in Dynamically Changing Context
	345	Insufficient Verification of Data Authenticity
	346	Origin Validation Error
	347	Improper Verification of Cryptographic Signature
	348	Use of Less Trusted Source
	349	Acceptance of Extraneous Untrusted Data With Trusted Data
	350	Reliance on Reverse DNS Resolution for a Security-Critical Action
	351	Insufficient Type Distinction
	353	Missing Support for Integrity Check
	354	Improper Validation of Integrity Check Value
	355	User Interface Security Issues
	356	Product UI does not Warn User of Unsafe Actions
	357	Insufficient UI Warning of Dangerous Operations
	358	Improperly Implemented Security Check for Standard
	359	Privacy Violation
	360	Trust of System Event Data
	370	Missing Check for Certificate Revocation after Initial Check
	372	Incomplete Internal State Distinction
	384	Session Fixation
	385	Covert Timing Channel
	412	Unrestricted Externally Accessible Lock

Kingdom	CWE #	Description
Security Features	417	Channel and Path Errors
	418	Channel Errors
	419	Unprotected Primary Channel
	420	Unprotected Alternate Channel
	421	Race Condition During Access to Alternate Channel
	422	Unprotected Windows Messaging Channel ('Shatter')
	424	Improper Protection of Alternate Path
	425	Direct Request ('Forced Browsing')
	446	UI Discrepancy for Security Feature
	447	Unimplemented or Unsupported Feature in UI
	450	Multiple Interpretations of UI Input
	451	UI Misrepresentation of Critical Information
	453	Insecure Default Variable Initialization
	454	External Initialization of Trusted Variables or Data Stores
	511	Logic/Time Bomb
	514	Covert Channel
	515	Covert Storage Channel
	520	.NET Misconfiguration: Use of Impersonation
	521	Weak Password Requirements
	522	Insufficiently Protected Credentials
	523	Unprotected Transport of Credentials
	547	Use of Hard-coded, Security-relevant Constants
	549	Missing Password Field Masking
	551	Incorrect Behavior Order: Authorization Before Parsing and Canonicalization
	555	J2EE Misconfiguration: Plaintext Password in Configuration File
	556	ASP.NET Misconfiguration: Use of Identity Impersonation
	565	Reliance on Cookies without Validation and Integrity Checking
	566	Authorization Bypass Through User-Controlled SQL Primary Key
	592	Authentication Bypass Issues
	593	Authentication Bypass: OpenSSL CTX Object Modified after SSL Objects are Created
	599	Missing Validation of OpenSSL Certificate
	602	Client-Side Enforcement of Server-Side Security

Kingdom	CWE #	Description
Security Features	603	Use of Client-Side Authentication
	613	Insufficient Session Expiration
	614	Sensitive Cookie in HTTPS Session Without 'Secure' Attribute
	616	Incomplete Identification of Uploaded File Variables (PHP)
	618	Exposed Unsafe ActiveX Method
	620	Unverified Password Change
	623	Unsafe ActiveX Control Marked Safe For Scripting
	625	Permissive Regular Expression
	636	Not Failing Securely ('Failing Open')
	638	Not Using Complete Mediation
	639	Authorization Bypass Through User-Controlled Key
	640	Weak Password Recovery Mechanism for Forgotten Password
	645	Overly Restrictive Account Lockout Mechanism
	646	Reliance on File Name or Extension of Externally-Supplied File
	647	Use of Non-Canonical URL Paths for Authorization Decisions
	648	Incorrect Use of Privileged APIs
	649	Reliance on Obfuscation or Encryption of Security-Relevant Inputs without Integrity Checking
	654	Reliance on a Single Factor in a Security Decision
	655	Insufficient Psychological Acceptability
	656	Reliance on Security Through Obscurity
	657	Violation of Secure Design Principles
	693	Protection Mechanism Failure
	697	Insufficient Comparison
	708	Incorrect Ownership Assignment
	732	Incorrect Permission Assignment for Critical Resource
	757	Selection of Less-Secure Algorithm During Negotiation ('Algorithm Downgrade')
	759	Use of a One-Way Hash without a Salt
	760	Use of a One-Way Hash with a Predictable Salt
	768	Incorrect Short Circuit Evaluation
	778	Insufficient Logging
	779	Logging of Excessive Data

Kingdom	CWE #	Description
Security Features	780	Use of RSA Algorithm without OAEP
	782	Exposed IOCTL with Insufficient Access Control
	784	Reliance on Cookies without Validation and Integrity Checking in a Security Decision
	798	Use of Hard-coded Credentials
	804	Guessable CAPTCHA
	807	Reliance on Untrusted Inputs in a Security Decision
	836	Use of Password Hash Instead of Password for Authentication
	841	Improper Enforcement of Behavioral Workflow
	842	Placement of User into Incorrect Group
	862	Missing Authorization
	863	Incorrect Authorization
	916	Use of Password Hash With Insufficient Computational Effort
	921	Storage of Sensitive Data in a Mechanism without Access Control
	922	Insecure Storage of Sensitive Information
	923	Improper Authentication of Endpoint in a Communication Channel
	924	Improper Enforcement of Message Integrity During Transmission in a Communication Channel
	925	Improper Verification of Intent by Broadcast Receiver
	926	Improper Restriction of Content Provider Export to Other Applications
	927	Use of Implicit Intent for Sensitive Communication
Time and State	8	J2EE Misconfiguration: Entity Bean Declared Remote
	179	Incorrect Behavior Order: Early Validation
	193	Off-by-one Error
	361	Time and State
	362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
	363	Race Condition Enabling Link Following
	364	Signal Handler Race Condition
	365	Race Condition in Switch
	366	Race Condition within a Thread
	367	Time-of-check Time-of-use (TOCTOU) Race Condition
	368	Context Switching Race Condition

Kingdom	CWE #	Description
Time and State	370	Missing Check for Certificate Revocation after Initial Check
	371	State Issues
	372	Incomplete Internal State Distinction
	376	Temporary File Issues
	377	Insecure Temporary File
	378	Creation of Temporary File With Insecure Permissions
	379	Creation of Temporary File in Directory with Incorrect Permissions
	380	Technology-Specific Time and State Issues
	381	J2EE Time and State Issues
	382	J2EE Bad Practices: Use of System.exit()
	383	J2EE Bad Practices: Direct Use of Threads
	385	Covert Timing Channel
	386	Symbolic Name not Mapping to Correct Object
	387	Signal Errors
	408	Incorrect Behavior Order: Early Amplification
	410	Insufficient Resource Pool
	411	Resource Locking Problems
	412	Unrestricted Externally Accessible Lock
	413	Improper Resource Locking
	414	Missing Lock Check
	421	Race Condition During Access to Alternate Channel
	430	Deployment of Wrong Handler
	431	Missing Handler
	432	Dangerous Signal Handler not Disabled During Sensitive Operations
	434	Unrestricted Upload of File with Dangerous Type
	453	Insecure Default Variable Initialization
	456	Missing Initialization of a Variable
	457	Use of Uninitialized Variable
	479	Signal Handler Use of a Non-reentrant Function
	543	Use of Singleton Pattern Without Synchronization in a Multithreaded Context
	551	Incorrect Behavior Order: Authorization Before Parsing and Canonicalization
	557	Concurrency Issues
	558	Use of getlogin() in Multithreaded Application
	562	Return of Stack Variable Address

Kingdom	CWE #	Description
Time and State	567	Unsynchronized Access to Shared Data in a Multithreaded Context
	572	Call to Thread run() instead of start()
	574	EJB Bad Practices: Use of Synchronization Primitives
	585	Empty Synchronized Block
	591	Sensitive Data Storage in Improperly Locked Memory
	593	Authentication Bypass: OpenSSL CTX Object Modified after SSL Objects are Created
	605	Multiple Binds to the Same Port
	609	Double-Checked Locking
	613	Insufficient Session Expiration
	642	External Control of Critical State Data
	662	Improper Synchronization
	663	Use of a Non-reentrant Function in a Concurrent Context
	664	Improper Control of a Resource Through its Lifetime
	665	Improper Initialization
	666	Operation on Resource in Wrong Phase of Lifetime
	667	Improper Locking
	672	Operation on a Resource after Expiration or Release
	675	Duplicate Operations on Resource
	691	Insufficient Control Flow Management
	696	Incorrect Behavior Order
	698	Execution After Redirect (EAR)
	705	Incorrect Control Flow Scoping
	736	CERT C Secure Coding Section 02 - Declarations and Initialization (DCL)
	764	Multiple Locks of a Critical Resource
	765	Multiple Unlocks of a Critical Resource
	768	Incorrect Short Circuit Evaluation
	769	File Descriptor Exhaustion
	770	Allocation of Resources Without Limits or Throttling
	774	Allocation of File Descriptors or Handles Without Limits or Throttling
	776	Improper Restriction of Recursive Entity References in DTDs ('XML Entity Expansion')
	783	Operator Precedence Logic Error
	799	Improper Control of Interaction Frequency
	820	Missing Synchronization

Kingdom	CWE #	Description
Time and State	821	Incorrect Synchronization
	824	Access of Uninitialized Pointer
	825	Expired Pointer Dereference
	826	Premature Release of Resource During Expected Lifetime
	828	Signal Handler with Functionality that is not Asynchronous-Safe
	831	Signal Handler Function Associated with Multiple Signals
	832	Unlock of a Resource that is not Locked
	833	Deadlock
	834	Excessive Iteration
	835	Loop with Unreachable Exit Condition ('Infinite Loop')
	837	Improper Enforcement of a Single, Unique Action
	908	Use of Uninitialized Resource
	909	Missing Initialization of Resource
	910	Use of Expired File Descriptor
	911	Improper Update of Reference Count

Appendix C: Discrimination Details on CVEs

Appendix C details how *Discrimination Rate* was calculated on the CVE-selected test cases. The *Reported CVEs* (or *True Positives* (*TP)*) column indicates how many CVEs were found in the vulnerable version of the test case by each tool. Subcolumn *Found* lists the number of CVEs directly reported, subcolumn *Hinted* lists the number of CVEs indirectly reported, and subcolumn *All* lists the sum of the previous two subcolumns. The *False Positives* (*FP*) column displays the number of CVEs that were incorrectly reported in the fixed version of the test case for which a TP was reported in the vulnerable version. The *N/A* column shows the number of CVEs that cannot be used to calculate discrimination rate, because the relevant code has been entirely removed from the fixed version of the test case.

Discrimination rate is calculated based on the formulae:

$$Discrimination\ rate(All) = (TP(All) - FP(All) - N/A(All)) \ / \ (TP(All) - N/A(All)) \tag{11}$$

$$Discrimination\ rate(Found) = (TP(Found) - FP(Found) - N/A(Found)) \ / \ (TP(Found) - N/A(Found)) \tag{12}$$

$$Discrimination\ rate\ (Hinted) = (TP(Hinted) - FP(Hinted) - N/A(Hinted)) \ / \ (TP(Hinted) - N/A(Hinted)) \tag{13}$$

Where

$$FP(All) = FP(Found) + FP(Hinted) \tag{14}$$

$$N/A(All) = N/A(Found) + N/A(Hinted) \tag{15}$$

Track	Test Case	Tool	Reported CVEs (TP)			False Positives (FP)		N/A		Discrimination Rate		
			All	Found	Hinted	Found	Hinted	Found	Hinted	All	Found	Hinted
C/C++	Asterisk	Tool J	2	2	0	0	0			100 %	100 %	
		Tool A	1	1	0	0	0			100 %	100 %	
		Tool B	1	1	0	0	0			100 %	100 %	
		Tool H	3	3	0	1	0			67 %	67 %	
		Tool C	0	0	0							
		Tool E	0	0	0							
		Tool G	0	0	0	0	0					
		Tool K	0	0	0	0	0					
	Wireshark	Tool E	1	1	0	0	0			100 %	100 %	
		Tool A	12	9	3	0	2			83 %	100 %	33 %
		Tool B	3	2	1	0	1			67 %	100 %	0 %
		Tool I	12	5	7	0	5	1		55 %	100 %	29 %
		Tool C	12	9	3	6	2			33 %	33 %	33 %
		Tool J	6	6	0	4	0			33 %	33 %	
		Tool H	4	4	0	3	0			25 %	25 %	
		Tool D	0	0	0	0	0					
		Tool K	0	0	0	0	0					
Java	JSPWiki	Tool L	1	1	0	1	0			0 %	0 %	
		Tool Q	1	0	1	0	1			0 %		0 %
		Tool M	0	0	0	0	0					
		Tool N	0	0	0	0	0					
		Tool O	0	0	0							
		Tool P	0	0	0	0	0					
	Openfire	Tool Q	6	6	0	4	0			33 %	33 %	
		Tool L	9	8	1	7	1			11 %	13 %	0 %
		Tool O	1	0	1	0	1			0 %		0 %
		Tool M	0	0	0	0	0					
		Tool N	0	0	0	0	0					
		Tool P	0	0	0	0	0					
PHP	WordPress	Tool R	7	7	0	2	0	1		67 %	67 %	

Appendix D: Recall Details on CVEs

Appendix D details how Recall was calculated on the CVE-selected test cases. The *Test Case's CVEs* (or *CVEs*) column lists the number of CVEs contained in each test case (subcolumn *Present*) and the number of CVE's applicable to each tool (subcolumn *App.*). The *Reported CVEs* (or *True Positives* (*TP)*) column indicates how many CVEs were found in the vulnerable version of the test case by each tool. Subcolumn *Found* lists the number of CVEs directly reported, subcolumn *Hinted* lists the number of CVEs indirectly reported, and subcolumn *All* lists the sum of the previous two subcolumns. Recall and applicable recall are calculated on *Found* CVEs and *All* CVEs.

Recall is calculated based on the formulae:

$$Recall(All) = TP(All) \ / \ CVEs(Present) \quad (16)$$

$$Recall(Found) = TP(Found) \ / \ CVEs(Present) \quad (17)$$

Applicable Recall is calculated as follows:

$$App\ Recall(All) = TP(All) \ / \ CVEs(App) \quad (18)$$

$$App\ Recall(Found) = TP(Found) \ / \ CVEs(App) \quad (19)$$

Track	Test Case	Tool	Test Case's CVEs (CVEs)		Reported CVEs (TP)			Recall		App. Recall	
			Present	App.	All	Found	Hinted	All	Found	All	Found
C/C++	Asterisk	Tool H	14	14	3	3	0	21 %	21 %	21 %	21 %
		Tool J	14	11	2	2	0	14 %	14 %	18 %	18 %
		Tool A	14	11	1	1	0	7 %	7 %	9 %	9 %
		Tool B	14	12	1	1	0	7 %	7 %	8 %	8 %
		Tool K	14	8	0	0	0	0 %	0 %	0 %	0 %
		Tool G	14	12	0	0	0	0 %	0 %	0 %	0 %
		Tool C	14	11	0	0	0	0 %	0 %	0 %	0 %
		Tool E	14	9	0	0	0	0 %	0 %	0 %	0 %
	Wireshark	Tool A	83	72	12	9	3	14 %	11 %	17 %	13 %
		Tool C	83	81	12	9	3	14 %	11 %	15 %	11 %
		Tool I	83	83	12	5	7	14 %	6 %	14 %	6 %
		Tool J	83	72	6	6	0	7 %	7 %	8 %	8 %
		Tool H	83	66	4	4	0	5 %	5 %	6 %	6 %
		Tool B	83	83	3	2	1	4 %	2 %	4 %	2 %
		Tool E	83	55	1	1	0	1 %	1 %	2 %	2 %
		Tool K	83	40	0	0	0	0 %	0 %	0 %	0 %
		Tool D	83	69	0	0	0	0 %	0 %	0 %	0 %
Java	JSPWiki	Tool L	1	1	1	1	0	100 %	100 %	100 %	100 %
		Tool Q	1	1	1	0	1	100 %	0 %	100 %	0 %
		Tool N	1	1	0	0	0	0 %	0 %	0 %	0 %
		Tool O	1	1	0	0	0	0 %	0 %	0 %	0 %
		Tool P	1	0	0	0	0	0 %	0 %		
		Tool M	1	0	0	0	0	0 %	0 %		
	Openfire	Tool L	10	10	9	8	1	90 %	80 %	90 %	80 %
		Tool Q	10	9	6	6	0	60 %	60 %	67 %	67 %
		Tool O	10	9	1	0	1	10 %	0 %	11 %	0 %
		Tool N	10	10	0	0	0	0 %	0 %	0 %	0 %
		Tool M	10	1	0	0	0	0 %	0 %	0 %	0 %
		Tool P	10	1	0	0	0	0 %	0 %	0 %	0 %
PHP	WordPress	Tool R	13	13	7	7	0	54 %	54 %	54 %	54 %

Appendix E: Reported and Unreported Weakness Classes on Juliet

Appendix E summarizes which CWEs in the Juliet C/C++ and Juliet Java test cases were reported and unreported by each tool. *Yes* means that the tool reported at least one *True Positive* (*TP*) for a given CWE. *No* indicates that the tool did not report a single *True Positive* for all test cases for this CWE.

Table 64 summarizes which CWEs in the Juliet C/C++ test cases were reported and unreported by each tool.

Table 64. Reported and Unreported Weakness Classes on Juliet C/C++.

Reported and Unreported Weakness Classes on Juliet C/C++								
CWE	**Tool B**	**Tool G**	**Tool H**	**Tool A**	**Tool C**	**Tool D**	**Tool E**	**Tool F**
CWE-121: Stack-based Buffer Overflow	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
CWE-457: Use of Uninitialized Variable	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
CWE-122: Heap-based Buffer Overflow	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
CWE-126: Buffer Over-read	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
CWE-476: NULL Pointer Dereference	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
CWE-124: Buffer Underwrite ('Buffer Underflow')	Yes	Yes	Yes	Yes	Yes	No	No	Yes
CWE-127: Buffer Under-read	Yes	Yes	Yes	Yes	Yes	No	No	Yes
CWE-134: Uncontrolled Format String	Yes	Yes	Yes	Yes	Yes	Yes	No	No
CWE-369: Divide By Zero	Yes	Yes	No	Yes	No	Yes	Yes	Yes
CWE-401: Memory Leak	Yes	Yes	Yes	Yes	Yes	No	Yes	No
CWE-415: Double Free	Yes	Yes	Yes	Yes	No	Yes	Yes	No
CWE-416: Use After Free	Yes	Yes	Yes	Yes	Yes	No	Yes	No
CWE-562: Return of Stack Variable Address	Yes	Yes	No	Yes	No	Yes	Yes	Yes
CWE-078: OS Command Injection	Yes	Yes	Yes	Yes	Yes	No	No	No
CWE-252: Unchecked Return Value	Yes	Yes	Yes	Yes	Yes	No	No	No
CWE-563: Unused Variable	No	Yes	Yes	Yes	Yes	No	Yes	No
CWE-762: Mismatched Memory Management Routines	Yes	No	No	Yes	Yes	Yes	Yes	No

Reported and Unreported Weakness Classes on Juliet C/C++								
CWE	**Tool B**	**Tool G**	**Tool H**	**Tool A**	**Tool C**	**Tool D**	**Tool E**	**Tool F**
CWE-188: Reliance on Data/Memory Layout	Yes	Yes	Yes	Yes	No	No	No	No
CWE-194: Unexpected Sign Extension	Yes	No	Yes	Yes	No	No	Yes	No
CWE-195: Signed to Unsigned Conversion Error	Yes	No	Yes	Yes	No	No	Yes	No
CWE-242: Use of Inherently Dangerous Function	No	Yes	Yes	No	Yes	No	Yes	No
CWE-427: Uncontrolled Search Path Element	Yes	No	Yes	Yes	No	Yes	No	No
CWE-468: Incorrect Pointer Scaling	Yes	Yes	Yes	No	No	Yes	No	No
CWE-561: Dead Code	Yes	Yes	Yes	Yes	No	No	No	No
CWE-570: Expression is Always False	Yes	Yes	No	Yes	No	Yes	No	No
CWE-571: Expression is Always True	Yes	Yes	No	Yes	No	Yes	No	No
CWE-590: Free of Memory not on the Heap	Yes	Yes	Yes	No	No	No	Yes	No
CWE-680: Integer Overflow to Buffer Overflow	Yes	No	Yes	Yes	Yes	No	No	No
CWE-023: Relative Path Traversal	No	No	Yes	Yes	No	Yes	No	No
CWE-190: Integer Overflow or Wraparound	No	No	No	Yes	No	Yes	No	Yes
CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition	Yes	No	Yes	No	Yes	No	No	No
CWE-377: Insecure Temporary File	Yes	No	Yes	No	No	No	Yes	No
CWE-400: Resource Exhaustion	Yes	No	Yes	No	Yes	No	No	No
CWE-404: Improper Resource Shutdown or Release	Yes	Yes	Yes	No	No	No	No	No
CWE-467: Use of sizeof() on a Pointer Type	Yes	Yes	No	Yes	No	No	No	No
CWE-481: Assigning instead of Comparing	Yes	Yes	No	No	No	Yes	No	No
CWE-483: Incorrect Block Delimitation	Yes	Yes	No	No	No	Yes	No	No

Reported and Unreported Weakness Classes on Juliet C/C++								
CWE	**Tool B**	**Tool G**	**Tool H**	**Tool A**	**Tool C**	**Tool D**	**Tool E**	**Tool F**
CWE-688: Function Call With Incorrect Variable as Argument	Yes	Yes	No	No	Yes	No	No	No
CWE-690: Unchecked Return Value to NULL Pointer Dereference	Yes	No	Yes	No	Yes	No	No	No
CWE-036: Absolute Path Traversal	No	No	Yes	Yes	No	No	No	No
CWE-191: Integer Underflow (Wrap or Wraparound)	No	No	No	Yes	No	Yes	No	No
CWE-196: Unsigned to Signed Conversion Error	No	Yes	No	Yes	No	No	No	No
CWE-197: Numeric Truncation Error	No	Yes	No	Yes	No	No	No	No
CWE-253: Incorrect Check of Function Return Value	Yes	No	No	No	Yes	No	No	No
CWE-390: Detection of Error Condition Without Action	No	No	No	No	Yes	Yes	No	No
CWE-398: Indicator of Poor Code Quality	No	Yes	No	No	No	Yes	No	No
CWE-480: Use of Incorrect Operator	Yes	Yes	No	No	No	No	No	No
CWE-482: Comparing instead of Assigning	Yes	Yes	No	No	No	No	No	No
CWE-484: Omitted Break Statement in Switch	Yes	Yes	No	No	No	No	No	No
CWE-587: Assignment of a Fixed Address to a Pointer	No	Yes	No	No	No	Yes	No	No
CWE-588: Attempt to Access Child of a Non-structure Pointer	Yes	No	No	No	No	No	Yes	No
CWE-606: Unchecked Input for Loop Condition	Yes	Yes	No	No	No	No	No	No
CWE-675: Duplicate Operations on Resource	Yes	No	Yes	No	No	No	No	No
CWE-685: Function Call With Incorrect Number of Arguments	Yes	No	No	No	Yes	No	No	No

Reported and Unreported Weakness Classes on Juliet C/C++								
CWE	**Tool B**	**Tool G**	**Tool H**	**Tool A**	**Tool C**	**Tool D**	**Tool E**	**Tool F**
CWE-773: Missing Reference to Active File Descriptor or Handle	Yes	No	Yes	No	No	No	No	No
CWE-775: Missing Release of File Descriptor after Effective Lifetime	Yes	No	Yes	No	No	No	No	No
CWE-789: Uncontrolled Memory Allocation	Yes	No	No	Yes	No	No	No	No
CWE-123: Write-what-where Condition	No	No	No	No	No	No	No	Yes
CWE-338: Use of Cryptographically Weak PRNG	No	Yes	No	No	No	No	No	No
CWE-396: Declaration of Catch for Generic Exception	Yes	No	No	No	No	No	No	No
CWE-426: Untrusted Search Path	No	No	No	No	Yes	No	No	No
CWE-459: Incomplete Cleanup	Yes	No	No	No	No	No	No	No
CWE-469: Use of Pointer Subtraction to Determine Size	No	Yes	No	No	No	No	No	No
CWE-475: Undefined Behavior for Input to API	No	Yes	No	No	No	No	No	No
CWE-478: Missing Default Case in Switch Statement	No	Yes	No	No	No	No	No	No
CWE-500: Public Static Field Not Marked Final	No	No	No	No	Yes	No	No	No
CWE-506: Embedded Malicious Code	No	No	No	No	Yes	No	No	No
CWE-511: Logic/Time Bomb	No	No	No	No	Yes	No	No	No
CWE-526: Information Exposure Through Environmental Variables	No	No	Yes	No	No	No	No	No
CWE-665: Improper Initialization	No	No	Yes	No	No	No	No	No
CWE-667: Improper Locking	Yes	No	No	No	No	No	No	No
CWE-672: Operation on a Resource after Expiration or Release	No	No	No	Yes	No	No	No	No

Reported and Unreported Weakness Classes on Juliet C/C++								
CWE	**Tool B**	**Tool G**	**Tool H**	**Tool A**	**Tool C**	**Tool D**	**Tool E**	**Tool F**
CWE-674: Uncontrolled Recursion	No	Yes	No	No	No	No	No	No
CWE-676: Use of Potentially Dangerous Function	No	No	Yes	No	No	No	No	No
CWE-758: Reliance on Undefined Behavior	No	Yes	No	No	No	No	No	No
CWE-761: Free of Pointer not at Start of Buffer	No	No	No	No	No	No	Yes	No
CWE-015: External Control of System or Configuration Setting	No	No	No	No	No	No	No	No
CWE-090: LDAP Injection	No	No	No	No	No	No	No	No
CWE-114: Process Control	No	No	No	No	No	No	No	No
CWE-176: Improper Handling of Unicode Encoding	No	No	No	No	No	No	No	No
CWE-222: Truncation of Security-relevant Information	No	No	No	No	No	No	No	No
CWE-223: Omission of Security-relevant Information	No	No	No	No	No	No	No	No
CWE-226: Sensitive Information Uncleared Before Release	No	No	No	No	No	No	No	No
CWE-244: Heap Inspection	No	No	No	No	No	No	No	No
CWE-247: Reliance on DNS Lookups in a Security Decision	No	No	No	No	No	No	No	No
CWE-256: Plaintext Storage of a Password	No	No	No	No	No	No	No	No
CWE-259: Use of Hard-coded Password	No	No	No	No	No	No	No	No
CWE-272: Least Privilege Violation	No	No	No	No	No	No	No	No
CWE-273: Improper Check for Dropped Privileges	No	No	No	No	No	No	No	No
CWE-284: Improper Access Control	No	No	No	No	No	No	No	No
CWE-319: Cleartext Transmission of Sensitive Information	No	No	No	No	No	No	No	No

Reported and Unreported Weakness Classes on Juliet C/C++								
CWE	**Tool B**	**Tool G**	**Tool H**	**Tool A**	**Tool C**	**Tool D**	**Tool E**	**Tool F**
CWE-321: Use of Hard-coded Cryptographic Key	No	No	No	No	No	No	No	No
CWE-325: Missing Required Cryptographic Step	No	No	No	No	No	No	No	No
CWE-327: Use of a Broken or Risky Cryptographic Algorithm	No	No	No	No	No	No	No	No
CWE-328: Reversible One-Way Hash	No	No	No	No	No	No	No	No
CWE-364: Signal Handler Race Condition	No	No	No	No	No	No	No	No
CWE-366: Race Condition within a Thread	No	No	No	No	No	No	No	No
CWE-391: Unchecked Error Condition	No	No	No	No	No	No	No	No
CWE-397: Declaration of Throws for Generic Exception	No	No	No	No	No	No	No	No
CWE-440: Expected Behavior Violation	No	No	No	No	No	No	No	No
CWE-464: Addition of Data Structure Sentinel	No	No	No	No	No	No	No	No
CWE-479: Signal Handler Use of a Non-reentrant Function	No	No	No	No	No	No	No	No
CWE-510: Trapdoor	No	No	No	No	No	No	No	No
CWE-534: Information Exposure Through Debug Log Files	No	No	No	No	No	No	No	No
CWE-535: Information Exposure Through Shell Error Message	No	No	No	No	No	No	No	No
CWE-546: Suspicious Comment	No	No	No	No	No	No	No	No
CWE-591: Sensitive Data Storage in Improperly Locked Memory	No	No	No	No	No	No	No	No
CWE-605: Multiple Binds to the Same Port	No	No	No	No	No	No	No	No
CWE-615: Information Exposure Through Comments	No	No	No	No	No	No	No	No

Reported and Unreported Weakness Classes on Juliet C/C++								
CWE	**Tool B**	**Tool G**	**Tool H**	**Tool A**	**Tool C**	**Tool D**	**Tool E**	**Tool F**
CWE-617: Reachable Assertion	No	No	No	No	No	No	No	No
CWE-620: Unverified Password Change	No	No	No	No	No	No	No	No
CWE-666: Operation on Resource in Wrong Phase of Lifetime	No	No	No	No	No	No	No	No
CWE-681: Incorrect Conversion between Numeric Types	No	No	No	No	No	No	No	No
CWE-780: Use of RSA Algorithm without OAEP	No	No	No	No	No	No	No	No
CWE-785: Path Manipulation Function w/o Max-sized Buffer	No	No	No	No	No	No	No	No
CWE-832: Unlock of a Resource that is not Locked	No	No	No	No	No	No	No	No
CWE-835: Infinite Loop	No	No	No	No	No	No	No	No
CWE-843: Type Confusion	No	No	No	No	No	No	No	No
Number of supported CWEs on Juliet C	**49**	**41**	**36**	**34**	**26**	**22**	**18**	**11**
Number of unsupported CWEs on Juliet C	**69**	**77**	**82**	**84**	**92**	**96**	**100**	**107**

Table 65 summarizes which CWEs in the Juliet Java test cases were reported and unreported by each tool.

Table 65. Reported and Unreported Weakness Classes on Juliet Java.

Reported and Unreported Weakness Classes on Juliet Java				
CWE	**Tool L**	**Tool N**	**Tool O**	**Tool M**
CWE-382: J2EE Bad Practices: Use of System.exit()	Yes	Yes	Yes	Yes
CWE-404: Improper Resource Shutdown or Release	Yes	Yes	Yes	Yes
CWE-481: Assigning instead of Comparing	Yes	Yes	Yes	Yes
CWE-563: Unused Variable	Yes	Yes	Yes	Yes
CWE-570: Expression is Always False	Yes	Yes	Yes	Yes
CWE-572: Call to Thread run() instead of start()	Yes	Yes	Yes	Yes
CWE-585: Empty Synchronized Block	Yes	Yes	Yes	Yes
CWE-586: Explicit Call to Finalize()	Yes	Yes	Yes	Yes
CWE-597: Use of Wrong Operator in String Comparison	Yes	Yes	Yes	Yes
CWE-772: Missing Release of Resource after Effective Lifetime	Yes	Yes	Yes	Yes
CWE-833: Deadlock	Yes	Yes	Yes	Yes
CWE-023: Relative Path Traversal	Yes	Yes	Yes	No
CWE-036: Absolute Path Traversal	Yes	Yes	Yes	No
CWE-078: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	Yes	Yes	Yes	No
CWE-080: Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)	Yes	Yes	Yes	No
CWE-083: Improper Neutralization of Script in Attributes in a Web Page	Yes	Yes	Yes	No
CWE-089: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	Yes	Yes	Yes	No
CWE-113: Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')	Yes	Yes	Yes	No
CWE-252: Unchecked Return Value	Yes	Yes	Yes	No
CWE-259: Use of Hard-coded Password	Yes	Yes	Yes	No
CWE-398: Indicator of Poor Code Quality	No	Yes	Yes	Yes
CWE-476: NULL Pointer Dereference	Yes	Yes	Yes	No
CWE-482: Comparing instead of Assigning	Yes	Yes	No	Yes

Reported and Unreported Weakness Classes on Juliet Java				
CWE	**Tool L**	**Tool N**	**Tool O**	**Tool M**
CWE-571: Expression is Always True	Yes	Yes	No	Yes
CWE-601: URL Redirection to Untrusted Site ('Open Redirect')	Yes	Yes	Yes	No
CWE-764: Multiple Locks of a Critical Resource	Yes	Yes	No	Yes
CWE-765: Multiple Unlocks of a Critical Resource	Yes	Yes	No	Yes
CWE-775: Missing Release of File Descriptor or Handle after Effective Lifetime	Yes	Yes	Yes	No
CWE-832: Unlock of a Resource that is not Locked	Yes	Yes	No	Yes
CWE-081: Improper Neutralization of Script in an Error Message Web Page	Yes	No	Yes	No
CWE-114: Process Control	Yes	No	No	Yes
CWE-209: Information Exposure Through an Error Message	Yes	No	No	Yes
CWE-319: Cleartext Transmission of Sensitive Information	Yes	No	Yes	No
CWE-328: Reversible One-Way Hash	Yes	No	Yes	No
CWE-338: Use of Cryptographically Weak PRNG	Yes	No	Yes	No
CWE-383: J2EE Bad Practices: Direct Use of Threads	Yes	No	No	Yes
CWE-390: Detection of Error Condition Without Action	Yes	No	No	Yes
CWE-395: Use of NullPointerException Catch to Detect NULL Pointer Dereference	Yes	No	No	Yes
CWE-478: Missing Default Case in Switch Statement	No	No	Yes	Yes
CWE-483: Incorrect Block Delimitation	No	Yes	No	Yes
CWE-484: Omitted Break Statement in Switch	No	Yes	Yes	No
CWE-584: Return Inside Finally Block	Yes	No	No	Yes
CWE-609: Double-Checked Locking	Yes	No	No	Yes
CWE-674: Uncontrolled Recursion	No	Yes	Yes	No
CWE-760: Use of a One-Way Hash with a Predictable Salt	Yes	No	Yes	No
CWE-015: External Control of System or Configuration Setting	Yes	No	No	No
CWE-090: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')	Yes	No	No	No
CWE-111: Direct Use of Unsafe JNI	Yes	No	No	No

Reported and Unreported Weakness Classes on Juliet Java				
CWE	**Tool L**	**Tool N**	**Tool O**	**Tool M**
CWE-226: Sensitive Information Uncleared Before Release	Yes	No	No	No
CWE-253: Incorrect Check of Function Return Value	No	Yes	No	No
CWE-256: Plaintext Storage of a Password	Yes	No	No	No
CWE-315: Cleartext Storage of Sensitive Information in a Cookie	Yes	No	No	No
CWE-327: Use of a Broken or Risky Cryptographic Algorithm	Yes	No	No	No
CWE-336: Same Seed in PRNG	Yes	No	No	No
CWE-396: Declaration of Catch for Generic Exception	No	No	No	Yes
CWE-397: Declaration of Throws for Generic Exception	No	No	No	Yes
CWE-400: Uncontrolled Resource Consumption ('Resource Exhaustion')	Yes	No	No	No
CWE-470: Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')	Yes	No	No	No
CWE-477: Use of Obsolete Functions	Yes	No	No	No
CWE-523: Unprotected Transport of Credentials	Yes	No	No	No
CWE-526: Information Exposure Through Environmental Variables	Yes	No	No	No
CWE-533: Information Exposure Through Server Log Files	Yes	No	No	No
CWE-534: Information Exposure Through Debug Log Files	Yes	No	No	No
CWE-535: Information Exposure Through Shell Error Message	Yes	No	No	No
CWE-539: Information Exposure Through Persistent Cookies	Yes	No	No	No
CWE-549: Missing Password Field Masking	Yes	No	No	No
CWE-566: Authorization Bypass Through User-Controlled SQL Primary Key	Yes	No	No	No
CWE-579: J2EE Bad Practices: Non-serializable Object Stored in Session	Yes	No	No	No
CWE-614: Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	Yes	No	No	No
CWE-643: Improper Neutralization of Data within XPath Expressions ('XPath Injection')	Yes	No	No	No

Reported and Unreported Weakness Classes on Juliet Java				
CWE	**Tool L**	**Tool N**	**Tool O**	**Tool M**
CWE-690: Unchecked Return Value to NULL Pointer Dereference	Yes	No	No	No
CWE-129: Improper Validation of Array Index	No	No	No	No
CWE-134: Uncontrolled Format String	No	No	No	No
CWE-190: Integer Overflow or Wraparound	No	No	No	No
CWE-191: Integer Underflow (Wrap or Wraparound)	No	No	No	No
CWE-193: Off-by-one Error	No	No	No	No
CWE-197: Numeric Truncation Error	No	No	No	No
CWE-248: Uncaught Exception	No	No	No	No
CWE-321: Use of Hard-coded Cryptographic Key	No	No	No	No
CWE-325: Missing Required Cryptographic Step	No	No	No	No
CWE-329: Not Using a Random IV with CBC Mode	No	No	No	No
CWE-369: Divide By Zero	No	No	No	No
CWE-378: Creation of Temporary File With Insecure Permissions	No	No	No	No
CWE-379: Creation of Temporary File in Directory with Incorrect Permissions	No	No	No	No
CWE-459: Incomplete Cleanup	No	No	No	No
CWE-486: Comparison of Classes by Name	No	No	No	No
CWE-491: Public cloneable() Method Without Final ('Object Hijack')	No	No	No	No
CWE-500: Public Static Field Not Marked Final	No	No	No	No
CWE-506: Embedded Malicious Code	No	No	No	No
CWE-510: Trapdoor	No	No	No	No
CWE-511: Logic/Time Bomb	No	No	No	No
CWE-546: Suspicious Comment	No	No	No	No
CWE-561: Dead Code	No	No	No	No
CWE-568: finalize() Method Without super.finalize()	No	No	No	No
CWE-580: clone() Method Without super.clone()	No	No	No	No
CWE-581: Object Model Violation: Just One of Equals and Hashcode Defined	No	No	No	No
CWE-582: Array Declared Public, Final, and Static	No	No	No	No
CWE-598: Information Exposure Through Query Strings in GET Request	No	No	No	No
CWE-600: Uncaught Exception in Servlet	No	No	No	No

Reported and Unreported Weakness Classes on Juliet Java				
CWE	**Tool L**	**Tool N**	**Tool O**	**Tool M**
CWE-605: Multiple Binds to the Same Port	No	No	No	No
CWE-606: Unchecked Input for Loop Condition	No	No	No	No
CWE-607: Public Static Final Field References Mutable Object	No	No	No	No
CWE-613: Insufficient Session Expiration	No	No	No	No
CWE-615: Information Exposure Through Comments	No	No	No	No
CWE-617: Reachable Assertion	No	No	No	No
CWE-667: Improper Locking	No	No	No	No
CWE-681: Incorrect Conversion between Numeric Types	No	No	No	No
CWE-698: Execution After Redirect (EAR)	No	No	No	No
CWE-759: Use of a One-Way Hash without a Salt	No	No	No	No
CWE-789: Uncontrolled Memory Allocation	No	No	No	No
CWE-835: Loop with Unreachable Exit Condition ('Infinite Loop')	No	No	No	No
Number of supported CWEs on Juliet Java	**63**	**33**	**32**	**28**
Number of unsupported CWEs on Juliet Java	**48**	**78**	**79**	**83**

Appendix F: Recall per CWE on Juliet C and Java

Appendix F summarizes the recall results per CWE for each tool on the Juliet test suites (C/C++ and Java).

Table 66 summarizes the recall results per CWE for each tool on the Juliet C/C++ test cases.

Table 66. Recall per CWE on Juliet C/C++.

Recall per CWE on Juliet C/C++		
CWE	**Tool**	**Recall**
CWE-23: Relative Path Traversal	Tool A	11 %
	Tool D	20 %
	Tool H	10 %
CWE-36: Absolute Path Traversal	Tool A	10 %
	Tool H	10 %
CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	Tool A	11 %
	Tool B	13 %
	Tool C	20 %
	Tool G	2 %
	Tool H	20 %
CWE-121: Stack-based Buffer Overflow	Tool A	12 %
	Tool B	14 %
	Tool C	25 %
	Tool D	2 %
	Tool E	3 %
	Tool F	74 %
	Tool G	1 %
	Tool H	21 %
CWE-122: Heap-based Buffer Overflow	Tool A	12 %
	Tool B	5 %
	Tool C	23 %
	Tool D	1 %
	Tool E	1 %
	Tool F	38 %
	Tool H	24 %
CWE-123: Write-what-where Condition	Tool F	79 %
CWE-124: Buffer Underwrite ('Buffer Underflow')	Tool A	21 %
	Tool B	5 %
	Tool C	9 %

Recall per CWE on Juliet C/C++		
CWE	**Tool**	**Recall**
	Tool F	61 %
	Tool G	0 %
	Tool H	9 %
CWE-126: Buffer Over-read	Tool A	7 %
	Tool B	7 %
	Tool C	24 %
	Tool D	0 %
	Tool F	60 %
	Tool G	0 %
	Tool H	2 %
CWE-127: Buffer Under-read	Tool A	13 %
	Tool B	14 %
	Tool C	17 %
	Tool F	61 %
	Tool G	0 %
	Tool H	9 %
CWE-134: Uncontrolled Format String	Tool A	19 %
	Tool B	26 %
	Tool C	25 %
	Tool D	42 %
	Tool G	2 %
	Tool H	42 %
CWE-188: Reliance on Data/Memory Layout	Tool A	50 %
	Tool B	47 %
	Tool G	14 %
	Tool H	50 %
CWE-190: Integer Overflow or Wraparound	Tool A	21 %
	Tool D	0 %
	Tool F	40 %
CWE-191: Integer Underflow (Wrap or Wraparound)	Tool A	18 %
	Tool D	0 %
CWE-194: Unexpected Sign Extension	Tool A	72 %
	Tool B	24 %
	Tool E	7 %
	Tool H	63 %
CWE-195: Signed to Unsigned Conversion Error	Tool A	87 %
	Tool B	32 %

Recall per CWE on Juliet C/C++		
CWE	**Tool**	**Recall**
	Tool E	11 %
	Tool H	59 %
CWE-196: Unsigned to Signed Conversion Error	Tool A	100 %
	Tool G	6 %
CWE-197: Numeric Truncation Error	Tool A	25 %
	Tool G	2 %
CWE-242: Use of Inherently Dangerous Function	Tool C	100 %
	Tool E	100 %
	Tool G	6 %
	Tool H	100 %
CWE-252: Unchecked Return Value	Tool A	40 %
	Tool B	34 %
	Tool C	14 %
	Tool G	7 %
	Tool H	11 %
CWE-253: Incorrect Check of Function Return Value	Tool B	5 %
	Tool C	13 %
CWE-338: Use of Cryptographically Weak PRNG	Tool G	6 %
CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition	Tool B	100 %
	Tool C	100 %
	Tool H	50 %
CWE-369: Divide By Zero	Tool A	31 %
	Tool B	19 %
	Tool D	1 %
	Tool E	6 %
	Tool F	79 %
	Tool G	1 %
CWE-377: Insecure Temporary File	Tool B	38 %
	Tool E	13 %
	Tool H	38 %
CWE-390: Detection of Error Condition Without Action	Tool C	20 %
	Tool D	20 %
CWE-396: Declaration of Catch for Generic Exception	Tool B	33 %
CWE-398: Indicator of Poor Code Quality	Tool D	20 %
	Tool G	2 %
CWE-400: Uncontrolled Resource Consumption ('Resource Exhaustion')	Tool B	54 %
	Tool C	5 %

Recall per CWE on Juliet C/C++		
CWE	**Tool**	**Recall**
	Tool H	24 %
CWE-401: Improper Release of Memory Before Removing Last Reference ('Memory Leak')	Tool A	29 %
	Tool B	54 %
	Tool C	0 %
	Tool E	31 %
	Tool G	1 %
	Tool H	67 %
CWE-404: Improper Resource Shutdown or Release	Tool B	3 %
	Tool G	2 %
	Tool H	4 %
CWE-415: Double Free	Tool A	35 %
	Tool B	44 %
	Tool D	2 %
	Tool E	41 %
	Tool G	1 %
	Tool H	48 %
CWE-416: Use After Free	Tool A	55 %
	Tool B	70 %
	Tool C	0 %
	Tool E	33 %
	Tool G	2 %
	Tool H	67 %
CWE-426: Untrusted Search Path	Tool C	50 %
CWE-427: Uncontrolled Search Path Element	Tool A	27 %
	Tool B	32 %
	Tool D	20 %
	Tool H	12 %
CWE-457: Use of Uninitialized Variable	Tool A	17 %
	Tool B	43 %
	Tool C	0 %
	Tool D	15 %
	Tool E	43 %
	Tool F	59 %
	Tool G	2 %
	Tool H	15 %
CWE-459: Incomplete Cleanup	Tool B	50 %
CWE-467: Use of sizeof() on a Pointer Type	Tool A	100 %

Recall per CWE on Juliet C/C++		
CWE	**Tool**	**Recall**
	Tool B	100 %
	Tool G	22 %
CWE-468: Incorrect Pointer Scaling	Tool B	51 %
	Tool D	49 %
	Tool G	30 %
	Tool H	3 %
CWE-469: Use of Pointer Subtraction to Determine Size	Tool G	28 %
CWE-475: Undefined Behavior for Input to API	Tool G	28 %
CWE-476: NULL Pointer Dereference	Tool A	55 %
	Tool B	60 %
	Tool D	20 %
	Tool E	52 %
	Tool F	78 %
	Tool G	4 %
	Tool H	47 %
CWE-478: Missing Default Case in Switch Statement	Tool G	39 %
CWE-480: Use of Incorrect Operator	Tool B	100 %
	Tool G	6 %
CWE-481: Assigning instead of Comparing	Tool B	100 %
	Tool D	100 %
	Tool G	39 %
CWE-482: Comparing instead of Assigning	Tool B	100 %
	Tool G	39 %
CWE-483: Incorrect Block Delimitation	Tool B	95 %
	Tool D	5 %
	Tool G	45 %
CWE-484: Omitted Break Statement in Switch	Tool B	100 %
	Tool G	39 %
CWE-500: Public Static Field Not Marked Final	Tool C	100 %
CWE-506: Embedded Malicious Code	Tool C	22 %
CWE-511: Logic/Time Bomb	Tool C	50 %
CWE-526: Information Exposure Through Environmental Variables	Tool H	100 %
CWE-561: Dead Code	Tool A	100 %
	Tool B	50 %
	Tool G	50 %
	Tool H	50 %

Recall per CWE on Juliet C/C++		
CWE	**Tool**	**Recall**
CWE-562: Return of Stack Variable Address	Tool A	33 %
	Tool B	67 %
	Tool D	33 %
	Tool E	100 %
	Tool F	67 %
	Tool G	67 %
CWE-563: Unused Variable	Tool A	64 %
	Tool C	5 %
	Tool E	36 %
	Tool G	5 %
	Tool H	43 %
CWE-570: Expression is Always False	Tool A	56 %
	Tool B	13 %
	Tool D	6 %
	Tool G	38 %
CWE-571: Expression is Always True	Tool A	50 %
	Tool B	19 %
	Tool D	6 %
	Tool G	38 %
CWE-587: Assignment of a Fixed Address to a Pointer	Tool D	100 %
	Tool G	39 %
CWE-588: Attempt to Access Child of a Non-structure Pointer	Tool B	15 %
	Tool E	9 %
CWE-590: Free of Memory not on the Heap	Tool B	37 %
	Tool E	27 %
	Tool G	0 %
	Tool H	7 %
CWE-606: Unchecked Input for Loop Condition	Tool B	32 %
	Tool G	4 %
CWE-665: Improper Initialization	Tool H	1 %
CWE-667: Improper Locking	Tool B	6 %
CWE-672: Operation on a Resource after Expiration or Release	Tool A	91 %
CWE-674: Uncontrolled Recursion	Tool G	100 %
CWE-675: Duplicate Operations on Resource	Tool B	67 %
	Tool H	2 %
CWE-676: Use of Potentially Dangerous Function	Tool H	100 %

Recall per CWE on Juliet C/C++		
CWE	**Tool**	**Recall**
CWE-680: Integer Overflow to Buffer Overflow	Tool A	30 %
	Tool B	49 %
	Tool C	17 %
	Tool H	60 %
CWE-685: Function Call With Incorrect Number of Arguments	Tool B	100 %
	Tool C	100 %
CWE-688: Function Call With Incorrect Variable or Reference as Argument	Tool B	100 %
	Tool C	100 %
	Tool G	39 %
CWE-690: Unchecked Return Value to NULL Pointer Dereference	Tool B	2 %
	Tool C	15 %
	Tool H	59 %
CWE-758: Reliance on Undefined, Unspecified, or Implementation-Defined Behavior	Tool G	1 %
CWE-761: Free of Pointer not at Start of Buffer	Tool E	28 %
CWE-762: Mismatched Memory Management Routines	Tool A	25 %
	Tool B	58 %
	Tool C	14 %
	Tool D	2 %
	Tool E	47 %
CWE-773: Missing Reference to Active File Descriptor or Handle	Tool B	51 %
	Tool H	53 %
CWE-775: Missing Release of File Descriptor or Handle after Effective Lifetime	Tool B	51 %
	Tool H	53 %
CWE-789: Uncontrolled Memory Allocation	Tool A	20 %
	Tool B	26 %

Table 67 summarizes the recall results per CWE for each tool on the Juliet Java test cases.

Table 67. Recall per CWE on Juliet Java.

Recall per CWE on Juliet Java		
CWE	**Tool**	**Recall**
CWE-15: External Control of System or Configuration Setting	Tool L	100 %
CWE-23: Relative Path Traversal	Tool L	100 %
	Tool N	43 %
	Tool O	100 %
CWE-36: Absolute Path Traversal	Tool L	100 %
	Tool N	43 %
	Tool O	100 %
CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	Tool L	100 %
	Tool N	47 %
	Tool O	100 %
CWE-80: Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)	Tool L	54 %
	Tool N	63 %
	Tool O	3 %
CWE-81: Improper Neutralization of Script in an Error Message Web Page	Tool L	54 %
	Tool O	6 %
CWE-83: Improper Neutralization of Script in Attributes in a Web Page	Tool L	54 %
	Tool N	63 %
	Tool O	6 %
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	Tool L	100 %
	Tool N	47 %
	Tool O	80 %
CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')	Tool L	100 %
CWE-111: Direct Use of Unsafe JNI	Tool L	100 %
CWE-113: Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')	Tool L	41 %
	Tool N	4 %
	Tool O	4 %
CWE-114: Process Control	Tool L	100 %
	Tool M	100 %
CWE-209: Information Exposure Through an Error Message	Tool L	100 %
	Tool M	50 %

Recall per CWE on Juliet Java		
CWE	**Tool**	**Recall**
CWE-226: Sensitive Information Uncleared Before Release	Tool L	100 %
CWE-252: Unchecked Return Value	Tool L	100 %
	Tool N	100 %
	Tool O	100 %
CWE-253: Incorrect Check of Function Return Value	Tool N	100 %
CWE-256: Plaintext Storage of a Password	Tool L	100 %
CWE-259: Use of Hard-coded Password	Tool L	27 %
	Tool N	9 %
	Tool O	9 %
CWE-315: Cleartext Storage of Sensitive Information in a Cookie	Tool L	100 %
CWE-319: Cleartext Transmission of Sensitive Information	Tool L	100 %
	Tool O	40 %
CWE-327: Use of a Broken or Risky Cryptographic Algorithm	Tool L	53 %
CWE-328: Reversible One-Way Hash	Tool L	100 %
	Tool O	67 %
CWE-336: Same Seed in PRNG	Tool L	100 %
CWE-338: Use of Cryptographically Weak PRNG	Tool L	100 %
	Tool O	100 %
CWE-382: J2EE Bad Practices: Use of System.exit()	Tool L	50 %
	Tool M	100 %
	Tool N	50 %
	Tool O	50 %
CWE-383: J2EE Bad Practices: Direct Use of Threads	Tool L	100 %
	Tool M	100 %
CWE-390: Detection of Error Condition Without Action	Tool L	50 %
	Tool M	50 %
CWE-395: Use of NullPointerException Catch to Detect NULL Pointer Dereference	Tool L	100 %
	Tool M	100 %
CWE-396: Declaration of Catch for Generic Exception	Tool M	100 %
CWE-397: Declaration of Throws for Generic Exception	Tool M	75 %
CWE-398: Indicator of Poor Code Quality	Tool M	76 %
	Tool N	12 %
	Tool O	12 %

Recall per CWE on Juliet Java		
CWE	**Tool**	**Recall**
CWE-400: Uncontrolled Resource Consumption ('Resource Exhaustion')	Tool L	24 %
CWE-404: Improper Resource Shutdown or Release	Tool L	60 %
	Tool M	20 %
	Tool N	60 %
	Tool O	40 %
CWE-470: Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')	Tool L	100 %
CWE-476: NULL Pointer Dereference	Tool L	66 %
	Tool N	80 %
	Tool O	64 %
CWE-477: Use of Obsolete Functions	Tool L	100 %
CWE-478: Missing Default Case in Switch Statement	Tool M	100 %
	Tool O	100 %
CWE-481: Assigning instead of Comparing	Tool L	100 %
	Tool M	6 %
	Tool N	100 %
	Tool O	100 %
CWE-482: Comparing instead of Assigning	Tool L	65 %
	Tool M	6 %
	Tool N	6 %
CWE-483: Incorrect Block Delimitation	Tool M	89 %
	Tool N	95 %
CWE-484: Omitted Break Statement in Switch	Tool N	100 %
	Tool O	100 %
CWE-523: Unprotected Transport of Credentials	Tool L	100 %
CWE-526: Information Exposure Through Environmental Variables	Tool L	100 %
CWE-533: Information Exposure Through Server Log Files	Tool L	100 %
CWE-534: Information Exposure Through Debug Log Files	Tool L	100 %
CWE-535: Information Exposure Through Shell Error Message	Tool L	100 %
CWE-539: Information Exposure Through Persistent Cookies	Tool L	100 %
CWE-549: Missing Password Field Masking	Tool L	100 %

Recall per CWE on Juliet Java		
CWE	**Tool**	**Recall**
CWE-563: Unused Variable	Tool L	29 %
	Tool M	92 %
	Tool N	15 %
	Tool O	15 %
CWE-566: Authorization Bypass Through User-Controlled SQL Primary Key	Tool L	100 %
CWE-570: Expression is Always False	Tool L	69 %
	Tool M	6 %
	Tool N	6 %
	Tool O	6 %
CWE-571: Expression is Always True	Tool L	69 %
	Tool M	6 %
	Tool N	6 %
CWE-572: Call to Thread run() instead of start()	Tool L	100 %
	Tool M	100 %
	Tool N	100 %
	Tool O	100 %
CWE-579: J2EE Bad Practices: Non-serializable Object Stored in Session	Tool L	100 %
CWE-584: Return Inside Finally Block	Tool L	100 %
	Tool M	100 %
CWE-585: Empty Synchronized Block	Tool L	100 %
	Tool M	100 %
	Tool N	100 %
	Tool O	100 %
CWE-586: Explicit Call to Finalize()	Tool L	100 %
	Tool M	100 %
	Tool N	100 %
	Tool O	100 %
CWE-597: Use of Wrong Operator in String Comparison	Tool L	100 %
	Tool M	100 %
	Tool N	94 %
	Tool O	94 %
CWE-601: URL Redirection to Untrusted Site ('Open Redirect')	Tool L	54 %
	Tool N	6 %
	Tool O	100 %
CWE-609: Double-Checked Locking	Tool L	100 %

Recall per CWE on Juliet Java		
CWE	**Tool**	**Recall**
	Tool M	100 %
CWE-614: Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	Tool L	100 %
CWE-643: Improper Neutralization of Data within XPath Expressions ('XPath Injection')	Tool L	100 %
CWE-674: Uncontrolled Recursion	Tool N	50 %
	Tool O	50 %
CWE-690: Unchecked Return Value to NULL Pointer Dereference	Tool L	59 %
CWE-760: Use of a One-Way Hash with a Predictable Salt	Tool L	100 %
	Tool O	100 %
CWE-764: Multiple Locks of a Critical Resource	Tool L	50 %
	Tool M	50 %
	Tool N	100 %
CWE-765: Multiple Unlocks of a Critical Resource	Tool L	50 %
	Tool M	50 %
	Tool N	100 %
CWE-772: Missing Release of Resource after Effective Lifetime	Tool L	50 %
	Tool M	50 %
	Tool N	50 %
	Tool O	50 %
CWE-775: Missing Release of File Descriptor or Handle after Effective Lifetime	Tool L	100 %
	Tool N	100 %
	Tool O	100 %
CWE-832: Unlock of a Resource that is not Locked	Tool L	50 %
	Tool M	50 %
	Tool N	50 %
CWE-833: Deadlock	Tool L	67 %
	Tool M	50 %
	Tool N	100 %
	Tool O	67 %

Appendix G: Applicable Recall per CWE on Juliet

Appendix G summarizes the applicable recall results per CWE for each tool on the Juliet test suites (C/C++ and Java).

Table 68 summarizes the applicable recall results per CWE for each tool on the Juliet C/C++ test cases.

Table 68. Applicable Recall per CWE on Juliet C/C++.

Applicable Recall per CWE on Juliet C/C++									
CWE	**Tool F**	**Tool H**	**Tool B**	**Tool A**	**Tool E**	**Tool C**	**Tool D**	**Tool G**	**Recall/CWE**
CWE-685: Function Call With Incorrect Number of Arguments			100 %			100 %			100 %
CWE-676: Use of Potentially Dangerous Function		100 %							100 %
CWE-674: Uncontrolled Recursion								100 %	100 %
CWE-526: Information Exposure Through Environmental Variables		100 %							100 %
CWE-500: Public Static Field Not Marked Final						100 %			100 %
CWE-672: Operation on a Resource after Expiration or Release				91 %					91 %
CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition		50 %	100 %			100 %			83 %
CWE-688: Function Call With Incorrect Variable or Reference as Argument			100 %			100 %		39 %	80 %
CWE-481: Assigning instead of Comparing			100 %				100 %	39 %	80 %
CWE-123: Write-what-where Condition	79 %								79 %
CWE-242: Use of Inherently Dangerous Function		100 %			100 %	100 %		6 %	76 %
CWE-467: Use of sizeof() on a Pointer Type			100 %	100 %				22 %	74 %
CWE-587: Assignment of a Fixed Address to a Pointer							100 %	39 %	69 %

Applicable Recall per CWE on Juliet C/C++

CWE	Tool F	Tool H	Tool B	Tool A	Tool E	Tool C	Tool D	Tool G	Recall/CWE
CWE-484: Omitted Break Statement in Switch			100 %					39 %	69 %
CWE-482: Comparing instead of Assigning			100 %					39 %	69 %
CWE-561: Dead Code		50 %	50 %	100 %				50 %	63 %
CWE-562: Return of Stack Variable Address	67 %		67 %	33 %	100 %		33 %	67 %	61 %
CWE-480: Use of Incorrect Operator			100 %					6 %	53 %
CWE-196: Unsigned to Signed Conversion Error				100 %				6 %	53 %
CWE-775: Missing Release of File Descriptor or Handle after Effective Lifetime		53 %	51 %						52 %
CWE-773: Missing Reference to Active File Descriptor or Handle		53 %	51 %						52 %
CWE-511: Logic/Time Bomb						50 %			50 %
CWE-459: Incomplete Cleanup			50 %						50 %
CWE-426: Untrusted Search Path						50 %			50 %
CWE-483: Incorrect Block Delimitation			95 %				5 %	45 %	48 %
CWE-195: Signed to Unsigned Conversion Error		59 %	32 %	87 %	11 %				47 %
CWE-476: NULL Pointer Dereference	78 %	47 %	60 %	55 %	52 %		20 %	4 %	45 %
CWE-194: Unexpected Sign Extension		63 %	24 %	72 %	7 %				42 %
CWE-188: Reliance on Data/Memory Layout		50 %	47 %	50 %				14 %	40 %
CWE-478: Missing Default Case in Switch Statement								39 %	39 %
CWE-680: Integer Overflow to Buffer Overflow		60 %	49 %	30 %		17 %			39 %
CWE-416: Use After Free		67 %	70 %	55 %	33 %	0 %		2 %	38 %
CWE-675: Duplicate Operations on Resource		2 %	67 %						34 %

Applicable Recall per CWE on Juliet C/C++									
CWE	**Tool F**	**Tool H**	**Tool B**	**Tool A**	**Tool E**	**Tool C**	**Tool D**	**Tool G**	**Recall/CWE**
CWE-396: Declaration of Catch for Generic Exception			33 %						33 %
CWE-468: Incorrect Pointer Scaling		3 %	51 %				49 %	30 %	33 %
CWE-563: Unused Variable		43 %		64 %	36 %	5 %		5 %	31 %
CWE-401: Improper Release of Memory Before Removing Last Reference ('Memory Leak')		67 %	54 %	29 %	31 %	0 %		1 %	30 %
CWE-762: Mismatched Memory Management Routines			58 %	25 %	47 %	14 %	2 %		29 %
CWE-377: Insecure Temporary File		38 %	38 %		13 %				29 %
CWE-415: Double Free		48 %	44 %	35 %	41 %		2 %	1 %	28 %
CWE-761: Free of Pointer not at Start of Buffer					28 %				28 %
CWE-571: Expression is Always True			19 %	50 %			6 %	38 %	28 %
CWE-570: Expression is Always False			13 %	56 %			6 %	38 %	28 %
CWE-475: Undefined Behavior for Input to API								28 %	28 %
CWE-469: Use of Pointer Subtraction to Determine Size								28 %	28 %
CWE-400: Uncontrolled Resource Consumption ('Resource Exhaustion')		24 %	54 %			5 %			27 %
CWE-134: Uncontrolled Format String		42 %	26 %	19 %		25 %	42 %	2 %	26 %
CWE-690: Unchecked Return Value to NULL Pointer Dereference		59 %	2 %			15 %			25 %
CWE-457: Use of Uninitialized Variable	59 %	15 %	43 %	17 %	43 %	0 %	15 %	2 %	24 %
CWE-369: Divide By Zero	79 %		19 %	31 %	6 %		1 %	1 %	23 %
CWE-427: Uncontrolled Search Path Element		12 %	32 %	27 %			20 %		23 %

Applicable Recall per CWE on Juliet C/C++									
CWE	Tool F	Tool H	Tool B	Tool A	Tool E	Tool C	Tool D	Tool G	Recall/CWE
CWE-789: Uncontrolled Memory Allocation			26 %	20 %					23 %
CWE-506: Embedded Malicious Code						22 %			22 %
CWE-252: Unchecked Return Value		11 %	34 %	40 %		14 %		7 %	21 %
CWE-190: Integer Overflow or Wraparound	40 %			21 %			0 %		20 %
CWE-390: Detection of Error Condition Without Action						20 %	20 %		20 %
CWE-127: Buffer Under-read	61 %	9 %	14 %	13 %		17 %		0 %	19 %
CWE-121: Stack-based Buffer Overflow	74 %	21 %	14 %	12 %	3 %	25 %	2 %	1 %	19 %
CWE-606: Unchecked Input for Loop Condition			32 %					4 %	18 %
CWE-590: Free of Memory not on the Heap		7 %	37 %		27 %			0 %	18 %
CWE-124: Buffer Underwrite ('Buffer Underflow')	61 %	9 %	5 %	21 %		9 %		0 %	17 %
CWE-122: Heap-based Buffer Overflow	38 %	24 %	5 %	12 %	1 %	23 %	1 %		15 %
CWE-126: Buffer Over-read	60 %	2 %	7 %	7 %		24 %	0 %	0 %	14 %
CWE-23: Relative Path Traversal		10 %		11 %			20 %		14 %
CWE-197: Numeric Truncation Error				25 %				2 %	13 %
CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')		20 %	13 %	11 %		20 %		2 %	13 %
CWE-588: Attempt to Access Child of a Non-structure Pointer			15 %		9 %				12 %
CWE-398: Indicator of Poor Code Quality							20 %	2 %	11 %
CWE-36: Absolute Path Traversal		10 %		10 %					10 %
CWE-253: Incorrect Check of Function Return Value			5 %			13 %			9 %

Applicable Recall per CWE on Juliet C/C++									
CWE	**Tool F**	**Tool H**	**Tool B**	**Tool A**	**Tool E**	**Tool C**	**Tool D**	**Tool G**	**Recall/CWE**
CWE-191: Integer Underflow (Wrap or Wraparound)				18 %			0 %		9 %
CWE-667: Improper Locking			6 %						6 %
CWE-338: Use of Cryptographically Weak PRNG								6 %	6 %
CWE-404: Improper Resource Shutdown or Release		4 %	3 %					2 %	3 %
CWE-665: Improper Initialization		1 %							1 %
CWE-758: Reliance on Undefined, Unspecified, or Implementation-Defined Behavior								1 %	1 %
Average Applicable Recall	**56 %**	**25 %**	**25 %**	**21 %**	**19 %**	**18 %**	**8 %**	**2 %**	**21 %**

Table 69 summarizes the applicable recall results per CWE for each tool on the Juliet Java test cases.

Table 69. Applicable Recall per CWE on Juliet Java.

Applicable Recall per CWE on Juliet Java					
CWE	**Tool M**	**Tool L**	**Tool O**	**Tool N**	**Recall /CWE**
CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')		100 %			100 %
CWE-775: Missing Release of File Descriptor or Handle after Effective Lifetime		100 %	100 %	100 %	100 %
CWE-760: Use of a One-Way Hash with a Predictable Salt		100 %	100 %		100 %
CWE-643: Improper Neutralization of Data within XPath Expressions ('XPath Injection')		100 %			100 %
CWE-614: Sensitive Cookie in HTTPS Session Without 'Secure' Attribute		100 %			100 %
CWE-609: Double-Checked Locking	100 %	100 %			100 %
CWE-586: Explicit Call to Finalize()	100 %	100 %	100 %	100 %	100 %
CWE-585: Empty Synchronized Block	100 %	100 %	100 %	100 %	100 %
CWE-584: Return Inside Finally Block	100 %	100 %			100 %
CWE-579: J2EE Bad Practices: Non-serializable Object Stored in Session		100 %			100 %
CWE-572: Call to Thread run() instead of start()	100 %	100 %	100 %	100 %	100 %
CWE-566: Authorization Bypass Through User-Controlled SQL Primary Key		100 %			100 %
CWE-549: Missing Password Field Masking		100 %			100 %
CWE-539: Information Exposure Through Persistent Cookies		100 %			100 %
CWE-535: Information Exposure Through Shell Error Message		100 %			100 %
CWE-534: Information Exposure Through Debug Log Files		100 %			100 %
CWE-533: Information Exposure Through Server Log Files		100 %			100 %
CWE-526: Information Exposure Through Environmental Variables		100 %			100 %
CWE-523: Unprotected Transport of Credentials		100 %			100 %
CWE-484: Omitted Break Statement in Switch			100 %	100 %	100 %
CWE-478: Missing Default Case in Switch Statement	100 %		100 %		100 %
CWE-477: Use of Obsolete Functions		100 %			100 %
CWE-470: Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')		100 %			100 %
CWE-396: Declaration of Catch for Generic Exception	100 %				100 %

Applicable Recall per CWE on Juliet Java					
CWE	**Tool M**	**Tool L**	**Tool O**	**Tool N**	**Recall /CWE**
CWE-395: Use of NullPointerException Catch to Detect NULL Pointer Dereference	100 %	100 %			100 %
CWE-383: J2EE Bad Practices: Direct Use of Threads	100 %	100 %			100 %
CWE-338: Use of Cryptographically Weak PRNG		100 %	100 %		100 %
CWE-336: Same Seed in PRNG		100 %			100 %
CWE-315: Cleartext Storage of Sensitive Information in a Cookie		100 %			100 %
CWE-256: Plaintext Storage of a Password		100 %			100 %
CWE-253: Incorrect Check of Function Return Value				100 %	100 %
CWE-252: Unchecked Return Value		100 %	100 %	100 %	100 %
CWE-226: Sensitive Information Uncleared Before Release		100 %			100 %
CWE-15: External Control of System or Configuration Setting		100 %			100 %
CWE-114: Process Control	100 %	100 %			100 %
CWE-111: Direct Use of Unsafe JNI		100 %			100 %
CWE-597: Use of Wrong Operator in String Comparison	100 %	100 %	94 %	94 %	97 %
CWE-483: Incorrect Block Delimitation	89 %			95 %	92 %
CWE-328: Reversible One-Way Hash		100 %	67 %		83 %
CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')		100 %	100 %	47 %	82 %
CWE-36: Absolute Path Traversal		100 %	100 %	43 %	81 %
CWE-23: Relative Path Traversal		100 %	100 %	43 %	81 %
CWE-481: Assigning instead of Comparing	6 %	100 %	100 %	100 %	76 %
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')		100 %	80 %	47 %	76 %
CWE-397: Declaration of Throws for Generic Exception	75 %				75 %
CWE-209: Information Exposure Through an Error Message	50 %	100 %			75 %
CWE-833: Deadlock	50 %	67 %	67 %	100 %	71 %
CWE-319: Cleartext Transmission of Sensitive Information		100 %	40 %		70 %
CWE-476: NULL Pointer Dereference		66 %	64 %	80 %	70 %
CWE-765: Multiple Unlocks of a Critical Resource	50 %	50 %		100 %	67 %
CWE-764: Multiple Locks of a Critical Resource	50 %	50 %		100 %	67 %
CWE-382: J2EE Bad Practices: Use of System.exit()	100 %	50 %	50 %	50 %	63 %
CWE-690: Unchecked Return Value to NULL Pointer Dereference		59 %			59 %

Applicable Recall per CWE on Juliet Java					
CWE	**Tool M**	**Tool L**	**Tool O**	**Tool N**	**Recall /CWE**
CWE-601: URL Redirection to Untrusted Site ('Open Redirect')		54 %	100 %	6 %	53 %
CWE-327: Use of a Broken or Risky Cryptographic Algorithm		53 %			53 %
CWE-832: Unlock of a Resource that is not Locked	50 %	50 %		50 %	50 %
CWE-772: Missing Release of Resource after Effective Lifetime	50 %	50 %	50 %	50 %	50 %
CWE-674: Uncontrolled Recursion			50 %	50 %	50 %
CWE-390: Detection of Error Condition Without Action	50 %	50 %			50 %
CWE-404: Improper Resource Shutdown or Release	20 %	60 %	40 %	60 %	45 %
CWE-83: Improper Neutralization of Script in Attributes in a Web Page		54 %	6 %	63 %	41 %
CWE-80: Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)		54 %	3 %	63 %	40 %
CWE-563: Unused Variable	92 %	29 %	15 %	15 %	38 %
CWE-398: Indicator of Poor Code Quality	76 %		12 %	12 %	34 %
CWE-81: Improper Neutralization of Script in an Error Message Web Page		54 %	6 %		30 %
CWE-571: Expression is Always True	6 %	69 %		6 %	27 %
CWE-482: Comparing instead of Assigning	6 %	65 %		6 %	25 %
CWE-400: Uncontrolled Resource Consumption ('Resource Exhaustion')		24 %			24 %
CWE-570: Expression is Always False	6 %	69 %	6 %	6 %	22 %
CWE-113: Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')		41 %	4 %	4 %	16 %
CWE-259: Use of Hard-coded Password		27 %	9 %	9 %	15 %
Average Applicable Recall	**78 %**	**73 %**	**52 %**	**39 %**	**58 %**

Appendix H: Complete Versions of Tables of CVEs Found and Missed

For readability, in Sec. 3.2.3.3, Tables 29 to 33 omitted columns for tools that did not find any CVE. For completeness, we include the versions of the tables with all columns here.

Table 70. CVEs Found and Missed on Asterisk.

Difficulty	CVE	Type	Tool H	Tool J	Tool B	Tool A	Tool G	Tool C	Tool E	Tool K
Simple	CVE-2012-1183	BOF	Match	Match	Match	Miss	Miss	Miss	Miss	Miss
	CVE-2013-2686	REX	Match	Match	Miss	Match	Miss	Miss	Miss	Miss
	CVE-2012-2415	BOF	Match	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-1184	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-2416	NPD	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-2947	NPD	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-3553	NPD	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-2948	NPD	Miss	Miss	Miss	Miss	Miss	Miss		
Medium	CVE-2012-3812	FREE	Miss	Miss	Miss	Miss	Miss	Miss		
Extreme	CVE-2012-5977	REX	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-4737	IAC	Miss		Miss		Miss		Miss	
	CVE-2012-3863	REX	Miss	Miss	Miss	Miss	Miss	Miss		
	CVE-2012-2186	IAC	Miss							
	CVE-2012-2414	IAC	Miss							

Table 71. Simple-rated CVEs Found and Missed on Wireshark.

Difficulty	CVE	Type	Tool A	Tool C	Tool J	Tool I	Tool B	Tool H	Tool D	Tool E	Tool K
Simple	CVE-2012-5240	BOF	Match	Miss	Match	Miss	Match	Match	Miss	Miss	Miss
	CVE-2013-2475	NPD	Match	Miss	Match	Miss	Match	Miss	Miss	Match	Miss
	CVE-2013-2481	REX	Match	Miss	Miss	Hint	Miss	Miss	Miss	Miss	
	CVE-2012-4285	DIV	Match	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-4286	DIV	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-4296	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1587	ASRT	Miss	Miss	Miss	Miss	Miss	Miss	Miss		
	CVE-2012-4293	ASRT	Miss	Miss	Miss	Miss	Miss		Miss		
	CVE-2012-5238	ASRT	Miss	Miss	Miss	Miss	Miss		Miss		

Table 72. Medium-rated CVEs Found and Missed on Wireshark.

Difficulty	CVE	Type	Tool A	Tool C	Tool J	Tool I	Tool B	Tool H	Tool D	Tool E	Tool K
Medium	CVE-2013-3559 (1)	BOF	Miss	Partial	Match	Partial	Miss	Miss	Miss	Miss	Miss
	CVE-2012-4298	BOF	Partial	Miss	Miss	Miss	Miss	Match	Miss	Miss	Miss
	CVE-2013-3559 (2)	BOF	Miss	Partial	Miss	Partial	Miss	Miss	Miss	Miss	Miss
	CVE-2013-4074	REX	Hint	Miss	Match	Hint	Hint	Miss	Miss	Miss	Miss
	CVE-2013-4082	BOF	Miss	Partial	Miss	Miss	Miss	Partial	Miss	Miss	Miss
	CVE-2013-3562	REX	Miss	Hint	Miss	Match	Miss	Miss	Miss	Miss	Miss
	CVE-2012-4294 / CVE-2012-4295	BOF	Match	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-2480	BOF	Miss	Hint	Miss	Hint	Miss	Miss	Miss	Miss	Miss
	CVE-2013-2487	LOOP	Miss	Hint	Miss	Hint	Miss	Miss	Miss	Miss	
	CVE-2012-4048	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-4049	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-4297	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-6059	PTR	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1579	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1582	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1588	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1590	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-2483	DIV	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-2484	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-2488	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-3557	INI	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-4076	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-4935	INI	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-4081	LOOP	Miss	Miss	Miss	Miss	Miss	Miss		Miss	Miss
	CVE-2012-3548	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss		
	CVE-2013-1575	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss		
	CVE-2013-2476	LOOP		Miss		Hint	Miss				
	CVE-2013-4933	REX	Miss		Miss	Miss	Miss				
	CVE-2012-5237	LOOP		Miss		Miss	Miss				
	CVE-2013-2485	LOOP		Miss		Miss	Miss				
	CVE-2013-4080	LOOP		Miss		Miss	Miss				

Table 73. Hard-rated CVEs Found and Missed on Wireshark.

Difficulty	CVE	Type	Tool A	Tool C	Tool J	Tool I	Tool B	Tool H	Tool D	Tool E	Tool K
	CVE-2012-6062	LOOP	Partial	Miss	Miss		Miss	Miss	Miss	Miss	
	CVE-2013-1573	LOOP	Miss	Partial	Miss		Miss	Miss	Miss	Miss	
	CVE-2013-4930	REX	Miss	Miss	Match	Miss	Miss	Miss	Miss	Miss	
	CVE-2013-1585	BOF	Partial	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-2478	BOF		Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-6061	LOOP	Miss		Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2013-1574	LOOP	Miss		Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2013-1580	LOOP	Miss		Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2013-1572	LOOP	Hint	Miss	Miss	Hint	Miss	Miss	Miss	Miss	Miss
	CVE-2013-2482	LOOP	Miss	Miss	Miss	Hint	Miss	Miss	Miss	Miss	
	CVE-2012-4292	PTR	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
Hard	CVE-2012-6060	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1583	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1584	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-1586	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-4075	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-4077	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-6056	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2012-6058	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2012-4287	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss		
	CVE-2012-6055	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss		
	CVE-2012-6053	LOOP		Miss		Miss	Miss				
	CVE-2013-2479	LOOP		Miss		Miss	Miss				

Table 74. Extreme-rated CVEs Found and Missed on Wireshark.

Difficulty	CVE	Type	Tool A	Tool C	Tool J	Tool I	Tool B	Tool H	Tool D	Tool E	Tool K
Extreme	CVE-2013-3558	BOF	Miss	Miss	Match	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2012-4288	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss		
	CVE-2012-4289	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss		
	CVE-2012-4290	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss		
	CVE-2012-6054	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss		
	CVE-2013-3560	FSTR	Miss	Miss	Miss	Miss	Miss	Miss	Miss		
	CVE-2012-4291	REX	Miss		Miss	Miss	Miss				
	CVE-2013-4078	LOOP		Miss		Miss	Miss				
	CVE-2013-3561 (2)	LOOP	Miss		Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2013-3561 (1)	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2013-4927	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss		
	CVE-2013-1581	LOOP		Miss		Miss	Miss				
	CVE-2013-4079	LOOP		Miss		Miss	Miss				
	CVE-2013-4929	LOOP		Miss		Miss	Miss				
	CVE-2012-6057	LOOP	Miss		Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2013-4083	BOF	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss
	CVE-2013-4931	LOOP	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	
	CVE-2013-1577	LOOP	Miss	Miss	Miss	Miss	Miss		Miss		
	CVE-2013-1578	REX	Miss	Miss	Miss	Miss	Miss		Miss		
	CVE-2013-1576	LOOP		Miss		Miss	Miss				