NIST Special Publication 500-297

Report on the Static Analysis Tool Exposition (SATE) IV

Vadim Okun
Aurelien Delaitre
Paul E. Black

http://dx.doi.org/10.6028/NIST.SP.500-297

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

NIST Special Publication 500-297

Report on the Static Analysis Tool Exposition (SATE) IV

Vadim Okun
Aurelien Delaitre
Paul E. Black
Software and Systems Division
Information Technology Laboratory

http://dx.doi.org/10.6028/NIST.SP.500-297

January 2013

U.S. Department of Commerce
Rebecca Blank, Acting Secretary

National Institute of Standards and Technology
Patrick D. Gallagher, Under Secretary of Commerce for Standards and Technology and Director

National Institute of Standards and Technology Special Publication 500-297
Natl. Inst. Stand. Technol. Spec. Publ. 500-297, 46 pages (January 2013)
http://dx.doi.org/10.6028/NIST.SP.500-297
CODEN: NSPUE2

Abstract

The NIST Software Assurance Metrics And Tool Evaluation (SAMATE) project conducted the fourth Static Analysis Tool Exposition (SATE IV) to advance research in static analysis tools that find security defects in source code. The main goals of SATE were to enable empirical research based on large test sets, encourage improvements to tools, and promote broader and more rapid adoption of tools by objectively demonstrating their use on production software.

Briefly, eight participating tool makers ran their tools on a set of programs. The programs were four pairs of large code bases selected in regard to entries in the Common Vulnerabilities and Exposures (CVE) dataset and approximately 60 000 synthetic test cases, the Juliet 1.0 test suite. NIST researchers analyzed approximately 700 warnings by hand, matched tool warnings to the relevant CVE entries, and analyzed over 180 000 warnings for Juliet test cases by automated means. The results and experiences were reported at the SATE IV Workshop in McLean, VA, in March, 2012. The tool reports and analysis were made publicly available in January, 2013.

SATE is an ongoing research effort with much work still to do. This paper reports our analysis to date which includes much data about weaknesses that occur in software and about tool capabilities. Our analysis is not intended to be used for tool rating or tool selection.

This paper also describes the SATE procedure and provides our observations based on the data collected. Based on lessons learned from our experience with previous SATEs, we made the following major changes to the SATE procedure. First, we introduced the Juliet test suite that has precisely characterized weaknesses. Second, we improved the procedure for characterizing vulnerability locations in the CVE-selected test cases. Finally, we provided teams with a virtual machine image containing the test cases properly configured to compile the cases and ready for analysis by tools.

This paper identifies several ways in which the released data and analysis are useful. First, the output from running many tools on production software is available for empirical research. Second, our analysis of tool reports indicates the kinds of weaknesses that exist in the software and that are reported by the tools.

Third, the CVE-selected test cases contain exploitable vulnerabilities found in practice, with clearly identified locations in the code. These test cases can help practitioners and researchers improve existing tools and devise new techniques. Fourth, tool outputs for Juliet cases provide a rich set of data amenable to mechanical analysis. Finally, the analysis may be used as a basis for a further study of weaknesses in code and of static analysis.

Keywords

Software security; static analysis tools; security weaknesses; vulnerability

Disclaimer

Certain instruments, software, materials, and organizations are identified in this paper to specify the exposition adequately. Such identification is not intended to imply recommendation or endorsement by NIST, nor is it intended to imply that the instruments, software, or materials are necessarily the best available for the purpose.

Cautions on Interpreting and Using the SATE Data

SATE IV, 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 a small subset of tool warnings.

The procedure that was used for finding CVE locations in the CVE-selected test cases and selecting related tool warnings, though improved since SATE 2010[1], has limitations, so the results may not indicate tools' ability 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 in this exposition differently from their use in practice. We analyzed tool warnings for correctness and looked for related warnings from other tools, whereas developers use tools to determine what changes need to be made to software, and 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 August through October 2011. 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 particular application or the general benefit of using static analysis tools. In Section 4 we suggest appropriate uses of the SATE data.

[1] Previous SATEs included the year in their name, e.g., SATE 2010. Starting with SATE IV, the name has an ordinal number. The change is to prevent confusion about missing years since we no longer conduct SATE annually.

Table of Contents

Executive Summary

The NIST Software Assurance Metrics And Tool Evaluation (SAMATE) project conducted the fourth Static Analysis Tool Exposition (SATE IV) to advance research in static analysis tools. SATE focused on tools that find security defects in source code. The main goals of SATE were to enable empirical research based on large test sets, encourage improvements in tools, and promote broader and more rapid adoption of tools by objectively demonstrating their use on production software.

Briefly, eight participating tool makers ran their tools on a set of programs from two language tracks: C/C++ and Java. Each track included production open source programs selected based on entries in the Common Vulnerabilities and Exposures (CVE) database, called CVE-selected test cases. In addition, there were tens of thousands of synthetic programs, called the Juliet 1.0 test suite.

NIST researchers performed a partial analysis of tool reports. The results and experiences were reported at the SATE IV Workshop in McLean, VA, in March, 2012. The tool reports and analysis were made publicly available in January, 2013. Below, we describe the test cases and tool outputs in more detail.

The CVE-selected test cases were pairs of programs: a vulnerable version with publicly reported security vulnerabilities (CVEs) and a fixed version, that is, a newer version where some or all of the CVEs were fixed. The C programs were Dovecot, a secure mail server, and Wireshark, a network protocol analyzer. The Java programs were Jetty and Tomcat, both servlet containers. The largest program, Wireshark, had 1.6M lines of code. Tool makers ran their tools on both vulnerable and fixed versions. Having tool outputs for both versions allowed us to compare tool results for code with a weakness and code without the weakness.

Tool outputs were converted to the SATE format that included attributes such as warning name, Common Weakness Enumeration (CWE) ID, severity, and location(s) in code. The median number of tool warnings per 1 000 lines of code (kLOC) for tool reports in SATE IV varied from 1.5 warnings per kLOC for Jetty to 6 warnings per kLOC for Dovecot. The number of tool warnings varies widely by tool, due to differences in tool focus, reporting granularity, different tool configuration options, and inconsistencies in mapping of tool warnings to the SATE format.

For ease of presentation in this paper, we grouped warnings into weakness categories: buffer errors, numeric errors, race condition, information leak, improper input validation, security features, improper error handling, API abuse, code quality problems, and miscellaneous.

For the CVE-selected test cases in the C/C++ track, there were no information leak warnings, mostly because these test cases do not output to an external user. The most common warning categories for Dovecot and Wireshark were code quality problems (a broad category including memory allocation, NULL pointer dereference, and other issues), API abuse, improper error handling, and improper input validation. The great majority of API abuse warnings were use of potentially dangerous function.

In contrast, for the CVE-selected test cases in the Java track, there were no buffer errors - most buffer errors are precluded by the Java language. Also, there were no numeric errors, race conditions, or error handling warnings reported. Most warnings for Jetty and Tomcat were improper input validation, including cross-site scripting (XSS).

We used three methods to select tool warnings for analysis from the CVE-selected test cases: (1) random selection, (2) selection of warnings related to manual findings, and (3) selection of warnings related to CVEs. In the first method, we randomly selected a subset of 30 warnings from each tool report, based on weakness name and severity. We 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.

Based on our previous SATE experience, 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. Therefore, we assigned one of the following correctness categories to each warning analyzed: true security weakness, true quality weakness (poor code quality, but may not be reachable or may not be relevant to security), true but insignificant weakness, not a weakness (false positive), and weakness status unknown (we were unable to determine correctness). In the rest of the discussion for this method, we focus on the true security and true quality weaknesses due to their importance.

For both Dovecot and Wireshark, the majority of true security and true quality weaknesses were code quality problems, such as NULL pointer dereference and memory management issues. A possible explanation is that most tools in the C/C++ track were quality oriented. For Dovecot, we did not find any warnings to be true security weaknesses. Indeed, Dovecot was written with security in mind. Hence, it is not likely to have many security problems.

For Java test cases, the vast majority of true security and quality weaknesses were improper input validation, most of which were XSS. Other input validation weaknesses were log file injection, path manipulation, and URL redirection. There were also security features weaknesses, including weak cryptographic algorithm and hard-coded password.

We also considered overlap of warnings between tools, that is, the percentage of true security and true quality weaknesses that were reported by one tool (no overlap), two tools, and three or more tools, respectively. Tools mostly find different weaknesses. Over 2/3 of the weaknesses were reported by one tool only, with very few weaknesses reported by three or more tools. One reason for low overlap is that tools look for different weakness types. Another reason is limited participation; in particular, only two tools were run on the Java test cases.[2] Finally, while there may be many weaknesses in large software, only a relatively small subset may be reported by tools.

There was more overlap for some well-known and well-studied categories, such as buffer errors. Additionally, of 13 true security XSS weaknesses, 6 were reported by both tools that participated in the Java track.

In the second method, security experts performed fuzzing and manual source code review of one of Wireshark's protocol dissectors in order to identify the most important weaknesses. We called these manual findings. The experts found a buffer error, which was the same as one of the CVEs.

In the third method, we identified CVEs and matched warnings from tools. As a result of CVE identification, we collected the CVE description, weakness type or CWE ID where available, and relevant code blocks, such as the sink where user input is used, locations on the path leading to the sink, and locations where the vulnerability was fixed. We based the CVE identification on information in the public vulnerability databases, vulnerability notices, bug tracking, patch, and

[2] MARFCAT reports were submitted late; we did not analyze the reports

version control information, comparison of files from different versions, manual review of the source code, and dynamic analysis.

The 88 CVEs that we identified included a wide variety of weakness types - 30 different CWE IDs. The majority of CVEs in Wireshark were buffer errors and code quality problems, including NULL pointer dereference, while security feature weaknesses were most common in Dovecot. Most CVEs in the Java test cases were improper input validation, including XSS and path traversal, and information leaks.

After identifying the CVEs, we searched the tool reports for related warnings. We found related warnings for about 20% of the CVEs. One possible reason for a small number of matching tool warnings is that our procedure for finding CVE locations in code had limitations. Another reason is a significant number of design level flaw CVEs that are very hard to detect by automated analysis. Also, size and complexity of the code bases may reduce the detection rates of tools.

We found a higher proportion of related warnings for improper input validation CVEs, including XSS and path traversal, and also for pointer reference CVEs. On the other hand, we found no related warnings for information leaks.

Compared to SATE 2010,[3] which also included Wireshark and Tomcat as test cases, we found more related warnings from tools. However, the results cannot be compared directly across SATEs, since the sets of participating tools were different, the list of CVEs was expanded for SATE IV to include newly discovered vulnerabilities, and the procedure for identifying CVEs was improved for SATE IV.

In SATE IV, we introduced a large number of synthetic test cases, called the Juliet 1.0 test suite, consisting of about 60 000 test cases, representing 177 different CWE IDs, and covering various complexities, that is, control and data flow variants.

Since Juliet test cases contain precisely characterized weaknesses, we were able to analyze the tool warnings mechanically. Specifically, a tool warning matched a test case if their weakness types were related and at least one warning location was in an appropriate block of the test case.

Five tools were run on the C/C++ Juliet test cases, while one tool was run on the Java Juliet test cases. At least 4 of 5 tools that were run on the C/C++ test cases detected weaknesses in the categories of buffer errors, numeric errors, and code quality problems. On the other hand, no tool detected weaknesses in the race condition or information leak categories. This may be due to a relatively low number of test cases in these categories.

The numbers of true positives, false positives, and false negatives show that tool recall and ability to discriminate between bad and good code vary significantly by tool and weakness category. A manual reanalysis of a small subset of warnings revealed that our mechanical analysis had errors where warnings were marked incorrectly, including several systematic errors.

For SATE V, we plan to improve our warning analysis guidelines, produce a set of realistic but precisely characterized synthetic test cases by extracting weakness and control/data flow details from CVE-selected test cases, introduce another language track, focus our analysis on an important weakness category and specific aspect of tool performance such as ability to find or parse code, and begin transition to a more powerful tool output format.

[3] Previous SATEs included the year in their name, e.g., SATE 2010. Starting with SATE IV, the name has an ordinal number. The change is to prevent confusion about missing years since we no longer conduct SATE annually.

1 Introduction

SATE IV was the fourth in a series of static analysis tool expositions. It was designed to advance research in static analysis tools that find security-relevant defects in source code. Briefly, participating tool makers ran their tools on a set of programs. NIST researchers performed a partial analysis of test cases and tool reports. The results and experiences were reported at the SATE IV Workshop [29]. The tool reports and analysis were made publicly available in January, 2013. SATE had these goals:

- Enable empirical research based on large test sets.
- Encourage improvement of tools.
- Foster adoption of tools by objectively demonstrating their use on production software.

Our goal was neither to evaluate nor choose the "best" tools.

SATE was aimed at exploring the following characteristics of tools: relevance of warnings to security, their correctness, and prioritization. We based SATE analysis on the textual reports produced by tools — not the richer user interfaces of some tools — which limited our ability to understand the weakness reports.

SATE focused on static analysis tools that examine source code to detect and report weaknesses that can lead to security vulnerabilities. Tools that examine other artifacts, like requirements, and tools that dynamically execute code were not included.

SATE was organized and led by the NIST Software Assurance Metrics And Tool Evaluation (SAMATE) team [23]. The tool reports were analyzed by a small group of analysts, consisting of NIST researchers. The supporting infrastructure for analysis was developed by NIST researchers. Since the authors of this paper were among the organizers and the analysts, we sometimes use the first person plural (we) to refer to analyst or organizer actions. Security experts from Cigital performed time-limited analysis for a portion of one test case.

1.1 Terminology

In this paper, we use the following terminology. A *vulnerability* is a property of system security requirements, design, implementation, or operation that could be accidentally triggered or intentionally exploited and result in a security failure [27]. A vulnerability is the result of one or more *weaknesses* in requirements, design, implementation, or operation.

A *warning* is an issue (usually, a weakness) identified by a tool. A (tool) *report* is the output from a single run of a tool on a test case. A tool report consists of warnings. Many weaknesses can be described using source-to-sink paths. A *source* is where user input can enter a program. A *sink* is where the input is used.

1.2 Previous SATE Experience

We planned SATE IV based on our experience from SATE 2008 [30], SATE 2009 [18], and SATE 2010 [21].[4] The large number of tool warnings and the lack of the ground truth complicated the analysis task in SATE. To address this problem in SATE 2009, we selected a

[4] Previous SATEs included the year in their name, e.g., SATE 2010. Starting with SATE IV, the name has an ordinal number. The change is to prevent confusion about missing years since we no longer conduct SATE annually.

random subset of tool warnings and tool warnings related to findings by security experts for analysis. We found that while human analysis is best for some types of weaknesses, such as authorization issues, tools find weaknesses in many important weakness categories and can quickly identify and describe in detail many weakness instances.

In SATE 2010, we included an additional approach to this problem – CVE-selected test cases. Common Vulnerabilities and Exposures (CVE) [6] is a database of publicly reported security vulnerabilities. The CVE-selected test cases are pairs of programs: an older *vulnerable* version with publicly reported vulnerabilities (CVEs) and a *fixed* version, that is, a newer version where some or all of the CVEs were fixed. For the CVE-selected test cases, we focused on tool warnings that correspond with the CVEs.

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 them are amenable to mechanical analysis.

We also found that the tools' philosophies about static analysis and reporting were often very different, which is one reason they produced substantially different warnings. While tools often look for different types of weaknesses and the number of warnings varies widely by tool, there is a higher degree of overlap among tools for some well known weakness categories, such as buffer errors. More fundamentally, the SATE experience suggested that the notion that weaknesses occur as distinct, separate instances is not reasonable in most cases.

A *simple* weakness can be attributed to one or two specific statements and associated with a specific Common Weakness Enumeration (CWE) [3] entry. In contrast, a non-simple weakness has one or more of these properties:

- Associated with more than one CWE (e.g., chains and composites [5]).
- Attributed to many different statements.
- Has intermingled control flows.

In [30], we estimated that only between 1/8 and 1/3 of all weaknesses are simple weaknesses.

We found that the tool interface was important in understanding most weaknesses – a simple format with line numbers and little additional information often did not provide sufficient context for a user to efficiently and correctly understand a warning. Also, 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 and five in SATE 2010: true security, true quality, true but insignificant, unknown, and false. At the same time, we improved the warning analysis criteria.

1.3 Related Work

Many researchers have studied static analysis tools and collected test sets. Among these, Zheng et. al [36] analyzed the effectiveness of static analysis tools by looking at test and customer-reported failures for three large-scale network service software systems. They concluded that static analysis tools are effective at identifying code-level defects. Also, SATE 2008 found that tools can help find weaknesses in most of the SANS/CWE Top 25 [25] weakness categories [30].

Several collections of test cases with known security flaws are available [13] [15] [24] [37]. Several assessments of open-source projects by static analysis tools have been reported recently

[1] [10] [11]. Walden et al. [33] measured the effect of code complexity on the quality of static analysis. For each of the 35 format string vulnerabilities that they selected, they analyzed both vulnerable and fixed versions of the software. We took a similar approach with the CVE-selected test cases. Walden et al. [33] concluded that successful detection rates of format string vulnerabilities decreased with an increase in code size or code complexity.

Kupsch and Miller [14] 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 of the vulnerabilities requiring a deep understanding of the code or design.

The U.S. National Security Agency's Center for Assured Software [35] ran 9 tools on about 60 000 synthetic test cases covering 177 CWEs and found that static analysis tools differed significantly in precision and recall. Also, tools' precision and recall ordering varied for different weaknesses. One of the conclusions in [35] was that sophisticated use of multiple tools would increase the rate of finding weaknesses and decrease the false positive rate. The Juliet 1.0 test cases, used in SATE IV, are derived, with minor changes, from the set analyzed in [35].

A number of studies have compared different static analysis tools for finding security defects, e.g., [9] [12] [13] [16] [22] [37]. SATE was different in that teams ran their own tools on a set of open source programs. Also, the objective of SATE was to accumulate test data, not to compare tools.

The rest of the paper is organized as follows. Section 2 describes the SATE IV procedure and summarizes the changes from the previous SATEs. Since we made a few changes and clarifications to the SATE procedure after it started (adjusting the deadlines and clarifying the requirements), Section 2 describes the procedure in its final form. Section 3 gives our observations based on the data collected. Section 4 provides summary and conclusions, and Section 5 lists some future plans.

2 SATE Organization

The exposition had two language tracks: a C/C++ track and a Java track.[5] Each track included CVE-selected test cases (Dovecot, Wireshark, Jetty, and Tomcat) and thousands of Juliet test cases. At the time of registration, teams specified which track(s) they wished to enter. We performed separate analysis and reporting for each track. Also at the time of registration, teams specified the version of the tool that they intended to run on the test set(s). We required teams to use a version of the tool having a release or build date that was earlier than the date when they received the test set(s).

2.1 Steps in the SATE procedure

The following summarizes the steps in the SATE procedure. Deadlines are given in parentheses.

- Step 1 Prepare.
  - Step 1a Organizers choose test sets.
  - Step 1b Teams sign up to participate.
- Step 2 Organizers provide test sets via SATE web site (31 July 2011).

[5] We had a PHP track with one CVE-selected test case, WordPress. However, no team participated in the PHP track.

- Step 3 Teams run their tool on the test set(s) and return their report(s) (by 31 Oct 2011).
- Step 4 Organizers analyze the reports, provide the analysis to the teams (12 March 2012).
    - Organizers select a subset of tool warnings for analysis and share with the teams (6 Jan 2012).
    - (Optional) Teams return their review of the selected warnings from their tool's reports (3 Feb 2012).
    - (Optional) Teams check their tool reports for matches to the CVE-selected test cases and return their review (3 Feb 2012).
- Step 5 Report comparisons at SATE IV workshop [29] (29 March 2012).
- Step 6 Publish results (Jan 2013).

2.2 Tools

Table 1 lists, alphabetically, the tools and the tracks in which the tools were applied.

Tool	Version	Tracks	Analyzed Juliet?
Buguroo BugScout[6]	2.0	Java	
Concordia University MARFCAT[7]	SATE-IV.2	C/C++, Java	
Cppcheck	1.49	C/C++	Y
Grammatech CodeSonar[8]	3.7 (build 74177)	C/C++	
LDRA Testbed[9]	8.5.3	C/C++	Y
Monoidics INFER	1.5	C/C++	Y
Parasoft C++test and Jtest	C++test 9.1.1.25 Jtest 9.1.0.20110801	C/C++, Java	Y
Red Lizard Software Goanna	9994 (devel. branch)	C/C++	Y

Table 1: Tools

2.3 Tool Runs and Submissions

Teams ran their tools and submitted reports following these specified conditions.

- Teams did not modify the code of the test cases.
- For each test case, teams did one or more runs and submitted the report(s). See below for more details.
- Teams did not edit the tool reports manually.
- Teams converted the reports to a common XML format. See Section 2.10.1 for description of the format.
- Teams specified the environment (including the operating system and version of compiler) in which they ran the tools. These details can be found in the SATE tool reports available at [31].

Most teams submitted one tool report per test case for the track(s) that they participated in. Buguroo analyzed vulnerable versions of the CVE-selected test cases only. Buguroo did not

[6] Analyzed vulnerable versions of CVE-selected test cases only, did not analyze Juliet test cases
[7] MARFCAT reports were submitted late; we did not analyze the reports
[8] Analyzed CVE-selected test cases, but not the Juliet test cases
[9] Analyzed Dovecot and Juliet test cases only

analyze the Juliet test cases. Grammatech analyzed the CVE-selected test cases, but not the Juliet test cases. LDRA analyzed Dovecot and the Juliet test cases only.

Grammatech CodeSonar was configured to improve analysis of Dovecot's custom memory functions. See "Special case of Dovecot memory management" in Section 2.9.4.

Whenever tool runs were tuned, e.g., with configuration options, the tuning details were included in the teams' submissions.

MARFCAT reports were submitted late. As a result, we did not analyze the output from any of its reports.

In all, we analyzed the output from 28 tool runs for the CVE-selected programs. This counts tool outputs for vulnerable and fixed versions of the same CVE-selected program separately.

Additionally, we analyzed the output from five tool runs for the Juliet C/C++ test cases and one tool run for the Juliet Java test cases.

Several teams also submitted the original reports from their tools in addition to the reports in the SATE output format. During our analysis, we used some information, such as details of weakness paths, from several original reports to better understand the warnings.

Grammatech CodeSonar and LDRA Testbed did not assign severity to the warnings. CodeSonar uses rank, a combination of severity and likelihood, instead of severity. All warnings in their submitted reports had severity 1. We changed the severity for some warning classes in the CodeSonar and Testbed reports based on the weakness names, CWE IDs, and some additional information from the tools.

2.4 Matching warnings based on CWE ID

The following tasks in SATE benefit from a common language for expressing weaknesses: matching warnings to known weaknesses, such as CVEs, and finding warnings from different tools that refer to the same weakness. CWE is such a language, so having CWE IDs simplifies matching tool warnings.

However, an exact match in all cases is still unlikely. First, some weakness classes are refinements of other weakness classes, since CWE is organized into a hierarchy. For instance, XSS CWE-79 is a subclass of the more general Improper Input Validation CWE-20. Accordingly, two warnings labeled CWE-79 and CWE-20 may refer to the same weakness.

Second, a single vulnerability may be the result of a chain of weaknesses or the composite effect of several weaknesses. A *chain* is a sequence of two or more separate weaknesses that can be closely linked together within software [5]. For instance, an Integer Overflow CWE-190 in calculating size may lead to allocating a buffer that is smaller than needed, which leads to a Buffer Overflow CWE-120. Thus two warnings, one labeled as CWE-190 and one as CWE-120, might refer to the same vulnerability.

Before beginning analysis of tool warnings, we performed the following two steps. First, if a tool did not assign CWE IDs to its warnings, we assigned them based on our understanding of the weakness names used by the tool. Second, we analyzed CWE IDs used in tool warnings and Juliet test cases, and combined the CWE IDs into 30 *CWE groups*. The list of CWE groups is included in the released data [31].

We allowed for overlap between different groups, since weaknesses can be assigned different CWE IDs based on different facets. For example, we assigned Access of Uninitialized Pointer CWE 824 to two groups: pointer issues and initialization issues.

Additionally, while preparing this paper, in order to simplify presentation of data, we combined CWE IDs into a small set of non-overlapping weakness categories. The categories are listed in Section 3.1.

2.5 Juliet 1.0 Test Cases

Each Juliet test case was designed to represent a CWE ID and has blocks of bad and good code [4][3]. *Bad code* contains the weakness identified by the CWE ID. The corresponding *good code* is the same as the bad code, except that it does not have the weakness. Each test case targets one weakness instance, but other incidental weakness instances may be present. To simplify the automated analysis, described in the following Section, we ignore any other weaknesses in the test case.

Each test case consists of one or more source files. A test case may use some of a handful of common auxiliary files. Test cases can be compiled as a whole, by CWE ID, or individually.

Table 2 lists some statistics for the Juliet test cases. The second column provides the number of different CWE IDs covered by the test cases. The list of CWE IDs covered by test cases is available in the released data [31]. The last two columns give the number of files and the number of non-blank, non-comment lines of code (LOC). The lines of code and files were counted using SLOCCount by David A. Wheeler [34].

Track	CWE IDs	Test Cases	Files	LOC
C/C++	116	45 309	63 195	6 494 707
Java	106	13 783	19 845	3 226 448
All	177	59 092	83 040	9 721 155

Table 2: Juliet 1.0 test cases

Test cases cover various *complexities*, that is, control and data flow variants. First, *baseline* test cases are the simplest weakness instances without any 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, *control/data flow* test cases combine control and data flow constructs. The detailed list of complexities is available as part of released data [31].

We made the following modifications to Juliet test cases. First, we excluded a small subset of the C/C++ test cases, which were Windows specific and therefore did not compile on Linux. Second, since some other C/C++ test cases caused compiler errors, we made minor changes in order to compile them on Linux. Finally, we wrote a Makefile to support compilation.

2.6 Analysis of Tool Reports for Juliet Test Cases

Since Juliet test cases contain precisely characterized weaknesses, we were able to analyze the tool warnings mechanically. This section describes the analysis process, with the overview given in Figure 1. A tool warning matched a test case if their weakness types were related, that is, their CWE IDs belonged to the same group (explained in Section 2.4) and at least one warning location was in an appropriate block of the test case, detailed as follows.

- If a related warning was in bad code, the tool had a true positive (TP).
- If no related warning was in bad code, the tool had a false negative (FN).
- If a related warning was in good code, the tool had a false positive (FP).
- If no related warning was in good code, the tool had a true negative (TN).
- Any unrelated warnings were disregarded.

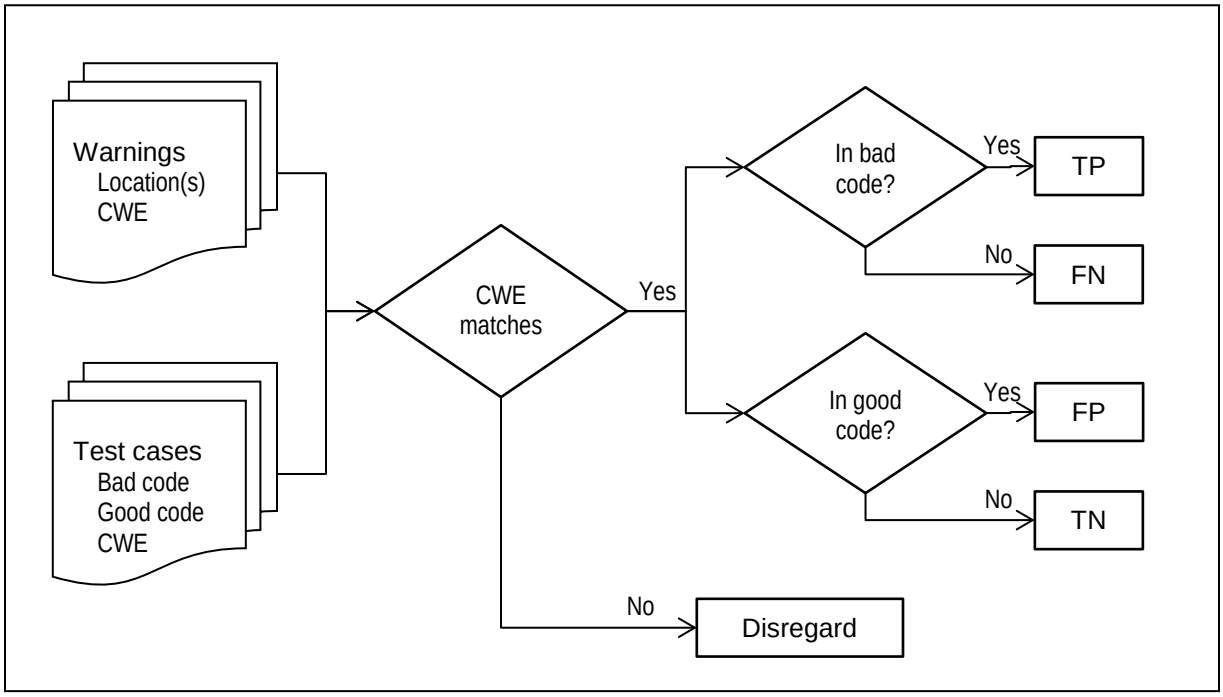

Figure 1: Analysis process for Juliet test cases

2.7 CVE-Selected Test Cases

This section explains how we chose the CVE-selected test cases. We list the test cases, along with some statistics, in Table 3. The last two columns give the number of files and the number of non-blank, non-comment lines of code (LOC) for the test cases. The lines of code and files were counted before compiling the programs. For several test cases, counting after the build process would have produced higher numbers. The table has separate rows for the vulnerable and fixed versions.

The counts for C test cases include C/C++ source (e.g., .c, .cpp, .objc) and header (.h) files. Both Dovecot and Wireshark are C programs. The counts for Dovecot include two C++ files. The counts for the Java test cases include Java (.java) and JSP (.jsp) files. Tomcat ver. 5.5.13 includes 192 C files. Tomcat ver. 5.5.33 does not include any C files. Each version of Jetty includes one C file. The C files were not included in the counts for Tomcat and Jetty. The counts do not include source files of other types: make files, shell scripts, Assembler, Perl, PHP, and SQL.

The lines of code and files were counted using SLOCCount by David A. Wheeler [34].

The links to the test case developer web sites, as well as links to download the exact versions analyzed, are available at the SATE web page [31].

Test case	Track	Description	Version	# Files	# LOC
Dovecot	C/C++	Secure IMAP and POP3 server	1.2.0	811	147 220
			1.2.17	818	149 991
Wireshark		Network protocol analyzer	1.2.0	2281	1 625 396
			1.2.18	2278	1 633 554
Jetty	Java	Servlet container	6.1.16	698	95 721
			6.1.26	727	104 326
Apache Tomcat		Servlet container	5.5.13	1494	180 966
			5.5.33	1602	197 758

Table 3: CVE-selected test cases

Wireshark and Tomcat were among the CVE-selected test cases in SATE 2010. This year, we used the same vulnerable versions as in SATE 2010, but we used newer fixed versions. In preparation to SATE IV, we reanalyzed Wireshark and Tomcat using an improved procedure for finding CVE locations in code. A newer version of Dovecot was used as a test case in SATE 2010, but it was not a CVE-selected test case. The rest of this Section describes the test case selection and CVE identification process.

We considered dozens of candidate programs while selecting the test cases. We looked for test cases with various security defects, over 10 000 lines of code, and compilable using a commonly available compiler. In addition, we used the following criteria to select the CVE-based test cases and also to select the specific versions of the test cases.

- Program had several, preferably dozens, of vulnerabilities reported in the CVE database.
- Reliable resources, such as bug databases and source code repositories, were available for locating the CVEs.
- We were able to find the source code for a version of the test case with CVEs present (vulnerable version).
- We were able to identify the location of some or all CVEs in the vulnerable version.
- We were able to find a newer version where some or all CVEs were fixed (fixed version).
- Both vulnerable and fixed versions were available for Linux OS.
- Many CVEs were in the vulnerable version, but not in the fixed versions.
- Both versions had similar design and directory structure.

There is a tradeoff between the last two items. Having many CVEs fixed between the vulnerable and fixed versions increased the chance of a substantial redesign between the versions.

We used several sources of information in selecting the test cases and identifying the CVEs. First, we exchanged ideas within the NIST SAMATE team and with other researchers. Second, we used several lists to search for open source programs [1] [11] [19] [28]. Third, we used several public vulnerability databases [6] [8] [17] [20] to identify the CVEs.

The selection process for the CVE-based test cases included the following steps. The process was iterative, and we adjusted it in progress.

- Identify potential test cases – popular open source software written in C, C++ or Java and likely to have vulnerabilities reported in CVE.
- Collect a list of CVEs for each program.

- For each CVE, collect several factors, including CVE description, versions where the CVE is present, weakness type (or CWE if available), version where the CVE is fixed, and patch.
- Choose a smaller number of test cases that best satisfy the above selection criteria.
- For each CVE, find where in the code it is located.

We used the following sources to identify the appropriate CWE IDs for the CVE entries. First, National Vulnerability Database (NVD) [17] entries often contain CWE IDs. Second, for some CWE entries, there is a section Observed Examples with links to CVE entries. Two CVE entries from SATE 2010 occurred as Observed Examples: CVE-2010-2299 for CWE-822 and CVE-2008-0128 for CWE-614. Finally, we sometimes assigned the CWE ids as a result of a manual review.

Locating a CVE in the code is necessary for finding related warnings from tools. Since a CVE location can be either a single statement or a block of code, we recorded the starting line number and block length in lines of code. If a warning refers to any statement within the block of code, it may be related to the CVE.

As we noted in Section 1, a weakness is often associated with a path, so it cannot be attributed to a single line of code. Also, sometimes a weakness can be fixed or corrected in a different part of code. Accordingly, we attempted to find three kinds of locations:

- Fix – a location where the code has been fixed.
- Sink – location where user input is used.
- Path – location that is part of the path leading to the sink.

The following example, a simplified version of CVE-2009-3243 in Wireshark, demonstrates different kinds of locations. The statements are far apart in the code.

```
  // Index of the missing array element
  #define SSL_VER_TLSv1DOT2   7

   const gchar* ssl_version_short_names[] = {
       "SSL",
       "SSLv2",
       "SSLv3",
       "TLSv1",
       "TLSv1.1",
       "DTLSv1.0",
       "PCT",
  // Fix: the following array element was missing in the vulnerable version
+      "TLSv1.2"
  };

  // Path: may point to SSL_VER_TLSv1DOT2
  conv_version =&ssl_session->version;

  // Sink: Array overrun
  ssl_version_short_names[*conv_version]
```

Since the CVE information is often incomplete, we used several approaches to find CVE locations in code. First, we searched the CVE description and references for relevant file or function names. Second, we reviewed the program's bug tracking, patch, and version control log

information, available online. We also reviewed relevant vulnerability notices from Linux distributions that included these programs. Third, we used "diff" to compare the corresponding source files in the last version with a CVE present and the first version where the CVE was fixed. This comparison often showed the fix locations. Fourth, we manually reviewed the source code. Finally in preparation for SATE IV, we expanded our understanding of some CVEs by performing dynamic analysis, as explained below.

2.7.1 Improving CVE Identification Dynamically

The fix for a given CVE is often situated somewhere between the source and the sink. Tools, on the other hand, usually report locations in the neighborhood of the sink and sometimes of the source. This means that there is little overlap between the fix location and the tool warning location, even if they target the same weakness. To address this issue, we expanded our description of the CVEs with as much of the source-to-sink path as we could determine.

A reliable and efficient way to do so consists of exploiting the vulnerabilities while the attacked program is under observation. For the C test cases, we used mainly the GNU debugger gdb and the profiling utility valgrind. These tools would notably produce a trace of the call stack at the time of the crash. They may also detect memory corruption, but sometimes in a delayed manner, which makes analysis more challenging.

Using these tools, it became possible to find the sink of a vulnerability, provided that we had the exploit to trigger it. Luckily, most Wireshark bug reports were augmented with a packet trace that triggered the bug. We just had to run the protocol analyzer inside the debugger and ask it to read the faulty trace file. This would typically produce a crash, caught by the debugger. From there, we were often able to track the control flow back to the sink. Note that the source was known, since we were always reading packets using the same mechanism.

Dovecot was more challenging; with help of its developers, we created several exploits. In particular, the default configuration of the service had to be altered, in order to enable some features that had vulnerabilities.

For Jetty and Tomcat, when CVE description or other sources of information provided inputs and configuration parameters that triggered the CVE, we attempted to reproduce it while running the test cases in debug mode, and then we identified relevant locations in code.

2.8 Analysis of Tool Reports for CVE-Selected Test Cases

Finding all weaknesses in a large program is impractical. Also, due to the large number of tool warnings, analyzing all warnings is impractical. Therefore, we selected subsets of tool warnings for analysis.

Figure 2 describes the high-level view of our analysis procedure. We used three methods to select tool warnings. In method 1, we randomly selected a subset of warnings from each tool report. In method 2, we selected warnings related to *manual findings* - weaknesses identified by security experts for SATE. In method 3, we selected warnings related to CVEs in the CVE-based test cases. We performed separate analysis and reporting for the resulting subsets of warnings.

For the selected tool warnings, we analyzed two characteristics. First, we associated (grouped together) warnings that refer to the same (or related) weakness. (See Section 3.4 of [30] for a

discussion of what constitutes a weakness.) Second, we analyzed correctness of the warnings. Also, we included our comments about warnings.

2.8.1 Three Methods for Tool Warning Selection

This section describes the three methods that we used to select tool warnings for analysis.

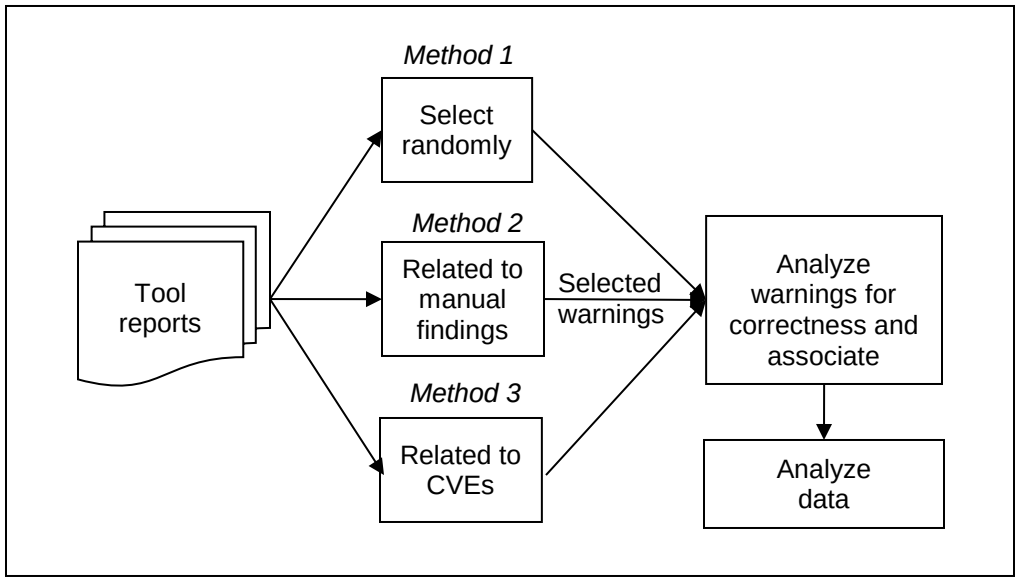

Figure 2: Analysis procedure overview

Method 1 – Select a subset of tool warnings

We selected a total of 30 warnings from each tool report (except one report, which had only 6 warnings) using the following procedure. In this paper, a warning class is a (weakness name, severity) pair, e.g., (Buffer Underrun, 1).

- Randomly selected 1 warning from each warning class with severities 1 through 4.
- While more warnings were needed, we took the following steps.
    - Randomly selected 3 of the remaining warnings (or all remaining warnings if there were less than 3 left) from each warning class with severity 1.
    - Randomly selected 2 of the remaining warnings (or all remaining warnings if there were less than 2 left) from each warning class with severity 2.
    - Randomly selected 1 of the remaining warnings from each warning class (if it still had any warnings left) with severity 3.
- If more warnings were still needed, randomly selected warnings from warning class with severity 4, then randomly selected warnings from warning class with severity 5.

If a tool did not assign severity, we assigned severity based on weakness names and our understanding of their relevance to security.

When finished selecting a set of warnings, we analyzed correctness of the selected warnings and also found associated warnings from other tools, see Section 2.8.3 for details.

Since MARFCAT reports were submitted late, we did not select any of its warnings.

Method 2 – Select tool warnings related to manually identified weaknesses

In this method, security experts analyzed a portion of Wireshark in order to identify the most important weaknesses. In this paper, we call these weaknesses *manual findings*. The human analysis looked for both design weaknesses and source code weaknesses, but focused on the latter. The human analysis combined multiple weakness instances with the same root cause. That is, the security experts did not look for every weakness instance, but instead gave a few (or just one) instances per root cause. Tools were used to aid human analysis, but tools were not the main source of manual findings.

We checked the tool reports to find warnings related to the manual findings. For each manual finding, for each tool, we found at least one related warning, or concluded that there were no related warnings.

Due to the limited resources (about 1.5 person-weeks), only the Intelligent Platform Management Interface (IPMI) protocol dissector – one of many Wireshark protocol decoders – was analyzed. We chose it in consultation with the security experts taking into account the availability of tools for fuzzing, likelihood of important weaknesses, and size of code.

The security experts performed network based fuzzing, packet capture (PCAP) file based fuzzing, and manual source code review. Security experts reported one buffer overrun, corresponding to CVE-2009-2559 in the vulnerable version, and validated that the issue was no longer present in the fixed version.

Assessment methodology is described in a document released as part of SATE data [31].

Method 3 – Select tool warnings related to CVEs

We chose the CVE-based test case pairs and pinpointed the CVEs in code using the criteria and process described in Section 2.4. For each test case, we produced a list of CVEs in SATE output format with additional location information, see Section 2.10.2 for details. We then searched, mechanically and manually, the tool reports to find warnings related to the CVEs.

2.8.2 Practical Analysis Aids

To simplify querying of tool warnings, we put all warnings into a relational database designed for this purpose.

To support human analysis of warnings, we developed a web interface that allows searching the warnings based on different criteria, viewing individual warnings, marking a warning with human analysis which includes opinion of correctness and comments, studying relevant source code files, associating warnings that refer to the same (or a related) weakness, etc.

2.8.3 Analysis Procedure

This section focuses on the procedure for analysis of warnings selected randomly, that is, using Method 1. First, an analyst searched for warnings to analyze (from the list of selected warnings). We analyzed some warnings that were not selected, either because they were associated with selected warnings or because we found them interesting. An analyst usually concentrated his or her efforts on a specific test case, since the knowledge of the test case gained enabled him to analyze other warnings for the same test case faster. Similarly, an analyst often concentrated

textually, e.g., choosing warnings nearby in the same source file. Sometimes an analyst concentrated on warnings of one type.

After choosing a particular warning, the analyst studied the relevant parts of the source code. If he formed an opinion, he marked correctness and/or added comments. If he was unsure about an interesting case, he may have investigated further by, for instance, extracting relevant code into a simple example and/or executing the code.

Next the analyst usually searched for warnings to associate among the warnings on nearby lines. Then the analyst proceeded to the next warning.

Below are two common scenarios for an analyst's work.

Search → View list of warnings → Choose a warning to work on → View source code of the file → Return to the warning → Add an evaluation

Search → View list of warnings → Choose a warning to work on → Associate the warning with another warning

Sometimes, an analyst returned to a warning that had already been analyzed, either because he changed his opinion after analyzing similar warnings or for other reasons. Also, to improve consistency, the analysts communicated with each other about application of the analysis criteria to some weakness classes and weakness instances.

Review by teams

We used feedback from participating teams to improve our analysis. In particular, we asked teams to review the selected tool warnings from their tool reports and provide their findings (optional step in Section 2.1). Several teams submitted a review of their tool's warnings.

In addition, several teams submitted reviews of their tool's results for the CVE selected test cases. Also, some teams presented a review of our analysis at the SATE IV workshop.

2.9 Warning Analysis Criteria for the CVE-Selected Test Cases

This Section describes the criteria that we used for marking correctness of the warnings and for associating warnings that refer to the same weakness.

2.9.1 Overview of Correctness Categories

We assigned one of the following categories to each warning analyzed.

- True security weakness – a weakness relevant to security.
- True quality weakness – poor code quality, but may not be reachable or may not be relevant to security. In other words, the issue requires the developer's attention.
    - Example: buffer overflow where input comes from the local user and the program is not run with super-user privileges, i.e., SUID.
    - Example: *locally true* - function has a weakness, but the function is always called with safe parameters.
- True but insignificant weakness.
    - Example: database tainted during configuration.
    - Example: a warning that describes properties of a standard library function without regard to its use in the code.

- Weakness status unknown – unable to determine correctness.
- Not a weakness – false – an invalid conclusion about the code.

The categories are ordered in the sense that a true security weakness is more important to security than a true quality weakness, which in turn is more important than a true but insignificant weakness.

We describe below the decision process for analysis of correctness, with more details for one weakness category. This is based on our past experience and advice from experts. We consider several factors in the analysis of correctness: context, code quality, and path feasibility.

2.9.2 Decision Process

This section gives an overview of the decision process. The following sections provide details about several factors (context, code quality, path feasibility) used in the decision process.

1. Mark a warning as false if any of the following holds.
    - Path is clearly infeasible.
    - Sink is never a problem, for example.
        - Tool confuses a function call with a variable name.
        - Tool misunderstands the meaning of a function, for example, tool warns that a function can return an array with less than 2 elements, when in fact the function is guaranteed to return an array with at least 2 elements.
        - Tool is confused about use of a variable, e.g., tool warns that “an empty string is used as a password,” but the string is not used as a password.
        - Tool warns that an object can be null, but it is initialized on every path.
    - For input validation issues, tool reports a weakness caused by unfiltered input, but in fact the input is filtered correctly.
2. Mark a warning as insignificant if a path is not clearly infeasible, does not indicate poor code quality, and any of the following holds.
    - A warning describes properties of a function (e.g., standard library function) without regard to its use in the code.
        - For example, "strncpy does not null terminate" is a true statement, but if the string is terminated after the call to strncpy in the actual use, then the warning is not significant.
    - A warning describes a property that may only lead to a security problem in unlikely (e.g., memory or disk exhaustion for a desktop or server system) or local (not caused by an external person) cases.
        - For example, a warning about unfiltered input from a command that is run only by an administrator during installation is likely insignificant.
    - A warning about coding inconsistencies (such as "unused value") does not indicate a deeper problem.

3. Mark a warning as true quality if
    - Poor code quality and any of the following holds.
        - Path includes infeasible conditions or values.
        - Path feasibility is hard to determine.
        - Code is unreachable.
    - Poor code quality and not a problem under the intended security policy, but could become a problem if the policy changes (e.g., a program not intended to run with privileges is run with privileges).
        - For example, for buffer overflow, program is intended not to run with privileges (e.g., setuid) and input not under control of remote user.
4. Mark a warning as true security if path is feasible and weakness is relevant to security.
    - For input validation issues, mark a warning as true security if input is filtered, but the filtering is not complete. This is often the case for cross-site scripting weaknesses.

The decision process is affected by the type of the weakness considered. The above list contains special cases for some weakness types. In Appendix A of [21], we list the decision process details that are specific to one particular weakness type: information leaks.

2.9.3 Context

In SATE runs, a tool does not know about context (environment and the intended security policy) for the program and may assume the worst case.

For example, if a tool reports a weakness that is caused by unfiltered input from command line or from local files, mark it as true (but it may be insignificant - see below). The reason is that the test cases are general purpose software and we did not provide any environmental information to the participants.

Often it is necessary to answer the following questions.

- Who can set the environment variables?
    - For web applications, the remote user.
    - For desktop applications, the user who started the application.
- Is the program intended to be run with privileges?
- Who is the user affected by the weakness reported?
    - Regular user.
    - Administrator.

2.9.4 Poor Code Quality vs. Intended Design

A warning that refers to poor code quality is usually marked as true security or true quality. On the other hand, a warning that refers to code that is unusual but appropriate should be marked as insignificant.

Some examples that always indicate poor code quality:

- Not checking size of a tainted string before copying it into a buffer.
- Outputting a password.

Some examples that may or may not indicate poor code quality:

- Not checking for disk operation failures.
- Many potential null pointer dereferences are due to the fact that methods such as malloc and strdup return null if no memory is available for allocation. If the caller does not check the result for null, this almost always leads to a null pointer dereference. However, this is not significant for some programs: if a program has run out of memory, seg-faulting is as good as anything else.
- Outputting a phone number is a serious information leak for some programs, but an intended behavior for other programs.

Special case of Dovecot memory management

Dovecot does memory allocation differently from other C programs. Its memory management is described in [26]. For example, all memory allocations (with some exceptions in the data stack) return memory filled with NULLs.

This information was provided to the tool makers, so if a tool reports a warning for this intended behavior, mark it as insignificant.

2.9.5 Path Feasibility

Determine path feasibility for a warning. Choose one of the following:

- Feasible - path shown by tool is feasible. If tool shows the sink only, the sink must be reachable.
- Feasibility difficult to determine – path is complex and contains many complicated steps involving different functions, or there are many paths from different entry points.
- Unreachable – a warning points to code within an unreachable function.
- Infeasible conditions or values – a “dangerous” function is always used safely or a path is infeasible due to a flag that is set in another portion of the code.
    - An example where a function is "dangerous," but always used so that there is no problem.

```
g(int j) {
      a[j] = ‘x’;    // potential buffer overflow
}
… other code …
if (i < size_of_a) {
      g(i);    // but g is called in a safe way
}
```

- An example where the path is infeasible due to a flag that is set elsewhere (e.g., in a different module). In the following example, a tool may mark NULL pointer dereference of arg, which is infeasible because the flag that is set *elsewhere in the code* is never equal FLAG_SET when value is not NULL.

```
if (value != NULL && flag == FLAG_SET) {
      *arg = TRUE;   // arg is never dereferenced when it is NULL
}
```

- Clearly infeasible.

  An example with infeasible path, local.

```
if (a) {
      if (!a) {
            sink
      }
}
```

- Another example: infeasible path, local, control flow within a complete stand-alone block (e.g., a function).

```
char a[10];
if (c)
      j = 10000;
else
      j = 5;
… other code that does not change j or c …
if (!c)
      a[j] = ‘x’;
```

- Infeasible path, another example.

```
if (x == null && y) {
      return 0;
} else if (x == null && !y) {
      return 1;
} else {
      String parts[] = x.split(“:”); // Tool reports NULL pointer
      deference for x – false because x cannot be null here
}
```

- Infeasible path, for example, two functions with the same name are declared in two different classes. Tool is confused about which function is called and considers a function from the wrong class.
- Infeasible path that shows a wrong case taken in a switch statement.

In SATE 2008 and 2009, we assumed perfect understanding of code by tools, so we implicitly had only two options for path feasibility. We marked any warning for an infeasible path as false. However, poor code that is infeasible now may become feasible one day, so it may be useful to bring a warning that points to such a weakness on an infeasible path to the attention of a programmer. Additionally, analysis of feasibility for some warnings took too much time. Therefore, starting in SATE 2010, we marked some warnings on an infeasible path as quality weakness or insignificant.

2.9.6 Criteria for Warning Association

Warnings from multiple tools may refer to the same weakness or weaknesses that are related. In this case, we associated warnings. (The notion of distinct weaknesses may be unrealistic. See Section 3.4 of [30] for a discussion.)

For each selected warning instance, our goal was to find at least one related warning instance (if one existed) from each of the other tools. While a tool may report many warnings related to a particular warning, we did not attempt to find all related warnings from the same tool.

We used the following degrees of association:

- Equivalent – weakness names are the same or semantically similar; locations are the same, or in case of paths the source and the sink are the same and the variables affected are the same.
- Strongly related – the paths are similar, and the sinks or sources are the same conceptually, e.g., one tool may report a shorter path than another tool.
- Weakly related – warnings refer to different parts of a chain or composite; weakness names are different but related in some ways, e.g., one weakness may lead to the other, even if there is no clear chain; the paths are different but have a filter location or another important attribute in common.

The following criteria apply to weaknesses that can be described using source-to-sink paths. Source and sink were defined in Section 1.1.

- If two warnings have the same sink, but the sources are two different variables, mark them as weakly related.
- If two warnings have the same source and sink, but paths are different, mark them as strongly related. However, if the paths involve different filters, mark them as weakly related.
- If one warning contains only the sink, the other warning contains a path, both warnings refer to the same sink, and both use a similar weakness name,
    - If there is no ambiguity as to which variable they refer to (and they refer to the same variable), mark them as strongly related.
    - If there are two or more variables affected and there is no way of knowing which variable the warnings refer to, mark them as weakly related.

2.9.7 Criteria for Matching Warnings to Manual Findings and CVEs

We used the same guidelines for matching warnings to manual findings and for matching warnings to CVEs.

This matching is sometimes different from matching tool warnings from different tools because the tool warnings may be at a different – lower – level than the manual findings or CVEs.

We marked tool warnings as related to manual findings or CVEs in the following cases:

- Directly related.
    - Same weakness instance.
    - Same weakness instance, different perspective. For example, consider a CVE involving NULL pointer dereference caused by a function call that returns NULL earlier on a path. A tool may report the lack of return value checking, not the NULL pointer dereference.
    - Same weakness instance, different paths. For example, a tool may report a different source, but the same sink.
- Indirectly related (or coincidental) – tool reports a lower level weakness that may point the user to the high level weakness.

2.10 SATE Data Formats

Teams converted their tool output to the SATE XML format. Section 2.10.1 describes this tool output format. Section 2.10.2 describes the extension of the SATE format for storing our analysis of CVEs, the extension for evaluated warnings, and the extension for matching tool warnings to CVEs and manual findings. Section 2.10.3 describes the format for storing the lists of associations of warnings. In the future we plan to use the SAFES format [2], instead of our own format. We offered it this year, but all teams used the SATE tool output format.

2.10.1 Tool Output Format

In devising the tool output format, we tried to capture aspects reported textually by most tools. In the SATE tool output format, each warning includes:

- ID - a simple counter.
- (Optional) tool specific ID.
- One or more paths (or traces) with one or more locations each, where each location has:
    - (Optional) ID – path ID. If a tool produces several paths for a weakness, ID can be used to differentiate between them.
    - Line - line number.
    - Path – pathname, e.g., wireshark-1.2.0/epan/dissectors/packet-smb.c.
    - (Optional) fragment - a relevant source code fragment at the location.
    - (Optional) explanation - why the location is relevant or what variable is affected.
- Name (class) of the weakness, e.g., buffer overflow.
- (Optional) CWE ID.
- Weakness grade (assigned by the tool):
    - Severity on the scale 1 to 5, with 1 being most severe.

    - (Optional) probability that the warning is a true positive, from 0 to 1.
    - (Optional) tool_specific_rank - tool specific metric – useful if a tool does not use severity and probability.
- Output - original message from the tool about the weakness. May be in plain text, HTML, or XML.
- (Optional) An evaluation of the issue by a human; not considered to be part of tool output. Note that each of the following fields is optional.
    - Correctness - human analysis of the weakness, one of five categories listed in Section 2.9.
    - Comments.

The XML schema file for the tool output format is available at the SATE web page [31].

2.10.2 Extensions to the Tool Output Format

For the CVE-selected test cases, we manually prepared XML files with lists of CVE locations in the vulnerable and/or fixed test cases. The lists use the tool output format with two additional attributes for the location element:

- Length - number of lines in the block of code relevant to the CVE.
- Type - one of fix/sink/path, described in Section 2.4.

The evaluated tool output format, including our analysis of tool warnings, has additional fields. Each warning includes:

- UID – another ID, unique across all reports.
- Selected – "yes" means that we selected the warning for analysis using Method 1.

The format for analysis of manual findings and CVEs extends the tool output format with the element named Related – one or more tool warnings related to a manual finding:

- UID – unique warning ID.
- ID – warning ID from the tool report.
- Tool – the name of the tool that reported the warning.
- Comment – our description of how this warning is related to the manual finding. For CVEs, the comment included whether the warning was reported in the vulnerable version only or in the fixed version also.

2.10.3 Association List Format

The association list consists of associations - pairs of associated warnings identified by unique warning ids (UID). Each association also includes:

- Degree of association – equivalent, strongly related or weakly related.
- (Optional) comment.

There is one association list per test case.

2.11 Summary of changes since previous SATEs

Based on our experience conducting previous SATEs, we made the following changes to the SATE procedure. First, we introduced thousands of Juliet test cases. These synthetic test cases contain precisely characterized weaknesses, which made mechanical analysis possible.

Second, we used the same CVE-selected test cases for warning subset analysis and for analysis based on CVEs and manual findings. Third, we described CVEs better than in SATE 2010, which resulted in improved matching of tool warnings to the CVEs.

Additionally, the following improvements made SATE easier for participants and analysts. First, we allowed teams more time to run their tools and analysts more time to analyze the tool reports.

Second, since installing a tool is often easier than installing multiple test cases, we provided teams with a virtual machine image containing the test cases properly configured and ready for analysis by tools. Finally, we worked with teams to detect and correct reporting and formatting inconsistencies early in the SATE process.

3 Data and Observations

This section describes our observations based on our analysis of the data collected.

3.1 Warning Categories

The tool reports contain 285 different weakness names. These names correspond to 90 different CWE IDs, as assigned by tools or determined by us. In order to simplify the presentation of data in this paper, we defined categories of similar weaknesses and placed tool warnings into the categories based on their CWE IDs.

Table 4 describes the weakness categories. The detailed list of which CWE IDs are in which category is part of the released data available at the SATE web page [31]. Some categories, such as improper input validation, are broad groups of weaknesses; others, such as race condition and information leak, are more narrow weakness classes. We included categories based on their prevalence and severity.

The categories are similar to those used for previous SATEs. The differences are due to a different set of tools used, differences in the test cases, and a different approach to mapping weakness names to weakness categories. In previous SATEs, we placed warnings into categories based on the weakness names or CWE IDs. In SATE IV, we used a more systematic approach, described in Section 2.4, based on CWE IDs. This resulted in placing some warnings under different weakness categories than in previous SATEs.

We made some changes to the categories in SATE IV based on our experience. First there are no separate categories for insufficient encapsulation and time and state weaknesses, since there were only 12 insufficient encapsulation warnings and no time and state warnings reported. Race condition, which was under time and state category, is a separate category. Second there is no separate category for cross-site scripting (XSS); it is now under improper input validation.

Third improper initialization is now under resource management problems, instead of being directly under code quality problems. Finally there is no separate category for null pointer dereference; it is under pointer and reference problems category.

Name	Abbre-viation	Description	Example types of weaknesses
Buffer errors	buf	Buffer overflows (reading or writing data beyond the bounds of allocated memory) and use of functions that lead to buffer overflows	Buffer overflow and underflow, improper null termination
Numeric errors	num-err	Improper calculation or conversion of numbers	Integer overflow, incorrect numeric conversion
Race condition	race	The code requires that certain state not be modified between two operations, but a timing window exists in which the state can be modified by an unexpected actor or process.	File system race condition
Information leak	info-leak	The disclosure of information to an actor that is not explicitly authorized to have access to that information	Verbose error reporting, system information leak
Improper input validation	input-val	Absent or incorrect protection mechanism that fails to properly validate input	XSS, SQL injection, HTTP response splitting, command injection, path manipulation, uncon-trolled format string
Security features	sec-feat	Security features, such as authentication, access control, confidentiality, cryptography, and privilege management	Hard-coded password, insecure randomness, least privilege violation
Improper error handling	err-handl	An application does not properly handle errors that may occur during processing	Incomplete error handling, missing check against null
API abuse	api-abuse	The software uses an API in a manner inconsistent with its intended use	Use of potentially dangerous function
Code quality problems	code-qual	Features that indicate that the software has not been carefully developed or maintained	See below
Resource manage-ment problems	res-mgmt	Improper management of resources	Use after free, double unlock, memory leak, uninitialized variable
Pointer and reference problems	ptr-ref	Improper pointer and reference handling	Null pointer dereference, use of sizeof() on a pointer type
Other quality	qual-other	Other code quality problems	Dead code, violation of coding standards
Miscel-laneous	misc	Other issues that we could not easily assign to any category	

Table 4: Weakness categories

The categories are derived from [4] [7] [32] and other taxonomies. We designed this list specifically for presenting the SATE data only and do not consider it to be a generally applicable classification. We use the abbreviations of weakness category names (the second column of Table 4) in Sections 3.2 and 3.3.

When a weakness type had properties of more than one weakness category, we tried to assign it to the most closely related category.

3.2 Test Case and Tool Properties

In this section, we present all tool warnings grouped in various ways. Figure 3 presents the numbers of tool warnings by test case, for CVE-selected test cases. The number of tools that were run on each test case is included in parenthesis. Related to this figure, five tools were run on the Juliet C/C++ test cases, producing 183 566 warnings; one tool was run on the Juliet Java test cases, producing 2017 warnings.

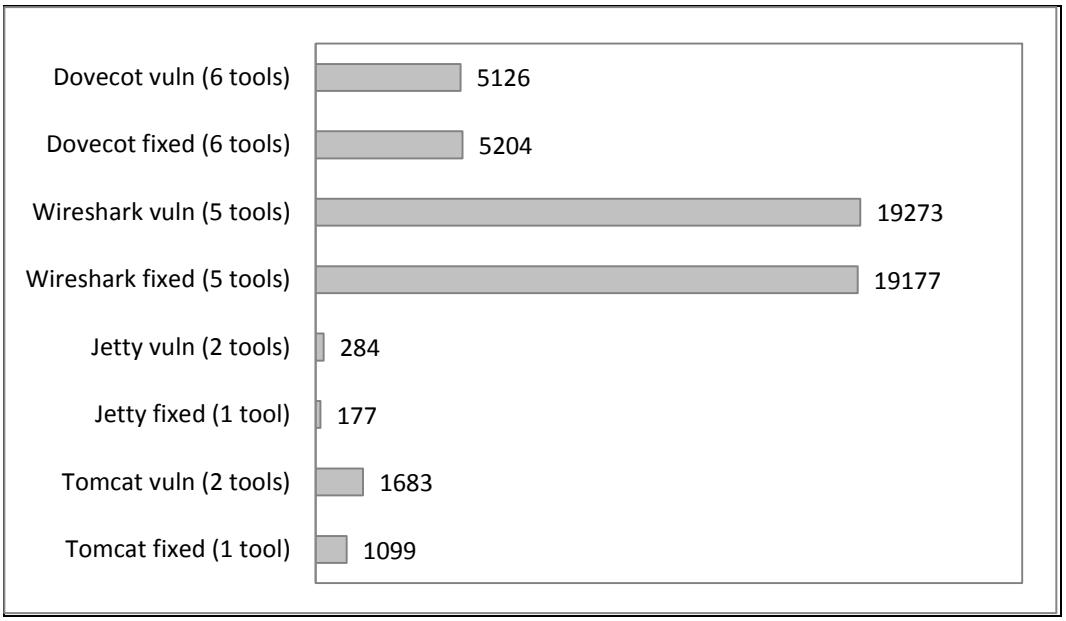

Figure 3: Warnings by test case, for CVE-selected test cases (total 52 023)

Figure 4 and Figure 5 present, for CVE-selected and Juliet test cases respectively, the numbers of tool warnings by severity as determined by the tool, with some changes noted in the next paragraph.

Grammatech CodeSonar and LDRA Testbed did not assign severity to the warnings. For example, Grammatech CodeSonar uses rank (a combination of severity and likelihood) instead of severity. We assigned severity for some warning classes in the CodeSonar and Testbed reports based on the weakness names, CWE IDs, and additional information in the tool outputs.

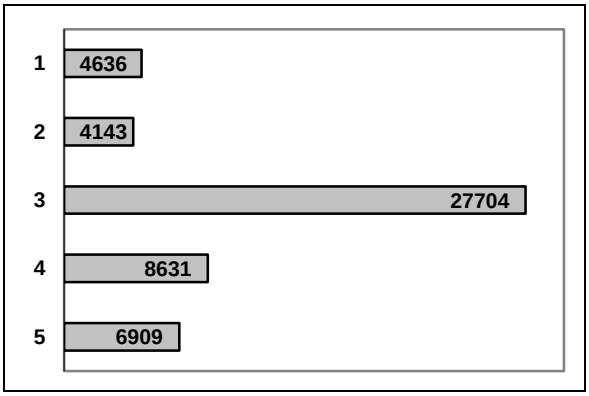

Figure 4: Warnings by severity, for CVE-selected test cases (total 52 023)

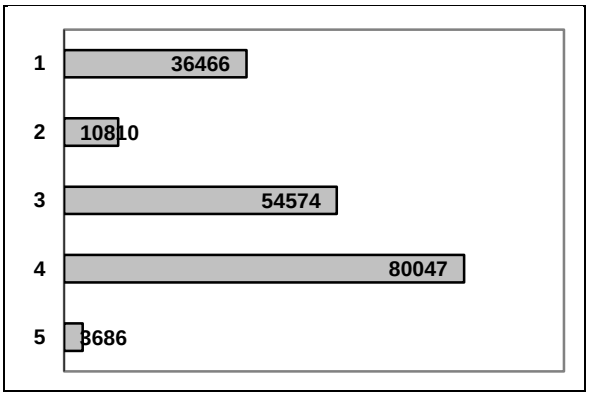

Figure 5: Warnings by severity, for Juliet test cases (total 185 583)

Table 5 below presents, for each CVE-selected test case, the number of warnings per 1000 lines of non-blank, non-comment code (kLOC) in the report with the most warnings (high), the report with the least warnings (low), and the median across all reports. The table presents the numbers for vulnerable versions only. For consistency, we only included the reports from tools that were run on every vulnerable version in a track. In other words, we included reports from 5 tools for the C/C++ track and from 2 tools for the Java track. Accordingly, the numbers in the "median" row for Jetty and Tomcat are the averages of the numbers in the "low" and "high" rows.

The number of warnings varies widely by tool for several reasons. First, tools report different kinds of warnings. In particular, tools focused on compliance may report a very large number of standards violations, while tools focused on security may report a small number of weaknesses. Second, as noted in Section 1.2, the notion that weaknesses occur as distinct, separate instances is not reasonable in most cases; a *single weakness* may be reported based on several distinct criteria. Third, the choice of configuration options greatly affects the number of warnings produced by tool. Finally, there were inconsistencies in the way tool output was mapped to the SATE output format. For example, in one tool's reports for SATE 2010, each weakness path, or trace, was presented as a separate warning, which increased the number of warnings greatly. Hence, tools should not be compared using numbers of warnings.

	Dovecot	Wireshark	Jetty	Tomcat
High	13.54	6.35	1.67	6.47
Median	6.03	1.53	1.49	4.65
Low	0.04	0.04	1.30	2.83
Tool reports	5	5	2	2

Table 5: Low, high, and median number of tool warnings per kLOC for reports in SATE IV

	Dovecot[10]	Wireshark	Chrome	Pebble	Tomcat
High	3.88	1.33	1.27	263.92	27.98
Median	0.905	0.375	0.315	134.87	14.17
Low	0.32	0.18	0.07	5.81	0.36

Table 6: Low, high, and median number of tool warnings per kLOC for reports in SATE 2010

[10] SATE 2010 used Dovecot ver. 2.0 Beta 6, whereas SATE IV used Dovecot ver. 1.2.0

	IRSSI	PVM3	Roller	DMDirc
High	71.64	33.69	64.00	12.62
Median	23.50	8.94	7.86	6.78
Low	0.21	1.17	4.55	0.74

Table 7: Low, high, and median number of tool warnings per kLOC for reports in SATE 2009

	Naim	Nagios	Lighttpd	OpenNMS	MvnForum	DSpace
High	37.05	45.72	74.69	80.81	28.92	57.18
Median	16.72	23.66	12.27	8.31	6.44	7.31
Low	4.83	6.14	2.22	1.81	0.21	0.67

Table 8: Low, high, and median number of tool warnings per kLOC for reports in SATE 2008

For comparison, Table 6, Table 7, and Table 8 present the same numbers as Table 5 for the reports in SATE 2010, SATE 2009, and SATE 2008, respectively. The tables are not directly comparable because not all tools were run in each of the three SATEs. In calculating the numbers in Table 8, we omitted the reports from one of the teams, Aspect Security, which did a manual review.

Weakness category	C/C++ track				Java track			
	All C/C++	Dovecot	Wireshark	Juliet C/C++	All Java	Jetty	Tomcat	Juliet Java
buf	10428	560	978	8890	0	0	0	0
num-err	4348	53	209	4086	0	0	0	0
race	143	102	41	0	0	0	0	0
info-leak	655	0	0	655	95	33	62	0
input-val	23599	753	455	22391	2654	211	1343	1100
sec-feat	2	2	0	0	908	27	12	869
err-handl	36976	459	9068	27449	0	0	0	0
api-abuse	84458	1225	2975	80258	47	0	0	47
code-qual	47353	1972	5544	39837	274	11	262	1
res-mgmt	29862	937	1917	27008	4	4	0	0
ptr-ref	8733	585	979	7169	0	0	0	0
qual-other	8758	450	2648	5660	270	7	262	1
misc	3	0	3	0	6	2	4	0
Total	207965	5126	19273	183566	3984	284	1683	2017

Table 9: Reported warnings by weakness category

Table 9 presents the numbers of reported tool warnings by weakness category for the C/C++ and Java tracks, for individual CVE-selected test cases, and for combined Juliet test cases. For CVE-selected test cases, the table presents the numbers for vulnerable versions only. The weakness categories are described in Table 4.

For the CVE-selected test cases in the C/C++ track, there were no info-leak warnings, mostly because these test cases are not web applications. The most common warning categories for C/C++ track included api-abuse, code-qual, err-handl, and input-val. The great majority of api-abuse warnings were CWE-676 Use of potentially dangerous function. Most err-handl warnings were CWE-252 Unchecked return value.

For the Java track, there were no buf warnings - most buffer errors are not possible in Java. Also, there were no warnings for num-err, race, and err-handl. Most warnings for Java track were input

validation errors, including cross-site scripting (XSS). The second most common warning category was sec-feat. The great majority of sec-feat warnings were CWE-311 Missing encryption of sensitive data.

Using Method 1, introduced in Section 2.8.1, we randomly selected a subset of tool warnings for CVE-selected test cases for analysis. The analysis confirmed that tools are capable of finding weaknesses in a variety of categories. Table 10 presents the numbers of true security and true quality *weaknesses* as determined by the analysts, by weakness category for the tracks and for individual test cases. This counts weaknesses, not individual warnings, since several warnings may be associated with one weakness.

In six cases where warnings for the same weakness belonged to different weakness categories, we chose the most appropriate weakness category manually. In three of these cases, one tool's warning referred to a race condition, while another tool's warning referred to use of a dangerous function that may cause the race condition. The former warning belonged to category race, while the latter belonged to category api-abuse. We chose category race for these weaknesses.

Weakness category	C/C++ track			Java track		
	All C/C++	Dovecot	Wireshark	All Java	Jetty	Tomcat
buf	6	0	6	0	0	0
num-err	7	0	7	0	0	0
race	5	1	4	0	0	0
info-leak	0	0	0	0	0	0
input-val	4	4	0	23	8	15
sec-feat	0	0	0	9	5	4
err-handl	1	0	1	0	0	0
api-abuse	4	2	2	0	0	0
code-qual	45	14	31	0	0	0
res-mgmt	26	8	18	0	0	0
ptr-ref	12	2	10	0	0	0
qual-other	7	4	3	0	0	0
misc	0	0	0	0	0	0
Total	72	21	51	32	13	19

Table 10: True security/quality weaknesses for CVE-selected test cases by weakness category

For Dovecot, we did not find any warnings to be security weaknesses. Indeed, Dovecot was written with security in mind. Hence, it is not likely to have many security problems. For both Dovecot and Wireshark, the majority of weaknesses belonged to code-qual category. A possible reason is that most of the tools in the C/C++ track were quality oriented.

The majority of input-val weaknesses in Java test cases were XSS. Other input-val weaknesses were log file injection, path manipulation, and URL redirection. Sec-feat weaknesses included weak cryptographic algorithm and passwords in source code.

We use Figure 6 and Figure 7 to present overlap of true security and true quality weaknesses between tools. Figure 6 presents overlap by test case: for the CVE-selected test cases, it shows the percentage of weaknesses that were reported by 1 tool (no overlap), 2 tools, and 3 or more tools. The bars have the numbers of weaknesses reported by different numbers of tools.

In parentheses next to the test case name is the number of tools that were run on the test cases. This number does not include MARFCAT, since we did not analyze its results. No true security or quality weakness was reported by more than 4 tools. For example, of 51 true security or quality weaknesses for Wireshark, 35 were reported by 1 tool, 13 were reported by 2 tools, and 3 were reported by 3 or 4 tools.

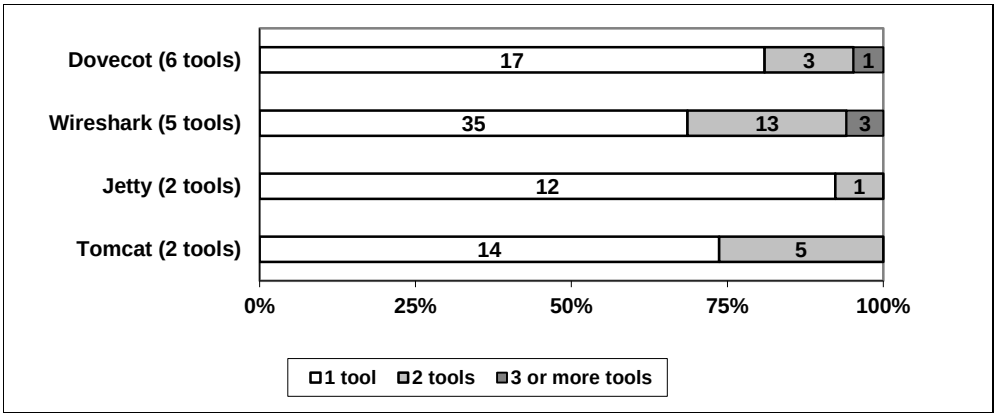

Figure 6: Weaknesses, by number of tools that reported them

As Figure 6 shows, tools mostly find different weaknesses. This is partly due to the fact that tools often look for different weakness types.

We next consider overlap by weakness category for the CVE-selected test cases. Figure 7 shows the percentage of weaknesses that were reported by 1 tool (no overlap), 2 tools, and 3 or more tools. The bars have the numbers of weaknesses reported by different numbers of tools.

In parentheses next to the weakness category is the applicable language track. The Figure excludes weakness categories, such as info-leak, with no confirmed true security and true quality weaknesses. It also excludes weakness categories, such as num-err and sec-feat, which have no overlap between tools. The Figure includes code-qual, but not its subcategories.

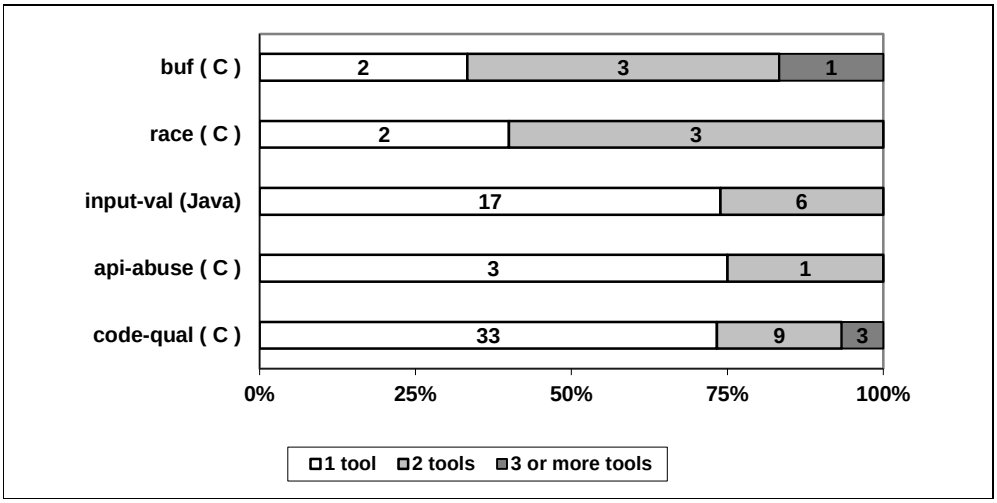

Figure 7: Weaknesses, by number of tools that reported them (excluding categories with no overlap)

Figure 7 shows that there is more overlap for some well-known and well-studied categories, such as buf. Additionally, there is more overlap among cross-site scripting (XSS) weaknesses, a

subset of input-val category. In particular, of 13 true security XSS weaknesses, 6 were reported by 2 tools. Note that only 2 tools were run on the Java test cases. Higher overlap for these categories is consistent with results from earlier SATEs, for example, see Section 3.2 of [18].

Overall, tools handled the code well, which is not an easy task for test cases of this size.

3.3 On our Analysis of Tool Warnings

Using Method 1 introduced in Section 2.8.1, we randomly selected 426 warnings for the CVE-selected test cases (vulnerable versions only) for analysis. It is about 1.6% of the total number of warnings for these four test cases (26458). In the course of analysis we also analyzed 249 other warnings for various reasons. In all we analyzed (associated or marked correctness of) 675 warnings, about 2.6% of the total. In this section, we present data on what portion of test cases was selected for analysis. We also briefly describe the effort that we spent on the analysis.

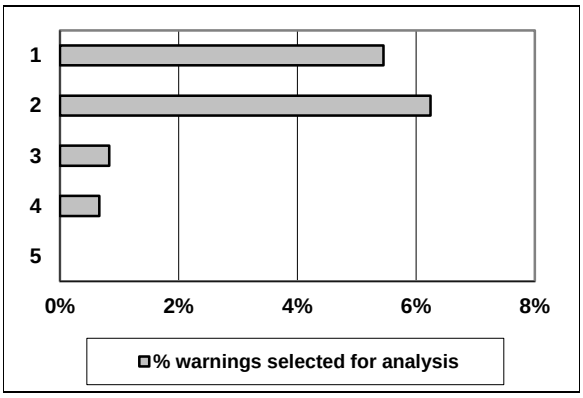

Figure 8: Proportion of warnings for CVE-selected test cases (vulnerable versions only) selected for analysis, by severity

Our selection procedure ensured that we analyzed warnings from each warning class for severities 1 through 4. However, for many warning classes we selected for analysis only a small subset of warnings. Figure 8 presents the percentage of warnings of each severity class selected for analysis. Due to a very large number of severity 3 warnings – about 52% of the total – a small percentage of these warnings were selected.

Three researchers analyzed the tool warnings. All analysts were competent software engineers with knowledge of security; however, the analysts were only occasional users of static analysis tools. The SATE analysis interface recorded when an analyst chose to view a warning and when he or she submitted an evaluation for a warning. The analyst productivity during SATE IV was similar to previous SATEs, see [18] [30] for details.

3.4 CVEs and Manual Findings by Weakness Category

Security experts analyzed a portion of Wireshark and reported one manual finding – a buffer overrun, the same as CVE-2009-2559 – in the vulnerable version, and validated that the issue was no longer present in the fixed version. Since the manual finding is a subset of CVEs, we do not consider it separately in this section.

Table 11 presents the numbers of CVEs in the CVE-selected test cases by weakness category. The table also lists the CWE ids of the CVEs in each weakness category.

The weakness categories were described in Table 4. When a CVE had properties of more than one weakness category, we assigned it to the most closely related category.

Weakness category	CWE ids	All	Dovecot	Wireshark	Jetty	Tomcat
buf	119, 125, 464	14	1	13	0	0
num-err	190, 191	4	0	4	0	0
race	364	1	1	0	0	0
info-leak	200	8	0	0	1	7
input-val	20, 22, 79	22	0	0	4	18
sec-feat	264, 284, 327, 614, 732	11	5	0	0	6
err-handl	391, 460	2	0	1	0	1
api-abuse	628	2	0	2	0	0
code-qual						
res-mgmt	400, 415, 416, 457, 789	8	0	8	0	0
ptr-ref	476, 690	6	0	6	0	0
qual-other	188, 674, 834, 835	9	1	8	0	0
misc	426	1	0	1	0	0
Total		88	8	43	5	32

Table 11: CVEs by weakness category

3.4.1 CVE Analysis Details and Changes since SATE 2010

Three CVEs applied exclusively to the C code portion of Tomcat. Also one CVE applied exclusively to the Nullsoft Scriptable Install System (NSIS) installation script. Since Tomcat was in the Java track, the numbers in this paper do not include these CVEs.

SATE IV used the same vulnerable versions of Wireshark and Tomcat as in SATE 2010. However, SATE 2010 used earlier fixed versions. The CVE analysis for these test cases changed somewhat. First, more vulnerabilities became publicly known since SATE 2010. The number of CVEs for Wireshark increased from 24 to 43, and the number of CVEs for Tomcat increased from 26 to 32.

Second, in preparation to SATE IV, we reanalyzed Wireshark and Tomcat using an improved procedure. As a result, we assigned different CWE IDs to 11 Wireshark CVEs and one Tomcat CVE. Four of these were changes within the same weakness category, to or from a more specific CWE ID. The rest were changes to a different weakness category. For example, CVE-2009-2562 in Wireshark represents a chain of integer overflow leading to buffer overflow. In SATE 2010, we classified it as CWE-190 and placed under num-err category, while in SATE IV we classified it as CWE-125 and placed under buf category.

More substantively, the improved analysis procedure allowed us to find CVE locations in code more accurately, and thus better match tool warnings to the CVEs.

3.5 Tool Warnings Related to CVEs

The description of the CVEs, as well as our listing of the related tool warnings, is available at [31]. Figure 9 presents by test case the numbers of CVEs for which at least one tool produced a directly related warning, an indirectly related warning, or no tool produced a related warning.[11] For definitions of directly related and indirectly related warnings, see Section 2.9.7. The number of CVEs for each test case is in parentheses.

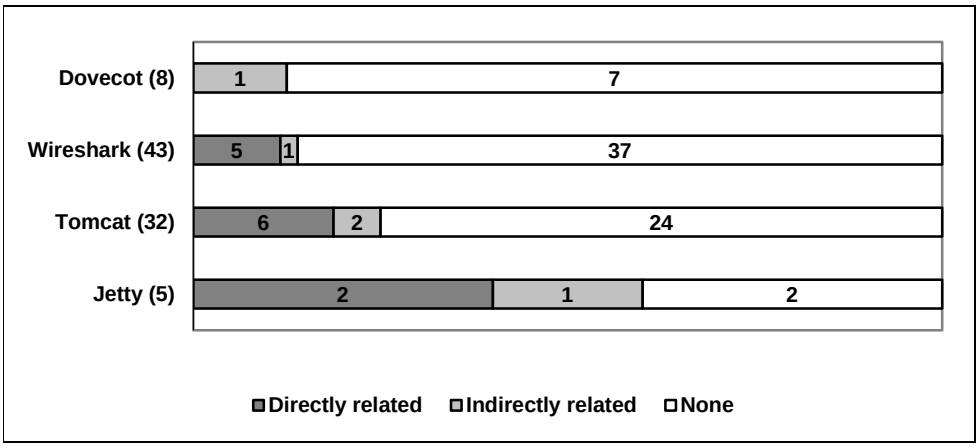

Figure 9: Related warnings from tools, by test case

Figure 10 presents by weakness category the numbers of CVEs for which at least one tool produced a directly related warning, an indirectly related warning, or no tool produced a related warning. Of the 10 XSS weaknesses, counted as part of the input-val category, 7 were found by at least one tool. Similarly, a high proportion of XSS weaknesses was found by tools in SATE 2010. The number of CVEs for each weakness category is in parentheses.

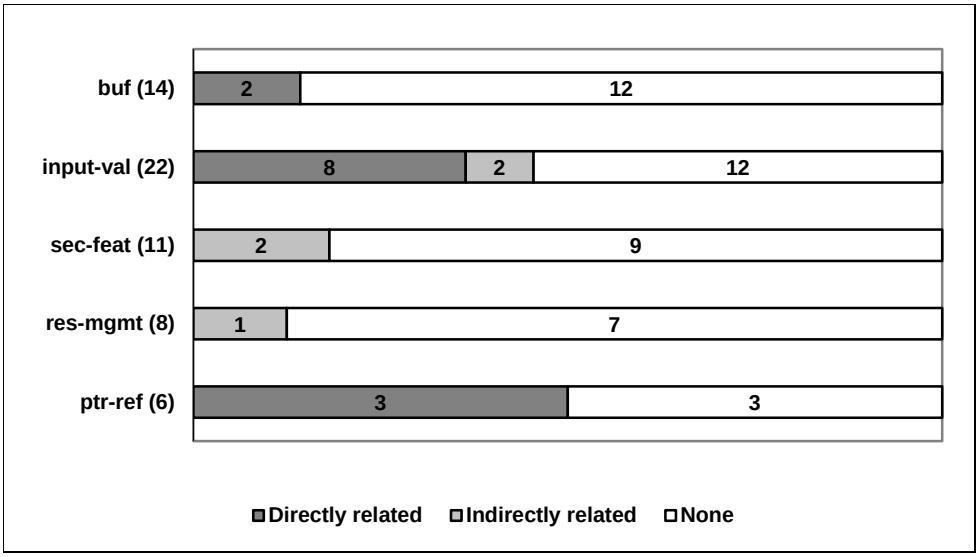

Figure 10: Related warnings from tools, by weakness category (excluding categories with no related warnings)

[11] This Section does not include the results from one of the tools, MARFCAT.

Figure 10 excludes weakness categories with no related warnings. In particular, there were no related warnings for any of the 8 CVEs from info-leak category. The number of CVEs by weakness category for each CVE-selected test case is shown in Table 11.

As detailed in Section 3.6 of [21], here are possible reasons for a low number of matching tool warnings:

- Some CVEs were not identified by tools in default configuration, but could have been identified with tool tuning.
- We may have missed some matches due to the limitations in our procedure for finding CVE locations in code and selecting tool warnings related to the CVEs.
- Some CVEs, such as design level flaws, are very hard to detect by computer analysis.
- There may be other important vulnerabilities in the test cases, which were found by tools, but are unknown to us. Since we do not know about them, we could not credit tools with finding them.
- The CVE-selected test cases are large, have complex data structures, program-specific functions, and complicated control and data flow. This complexity presents a challenge for static analysis tools, especially when run in default configuration.

Compared to SATE 2010, which also included Wireshark and Tomcat as test cases, we found more related warnings. However, the results cannot be compared directly across SATEs, since the sets of participating tools were different, the list of CVEs in SATE IV included newly discovered vulnerabilities, and CVE descriptions were improved for SATE IV.

3.6 Tool Results for Juliet

As explained in Section 2.6, we mechanically analyzed the tool warnings for Juliet. Table 12 presents the number of test cases and tool results per weakness category for the C/C++ test cases in Juliet. Tool results are numbers of true positives (TP) and false positives (FP). Since Juliet is synthetic cases, we mark results as true positive if there is an appropriate warning in flawed (bad) code or false positive if there is an appropriate warning in non-flawed (good) code. Any warning not related to the subject of the test case is not included. For instance, a Java case for CWE-489 Leftover Debug Code had an incidental CWE-259 Hardcoded Password weakness. Warnings about the CWE-259 were ignored. Five tools were run on the Juliet C/C++ test cases. We designated them Tool A, B, C, D, and E.

As shown in Table 12, at least 4 of 5 tools detected weaknesses in buf, num-err, and code-qual weakness categories. On the other hand, no tool detected weaknesses in race, info-leak, and misc categories. This may be due to a relatively low number of test cases in these categories.

The numbers of true positives, false positives, false negatives, as well as other metrics, can be used to identify tool strengths and limitations. Tool E had the highest number of true positives and also covered more weakness categories than any other tool. Tool A, as well as tool E, had twice as many true positives as false positives. Tool B and tool D had almost the same number of false positives as true positives. Tool C had the least false positives; for the weakness that it found, it showed an excellent ability to discriminate between bad and good code.

In looking at the tool results in Table 12, it is important to remember that in Juliet, there are an equal number of good and bad code blocks, whereas in practice, sites with weaknesses appear much less frequently than sites without weaknesses.

Weakness category	Tests	Tool A		Tool B		Tool C		Tool D		Tool E	
		TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
buf	10942	82	71	29	29	833	66	0	0	1117	873
num-err	6746	115	34	64	63	108	0	0	0	751	962
race	133	0	0	0	0	0	0	0	0	0	0
info-leak	209	0	0	0	0	0	0	0	0	0	0
input-val	9343	0	0	52	46	0	0	1063	1062	7699	2995
sec-feat	1185	21	11	0	0	0	0	0	0	0	0
err-handl	2770	0	0	0	0	38	0	93	95	0	0
api-abuse	172	0	0	76	68	0	0	114	114	19	14
code-qual	13623	1388	707	303	278	2476	82	10	38	1664	932
res-mgmt	9712	1331	666	37	35	1437	16	10	38	1358	782
ptr-ref	1412	57	41	43	41	1019	66	0	0	184	90
qual-other	2499	0	0	223	202	20	0	0	0	122	60
misc	186	0	0	0	0	0	0	0	0	0	0
Total	45309	1606	823	524	484	3455	148	1280	1309	11250	5776
Total (as %)	100	3.54	1.82	1.16	1.07	7.63	0.33	2.83	2.89	24.83	12.75

Table 12: True positives and false positives for Juliet C/C++ test cases

Here are some measures that can be produced from the numbers of true positives (TP), false positives (FP), and false negatives (FN).

- False positive rate is FP / (TP + FP).
- Precision, or true positive rate, is TP / (TP + FP).
- Recall is TP / (TP + FN). Recall represents the fraction of weaknesses reported by tool.
- Discriminations and discrimination rate are used to determine whether a tool can discriminate between bad and good code. A tool is given credit for discrimination when it reports a weakness in bad code and does not report the weakness in good code. For every test case, each tool is assigned 0 or 1 discriminations. Over a set of test cases, the discrimination rate is the number of discriminations divided by the number of weaknesses.

A detailed description of these and other metrics can be found in [4].

3.7 Manual Reanalysis of Juliet Tool Results

After completion of the SATE IV workshop, we used the following procedure to manually analyze a small sample of tool results for Juliet.

A status type is either true positive (TP), false positive (FP), false negative (FN), or true negative (TN). We randomly chose 10 results for each of the 5 tools and for each of the 4 status types, for a total of 200 results. Here, a result is uniquely identified by (tool, test case, status) triplet. One of the authors manually checked whether the status type assigned mechanically was correct.

As a result of manual reanalysis, we found several systematic errors in the mechanical analysis. The analysis can be improved by changing CWE groups and fine-tuning the procedure for matching a tool warning location to a test case block. In particular, for some API abuse CWEs where a weakness is manifested in a specific function call a tool warning location should be matched to a specific line, instead of anywhere in the bad code.

The main observations from reanalysis are as follows. First, we marked status as incorrect for 4% of results for the C/C++ test cases only.

Second, one common error type involved matching a warning about memory access weakness to a different memory access weakness in the test case. This error was due to having a CWE group that was too broad and can be corrected by splitting the CWE group.

We are improving the mechanical analysis for the next SATE using the following iterative process. First, we manually reanalyze a small sample of the tool results and record the errors. Second, we make improvements to the mechanical analysis procedure based on observations from reanalysis. We then choose a new sample for reanalysis and repeat. The process ends when we do not notice any systematic errors. However, some errors are unavoidable due to the number of the Juliet test cases and differences in reporting between tools. This is in contrast to the limited number of CVEs in the CVE-selected test cases.

4 Summary and Conclusions

We conducted the Static Analysis Tool Exposition (SATE) IV to enable empirical research on large data sets and encourage improvement and adoption of tools. Based on our observations from the previous SATEs, we made several improvements including a better procedure for characterizing CVE-selected test cases and introduction of the Juliet 1.0 test suite.

Teams ran their tools on four CVE-selected test cases - eight code bases - open source programs from 96k to 1.6M non-blank, non-comment lines of code. Eight teams returned 28 tool reports[12] with about 52k tool warnings. The median number of tool warnings per 1 000 non-blank non-comment lines of code (kLOC) for tool reports varied from 1.5 warnings per kLOC for Jetty to 6 warnings per kLOC for Dovecot. The number of tool warnings varies widely by tool, due to differences in tool focus, reporting granularity, different tool configuration options, and inconsistencies in mapping of tool warnings to the SATE output format.

The types of warnings reported vary by tool, test case properties, and programming languages. There were no information leak warnings for the CVE-selected test cases, Dovecot and Wireshark, in the C/C++ track, mostly because these test cases do not output to an external user.

There were no buffer errors reported for the CVE-selected test cases, Jetty and Tomcat, in the Java track - most buffer errors are precluded by the Java language. Most warnings for Jetty and Tomcat were improper input validation, including cross-site scripting (XSS).

We analyzed less than 3% of tool warnings for the CVE-selected test cases. We selected the warnings for analysis randomly, based on findings by security experts, and based on CVEs.

For both Dovecot and Wireshark, the majority of true security and true quality weaknesses were code quality problems, such as NULL pointer dereference and memory management issues. A possible explanation is that most tools in the C/C++ track were quality oriented.

For Java test cases, the vast majority of true security and quality weaknesses were improper input validation, most of which were XSS.

Tools mostly find different weaknesses. Over 2/3 of the weaknesses were reported by one tool only. Very few weaknesses were reported by three or more tools. One reason for low overlap is

[12] MARFCAT reports were submitted late; we did not analyze the reports

that tools look for different weakness types. Another reason is limited participation; in particular, only two tools were run on the Java test cases. Finally, while there are many weaknesses in large software, only a relatively small subset may be reported by tools. There was more overlap for some well-known and well-studied categories, such as buffer errors and XSS.

The 88 CVEs that we identified included a wide variety of weakness types - 30 different CWE IDs. We found tool warnings related to about 20% of the CVEs. One possible reason for a small number of matching tool warnings is that our procedure for finding CVE locations in code had limitations. Another reason is a significant number of design level flaw CVEs that are very hard to detect by computer analysis. Also, size and complexity of the code bases may reduce the detection rates of tools. A significant effect of code complexity and code size on quality of static analysis results was found in [33].

We found a higher proportion of related warnings for improper input validation CVEs, including XSS and path traversal, and also for pointer reference CVEs. On the other hand, we found no related warnings for information leaks.

For the first time, teams ran their tools on the Juliet suite, consisting of about 60 000 test cases, representing 177 different CWE IDs and covering various complexities, that is, control and data flow variants. 5 teams returned 6 tool reports with about 186k tool warnings. The numbers of true positives, false positives, and false negatives show that tool recall and tool ability to discriminate between bad and good code vary significantly by tool and by weakness category.

Several teams improved their tools based on their SATE experience.

The released data is useful in several ways. First, the output from running many tools on production software is available for empirical research. Second, our analysis of tool reports indicates the kinds of weaknesses that exist in the software and that are reported by the tools.

Third, the CVE-selected test cases contain exploitable vulnerabilities found in practice with clearly identified locations in the code. These test cases can serve as a challenge to the practitioners and researchers to improve existing tools and devise new techniques.

Fourth, tool outputs for Juliet test cases provide a rich set of data amenable to mechanical analysis. Finally, the analysis may be used as a basis for a further study of the weaknesses in the code and of static analysis.

SATE is an ongoing research effort with much work still to do. This paper reports our analysis to date which includes much data about weaknesses that occur in software and about tool capabilities. Our analysis is not intended to be used for tool rating or tool selection.

5 Future Plans

We plan to improve our future analysis in several ways. First, we intend to improve the analysis guidelines by making the structure of the decision process (Section 2.9) more precise, clarifying ambiguous statements, and providing more details for some important weakness categories. Second, we plan to produce more realistic synthetic test cases by extracting weakness and control/data flow details from CVE-selected test cases. These test cases will combine the realism of production code with the ease of analysis of synthetic test cases.

Additionally, we may be able to make the following improvements, which will make SATE easier for participants and more useful to the community.

- Introduce a PHP or .Net language track in addition to the C/C++ and Java tracks.
- Focus analysis on one important weakness category, such as Information Leaks.
- Focus analysis on a specific aspect of tool performance, such as ability to find and parse code.
- Use the new unified Software Assurance Findings Expression Schema (SAFES) [2] as the common tool output format.

6 Acknowledgements

Bill Pugh came up with the idea of SATE. SATE is modeled on the NIST Text Retrieval Conference (TREC): http://trec.nist.gov/. Paul Anderson wrote a detailed proposal for using CVE-selected test cases to provide ground truth for analysis. We thank the NSA Center for Assured Software for contributing the Juliet test suite and helping analyze CVE-selected test cases. Mike Cooper and David Lindsay of Cigital are the security experts that quickly and accurately performed human analysis of a Wireshark dissector.

We thank other members of the NIST SAMATE team for their help during all phases of the exposition. In particular, Kerry Cheng wrote a utility for suggesting associations between warnings from different tools.

We especially thank those from participating teams – Paul Anderson, Fletcher Hirtle, Daniel Marjamaki, Ralf Huuck, Ansgar Fehnker, Clive Pygott, Arthur Hicken, Erika Delgado, Dino Distefano, Pablo de la Riva Ferrezuelo, Alexandro López Franco, David García Muñoz, David Morán, and Serguei Mokhov - for their effort, valuable input, and courage.

7 References

[1] Accelerating Open Source Quality, http://scan.coverity.com/.

[2] Barnum, Sean, Software Assurance Findings Expression Schema (SAFES) Framework, Presentation, Static Analysis Tool Exposition (SATE 2009) Workshop, Arlington, VA, Nov 6, 2009.

[3] Boland, Tim, Overview of the Juliet test suite, Presentation, Static Analysis Tool Exposition (SATE 2010) Workshop, Gaithersburg, MD, Oct 1, 2010.

[4] Center for Assured Software, CAS Static Analysis Tool Study – Methodology, Dec. 2011, http://samate.nist.gov/docs/CAS_2011_SA_Tool_Method.pdf.

[5] Chains and Composites, The MITRE Corporation, http://cwe.mitre.org/data/reports/chains_and_composites.html.

[6] Common Vulnerabilities and Exposures (CVE), The MITRE Corporation, http://cve.mitre.org/.

[7] Common Weakness Enumeration, The MITRE Corporation, http://cwe.mitre.org/.

[8] CVE Details, Serkan Özkan, http://www.cvedetails.com/.

[9] Emanuelsson, Par, and Ulf Nilsson, A Comparative Study of Industrial Static Analysis Tools (Extended Version), Linkoping University, Technical report 2008:3, 2008.

[10] Frye, Colleen, Klocwork static analysis tool proves its worth, finds bugs in open source projects, SearchSoftwareQuality.com, June 2006.

[11] Java Open Review Project, Fortify Software, http://opensource.fortifysoftware.com/.

[12] 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.

[13] Kratkiewicz, Kendra, and Richard Lippmann, Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools, In Workshop on the Evaluation of Software Defect Tools, 2005.
[14] Kupsch, James A. and Barton P. Miller, Manual vs. Automated Vulnerability Assessment: A Case Study, First International Workshop on Managing Insider Security Threats (MIST 2009), West Lafayette, IN, June 2009.
[15] Livshits, Benjamin, Stanford SecuriBench, http://suif.stanford.edu/~livshits/securibench/
[16] Michaud, Frédéric, and Richard Carbone, Practical verification & safeguard tools for C/C++, DRDC Canada – Valcartier, TR 2006-735, 2007.
[17] National Vulnerability Database (NVD), NIST, http://nvd.nist.gov/.
[18] Okun, Vadim, Aurelien Delaitre, and Paul E. Black, The Second Static Analysis Tool Exposition (SATE) 2009, NIST Special Publication 500-287, June 2010.
[19] Open Source Software in Java, http://java-source.net/.
[20] Open Source Vulnerability Database (OSVDB), Open Security Foundation, http://osvdb.org/.
[21] Report on the Third Static Analysis Tool Exposition (SATE 2010), NIST Special Publication 500-283, October 2011, http://samate.nist.gov/docs/NIST_Special_Publication_500-283.pdf, Vadim Okun, Aurelien Delaitre, Paul E. Black, editors.
[22] 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, Nov 2004, http://dx.doi.org/10.1109/ISSRE.2004.1.
[23] SAMATE project, https://samate.nist.gov/.
[24] SAMATE Reference Dataset (SRD), http://samate.nist.gov/SRD/.
[25] SANS/CWE Top 25 Most Dangerous Programming Errors, http://cwe.mitre.org/top25/.
[26] Sirainen, Timo, Dovecot Design/Memory, http://wiki2.dovecot.org/Design/Memory.
[27] Source Code Security Analysis Tool Functional Specification Version 1.1, NIST Special Publication 500-268, Feb 2011, http://samate.nist.gov/docs/source_code_ security_analysis_spec_SP500-268_v1.1.pdf.
[28] SourceForge, Geeknet, Inc., http://sourceforge.net/.
[29] Static Analysis Tool Exposition (SATE IV) Workshop, Co-located with the Software Assurance Forum, McLean, VA, March 29, 2012, http://samate.nist.gov/SATE4Workshop.html.
[30] Static Analysis Tool Exposition (SATE) 2008, NIST Special Publication 500-279, June 2009, Vadim Okun, Romain Gaucher, and Paul E. Black, editors.
[31] Static Analysis Tool Exposition (SATE), http://samate.nist.gov/SATE.html.
[32] Tsipenyuk, Katrina, Brian Chess, and Gary McGraw, "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors," in *Proc. NIST Workshop on Software Security Assurance Tools, Techniques, and Metrics (SSATTM)*, US Nat'l Inst. Standards and Technology, 2005.
[33] 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.
[34] Wheeler, David A., SLOCCount, http://www.dwheeler.com/sloccount/.
[35] Willis, Chuck, CAS Static Analysis Tool Study Overview, In Proc. Eleventh Annual High Confidence Software and Systems Conference, page 86, National Security Agency, 2011, http://hcss-cps.org/.
[36] 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.
[37] Zitser, Misha, Richard Lippmann, and Tim Leek, Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code. In SIGSOFT Software Engineering Notes, 29(6):97-106, ACM Press, New York (2004), http://dx.doi.org/10.1145/1041685.1029911.