NIST Special Publication 500-287

The Second Static Analysis Tool Exposition (SATE) 2009

Vadim Okun

Aurelien Delaitre

Paul E. Black

Software and Systems Division
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899

June 2010

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

Abstract:

The NIST Software Assurance Metrics And Tool Evaluation (SAMATE) project conducted the second Static Analysis Tool Exposition (SATE) in 2009 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, participating tool makers ran their tool on a set of programs. Researchers led by NIST performed a partial analysis of tool reports. The results and experiences were reported at the SATE 2009 Workshop in Arlington, VA, in November, 2009. The tool reports and analysis were made publicly available in 2010.

This paper describes the SATE procedure and provides our observations based on the data collected. We improved the procedure based on lessons learned from the SATE 2008 experience. The changes included random selection of subsets of tool warnings for analysis and also selection based on human analysis, more detailed analysis categories and criteria, an enhanced output format that provides a richer description of weakness paths, and a more detailed and accurate analysis of tool warnings.

The SATE data suggests that while tools often look for different types of weaknesses and the number of warnings varies widely by tool, there is a significant degree of agreement among tools for well-known weakness categories, such as buffer errors. The data also provides evidence that, while human analysis is best suited for identifying some types of weaknesses, tools find a significant portion of weaknesses considered important by human experts.

This paper identifies several ways in which the released data and analysis are useful. First, the output from running many tools on production software can be used for empirical research. Second, the analysis of tool reports indicates actual weaknesses that exist in the software and that are reported by the tools. Finally, the analysis may also be used as a basis for a further study of the security weaknesses and of static analysis.

Keywords:

Software security; static analysis tools; security weaknesses; vulnerability

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 2009, as well as its predecessor, SATE 2008, 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 of SATE.

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 variety and different nature of security weaknesses, defining clear and comprehensive analysis criteria is difficult. While the analysis criteria have been improved since SATE 2008, refinements are necessary and are in progress.

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 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 user interface, 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 late August – early September 2009. 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.

Table of Contents

1 Introduction

SATE 2009 was the second 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 tool on a set of programs. Researchers led by NIST performed a partial analysis of test cases and tool reports. The results and experiences were reported at the SATE 2009 Workshop [19]. The tool reports and analysis were made publicly available in 2010. SATE had these goals:

- To enable empirical research based on large test sets
- To encourage improvement of tools
- To foster adoption of the tools by objectively demonstrating their use on production software

Our goal was not 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 their user interfaces - which limited our ability to understand the weakness reports.

SATE was 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 [13]. The tool reports were analyzed by a small group of analysts, consisting of the NIST and MITRE researchers. The supporting infrastructure for analysis was developed by the NIST researchers. Since the authors of this report 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 2 of the 4 test cases [9].

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 [16]. 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.

We planned SATE 2009 based on our experience from SATE 2008 [18]. In particular, we found that the tool interface was important in understanding most weaknesses – a simple format with line numbers and little additional information did not always provide sufficient context for a user to efficiently and correctly understand a weakness report. Also, a binary true/false positive verdict on tool warnings did not provide adequate resolution to communicate the relation of the warning to the underlying weakness.

We also found that the tools' philosophies about static analysis and reporting were often very different, so they produced substantially different warnings. This complicated our task of analyzing warnings and associating warnings from different tools that refer to the same weakness. For example, tools reported weaknesses at different granularity levels. The SATE 2008 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 [2]).
- Attributed to many different statements.
- Has intermingled flows.

In [18], we estimated that only between 1/8 and 1/3 of all weaknesses are simple weaknesses.

The large number of tool warnings and the lack of the ground truth further complicated the analysis task in SATE 2008. To address this problem, we selected a random subset of tool warnings and tool warnings related to findings by security experts for analysis.

Researchers have studied static analysis tools and collected test sets. Zheng et. al [22] 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 [15] weakness categories [18].

Several collections of test cases with known security flaws are available [8] [23] [10] [14]. Several assessments of open-source projects by static analysis tools have been reported recently [1] [5] [6]. A number of studies have compared different static analysis tools for finding security defects, e.g., [12] [8] [23] [7] [11] [4]. SATE was different in that many 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 2009 procedure and summarizes the changes from SATE 2008. Since we made a few changes and clarifications to the SATE procedure after it started (adjusting the deadlines, clarifying the requirements, and adding the reanalysis step), Section 2 describes the procedure in its final form. Section 3 gives our observations based on the data collected and a summary of the reanalysis results. Section 4 summarizes conclusions and Section 5 lists some future plans.

2 SATE Organization

The exposition had two language tracks: C track and Java track. 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 (by 14 Aug 2009)
- Step 2 Organizers provide test sets via SATE web site (19 Aug 2009)
- Step 3 Teams run their tool on the test set(s) and return their report(s) (by 4 Sep 2009)
- Step 4 Organizers analyze the reports, provide the analysis to the teams (preliminary analysis by 16 Oct 2009, updated analysis by 23 Oct 2009)
    - Organizers select a subset of tool warnings for analysis and share with the teams (by 25 Sep 2009)
    - (Optional) Teams return their review of the selected warnings from their tool's reports (by 6 Oct 2009)
- Step 5 Report comparisons at SATE 2009 workshop [19] (6 Nov 2009)
- Step 6 Organizers reanalyze the warnings that were analyzed previously, provide the updated analysis to the teams (Not planned prior to the exposition, done by April 23 2009)
- Step 7 Publish results (Originally planned for Feb - May 2010, but delayed until June 2010)

2.2 Test Sets

We list the test cases we selected, along with some statistics for each test case, in Table 1. 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 counts for C test cases include source (.c) and header (.h) files. The counts for the Java test cases include Java (.java) and JSP (.jsp) files. The counts do not include source files of other types: make files, shell scripts, Perl, PHP, and SQL. The lines of code were counted using SLOCCount by David A. Wheeler [21].

Test case	Track	Description	Version	# Files	# LOC
IRSSI	C	IRC client	0.8.14	347	52,803
PVM3	C	Parallel virtual machine	3.4	320	72,032
Roller	Java	Weblog server	4.0.1	1057	64,888
DMDirc	Java	IRC client	0.6.3m1	926	63,333

Table 1 Test cases

The links to the test case developer web sites, as well as links to download the versions analyzed, are available at the SATE web page [17].

We spent about 3 weeks selecting the test cases and considered dozens of candidates. In particular, we looked for test cases with various security defects, over 10k lines of code, compilable using a commonly available compiler, etc.

2.3 Tools

Table 2 lists, alphabetically, the participating tools and the tracks in which the tools were applied. One of the teams, Veracode, performed a human review of its reports to remove anomalies such as high false positives in a particular weakness category.

2.4 Tool Runs and Submissions

Teams ran their tools and submitted reports following 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.
- Except for Veracode, the teams did not do any hand editing of tool reports. Veracode performed a human quality review of its reports to remove anomalies such as high false positives in a particular weakness category. This quality review did not add any results.
- Teams converted the reports to a common XML format. See Section 2.6.1 for description of the format.
- Teams specified the environment (including the operating system and version of compiler) in which they ran the tool. These details can be found in the SATE tool reports available at [17].

Tool	**Version**	**Tracks**
Armorize CodeSecure	3.5.9	Java
Checkmarx CxSuite	2.7.5.0	Java
Coverity Prevent	4.5.0	C
Grammatech CodeSonar	3.4p0	C
Klocwork Insight[1]	8.2	C, Java
LDRA Testbed	8.1.0	C
SofCheck Inspector for Java	2.17250, 2.18479[2]	Java
Veracode SecurityReview[3]	As of 08/31/2009	C, Java

Table 2 Tools

Most teams submitted one tool report per test case for the track(s) that they participated in. Klocwork analyzed one test case per track: PVM3 and DMDirc.

Klocwork submitted two runs for DMDirc: the first run used the default settings, while the second run used custom settings. In the custom run, two checkers were turned off and two checkers were tuned to suppress some warnings. We analyzed the output from the second run only. The tuning details were included in their submission and are available as part of the released data.

In all, we analyzed the output from 18 tool runs: 4 from Veracode (participated in 2 tracks) and 2 each from the other 7 tools.

[1] Analyzed PVM3 and DMDirc

[2] SofCheck Inspector build version 17250 was used for DMDirc, build version 18479 – for Roller

[3] A service

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 of the information, such as details of weakness paths, from some of the original reports to better understand the warnings.

Several tools (Grammatech CodeSonar, Coverity Prevent, 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. All warnings in their submitted reports had severity 1. We changed the severity field for some warning classes in the CodeSonar, Prevent, and Testbed reports based on the weakness names and some additional information from the tools.

In the Grammatech CodeSonar report for IRSSI, 6 of 8 buffer overrun warnings appeared due to a tool configuration error: the analysis was done by a compiler configured for 64 bits, but with models configured for 32 bits. We analyzed the warnings from this run. Later, Grammatech submitted the updated run with the tool configured correctly. The updated run is available as part of the released data.

2.5 Analysis of Tool Reports

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

Figure 1 describes the high-level view of our analysis procedure. We used two complementary methods to select tool warnings. In the first method, we randomly selected a subset of warnings from each tool report. In the second method, we selected tool warnings related to manually identified weaknesses. We performed separate analysis and reporting for the two resulting subsets of warnings.

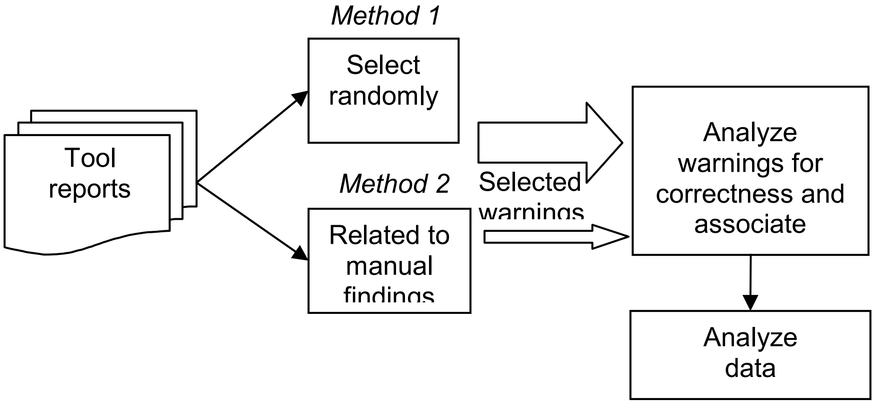
Method 1
Select randomly
Tool reports
Method 2
Related to manual findings
Selected warnings
Analyze warnings for correctness and associate
Analyze data

Figure 1 Analysis procedure overview

For selected tool warnings, we analyzed the following characteristics. First, we associated (grouped together) warnings that refer to the same (or related) weakness. (See Section 3.4 of [18] for a discussion of what constitutes a weakness.) Second, we analyzed correctness of the warnings. Also, we included our comments about warnings.

2.5.1 Two Methods for Tool Warning Selection

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

Method 1 – Select a subset of tool warnings

We selected 30 warnings from each tool report (except one report, which had only 11 warnings) using the following procedure. Here, a warning class is identified by a (weakness name, severity) pair, e.g., (Buffer Underrun, 1).

- Randomly selected one warning from each warning class with severities 1 through 4.
- While more warnings were needed, repeated:
    - 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, selected warnings from warning class with severity 4, then 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.

We analyzed correctness of the selected warnings and also found associated warnings from other tools.

Method 2 – Select tool warnings related to manually identified weaknesses

In this method, security experts manually analyzed one C and one Java test case and identified the most important weaknesses (manual findings). The time-limited human analysis identified both design weaknesses and source code weaknesses focusing on the latter. The human analysis combined multiple weaknesses with the same root cause. Rapid threat modeling was used to guide specific testing activities, including automated analysis, code review, penetration testing, and fuzzing. Tools were used to aid human analysis, but tools were not the main source of manual findings. The methodology of human analysis used is presented in [9]. Due to the limited resources (about 1.5 person-weeks), security experts analyzed two of the four test cases, IRSSI and Roller.

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.

2.5.2 Practical Analysis Aids

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

To support human analysis of warnings, we developed a web interface which allows searching the warnings based on different search 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 related) weakness, etc.

2.5.3 Analysis Procedure

This section focuses on the procedure for analysis of warnings selected 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 that he gained enabled him to analyze other warnings for the same test case faster. Similarly, an analyst often concentrated textually, e.g., choosing warnings near by in the same source file. An analyst also tended to concentrate on warnings of the same 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. 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 → Submit an evaluation

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

Sometimes, an analyst may have 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 had a series of communications about application of the analysis criteria to some weakness classes and weakness instances.

To save time, we used heuristics to partially automate the analysis of some similar warnings. For example, when we determined that a particular source file is executed during installation only, we downgraded severity of certain warning types referring to that source file.

Review by teams

We used feedback from 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.

Additionally, several teams presented a review of our analysis at the SATE 2009 workshop.

2.5.4 Analysis Criteria

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

Correctness categories

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

- True weakness
- True but insignificant weakness
    - Examples: database tainted during configuration or 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 - an invalid conclusion about the code

In the above categories, there are two distinct and independent dimensions: correctness and significance for security.

Criteria for correctness and significance marking

In our analysis of correctness we assumed that:

- A tool has (or should have) perfect knowledge of control/data flow that is explicitly in the code.
    - If a tool reports a weakness on an infeasible path, mark it as false (not a weakness).
    - If a tool reports a weakness that is not present, mark it as false. For example, if a tool reports an error caused by unfiltered input, but in fact the input is filtered correctly, mark it as false.
    - If the input is filtered, but the filtering is not complete, mark it as true. This is often the case for cross-site scripting weaknesses.
    - If a warning says that a function can be called with a bad parameter, but in the test case it is always called with safe values, mark the warning as false.
- A tool does not know about context or environment 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 teams.

In the analysis of significance of a warning, we considered its possible effects on security (integrity, confidentiality, availability). We marked a warning as true but insignificant in these cases:

- A warning describes properties of a function (e.g., standard library function) without regard to its use in the code.
- A warning describes a property that may only lead to a security problem in unlikely and 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.
    - If a warning about coding inconsistencies does not indicate a deeper problem, then it is insignificant.

Criteria for warning association

Tool warnings may refer to the same (or related) weakness. (The notion of distinct weaknesses may be unrealistic. See Section 3.4 of [18] for a discussion.) In this case, we associated them. In contrast to SATE 2008, where any analysis for one warning applied to every associated warning, in SATE 2009, each warning could have a separate analysis.

For each selected warning instance, our goal was to find at least one related warning instance (if it exists) from each of the other tools. While there may be many warnings reported by a tool that are related to a particular warning, we did not attempt to find all of them.

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, where 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. A source is where user input can enter a program. A sink is where the input is used.

- 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, and the other contains a path, the two warnings refer to the same sink and 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.

Criteria for matching warnings related to manual findings

Matching tool warnings to the manual findings is often different from matching tool warnings from different tools because the tool warnings may be at a different – lower – level than the manual findings.

We marked tool warnings related to manual findings with one or more of the following labels:

- Same instance
- Same instance, different perspective
- Same instance, different paths
    - Example: different paths, e.g., different sources, but the same sink
- Coincidental – tool reports a lower level weakness that may point the user to the high level weakness
- Other instance – tool reports a similar weakness (the same weakness type) elsewhere in the code

Due to the possibility of a large number of tool warnings related to a manual finding, we did not attempt to find all associated tool warnings for each manual finding.

2.5.5 Reanalysis

After completion of the SATE 2009 workshop, we reanalyzed all SATE warnings that were analyzed previously (as in the original analysis, we focused on the 521 selected warnings).

Our goals were to (1) improve the analysis quality, (2) identify the areas of the analysis criteria that need improvement, and (3) better understand the types and frequency of errors that we made during the original analysis. We watched for cases where we made a mistake in marking correctness of a warning and where we did not associate a warning with other warnings that refer to the same weakness (we focused on association of warnings from different tools). We used the same analysis criteria as during the original analysis.

The data and observations presented in this paper, unless otherwise specified, include the changes from reanalysis.

2.6 SATE Data Format

Teams converted their tool output to the common SATE XML format. Section 2.6.1 describes this tool output format. Section 2.6.2 describes the extension of the SATE format for storing our analysis of the warnings. Section 2.6.3 describes the extension of the SATE format for our analysis of which tool warnings are related to the manual findings. Section 2.6.4 describes the format for storing the lists of associations of warnings.

2.6.1 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 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.
    - (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, where applicable.
- Weakness grade (assigned by the tool):
    - Severity on the scale 1 to 5, with 1 - the highest.
    - (Optional) probability that the problem 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, either 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 four categories listed in Section 2.5.4.
    - Comments.

The XML schema file for the tool output format is available at the SATE web page [17].

2.6.2 Evaluated Tool Output Format

The evaluated tool output format, including our analysis of tool warnings, has other fields in addition to the tool output format above. Specifically, each warning includes:

- UID – another id, unique across all tool reports.
- Selected – “yes” means that we selected the warning for analysis.

2.6.3 Manual Findings Analysis Format

The format for analysis of manual findings extends the tool output format with the following:

- Related – one or more tool warnings related to a manual finding:
    - UID – unique warning id
    - Summary – one or more of "same instance," "same instance, different perspective," "same instance, different paths," "coincidental," or "other instance"
    - Tool – the name of the tool that reported the warning
    - Comment – our description of how this warning is related to the manual finding.

2.6.4 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.

2.7 Summary of changes since SATE 2008

Based on our experience conducting SATE 2008, we made the following changes to the SATE procedure.

First, we improved the procedure for selecting tool warnings for analysis. For method 1, we randomly selected a subset of warnings from each tool report. This selection method is useful to the tool users because it considers warnings from each tool. For method 2, we selected warnings related to findings by security experts. This selection method is useful to the tool users because it is largely independent of tools and thus includes weaknesses that may not be found by any tools. It also focused analysis on weaknesses found most important by security experts.

Second, based on analysis of SATE 2008 tool warnings, we realized that a binary true/false positive verdict on tool warnings is not enough. Instead, we used 4 correctness categories in SATE 2009. They are true, true but insignificant, false, and unknown.

Third, we added more details to the analysis criteria and modified the association criteria. In particular, since the notion of a distinct weakness is often unrealistic, we associated pairs of warnings instead of larger sets. Also, we allowed for two associated warnings to have a different correctness evaluation.

Fourth, we improved the SATE output format. In particular, we added a richer description of weakness paths. The SATE 2009 output format is backward compatible with the SATE 2008 format.

Finally, to provide more useful feedback to the developers, we selected the latest, beta versions of test cases. Other changes include selecting 2 instead of 3 test cases per track and performing a full reanalysis after completion of the SATE 2009 workshop.

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 83 different valid CWE ids. In addition, there are 81 weakness names for warnings that do not have a valid CWE id. In all, there are 221 different weakness names. This exceeds 83+81 since tools sometimes use different weakness names for the same CWE id. In order to simplify the presentation of data in this report, we placed warnings into categories based on the CWE id and the weakness name, as assigned by tools.

Table 3 describes the weakness categories. The detailed list is part of the released data available at the SATE web page [17]. Some categories are individual weakness classes such as XSS; others are broad groups of weaknesses. We included categories based on their prevalence and severity.

Name	Abbre-viation	Description	Example types of weaknesses
Cross-site scripting (XSS)	xss	The software does not sufficiently validate, filter, escape, and encode user-controllable input before it is placed in output that is used as a web page that is served to other users.	Reflected XSS, stored XSS
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, unchecked array indexing, improper null termination
Numeric errors	num-err	Improper calculation or conversion of numbers	Integer overflow, incorrect numeric conversion, divide by zero
Command injection	cmd-inj	The software fails to adequately filter command (control plane) syntax from user-controlled input (data plane) and then allows potentially injected commands to execute within its context.	OS command injection
Cross-site request forgery (CSRF)	csrf	The web application does not, or cannot, sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request.	
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 intentional or unintentional disclosure of information to an actor that is not explicitly authorized to have access to that information	Verbose error reporting, system information leak

Name	Abbreviation	Description	Example types of weaknesses
Broad categories			
Improper input validation	input-val	Absent or incorrect protection mechanism that fails to properly validate input	Log forging, HTTP response splitting, resource injection, file injection, path manipulation, uncontrolled 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 occur during processing	Incomplete, missing error handling, missing check against null
Insufficient encapsulation	encaps	The software does not sufficiently encapsulate critical data or functionality	Trust boundary violation, leftover debug code
API abuse	api-abuse	The software uses an API in a manner contrary to its intended use	Heap inspection, use of inherently dangerous function
Time and state	time-state	Improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads	Concurrency weaknesses, session management problems
Quality problems	quality	Features that indicate that the software has not been carefully developed or maintained	Null pointer dereference, dead code, uninitialized variable, resource management problems, incl. denial of service due to unreleased resources, use after free, double unlock, memory leak, potential violation of coding standards
Uncategorized	uncateg	Other issues that we could not easily assign to any category	

Table 3 Weakness categories

The categories are similar to those used for SATE 2008. The differences from SATE 2008 are due to a different set of tools used and to differences in the test cases. In particular, there is no separate category for SQL injection in SATE 2009, since there was only one SQL injection warning reported (marked as insignificant). SQL injection is included as part of the broader Improper input validation category. Also, several weakness types concerned with potential violation of coding standards are included in the broad Quality problems category, not in a more specific category such as Numeric errors.

The categories are derived from [3], [20], 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 abbreviations of weakness category names (the second column of Table 3) in Sections 3.2 and 3.3.

Some weakness categories in Table 3 are subcategories of other, broader, categories. In particular, Cross-site scripting (XSS) and Command injection are kinds of improper input validation. Also, Race condition is a kind of Time and state weakness category. Due to their prevalence, we decided to use separate categories for these weaknesses.

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 the division of tool warnings by test case and by severity, the number of tool warnings per report, the division of reported tool warnings by weakness category, and the division of true significant weaknesses by weakness category and by number of tools that reported them.

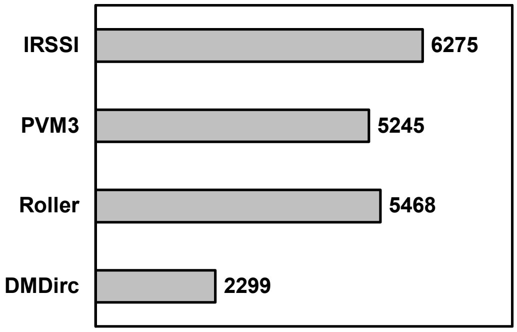

Figure 2 Warnings by test case (total 19287)

Figure 2 presents the numbers of tool warnings by test case. Since LDRA Testbed reports contained a very large number of warnings, we did not include two of the most numerous warnings categories – “Function call with no prior declaration” (7908 warnings) and “Procedure call has no prototype and no defn” (11242 warnings). The numbers in Figure 2 and elsewhere reflect this change.

Figure 3 presents the numbers of tool warnings by severity as determined by the tool. Several tools (Grammatech CodeSonar, Coverity Prevent, 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. All warnings in their submitted reports had severity 1. We changed the severity field for some warning classes in the CodeSonar, Prevent, and Testbed reports based on the weakness names, some additional information from the tools. The numbers in Figure 3 and elsewhere in the report reflect this change.

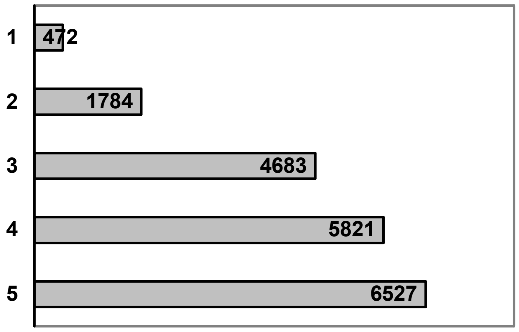

Figure 3 Warnings by severity (total 19287)

Table 4 presents, for each test case, the number of tool warnings per 1,000 lines of non-blank, non-comment code (KLOC) in a report with the most warnings (high), a report with the least warnings (low), and the median number. The number of warnings varies widely by tool, since tools report different kinds of warnings.

	IRSSI	PVM3	Roller	DMDirc
Low	0.21	1.17	4.55	0.74
High	71.64	33.69	64	12.62
Median	23.5	8.94	7.86	6.78

Table 4 Low, high, and median number of tool warnings per KLOC

For comparison, Table 5 presents the same number as Table 4 for the reports in SATE 2008. The tables are not directly comparable, because not all tools were run in both SATE 2008 and SATE 2009. In calculating the numbers in Table 5, we omitted the reports from one of the teams, Aspect Security, which did a manual review.

	Naim	Nagios	Lighttpd	OpenNMS	MvnForum	DSpace
Low	4.83	6.14	2.22	1.81	0.21	0.67
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

Table 5 Low, high, and median number of tool warnings per KLOC for reports in SATE 2008

Table 6 presents the numbers of reported tool warnings by weakness category for the C and Java tracks, as well as for individual test cases. The weakness categories are described in Table 3.

For the C track, there were no xss, csrf, info-leak, and encaps warnings, mostly because the C test cases are not web applications. Also, since it is uncommon to write web applications in C, the tools tend not to look for web application vulnerabilities in the C code. For the Java track, there were no buf warnings - most buffer errors are not possible in Java.

Figure 6 in Section 3.3 shows warnings selected for analysis by weakness category. Table 7 presents the numbers of true significant weaknesses, as determined by the analysts, by weakness category for the C and Java tracks, as well as for individual test cases. This counts weaknesses, not individual warnings referring to the weaknesses.

Weakness category	C track			Java track		
	All C	IRSSI	PVM3	All Java	Roller	DMDirc
xss	0	0	0	59	58	1
buf	182	10	172	0	0	0
num-err	53	22	31	195	59	136
cmd-inj	1	0	1	8	0	8
csrf	0	0	0	106	106	0
race	14	12	2	10	2	8
info-leak	0	0	0	175	150	25
input-val	24	0	24	695	556	139
sec-feat	6	6	0	55	37	18
quality	6545	3486	3059	3119	2212	907
err-handl	84	30	54	2385	1703	682
encaps	0	0	0	295	205	90
api-abuse	2072	2072	0	17	16	1
time-state	1	1	0	160	144	16
uncateg	2538	636	1902	488	220	268
Total	11520	6275	5245	7767	5468	2299

Table 6 Reported warnings by weakness category

Of the 13 input-val weaknesses in Roller, 6 were HTTP response splitting weaknesses. Some tools referred to these weaknesses as XSS, which can indeed be a consequence of HTTP response splitting. In Table 7, these weaknesses are in the input-val category. One weakness in DMDirc was incorrectly marked by tool as an instance of XSS. We assigned it to input-val category instead.

Weakness category	C track			Java track		
	All C	IRSSI	PVM3	All Java	Roller	DMDirc
xss	0	0	0	7	7	0
buf	22	0	22	0	0	0
num-err	2	2	0	3	2	1
cmd-inj	0	0	0	0	0	0
csrf	0	0	0	2	2	0
race	2	2	0	1	1	0
info-leak	0	0	0	8	4	4
input-val	2	0	2	14	13	1
sec-feat	2	2	0	4	1	3
quality	16	8	8	8	2	6
err-handl	8	8	0	0	0	0
encaps	0	0	0	1	1	0
api-abuse	0	0	0	0	0	0
time-state	0	0	0	3	1	2
uncateg	0	0	0	0	0	0
Total	54	22	32	51	34	17

Table 7 True significant weaknesses by weakness category

Table 7 shows that tools are capable of finding weaknesses in a variety of categories. These include not just resource management, XSS and other input validation problems, but also some classes of authentication errors (e.g., hard-coded password, insecure randomness) and information disclosure problems.

Figure 4 presents, for all weakness categories, for buf category, and for all categories except buf, the percentage of true weaknesses that were reported by 1, 2, 3, or 4 of the tools. It also gives, on the bars, the numbers of true weaknesses reported by different numbers of tools. True buf weaknesses were found only in PVM3. Only one true weakness was reported by 4 tools. For reference, 4 tools were run on IRSSI and Roller, 5 tools were run on PVM3 and DMDirc.

As Figure 4 shows, tools find different weaknesses. If the figure were not restricted to true significant weaknesses only, it would show even less overlap between tool results. This is partly due to the fact that tools often look for different weakness types. However, there is more overlap for some well known and well studied categories, such as buf.

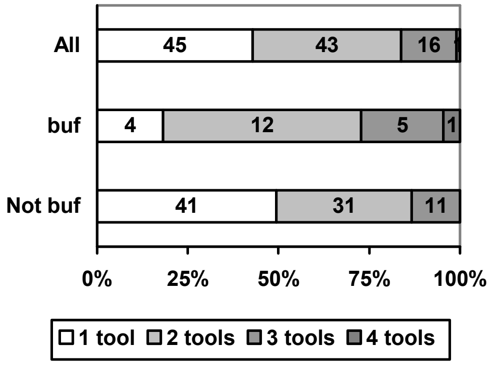

Figure 4 True significant weaknesses, by number of tools that reported them

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

We randomly selected 521 warnings for analysis. It is about 2.7% of the total number of warnings (19287). We also analyzed some other warnings. In all, we analyzed (associated or marked correctness of) 778 warnings, about 4% of the total. In this section, we present data on what portions of test cases and weakness categories were selected for analysis. We also briefly describe the effort that we spent on the analysis.

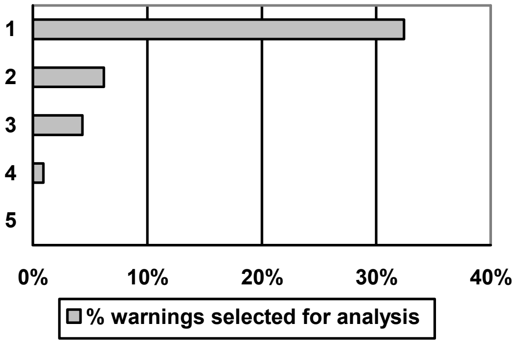

Figure 5 Warnings selected for analysis, by severity

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 5 presents, by severity, the percentage of warnings selected for analysis. Note that this does not consider all warnings that were analyzed, just those selected for analysis.

Figure 6 presents, by weakness category and for all categories, the percentage of warnings that were selected for analysis. It also gives, on the bars, the numbers of warnings that were selected/not selected. We use abbreviations of weakness category names from Table 3. As the figure shows, we analyzed a relatively large portion of xss, buf, cmd-inj, race and sec-feat categories. These are among the most common categories of weaknesses. We were able to analyze all cmd-inj warnings because there were only 9.

Eight people analyzed the tool warnings (spending anywhere from a few hours to a few weeks). All analysts were competent software engineers with knowledge of security; however, most of the analysts were only casual users of static analysis tools.

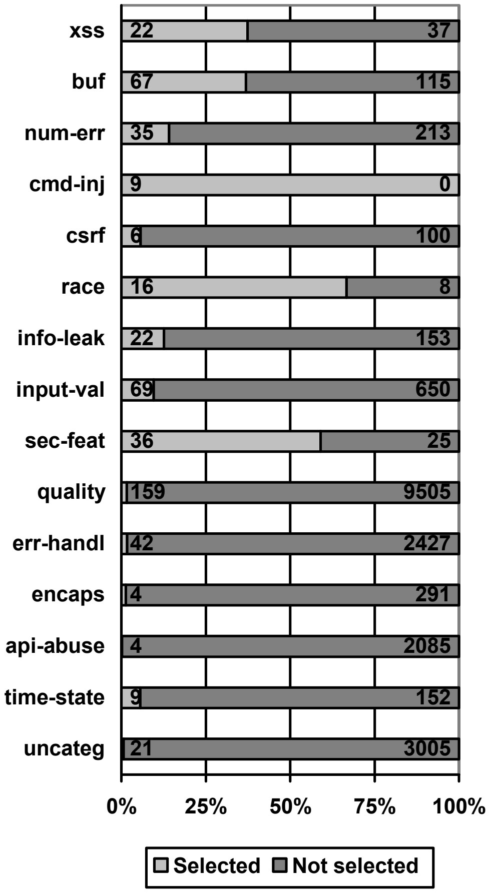

Figure 6 Warnings selected for analysis, by weakness category

The SATE analysis interface recorded when an analyst chose to view a warning and when he submitted an evaluation for a warning. According to these records, the analysis time for an individual warning ranged from less than 1 minute to well over 30 minutes. In particular, analysis of significance was very time consuming, while association of warnings did not take too long and was partially automated based on weakness name and path information.

We did not have a controlled environment for the analysis phase, so these numbers are approximate and may not reflect the actual time the analysts spent. Also, these numbers are not indicative of the time tool users can be expected to spend, because we used the tools differently and had different goals in our analysis.

3.4 Summary of Reanalysis

After completion of the SATE 2009 workshop, we reanalyzed all SATE warnings that were analyzed previously (as in the original analysis, we focused on the 521 selected warnings).

The main data findings are:

1. We changed our analysis of correctness for 131 of 521 selected warnings (25%). Table 8 presents the numbers of warnings for which we changed our analysis of correctness, by type of change. The diagonal contains the numbers of warnings for which we did not change our analysis. In a few cases, we made multiple changes because we did not apply the criteria correctly during the original analysis or because of a change in the assumptions about the test case environment.

To From	True	Insignificant	False	Unknown
True	114	36	9	2
Insignificant	22	128	16	3
False	3	26	120	5
Unknown	1	2	6	28

Table 8 Numbers of warnings with change in analysis, by type of change

2. In the original analysis, there were 259 associations between pairs of related warnings. During reanalysis, we added 104 associations, removed 7 associations, and changed the degree of association in 6 cases. One of the goals of analysis was to find, for each selected warning, associated warnings from other tools. In the original analysis, there were 151 associations between warnings from different tools, where at least one warning was selected for analysis. After reanalysis, there were 211 such associations, a 40% increase.

Other significant observations include:

1. The quality of our analysis was uneven across different test cases. In particular, for DMDirc, we changed our analysis of correctness for 66 of 150 selected warnings (44%) - much higher than average error rate.
2. We found further evidence that analysis criteria need to be improved and the list of correctness categories needs to be modified. In particular, it is often hard to determine significance of a true warning. We found that the criteria for associating warnings are easier to apply than the criteria for analysis of correctness.

3.5 Tool Warnings Related to Manual Findings

The security experts found 10 weaknesses for Roller and 3 for IRSSI. The description of the manual findings, as well as our listing of the related instances, is available at [17]. The human analysis combined multiple weaknesses with the same root cause. Due to this and to a limited time allotted to it, the number of manual findings was small.

Figure 7 presents the numbers of manual findings for which at least one tool identified the same or other similar instance, at least one tool produced a coincidental warning or no tool produced a related warning. Overall, tools produced related warnings for 5 of 10 manual findings for Roller and for 2 of 3 manual findings for IRSSI.

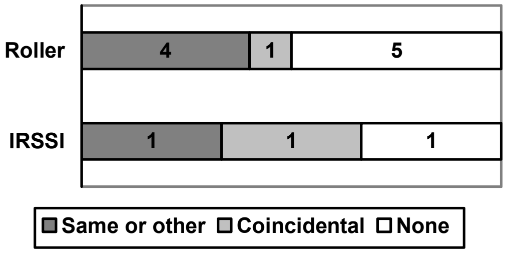

Figure 7 Related warnings from tools

The following briefly describes the manual findings which had coincidental tool warnings. Security experts found the use of weak CAPTCHA in Roller. Coincidentally, 2 tools reported weak random number generation that is used for generating the CAPTCHA challenge. While strength or random number generation is not important here, the tool warnings may point user to the higher level problem. Similarly, security experts reported the use of a small space of random values for an authentication token (an integer modulo 64) for IRSSI. Coincidentally, 2 tools reported poor entropy of the pseudo-random number generator used to generate the authentication token.

2 of the 5 manual findings for Roller that had no related tool warnings were access control weaknesses, where a user is authenticated, but no authorization checks are performed, so a user with no administrator rights to a weblog can perform actions that only an administrator should be allowed to do. Such issues are very hard to identify by automated analysis.

For all but one manual finding that had at least one related tool warning, there were 2 or 3 tools (out of 4 tools that were run for each of the two test cases) that reported the related warnings.

4 Summary and Conclusions

We conducted the Static Analysis Tool Exposition (SATE) 2009 to enable empirical research on large data sets and encourage improvement and adoption of tools. Based on our observations from SATE 2008, we improved the SATE procedure, including selection of warnings for analysis, analysis criteria, and the output format.

Teams ran their tools on 4 test cases - open source programs from 53k to 72k non-blank non-comment lines of code. Eight teams returned 18 tool reports with a total of 19,287 tool warnings. We analyzed approximately 4% of the tool warnings. We selected the warnings for analysis randomly and based on findings by security experts. Several teams improved their tools based on their SATE experience.

Communication with developers of the test cases improved the accuracy of our analysis and resulted in fixes to the software. Reanalysis, completed after the SATE workshop, further improved the analysis quality.

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 weaknesses that exist in the software and that are reported by the tools. The analysis may also be used as a basis for a further study of the weaknesses in the code and of static analysis.

SATE data suggests that 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 well known weakness categories, such as buffer errors.

As part of SATE 2009, we selected tool warnings related to findings by security experts. Tools reported related warnings for 5 of 10 manual findings in Roller and for 2 of 3 manual findings in IRSSI. 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.

Due to complexity of the task and limited resources, our analysis of the tool reports is imperfect. For this and other reasons, our analysis must not be used as a direct source for rating or choosing tools or even in making a decision whether or not to use tools.

5 Future Plans

For the next SATE, analysis of tool reports must be improved. First, while having four correctness categories instead of two helped, analysis of significance was time-consuming and error-prone. Therefore, the set of correctness categories needs to be updated. Second, while analysis criteria were improved since SATE 2008, the criteria need to be clarified further. Third, a large number of tool warnings and lack of the ground truth made analysis very difficult. In SATE 2009, we partially addressed this problem by selecting a random subset of tool warnings for analysis and also selecting tool warnings related to findings by security experts. We are considering other ways of removing uncertainty in analysis, in particular, selecting source code with publicly reported security vulnerabilities.

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/. We thank Romain Gaucher for help with planning SATE 2009. David Lindsay and Romain Gaucher of Cigital are the security experts that quickly and accurately performed human analysis of the test cases. We thank Drew Buttner and Steve Christey of MITRE, Charline Cleraux, Lingda Tang, and Michael Koo of NIST for playing an important role in the analysis of tool reports. We thank other members of the NIST SAMATE team for their help during all phases of the exposition. In particular, Michael Kass helped organize the SATE workshop, and Will

Guthrie helped develop the method for randomly selecting tool warnings for analysis. Chris Johnson reviewed the paper and provided many insightful comments and suggestions for its improvement. We especially thank those from participating teams – Tucker Taft, Maty Siman, Paul Anderson, Fletcher Hirtle, Andy Chou, Peter Henriksen, Benson Wu, Stephen Liu, Michael Su, Alen Zukich, Gwyn Fisher, Todd Landry, John Greenland, Nat Hillary, and Chris Wysopal - for their effort, valuable input, and courage.

7 References

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

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

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

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

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

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

[7] Johns, Martin, Moritz Jodeit, Wolfgang Koeppl and Martin Wimmer, Scanstud - Evaluating Static Analysis Tools, OWASP Europe 2008.

[8] Kratkiewicz, K., and Lippmann, R., 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.

[9] Lindsay, David and Romain Gaucher, Threat Modeling and Manual Assessment, Presentation, Static Analysis Tool Exposition (SATE 2009) Workshop, Arlington, VA, Nov 6, 2009.

[10] Livshits, Benjamin, Stanford SecuriBench, http://suif.stanford.edu/~livshits/securibench/

[11] Michaud, F., and R. Carbone, Practical verification & safeguard tools for C/C++, DRDC Canada – Valcartier, TR 2006-735, 2007.

[12] Rutar, N., C. B. Almazan and J. S. Foster, A Comparison of Bug Finding Tools for Java, 15th IEEE Int. Symp. on Software Reliability Eng. (ISSRE'04), France, Nov 2004.

[13] SAMATE project, https://samate.nist.gov/

[14] SAMATE Reference Dataset (SRD), http://samate.nist.gov/SRD/

[15] SANS/CWE Top 25 Most Dangerous Programming Errors, http://cwe.mitre.org/top25/

[16] Source Code Security Analysis Tool Functional Specification Version 1.0, NIST Special Publication 500-268. May 2007, http://samate.nist.gov/docs/source_code_security_analysis_spec_SP500-268.pdf

[17] Static Analysis Tool Exposition (SATE), http://samate.nist.gov/SATE.html

[18] Static Analysis Tool Exposition (SATE) 2008, NIST Special Publication 500-279, June 2009, Vadim Okun, Romain Gaucher, and Paul E. Black, editors.

[19] Static Analysis Tool Exposition (SATE 2009) Workshop, Co-located with 11th semi-annual Software Assurance Forum, Arlington, VA, Nov 6, 2009, http://samate.nist.gov/SATE2009Workshop.html

[20] Tsipenyuk, K., B. Chess, and G. McGraw, “Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors,” to be published in *Proc. NIST Workshop on Software Security Assurance Tools, Techniques, and Metrics (SSATTM)*, US Nat’l Inst. Standards and Technology, 2005.

[21] Wheeler, David A., SLOCCount, http://www.dwheeler.com/sloccount/

[22] Zheng, J., L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. Vouk, On the Value of Static Analysis for Fault Detection in Software, IEEE Trans. on Software Engineering, v. 32, n. 4, Apr. 2006.

[23] Zitser, M., Lippmann, R., Leek, T., 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).