NIST Special Publication 500-283

Report on the Third Static Analysis Tool Exposition (SATE 2010)

Editors:
Vadim Okun
Aurelien Delaitre
Paul E. Black

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

October 2011

U.S. Department of Commerce
John E. Bryson, Secretary

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

Abstract:

The NIST Software Assurance Metrics And Tool Evaluation (SAMATE) project conducted the third Static Analysis Tool Exposition (SATE) in 2010 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 2010 Workshop in Gaithersburg, MD, in October, 2010. The tool reports and analysis were made publicly available in 2011.

This special publication consists of the following three papers. "The Third Static Analysis Tool Exposition (SATE 2010)," by Vadim Okun, Aurelien Delaitre, and Paul E. Black, describes the SATE procedure and provides observations based on the data collected. The other two papers are written by participating tool makers.

"Goanna Static Analysis at the NIST Static Analysis Tool Exposition," by Mark Bradley, Ansgar Fehnker, Ralf Huuck, and Paul Steckler, introduces Goanna, which uses a combination of static analysis with model checking, and describes its SATE experience, tool results, and some of the lessons learned in the process.

Serguei A. Mokhov introduces a machine learning approach to static analysis and presents MARFCAT's SATE 2010 results in "The use of machine learning with signal- and NLP processing of source code to fingerprint, detect, and classify vulnerabilities and weaknesses with MARFCAT."

Keywords:

Software security; static analysis tools; security weaknesses; vulnerability

Table of Contents

The Third Static Analysis Tool Exposition (SATE 2010)

Vadim Okun Aurelien Delaitre Paul E. Black
{vadim.okun, aurelien.delaitre, paul.black}@nist.gov
National Institute of Standards and Technology
Gaithersburg, MD 20899

Abstract

The NIST Software Assurance Metrics And Tool Evaluation (SAMATE) project conducted the third Static Analysis Tool Exposition (SATE) in 2010 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 2010 Workshop in Gaithersburg, MD, in October, 2010. The tool reports and analysis were made publicly available in 2011.

This paper describes the SATE procedure and provides our observations based on the data collected. We improved the procedure based on lessons learned from our experience with previous SATEs. One improvement was selecting programs based on entries in the Common Vulnerabilities and Exposures (CVE) dataset. Other improvements were selection of tool warnings that identify the CVE entries, expanding the C track to a C/C++ track, having larger — up to 4 million lines of code — test cases, clarifying further the analysis categories, and having much more detailed analysis criteria.

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.

Third, the CVE-selected test cases contain real-life exploitable vulnerabilities, with additional information about the CVE entries, including their locations in the code. These test cases can serve as a challenge to practitioners and researchers to improve existing tools and devise new techniques. Finally, the analysis may be used as a basis for a further study of the weaknesses in the code and of static analysis.

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 2010, 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 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 much improved since the previous SATEs, 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.

In SATE 2010, we added CVE-selected programs to the test sets for the first time. The procedure that was used for finding CVE locations in code and selecting tool warnings related to the CVEs has limitations, so the results may not indicate tools' ability to find important security weaknesses.

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 July 2010. 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 Introduction

SATE 2010 was the third 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 2010 Workshop [26]. The tool reports and analysis were made publicly available in 2011. SATE had these goals:

- To enable empirical research based on large test sets
- To encourage improvement of tools
- To 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 their user interfaces — 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 [21]. The tool reports were analyzed by a small group of analysts, consisting of the NIST researchers and a volunteer. 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 test cases.

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 [24]. 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 2010 based on our experience from SATE 2008 [27] and SATE 2009 [30]. 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 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) [5] 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.

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 [4]).
- Attributed to many different statements.
- Has intermingled flows.

In [27], 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 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 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 [37] 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 [23] weakness categories [27].

Several collections of test cases with known security flaws are available [12] [38] [15] [22]. Several assessments of open-source projects by static analysis tools have been reported recently [1] [9] [10]. 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 the vulnerable and the fixed version 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 [13] 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.

National Security Agency's Center for Assured Software [36] ran 9 tools on over 59000 synthetic test cases covering 177 CWEs and found that static analysis tools differed significantly in precision and recall, and their precision and recall ordering varied for different weaknesses. They concluded that the sophisticated use of multiple tools would increase the rate of finding weaknesses and decrease the false positive rate.

A number of studies have compared different static analysis tools for finding security defects, e.g., [8] [12] [16] [20] [38] [11]. 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 2010 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. 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 25 June 2010)
- Step 2 Organizers provide test sets via SATE web site (28 June 2010)
- Step 3 Teams run their tool on the test set(s) and return their report(s) (by 30 July 2010)
- Step 4 Organizers analyze the reports, provide the analysis to the teams (preliminary analysis by 14 Sep 2010, updated analysis by 22 Sep 2010)
  - Organizers select a subset of tool warnings for analysis and share with the teams (by 24 Aug 2010)
  - (Optional) Teams check their tool reports for matches to the CVE-selected test cases and return their review (by 1 Sep 2010)
  - (Optional) Teams return their review of the selected warnings from their tool's reports (by 8 Sep 2010)
- Step 5 Report comparisons at SATE 2010 workshop [26] (1 Oct 2010)

- Step 6 Publish results[1] (Originally planned for Feb - May, but delayed until Oct 2011)

2.2 Test Sets

This Section lists 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 lines of code and files were counted before building the programs. For several test cases, counting after the build process would have produced higher numbers. For each CVE-selected test case, the table has separate rows for the vulnerable and fixed versions.

The counts for C and C++ test cases include C/C++ source (e.g., .c, .cpp, .objc) and header (.h) files. Dovecot and Wireshark are C programs, whereas Chrome is a C++ program. The counts for Dovecot include 2 C++ files. The counts for the Java test cases include Java (.java) and JSP (.jsp) files. Pebble's test code (in src/test subdirectory) is not included in its counts. Tomcat ver. 5.5.13 includes 192 C files. Tomcat ver. 5.5.29 does not include any C files. The C files were not included in the counts for Tomcat. The counts do not include source files of other types: make files, shell scripts, Assembler, Perl, PHP, and SQL.

Mitigation for Cross-Site Request Forgery (CSRF) CWE-352 was introduced in the version of Pebble that was analyzed by tools. However, an implementation bug in CSRF mitigation code prevented many features of Pebble from working properly. Security experts analyzed, as part of SATE 2010, a newer build of Pebble, where the only change was removal of CSRF mitigation code. The expert analysis is described in Section 2.6.1, Method 2.

The lines of code and files were counted using SLOCCount by David A. Wheeler [35].

Test case	Track	Description	Version	# Files	# LOC
Dovecot	C/C++	Secure IMAP and POP3 server	2.0 Beta 6	1111	193501
Wireshark		Network protocol analyzer	1.2.0	2281	1625396
			1.2.9	2278	1630282
Google Chrome		Web browser	5.0.375.54	19070	3958861
			5.0.375.70	19070	3958998
Pebble	Java	Weblog software	2.5 M2	603	29422
Apache Tomcat		Servlet container	5.5.13	1494	180966
			5.5.29	1603	197060

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 [28].

We spent several weeks selecting the test cases and considered dozens of candidates. In particular, we looked for test cases with various security defects, over 10000 lines of code, and compilable using a commonly available compiler. We used different selection criteria and selection process for the CVE-based test cases and the general test cases. The following section describes how we selected the CVE-based test cases.

[1] Per requests by Coverity and Grammatech, their tool output is not released as part of SATE data. Consequently, our detailed analysis of their tool warnings is not released either. However, the observations and summary analysis in this paper are based on the complete data set.

2.3 CVE-Selected Test Cases

In addition to the criteria listed above, we used the following selection criteria for the CVE-based test cases and also for selecting the specific versions of the test cases.

- Program had several, preferably dozens, of vulnerabilities reported in the CVE database.
- 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).
- Reliable resources, such as bug databases and source code repositories, were available for locating the CVEs.
- 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 the following 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 [25] [18] [1] [10]. Third, we used several public vulnerability databases [5] [7] [17] [19] 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, 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 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 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 can often be located on a path, so it cannot be attributed to a single line of code. Also, sometimes a weakness can be fixed 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, and
- 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 the 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 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 the file comparison utility "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. Finally, we manually reviewed the source code.

Some of the information links, such as which bug ID in the bug database corresponds to the CVE, were missing. We reviewed change logs and used other hints to find the CVE locations.

The following is one scenario for pinpointing the CVE in code. First, view a CVE entry in the NVD. Second, follow a link to the corresponding bug description in the program's bug database. Third, from the bug description, find the program's revision where the bug was fixed. Fourth, for each source file that was changed in the revision, examine the patch, that is, the difference with the previous version of the file. The patch often contains the fix location. Finally, examine the source code to find the corresponding sink and/or path locations.

2.4 Tools

Table 2 lists, alphabetically, the tools and the tracks in which the tools were applied.

One of the teams, Veracode, is a service. Its tool is not available publicly. Veracode performed a human quality review of its reports to remove spurious warnings such as high false positives in a particular weakness category. This quality review is part of its usual analysis procedure.

Tool	Version	Tracks
Armorize CodeSecure	4.0.0.2	Java
Concordia University MARFCAT[2]	SATE2010.6	C/C++, Java
Coverity Static Analysis for C/C++[3]	5.2.1	C/C++
Cppcheck	1.43	C/C++
Grammatech CodeSonar[4][5]	3.6 (build 59634)	C/C++
LDRA Testbed[6]	8.3.0	C/C++
Red Lizard Software Goanna	2.0	C/C++
Seoul National University Sparrow[7]	1.0	C/C++
SofCheck Inspector for Java[8]	2.22510	Java
Veracode SecurityReview[9]	As of 07/12/2010	C/C++, Java

Table 2 Tools

2.5 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.
- Except for Veracode, the teams did not do any hand editing of tool reports. Veracode, a service, performed a human quality review of its reports to remove spurious warnings 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 2.8.1 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 [28].

Most teams submitted one tool report per test case for the track(s) that they participated in. LDRA and Grammatech analyzed Dovecot only. Sparrow works on C only, so it was run on Dovecot and Wireshark, but not on Chrome. Sparrow was run with an option which lets it distinguish up to two array elements, whereas in default configuration, array elements are combined into a single abstract location.

SofCheck analyzed Pebble only. SofCheck Inspector was run in default mode, which did not include its checks for XSS and SQL injection weaknesses.

Coverity and Grammatech tools were configured to improve analysis of Dovecot's custom memory functions. See "Special case of Dovecot memory management" in Section 2.7.4.

[2] Marfcat was in early stages of development; reports were submitted late; we did not analyze the reports
[3] Per Coverity's request, Coverity tool output is not released as part of SATE data
[4] Analyzed Dovecot only
[5] Per Grammatech's request, Grammatech tool output is not released as part of SATE data
[6] Analyzed Dovecot only
[7] Sparrow works on C only, so analyzed Dovecot and Wireshark
[8] Analyzed Pebble only
[9] A service

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

MARFCAT was in the early stages of development during SATE 2010, so the reports were submitted late. As a result, we did not analyze the output from any of MARFCAT reports. Serguei Mokhov described MARFCAT methodology and results in "The use of machine learning with signal- and NLP processing of source code to fingerprint, detect, and classify vulnerabilities and weaknesses with MARFCAT," included in this publication.

In all, we analyzed the output from 32 tool runs. This counts tool outputs for vulnerable and fixed versions of the same CVE-based program separately.

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 Static Analysis, 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 Grammatech CodeSonar, Coverity Static Analysis, and LDRA Testbed reports based on the weakness names and some additional information from the tools.

After the analysis step of the SATE protocol (Section 2.1) was completed, Armorize discovered a flaw in the engine of the CodeSecure version used for SATE. This flaw caused many false positives and false negatives for the JSP-related weaknesses. JSP accounts for 19% of lines of code in Pebble (not including Pebble's test code), 3.7% of lines of code in the vulnerable version of Tomcat, and 3.4% of lines of code in the fixed version of Tomcat.

2.6 Analysis of Tool Reports

For SATE, 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 1 describes the high-level view of our analysis procedure. We used three 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 warnings related to manually identified weaknesses. In the third method, 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 the following characteristics. First, we associated (grouped together) warnings that refer to the same (or related) weakness. (See Section 3.4 of [27] for a discussion of what constitutes a weakness.) Second, we analyzed correctness of the warnings. Also, we included our comments about warnings.

2.6.1 Three Methods for Tool Warning Selection

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

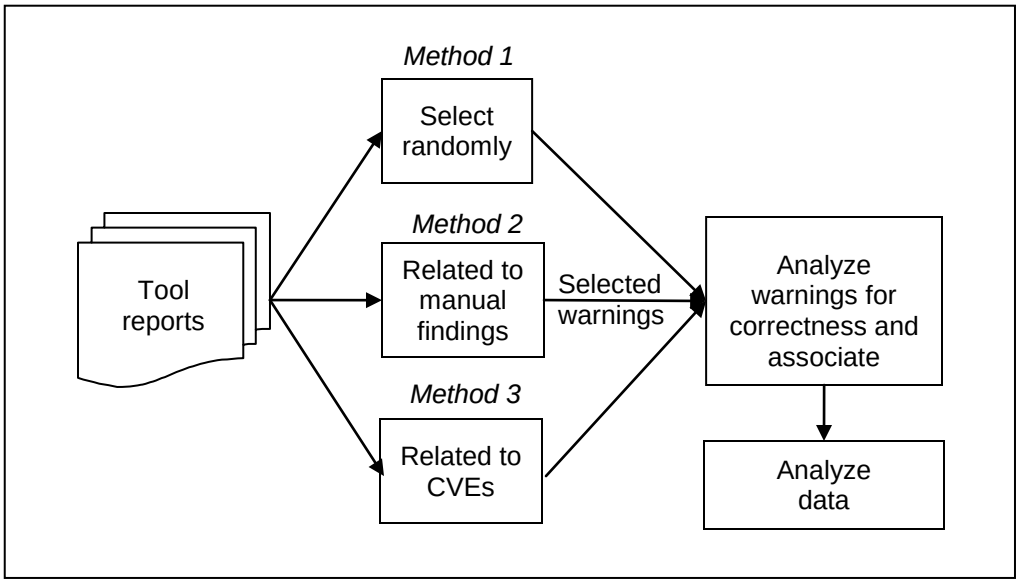

Figure 1 Analysis procedure overview

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 identified by a (weakness name, severity) or (CWE ID, 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, 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, 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.

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

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 manually analyzed Dovecot and Pebble and identified the most important weaknesses in these test cases. In this paper, we call these weaknesses *manual findings*. The 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. That is, the security experts did not look for every weakness instance, but instead gave a few (or just one) instances per root cause. Rapid threat modeling was used to guide specific testing activities, including code review, automated analysis, 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 [14]. Due to the limited resources (about 3.5 person-weeks), security experts analyzed two of the five test cases only.

Specifically for Dovecot, the security experts considered only the external facing mechanisms as targets of interest. First, fuzzing was performed on the externally exposed mail protocols (LMTP, POP3, and IMAP). This involved sending data to be interpreted by the protocol implementation in an attempt to cause unexpected application behavior. Dovecot did not exhibit any insecure or unwanted behavior when under stress from the fuzzing tools.

Second, a code review was conducted to determine if the Dovecot source code contained weaknesses that can lead to exploitable vulnerabilities. The review focused on manual analysis of the memory management routines in the code, as well as problems reported by tools which they ran, including the Clang static analysis tool (part of the LLVM project)[10]. No weakness was determined to be exploitable by an external attacker on a Dovecot system configured using default security constraints. To summarize, the security experts reported no findings for Dovecot based on the limited analysis they had time to perform.

For Pebble, penetration testing and code review were performed. Security experts analyzed a newer build of Pebble, where the only change was removal of the incorrect CSRF mitigation code. See Section 2.2 for details. Security experts reported 10 findings for Pebble: 8 stored Cross-Site Scripting (XSS) CWE-79 and 2 URL Redirection CWE-601 weaknesses.

Security experts, with our assistance, checked the tool reports to find warnings related to the manual findings. For each manual finding, for each tool, they found at least one related warning, or concluded that there were no related warnings.

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 2.3. For each test case, we produced a list of CVEs in SATE output format with additional location information, see Section 2.8.2 for details. We then searched, mechanically and manually, the tool reports to find warnings related to the CVEs.

2.6.2 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.

After the SATE workshop, we wrote a utility to find all warnings that were present in one report but not in another report. We used this utility to produce a subset of warnings reported by a tool

[10] Available at http://clang.llvm.org/

for the vulnerable version but not for the fixed version of a test case. We then checked whether any of these warnings are related to the CVEs.

2.6.3 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. Usually the analyst 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 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.

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.

Two teams, Red Lizard Software and Veracode, submitted reviews of their tool's results for the CVE selected test cases. The reviews helped identify mistakes in our analysis of the CVEs.

Additionally, several teams presented a review of our analysis at the SATE 2010 workshop.

2.7 Analysis Criteria

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

2.7.1 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 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 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 – we discuss these factors below.

2.7.2 Decision Process

This Section omits details about several factors (context, code quality, path feasibility) used in the decision process – see later sections for definitions and examples.

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 can 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 the 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, we list the decision process details specific to one particular weakness type: information leaks.

2.7.3 Context

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 determine the following:

- 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

2.7.4 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 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 [31]. For example, all memory allocations (with some exceptions in 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.

2.7.5 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 path is infeasible due to a flag that is set elsewhere (e.g., in a different module). In the following example, tool may mark NULL pointer dereference of arg, which is infeasible because 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 which shows a wrong case taken in a switch statement.

In previous SATEs, 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 a warning that points to such a weakness on an infeasible path should be brought to the attention of a programmer. Additionally, analysis of feasibility for some warnings took too much time. Therefore, we marked some warnings on an infeasible path as quality weakness or insignificant.

2.7.6 Criteria for Warning Association

Warnings from multiple tools may refer to the same (or a related) weakness. (The notion of distinct weaknesses may be unrealistic. See Section 3.4 of [27] 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 2010, as well as 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 one existed) 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. 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, 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.

2.7.7 Criteria for Matching Warnings to Manual Findings and CVEs

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

- Same weakness instance
- Same weakness instance, different perspective
- Same weakness instance, different paths
    - Example: different paths, e.g., different sources, but the same sink

Note that in SATE 2009, we considered two additional cases: "Coincidental" – tool reports a lower level weakness that may lead the user to the high level weakness, and "Other weakness instance" – tool reports a similar weakness (the same weakness type) elsewhere in the code.

In contrast, in SATE 2010, we did not find any cases of a coincidental warning. We also did not look for "other weakness instance," since for CVEs, we are interested in tools reporting the particular weakness instance, and the manual findings belonged to just two weakness types.

2.8 SATE Data Formats

Teams converted their tool output to the SATE XML format. Section 2.8.1 describes this tool output format. Section 2.8.2 describes the extension of the SATE format for storing our analysis of the warnings. Section 2.8.4 describes the extension of the SATE format for our analysis of which tool warnings are related to the manual findings. Section 2.8.5 describes the format for storing the lists of associations of warnings.

2.8.1 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 2.7.
    - Comments.

The XML schema file for the tool output format is available at the SATE web page [28].

2.8.2 CVE 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 SATE 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 2.3.

2.8.3 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 reports.
- Selected – “yes” means that we selected the warning for analysis using Method 1.

2.8.4 Manual Findings and CVEs Analysis Format

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

2.8.5 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.

2.9 Summary of changes since previous SATEs

Based on our experience conducting SATE 2008 and 2009, we made the following changes to the SATE procedure. First, we added CVE-selected programs to the test sets and selected tool warnings related to the CVEs. This selection method complements the two methods used in SATE 2009 by focusing our analysis on real-life exploitable vulnerabilities.

Second, we expanded the C track to C/C++ track by including Chrome and had larger test cases than in previous SATEs. Having vulnerable and fixed version pairs for CVE-selected test cases also meant that the total number of programs to scan, 8, was higher than in previous SATEs, which, together with the larger sizes, put extra burdens on participating teams.

Third, based on analysis of SATE 2009 tool warnings, we added a fifth correctness category, true quality weakness. We updated the SATE output format accordingly.

Fourth, we wrote much more detailed guidelines, including criteria and decision process, for analysis of correctness. Previously, we assumed perfect understanding of data and control flow by tools. Consider so called "locally true but globally false" cases. 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, the warning would be marked as false under the SATE 2009 criteria. However, determining whether a function is always called with safe values can be very resource consuming.

Therefore, in SATE 2010, we relaxed this assumption of perfect code understanding. For example, tool warnings involving infeasible paths can now be marked as quality or insignificant, depending on other considerations. The guidelines are in Section 2.7.

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 58 different CWE ids. In addition, there are 145 weakness names for warnings that do not have a CWE ID. In all, there are 224 different weakness names. This exceeds 58+145 since tools sometimes use different weakness names for the same CWE ID. In order to simplify the presentation of data in this report, we defined categories of similar

weaknesses and placed tool warnings into the categories based on CWE ID and 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 [28]. Some categories are individual weakness classes; others are broad groups of weaknesses. In particular, cross-site scripting (XSS) is a kind of improper input validation. Also, race condition is a kind of time and state weakness category. Due to their prevalence and severity, we decided to use separate categories for these weaknesses.

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
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
Broad categories			
Improper input validation	input-val	Absent or incorrect protection mechanism that fails to properly validate input	Log forging, HTTP response splitting, command injection, SQL injection, re-source injection, file 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 occur during processing	Incomplete, missing error handling, missing check against null
Insufficient encapsula-tion	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 inconsistent with its intended use	Heap inspection, use of inherently dangerous function

Name	Abbre-viation	Description	Example types of weaknesses
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 weak-nesses, session management problems
Code quality problems	code-qual	Features that indicate that the software has not been carefully developed or maintained	See below
Null de-reference	null-deref	A program dereferences a pointer or reference that is NULL	Null pointer dereference
Improper initializa-tion	uninit	A resource is not initialized or incorrectly initialized	Uninitialized variable
Resource manage-ment problems	res-mgmt	Improper management of resources	Denial of service due to unreleased re-sources, use after free, double unlock, memory leak
Other quality	qual-other	Other code quality problems	Dead code, potential violation of coding standards
Uncatego-rized	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 and 2009. 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 cross-site request forgery (CSRF) in SATE 2010, since there were no CSRF warnings reported. The version of Pebble that was analyzed by tools contained the CSRF mitigation code that broke other functionality. See Section 2.2 for details.

Also, there is no separate category for command injection, since there were only 2 warning instances reported, both for Chrome. Command injection, as well as SQL injection, is under improper input validation category.

Since the majority of warnings in SATE 2010 are under the code quality category, we included the data for its subcategories. Several weakness types concerned with potential violation of coding standards are included in the code quality category, not in a more specific category such as numeric errors. Also, to avoid confusion with the "quality" correctness category introduced in SATE 2010, we changed the abbreviation for the code quality category to code-qual.

The categories are derived from [6], [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 abbreviations of weakness category names (the second column of Table 3) 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 the division of tool warnings by various dimensions. Figure 2 presents the numbers of tool warnings by test case.

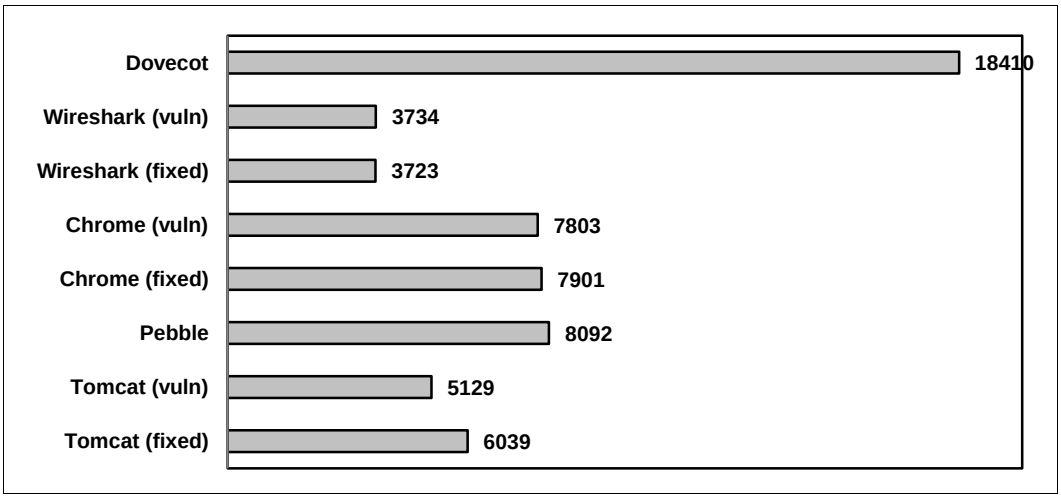

Figure 2 Warnings by test case (total 60 831)

Figure 3 presents the numbers of tool warnings by severity as determined by the tool, with some changes noted below. 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.

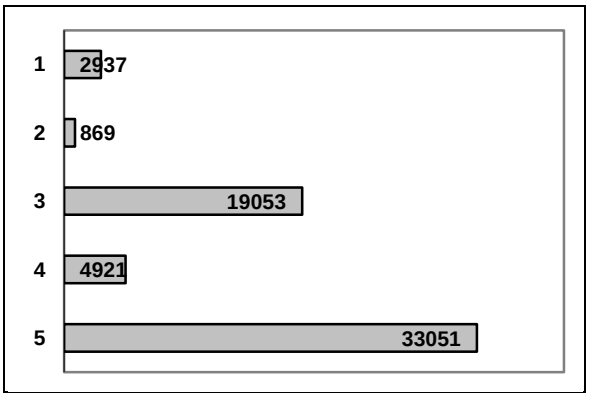

Figure 3 Warnings by severity (total 60 831)

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. For CVE-selected test cases, the table presents the numbers for vulnerable versions only. For consistency, we only included the reports from tools that were run on every test case in a track. In other words, we included reports from 4 tools for the C/C++ track and from 2 tools for the Java track. Accordingly, the numbers in the "median" row for Pebble 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. Second, as noted in Section 1.2, the notion that weaknesses occur as distinct, separate instances is not reasonable in most cases. Third, there were inconsistencies in the way

tool output was mapped to the SATE output format. For example, in the Armorize CodeSecure reports, 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	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 4 Low, high, and median number of tool warnings per kLOC

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

Table 5 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 6 Low, high, and median number of tool warnings per kLOC for reports in SATE 2008

For comparison, Table 5 and Table 6 present the same numbers as Table 4 for the reports in 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 6, 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	Chrome	All Java	Pebble	Tomcat
xss	0	0	0	0	2176	91	2085
buf	508	81	95	332	0	0	0
num-err	454	116	260	78	10	10	0
race	141	133	6	2	1	0	1
info-leak	0	0	0	0	68	5	63
input-val	153	104	9	40	10726	7819	2907
sec-feat	99	32	23	44	12	7	5
code-qual	28150	17829	3262	7059	87	84	3
null-deref	11965	11190	87	688	45	45	0
uninit	4637	3405	62	1170	0	0	0
res-mgmt	1731	1307	30	394	0	0	0
qual-other	9817	1927	3083	4807	42	39	3
err-handl	401	107	78	216	0	0	0
encaps	6	0	0	6	6	5	1
api-abuse	20	6	0	14	5	1	4
time-state	9	2	0	7	66	6	60
uncateg	6	0	1	5	64	64	0
Total	29947	18410	3734	7803	13221	8092	5129

Table 7 Reported warnings by weakness category

Table 7 presents the numbers of reported tool warnings by weakness category for the C/C++ and Java tracks, as well as for individual test cases. For CVE-selected test cases, the table presents the numbers for vulnerable versions only. The weakness categories are described in Table 3.

For the C/C++ track, there were no xss and info-leak warnings, mostly because these test cases are not web applications. Also, since it is uncommon to write web applications in C or C++, the tools tend not to look for web application vulnerabilities in the C or C++ code. However, as noted in Section 3.4, Chrome had a CVE entry in the xss category. For the Java track, there were no buf warnings - most buffer errors are not possible in Java. Most warnings for Java track were input validation errors, including xss.

The great majority of warnings reported for Dovecot were from the code-qual category. This is due to a combination of two factors. First, Dovecot was written with security in mind, so it was not likely to have many security problems. Second, most of the tools that were run on Dovecot were quality oriented.

In contrast, only 3 warnings reported for Tomcat were from the code-qual category. The main reason is that the two tools that were run on Tomcat, Armorize CodeSecure and Veracode, were security oriented.

Using Method 1 introduced in Section 2.6.1, we randomly selected a subset of tool warnings for Dovecot and Pebble for analysis. The analysis confirmed that tools are capable of finding weaknesses in a variety of categories.

For Dovecot, we judged weaknesses from the following weakness categories to be security or quality: buf, num_err, race, input_val, sec_feat, code-qual (including null pointer dereference and various resource management problems), err_handl, and api_abuse.

For Pebble, we judged weaknesses from the following weakness categories to be security or quality: xss, input_val (including HTTP response splitting, log forging, and external control of file name or path), sec_feat, code-qual, and encaps.

Since we selected a small subset of warnings for analysis, we do not report the numbers of security and quality weaknesses. See Table 7 in [30] for the numbers of true significant weaknesses, as determined by analysts, in SATE 2009.

Figure 4 presents, for Dovecot and Pebble, for true security correctness category and for all correctness categories, the percentage of weaknesses that were reported by 1 tool or 2 or more tools. It also gives, on the bars, the numbers of weaknesses reported by different numbers of tools. For example, of 6 true security weaknesses for Dovecot, 3 were reported by 2 tools and 3 were reported by 1 tool.

Of the 23 weaknesses reported by multiple tools for Dovecot, 21 weaknesses were reported by 2 tools and 2 weaknesses were reported by 3 tools. No other weaknesses were reported by 3 or more tools. For reference, we analyzed the results from 7 tools for Dovecot and 3 tools for Pebble. Although MARFCAT was run on both Dovecot and Pebble, we did not analyze its results.

For Pebble, SofCheck Inspector warnings did not overlap with the warnings from other tools at all. We see two reasons. First, it is a quality-oriented tool, in contrast with Armorize CodeSecure and Veracode. Second, SofCheck Inspector was run in default mode, which in particular did not include its checks for XSS weaknesses.

As Figure 4 shows, tools mostly find different weaknesses. This is partly due to the fact that tools often look for different weakness types. However, as shown in Section 3.2 of [30], there is more overlap for some well known and well studied categories, such as buf.

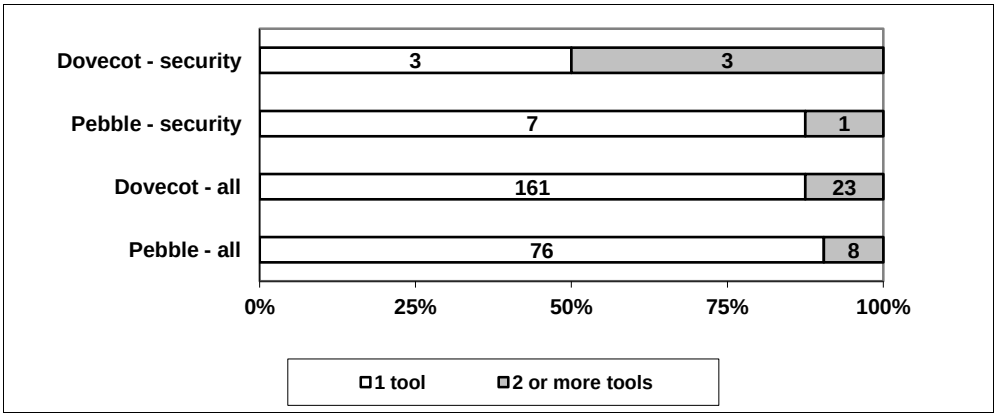

Figure 4 Weaknesses, by number of tools that reported them

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

Using Method 1 introduced in Section 2.6.1, we randomly selected 276 Dovecot and Pebble warnings for analysis. It is about 1 % of the total number of warnings for these 2 test cases (26 502). We also analyzed some other warnings. In all, we analyzed (associated or marked correctness of) 405 warnings, about 1.5 % 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.

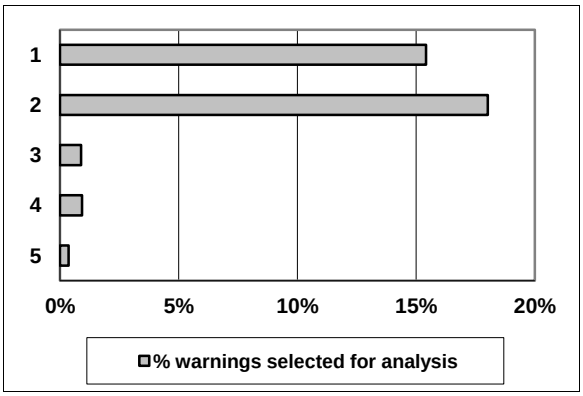

Figure 5 Warnings for Dovecot and Pebble selected for analysis, by severity

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 of that severity class for Dovecot and Pebble selected for analysis. Due to a very large number of severity 3 warnings for Dovecot and Pebble – about 55 % of the total – a small portion of these warnings were selected.

Five researchers analyzed the tool warnings, spending anywhere from a few days to a few weeks. 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 2010 was similar to previous SATEs, see [27] [30] for details.

3.4 Manual Findings and CVEs by Weakness Category

The security experts found 10 weaknesses for Pebble and no weaknesses for Dovecot. The weaknesses in Pebble included 8 stored XSS weaknesses (xss category) with several sources each and 2 URL redirection weaknesses (a subcategory of the input-val category).

Table 8 presents the numbers of CVEs in the CVE-selected test cases by weakness category. The table omits rows for the weakness categories that had no CVEs. The table also lists the CWE ids of the CVEs in each weakness category. See Section 2.3 for a description of how we identified the CWE ids.

The weakness categories are described in Table 3. When a CWE had properties of more than one weakness category, we tried to assign it to the most closely related category. For example, CVE-2009-2562 in Wireshark represents a chain of integer overflow leading to buffer overflow. We classified it as CWE-190 and placed under num-err category.

3.4.1 CVE Analysis Details and Changes

Three CVEs applied exclusively to the C code portion of Tomcat. Since Tomcat was in the Java track, the numbers in this report do not include these CVEs.

Weakness category	CWE ids	All	Wireshark	Chrome	Tomcat
xss	79	8	0	1	7
buf	119, 822	10	8	2	0
num-err	189, 190	5	4	1	0
info-leak	200	6	0	0	6
input-val	20, 22	11	2	1	8
sec-feat	255, 264, 327, 614	5	0	0	5
code-qual	399, 401, 457, 474, 476	13	10	3	0
uncateg		1	0	1	0
Total		59	24	9	26

Table 8 CVEs by weakness category

Our original analysis of CVEs had several errors. First, we included several CVEs in Chrome that were still present in the fixed version of the test case. We were alerted about this error by the tool makers and excluded these CVEs from consideration.

Second, during reanalysis of CVEs after the workshop, we found several mistakes in our original analysis and corrected those. One of these CVEs, CVE-2007-6286, was described incorrectly in our original analysis and was matched by a warning (UID 13216). UID is a warning identifier we assigned, unique across all tool reports. We later determined that the CVE was in a different part of the code and was unrelated to the warning.

Third, for Chrome, CVE-2010-2304 was replaced by CVE-2010-1773 and CVE-2010-2303 was replaced by CVE-2010-1772 in the CVE and NVD databases. We made the corresponding change in our data files.

Finally, we originally marked a warning (UID 13217) as related to CVE-2008-0128. During reanalysis, we determined that the warning was not related to the CVE.

3.5 Tool Warnings Related to Manual Findings and CVEs

The description of the manual findings and CVEs, as well as our listing of the related tool warnings, is available at [28].

Figure 6 presents the numbers of manual findings and CVEs for which at least one tool identified the same weakness instance or no tool produced a related warning. Since security experts did not find any weaknesses for Dovecot, and we did not find any matching warnings for Wireshark and Chrome[11], the figure presents the data for the two Java test cases only. Overall, tools produced related warnings for 7 of 10 manual findings for Pebble and for 4 of 26 CVEs for Tomcat.

Figure 6 Related warnings from tools

Of the three tools that were run on Pebble, only two found warnings related to manual findings. Of the 8 XSS weaknesses, 4 were found by one tool only and 1 was found by two tools. The two URL redirection weaknesses were found by one tool only.

The number of CVEs by weakness category for each CVE-selected test case is shown in Table 8. All 4 CVEs with related tool warnings were XSS in Tomcat. 3 of the 4 CVEs were reported for the vulnerable version only, 1 was reported in both the vulnerable and fixed versions.

3.6 On Detecting CVEs by Tools

The data collected about the CVEs during SATE 2010 can provide some insights into what properties of the vulnerabilities make them easier or harder to find by static analysis tools. In this Section, we describe sample CVEs, pointing out their properties relevant to their detectability by

[11] This Section does not include the results from one of the tools, MARFCAT, which was trained on the SATE 2010 CVE-based test cases and achieved good precision when used on these test cases.

tools. We then summarize the possible reasons for the low number of matching tool warnings for CVEs.

First, consider CVEs that were detected by a tool. For example, CVE-2007-2449 is in JSP code which is part of a sample web application. In the vulnerable version, request.getQueryString() is printed to the web page without filtering, whereas in the fixed version, it is filtered using util.HTMLFilter.filter() function. The tool recognized the filter function and reported the weakness in the vulnerable version only.

CVE-2006-7195 is in JSP code that uses JSP Standard Tag Library (JSTL) functions. In the vulnerable version, the Host request-header field is printed to the web page without filtering, whereas in the fixed version, it is filtered using a JSTL tag fn:escapeXml(). The tool did not recognize the filter tag and reported the weakness in both vulnerable and fixed versions. Tools can be made better by improving their understanding of popular libraries and frameworks.

The rest of the examples described in this Section were not detected by tools[12]. The following example, CVE-2009-3551 in Wireshark, is an off-by-one error in the SMB dissector. The incorrect condition on line 2195 is in bold font.

```
#define MAX_DIALECTS 20
struct negprot_dialects {
      int num;
2097  char *name[MAX_DIALECTS+1];
};
...
dissect_negprot_request(...) {
      ...
      struct negprot_dialects *dialects = NULL;
      ...
      if (... && dialects && dialects->num<MAX_DIALECTS) {
            dialects->name[dialects->num++] = se_strdup(str);
      }

dissect_negprot_response(...) {
      ...
      struct negprot_dialects *dialects = NULL;
      ...
      /* Dialect Index */
      dialect = tvb_get_letohs(tvb, offset);

      if (si->sip && si->sip->extra_info_type==SMB_EI_DIALECTS) {
```

[12] This Section does not include the results from one of the tools, MARFCAT, which was trained on the SATE 2010 CVE-based test cases and achieved good precision when used on these test cases.

```
            dialects = si->sip->extra_info;
2195        if (dialect <= dialects->num) {
                  dialect_name = dialects->name[dialect];
            }
      }


static smb_function smb_dissector[256] = {
      ...
      /* 0x72 Negotiate Protocol*/  {dissect_negprot_request,
dissect_negprot_response},
```

Here, dialects is populated in function dissect_negprot_request(). It is used in function dissect_negprot_response(), where the off-by-one access occurs. The fix is to replace <= with < on line 2195. The functions dissect_negprot_request() and dissect_negprot_response() are called indirectly via a table of function pointers.

Flawfinder [34], an early static analysis tool that does not do control flow or data flow analysis, produces a general warning for line 2097: “Statically-sized arrays can be overflowed.” This warning is too general and is also generated for a fixed version of the test case. Tools such as those participating in SATE 2010, that do deeper analysis, including inter-procedural and indirect function call handling, should be able to report this kind of weakness for the vulnerable version and not report it for the fixed version of the test case. However, this weakness was not detected.

Some CVEs represent the types of weaknesses that are hard to discover by tools. Some examples are as follows.

- CVE-2007-5342 in Tomcat is an access control problem, where the default policy for a logging component does not restrict certain permissions for web applications.
- CVE-2007-1858 in Tomcat is a design error where the default SSL cipher configuration uses certain insecure ciphers.
- As described in Section 7 of “Goanna Static Analysis at the NIST Static Analysis Tool Exposition” by Bradley, et al. included in this publication, CVE-2010-2286 in Wireshark is an infinite loop weakness for a loop implemented by goto statements that span approximately 2000 lines of code.

Table 9 provides a short description of CVEs in Chrome with our comments about how easily the tools could detect them. In most cases, some understanding of program design is required.

The following are possible reasons for the lack of matching tool warnings for Wireshark and Chrome and a low number of matching tool warnings for Tomcat.

- Some CVEs could have been identified with tool tuning, but were not identified by tools in default configuration. The accompanying paper “Goanna Static Analysis at the NIST Static Analysis Tool Exposition” by Bradley, et al. provides an example where CVE-2010-2283 was not detected in the SATE tool run which used a low timeout value, but was detected in another run with a higher timeout value.

- Not all tools were run on the CVE-selected test cases. In particular, Grammatech CodeSonar's results were not submitted to SATE; these results are described in [2].
- Due to the limitations in our procedure for finding CVE locations in code and selecting tool warnings related to the CVEs, we may have missed some matches.
- Some CVEs, such as design level flaws, are very hard to detect by automated analysis.
- There are other, unknown to us, important vulnerabilities in the test cases, which may have been found by tools. For example, since we selected SATE 2010 test cases, 5 new CVEs were reported for Tomcat 5.5.13. For one of these, CVE-2010-3718, Tomcat does not make the ServletContext attribute read-only, which allows local web applications to read or write files outside of the intended working directory. In fact, a SATE 2010 tool report contained related lower-level resource injection warnings about creation of the working directory.
- The CVE-selected test cases are large, from about 200k to about 4M non-blank, non-comment lines of code. The software has 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. A significant effect of code complexity and code size on quality of static analysis results was found in [33].

CVE ID	Brief description	Detectability by tools
CVE-2010-1773	Off-by-one leading to out of bounds array access in the rendering code. The fix was to decrement the dividend instead of subtracting one from the modulo operation's result.	Not difficult. Note that a tool reported an unrelated warning in the same function.
CVE-2010-1772	Use after free - geolocation timers were not stopped upon deletion of a document.	Requires understanding the geolocation data structures and its semantics.
CVE-2010-2302	Use-after-free in WebKit. Loading a remote font forces a style recalculation. However, the recalculation is not propagated into shadow DOM trees.	Requires understanding of complex classes.
CVE-2010-2301	HTML text area tag was not escaped.	XSS is uncommon in C++, so tools usually do not look for it.
CVE-2010-2300	Use after free in WebKit. The normalization process can result in removing attributes from the DOM element being normalized. This can lead to accessing attributes past the end of the vector.	Requires understanding of the normalization algorithm.
CVE-2010-2299	The untrusted pointer to shared memory based bitmaps is replaced when processing synchronous messages, but not when processing asynchronous messages. The fix is to remove the pointer when processing an asynchronous message.	Requires knowledge of application design
CVE-2010-2298	In OS_POSIX based systems, the browser sends a directory file descriptor to the renderer process. This enables escaping of the chroot()-based sandbox.	Not all platforms are affected.
CVE-2010-2297	Vector size is not checked in table layout calculation resulting in out of bounds read.	Not difficult, but presence of an assertion that aborts execution in debug mode only may have confused tools.
CVE-2010-2295	EventHandler can operate on a wrong frame if focus changes during keyboard event dispatch. This allows keystroke redirection via a crafted HTML document.	Requires knowledge of application design

Table 9 Summary of Chrome CVEs

4 Summary and Conclusions

We conducted the Static Analysis Tool Exposition (SATE) 2010 to enable empirical research on large data sets and encourage improvement and adoption of tools. Based on our observations from the previous SATEs, we improved the SATE procedure, including analysis categories and analysis criteria.

Teams ran their tools on eight code bases - open source programs from 29k to 4M non-blank, non-comment lines of code. Ten teams returned 40 tool reports with about 61 000 tool warnings. We analyzed approximately 1.5 % of the tool warnings. We selected the warnings for analysis randomly, based on findings by security experts, and based on CVEs. 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.

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.

Third, the CVE-selected test cases contain real-life exploitable vulnerabilities, 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. Finally, the analysis may be used as a basis for a further study of the weaknesses in the code and of static analysis.

SATE 2010 data, as well as the data from previous SATEs, suggests that tools often look for different types of weaknesses and the number of warnings varies widely by tool. The following important questions can be investigated further using SATE data:

- What is the degree of overlap among tools for different weakness categories?
- Does coincidence of warnings from multiple tools imply correctness (similar to the conclusion in [36])?
- Are important weaknesses, such as CVEs, more likely to be in higher complexity modules?

As part of SATE 2010, we selected tool warnings related to findings by security experts and to CVEs. Tools reported related warnings for 7 of 10 manual findings in Pebble and for 4 of 26 CVEs in Tomcat. The security experts did not find any weaknesses for Dovecot, and we did not find any matching warnings for Wireshark and Chrome. This does not include the results from one of the tools, MARFCAT, which was trained on the SATE 2010 CVE-based test cases and found almost all CVEs in these test cases.

In Section 3.6 we described sample CVEs, pointing out their properties relevant to their detectability by tools, and summarized possible reasons for the low number of matching tool warnings. While human analysis is better for some types of weaknesses, such as design and 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. The procedure that was used for finding CVE locations in code and selecting tool warnings related to the CVEs is new and has limitations, so the results may not indicate tools' ability to find important security weaknesses. 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

In previous SATEs, a large number of tool warnings and insufficiency of ground truth made analysis difficult. We partially addressed this problem by selecting a random subset of tool warnings for analysis and selecting tool warnings related to findings by security experts and CVEs. We plan to improve our future analysis as follows. First, we will improve the procedure for finding CVE locations in code and selecting tool warnings related to the CVEs. Second, we will introduce synthetic test cases which contain precisely characterized weaknesses and thus warnings for them are amenable to mechanical analysis. Third, we intend to improve the analysis guidelines by making the structure of the decision process more precise, clarifying ambiguous statements, and providing more details for some important weakness categories.

Additionally, the following improvements will make SATE easier for participants and more useful to the community.

- Allow teams more time to run their tools and analysts more time to analyze the tool reports.
- Since installing a tool is often easier than installing multiple test cases, provide teams with a virtual machine image containing the test cases properly configured and ready for analysis by the tools.
- Work with teams to detect and correct reporting and formatting inconsistencies early in the SATE procedure.
- Introduce a PHP language track in addition to the C/C++ and Java tracks.

Finally, we will begin a transition toward using the new unified Software Assurance Findings Expression Schema (SAFES) [3] 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-based test cases to provide ground truth for analysis. Romain Gaucher helped with planning SATE. Romain Gaucher and Ramchandra Sugasi of Cigital are the security experts that quickly and accurately performed human analysis of the test cases.

We thank Sue Wang, now at MITRE, for great help with all phases of SATE 2010, including planning, selection of CVE-based test cases, and analysis. We also thank Sue Wang, Charline Cleraux, and Jenise Reyes-Rodriguez for help with the analysis of tool reports. We thank other members of the NIST SAMATE team for their help during all phases of the exposition.

We especially thank those from participating teams – Tucker Taft, Paul Anderson, Fletcher Hirtle, Andy Chou, Peter Henriksen, Daniel Marjamaki, Ralf Huuck, Ansgar Fehnker, Benson Wu, Daniel Shih, Kwangkeun Yi, Hakjoo Oh, Nat Hillary, Serguei Mokhov, Chris Eng, and Chris Wysopal - for their effort, valuable input, and courage.

7 References

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

[2] Anderson, Paul, Bugs that Matter: True Positives and False Negatives in CodeSonar, Presentation, Static Analysis Tool Exposition (SATE 2010) Workshop, Gaithersburg, MD, Oct 1, 2010.
[3] Barnum, Sean, Software Assurance Findings Expression Schema (SAFES) Framework, Presentation, Static Analysis Tool Exposition (SATE 2009) Workshop, Arlington, VA, Nov 6, 2009.
[4] Chains and Composites, The MITRE Corporation, http://cwe.mitre.org/data/reports/chains_and_composites.html
[5] Common Vulnerabilities and Exposures (CVE), The MITRE Corporation, http://cve.mitre.org/.
[6] Common Weakness Enumeration, The MITRE Corporation, http://cwe.mitre.org/
[7] CVE Details, Serkan Özkan, http://www.cvedetails.com/.
[8] Emanuelsson, Par, and Ulf Nilsson, A Comparative Study of Industrial Static Analysis Tools (Extended Version), Linkoping University, Technical report 2008:3, 2008.
[9] Frye, C., Klocwork static analysis tool proves its worth, finds bugs in open source projects, SearchSoftwareQuality.com, June 2006.
[10] Java Open Review Project, Fortify Software, http://opensource.fortifysoftware.com/
[11] 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.
[12] 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.
[13] 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.
[14] Lindsay, David and Romain Gaucher, Threat Modeling and Manual Assessment, Presentation, Static Analysis Tool Exposition (SATE 2009) Workshop, Arlington, VA, Nov 6, 2009.
[15] Livshits, Benjamin, Stanford SecuriBench, http://suif.stanford.edu/~livshits/securibench/
[16] Michaud, F., and R. 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] Open Source Software in Java, http://java-source.net/.
[19] Open Source Vulnerability Database (OSVDB), Open Security Foundation, http://osvdb.org/.
[20] 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.
[21] SAMATE project, https://samate.nist.gov/
[22] SAMATE Reference Dataset (SRD), http://samate.nist.gov/SRD/
[23] SANS/CWE Top 25 Most Dangerous Programming Errors, http://cwe.mitre.org/top25/
[24] 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
[25] SourceForge, Geeknet, Inc., http://sourceforge.net/.
[26] Static Analysis Tool Exposition (SATE 2010) Workshop, Co-located with 13th semi-annual Software Assurance Forum, Gaithersburg, MD, Oct 1, 2010, http://samate.nist.gov/SATE2010Workshop.html
[27] Static Analysis Tool Exposition (SATE) 2008, NIST Special Publication 500-279, June 2009, Vadim Okun, Romain Gaucher, and Paul E. Black, editors.
[28] Static Analysis Tool Exposition (SATE), http://samate.nist.gov/SATE2010.html
[29] Steven M. Christey, Personal communication, October 2009.

[30] The Second Static Analysis Tool Exposition (SATE) 2009, NIST Special Publication 500-287, June 2010, Vadim Okun, Aurelien Delaitre, and Paul E. Black.
[31] Timo Sirainen, Dovecot Design/Memory, http://wiki2.dovecot.org/Design/Memory.
[32] 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.
[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., Flawfinder, http://www.dwheeler.com/flawfinder/
[35] Wheeler, David A., SLOCCount, http://www.dwheeler.com/sloccount/
[36] 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/.
[37] 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.
[38] 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).

Appendix A Decision Process Details for Information Leaks

This Appendix provides more details for one weakness category: information leaks. It is based on Steve Christey's suggestions [29].

- Mark these information leak warnings as false:
    - Error codes that communicate user-level errors or status. The "404 not found" error message is how the web server tells the client that the web page does not exist. (However, if the 404 message includes, for example, a full pathname, the warning is not false.)
    - Presentation information such as what color background should be used, or the font
    - process ID numbers
    - Version number of the software. (Exception: security software or other software that explicitly advertises itself as "invisible")
    - Inode numbers, file descriptor numbers, ...
    - Memory addresses (usually)
- Mark these information leak warnings as security or quality or insignificant:
    - Valid usernames
    - Passwords (encrypted or not)
    - Personally identifiable information (social security number, email address, phone number, address, etc.)

- Financial information (credit card number, etc.)
- Other privacy (e.g. list of books or movies most recently purchased)
- Installation path or other internal pathnames
- Session IDs, cookies, or other mechanisms for session management
- Source code of a program that should have been executed
- Entire configuration file
- Directory listings
- Process listings
- Response discrepancy information leaks, e.g. if authentication errors can return either "username not valid" or "password not valid," this would tell the attacker whether or not a given username is valid

Whether a warning is “true security,” “true quality” or insignificant, depends on the role of the person who sees the data. For example, providing unencrypted passwords to an administrator is at least a quality problem. However, providing encrypted passwords to an administrator is probably insignificant.

To distinguish between “true security” and insignificant: if the "attacker" already has access to the targeted information or functionality through legitimate means, then it may be true-but-insignificant. Any functionality advertised by the program counts as "legitimate." For example:

- On LinkedIn, your contacts are legitimately allowed to have access to your personal information such as email address and phone number – that would not be an information leak, and if flagged might be insignificant. But if any anonymous user can get your contact info without logging in - then it's a "true" information leak.
- A web application administrator should be expected to have access to the data files that list who the users on the system are, and probably knows the full path of the application, so that is probably insignificant.
- A denial-of-service weakness can only be exploited by an attacker with physical access to the machine. Well, that attacker can already cause DoS by throwing the machine out of a window, so this is insignificant.
- The administrator can trick the program into deleting itself. Presumably the program file already has the permissions to let the admin delete it, so this could be insignificant.

Generally, mark as insignificant or quality, if the information is only available to the person who started the program, or the program is remotely available but the information is only accessible to the program's administrator.

Goanna Static Analysis at the NIST Static Analysis Tool Exposition

Mark Bradley, Ansgar Fehnker, Ralf Huuck and Paul Steckler
Red Lizards Software
Email: info@redlizards.com
Url: redlizards.com

Abstract

In 2010 Red Lizard software participated for the first time in the Static Analysis Tool Exposition (SATE) organized by the National Institute of Standards and Technology (NIST) with the static analysis tool *Goanna*. The aim of SATE is to advance static analysis research and solutions that detect serious security and quality issues in source code. Goanna is a static analysis solution for the desktop and server, which detects bugs in C/C++ source code by a combination of static analysis techniques with model checking technology. This report will give a brief introduction to source code analysis with Goanna, it describes how the submission to SATE was prepared, the results that were obtained, and some of the lessons that were learned in the process.

1 Introduction

Software development cycles are a major competitive aspect in many market segments including mobile phone handsets, games, and consumer electronics. The obvious goal is to deliver software as fast as possible, as cheaply as possible at the highest possible quality. For these reasons automation and tool support play an increasing role. VDC estimates that around 50% of the software development costs result from testing and debugging.

A category tools that help to reduce these costs are source code analysis tools like *Goanna*. These tools use a combination of techniques to detect deficiencies of the source code in the programming phase. Integrating these tools in the SDLC has the numerous benefits. First, it reduces the number of defects detected by testing, and thus the number of test cycles. It is estimated that finding bug by testing can be up to 80 times more costly than finding them in the programming phase. These tools can also be used to ensure that code meets certain coding standards, which will help to make keep it maintainable. They furthermore help programmers with debugging their own code more efficiently, i.e. coding itself becomes more efficiently, especially if these tools are available in the development environment. Another benefit is that these tool can find potential bugs that are difficult, if not impossible, to find through traditional means. And of course there is the benefit that these tools give the programmer automatic, and often instant feedback on his programming.

Goanna by Red Lizards Software is an integrated C/C++ source code analysis tools for mission-critical industries. It is the first solution in the market that combines the automated technologies of static analysis with model checking. There are two product lines: *Goanna Studio*, the IDE version, and *Goanna Central*, the command line version.

Goanna Studio is the desktop solution, integrated with either the Eclipse IDE both for Linux and Windows, or Microsoft Visual Studio, version 2005, 2008 and 2010. Red Lizard Software was a SimShip partner when Microsoft launched Visual Studio 2010, the only static analysis solution in the market to be jointly launched. Goanna Studio is a developer tool, fully integrated into the IDEs, and offers the full solution to be used while programming.

Goanna Central is the command line version that can be integrated with the nightly build system. It supports all common common C/C++ dialects such as ANSI/ISO C, the Microsoft dialects of C/C++, and GNU C/C++, and the most common build systems such as make, cmake, or scons.

NIST specified for SATE five C/C++ code bases as test bed for the participating static analysis tools. One code base was for Dovecot, an open source IMAP and POP3 server, and two different version for each Wireshark, a network protocol analyzer, and for Chrome an operating system. To analyze this code we used Goanna Central for Linux. The next section introduces the different types of static analysis techniques, Section 3 desribes the tool architecture of Goanna, Section 4 describes which checks and which version of Goanna were used in SATE, Section 5 presents the overall results, while Section 6 selects and explain a few warnings from the tool reports.

2 Static Analysis Technologies

Goanna checks for bugs, memory leaks and security vulnerabilities, is fully path-sensitive and inter-procedural. It uses a combination of techniques, from pattern matching, to data flow analysis, and model checking. In the following we describe the main techniques used by modern static analysis tools for detecting security vulnerabilities in source code.

Tree-Pattern Matching. Pattern matching is the simplest technology applied by virtually all static analysis tools. Pattern matching means to identify points of interest in the program's syntax, typically by defining queries on either the plain source code or on the *abstract syntax tree* (AST), i.e. the representation of the source code after parsing. For instance, it is well known that in C/C++ programs the library function `strcopy` is vulnerable to security exploitation if not used absolutely correctly. Hence, a simple keyword scan for `strcopy` can help with this. Queries can become more complicated, and a series of interdependent queries may be used for more advanced checks, for example to identify inconsistent use of semantic attributes. Our analysis tool Goanna uses such tree-pattern matching [BFK02] on the AST. However, pattern matching is fundamentally limiting, since it only searches for keywords and their context, but is unable to take control flow, data flow, or other semantic information into account.

Data Flow Analysis. The next step up in terms of technology is about understanding how certain constructs are related. Data flow analysis [NNH99] makes use of the structure of program, in particular its *control flow graph* (CFG). It is a standard compiler technique to examine the flow of information between variables and other elements of interest that can be syntactically identified. An example is checking for uninitialized variables leading to unexpected arbitrary behavior. We can syntactically identify program locations that are variable declarations and uses of variables either as a reading operation (such as the right hand side of an assignment) or a write operation (such as the left hand side of an assignment). Data flow analysis enables us to examine if there exists a read operation to a variable without a prior write operation. These and similar methods are used to analyze how certain elements in a program are related. Typically, however, data flow analysis uses a number of approximation techniques that prohibit the precise identification of the program path. To remedy this *model checking* can be applied.

Model Checking. Model checking was developed in the early 1980s as a technique to check whether larger concurrent systems satisfy given temporal properties [CE82, QS82]. It is essentially a technique to determine whether all paths in a graph satisfy certain ordering of events along that path. Unlike other techniques that enumerate paths, model checking does not put a limit of the number, the length or the branching of paths. Another advantage is that if it finds a violation, it will also return a counterexample path. For their solution the original authors received the Turing Award in 2007.

Model checking was originally applied to the formal verification of protocols and hardware designs. In recent years a strong push has been made towards software verification, and effective methods have been developed to overcome scalability challenges. Our tool Goanna is the first tool that manages to apply model checking on a grand scale to static program analysis. This means that we use it to analyze millions of lines of code for over one hundred different classes of checks. The advantage of this approach is that it is oftentimes faster than existing data flow and path enumeration approaches. Moreover, it also tends to be more powerful as it allows specifying complex relations between program constructs.

Abstract Data Tracking. To detect buffer overflows, division-by-zero errors and other defects it is helpful to know as much as possible about the values that can occur at certain program locations. A technique to approximate and track data values is commonly called abstract interpretation [Cou81]. Abstract interpretation estimates for every variable at every program location all the potential range of variables. For example, for an array access, all potential index values can be estimated. Different domains can be used for data tracking, with varying levels of precision. There is a trade-off between the precision and speed of the analysis. Goanna uses a variant of interval constraint analysis that is reasonably precise while fast, but can also use automated theorem proving techniques (SMT solving) to validate individual program paths.

Inter-procedural Analysis. While the aforementioned techniques help to detect various security issues and are often complimentary, one of the key factors to success is to scale to large code bases. Many potential issues require to understand the overall call structure of a program, and being able to track data and control flow across function boundaries. To overcome this challenges advanced tools introduce the ability to generate *function summaries* automatically. These summaries contain only the essential information of a function that is needed for particular security check. Instead of propagating information of the whole function, which can be prohibitively large, only the summary information is used. Generating summaries is an iterative process taking the mutual impact (from recursion etc.) into account.

3 The Goanna Static Analysis

Goanna is an automated analysis tool that does not execute code as in traditional testing, but examines the code structure, the keywords, their inter-relation as well as the data and information flow across the whole source code base. These techniques can be fully automated and scale to millions of lines of code. As such Goanna is able to identify many classes of security issues automatically at software development time. Moreover, Goanna is the only tool that combines all the techniques mentioned in Section 2. The Goanna tool is fully path-sensitive and performs inter-procedural analysis. For a detailed technical description of the underlying formal techniques see [FHJ+07].

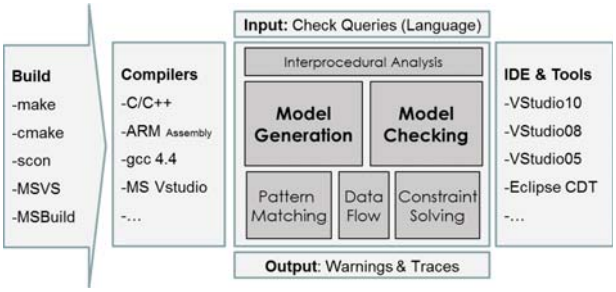

Figure 1: Goanna static analysis architecture

Supported Languages and Architectures. The Goanna tool is currently implemented to handle C/C++ source code. The architecture and the technology are in theory adaptable to other imperative programming languages such as $C\#$ or Java. However, the C/C++ programming language is still predominant in many mission-critical systems while at the same time it easily suffers from potential exploits.

The Goanna tool fully parses C/C++ source code including compiler specific dialects and extensions such as GNU, Visual C++ or GreenHills. It is available for Microsoft Windows as well as various Linux on either 32-bit or 64-bit machines. On top of that Goanna supports the cross-compilation to a number of embedded platforms. It is possible to integrate the tool in popular build environments, such as *make* or *scons*.

Analysis Framework. Figure 1 depicts how the different types of analysis are embedded in Goanna. The core of the tool uses model checking, while the other techniques are used for particular checks, and to assist in the generation of the models. A distinguishing feature of Goanna is that it architecturally modularizes the core analysis engine and the checks that are performed. Goanna has a specification language to define checks; this allow for the rapid development of new checks, separately from the core algorithms, and any future improvement of the engine will be effective for all checks.

In short, Goanna uses model checking for all path-sensitive check on how program information relates. It

uses tree-pattern matching to identify certain locations and operations of interest and abstract data tracking by constraint solving for specialized checks, such as buffer overruns, shifting beyond bounds and overflow errors. Moreover, Goanna uses a number of heuristics and advanced features such as interprocedural whole-program analysis to achieve speed and scalability.

4 The Entry

NIST specified five code bases for SATE, but the participants were free to configure their tool to fit the code base. For the entry to SATE we used a development snapshot of the then current version of Goanna 2.0 (r7225), running on Linux Ubuntu Server 8.04. None of the code bases required special configuration above what would be required to configure a normal build. All of the code bases specified by NIST use a combination of `configure` and `make` files; all that was required is to configure them with Goanna, and in addition, since we are using interprocedural analysis, to specify a database file. The remaining configuration dealt with formatting the output to comply with the NIST specification.

For analysis of Chrome 5.0.375.54 and 5.0.375.70 we used the standard version of goanna. For Dovecot and Wireshark 1.2.0 and 1.2.9 we used a prototype version. It differed from the standard version in that it used an additional SMT solver to evaluate paths, and an off-the-shelf model checker, rather than our custom build checker. The latter has no influence on the results. It was used to for the prototype because it supported a secondary feature that was not supported at the time by the in house checker. This came at an expense of performance, since it has not been optimised for the use with Goanna. The in-house checker can be up to 2 orders of magnitude faster than the off-the-shelf solution.

For our submission we selected the 55 default checks of Goanna 2.0. These checks are enabled by default, because they detect the more serious issues that should be addressed by any programmer. We omitted the other checks which typically deal more with either stylistic requirements, such as the unused parameter check, or warn very conservatively, i.e. even if an actual bug is unlikely. The remainder of this section will give a quick overview of the classes of checks that we included.

Array bounds. These checks are concerned with correct array access. They will warn if they detect an out of bound array access. We included three checks of this class.

Arithmetic errors. These checks are concerned with arithmetic errors such as division by zero or out of bound shifts. We included ten checks of this class.

C++ copy control. These checks are concerned with the correct initialization, construction or destruction. This class only applies to C++. We included three checks from this class.

C++ usage. These checks are concerned with the correct use of C++ features. For example, they warn if a nonvirtual destructor is defined for an abstract class.

Potentially unexpected behavior. These checks are concerned with features and idiosyncracies of C/C++ that are often poorly understood. An example is an ambiguous use of an `else`. Two of these checks have been included.

Function pointer usage. These are concerned with using accidentally a function pointer incorrectly, for example in an arithmetic expression. We included two checks of this class.

Memory usage. These checks deal with the correct usage of stacks, arrays and pointers, like storing a stack address in a global variable. Ten of these checks have been included.

Pointer misuse. These checks deal more specifically with the correct use of pointers, like testing for NULL pointer. We included five checks of this class.

Redundant code. These checks deal with code that may be redundant, like dead code or trivial conditions. We included six checks of this class.

Semantic attributes. These checks are concerned with the correct use of the GNU C language extension with semantic attributes like `const` or `pure`. We included four checks from this class.

Unspecified behavior. These checks deal with code for which the C standard does not define behavior. This

includes cases where the execution order is undefined, or cases where initialization is undefined. We included five checks of this class.

As mentioned before, we selected the default checks, which report serious issues, and assume that the programmer is programming defensively. They try not to warn in cases that are common and accepted programming practice. And example would be redundant `return` statements. While the warning would be technically correct, redundant `return` statements are often included to deal with compiler warnings on missing `return` statements.

5 The Results

The size of the code bases in SATE differed greatly. Dovecot is 360 kLoC in size, Wireshark 1.2.0 and 1.2.9 are both 1.7 MLoC, and Chrome 5.0.375.54 and 5.0.375.70 are 1.5MLoC and 1.7MLoC, respectively. The number of warnings Goanna issued range from 180 for Dovecot, 534 and 532 for the Wireshark code bases, to 1079 and 1173 for the Chrome code bases, respectively. But the result do not only differ in the number of warnings found, but also in the type of warnings. Table 1 list the top ten checks by the number of warnings for Dovecot.

Most of the Dovecot warnings relate to potential NULL pointer dereferences and to trivial conditions. A class of warning that only occurred for the Dovecot code base are the warnings on the correct use of the semantic attributes `const` and `pure`. These attributes are a GNU language extension, that can be used in compiler optimization, however the gnu compiler does not check for proper use of these attributes. This might explain the fairly high number of warnings of this type; the programmer is expected to use these attributes correctly, without being held to a correct use by the compiler.

The Wireshark code bases does not use this language extension, and this type of warnings is therefore absent. The majority of warnings concern potential NULL pointer, uninitialized variables and trivial conditions. One check with remarkably many warnings is the check for unused pointer values. This check will warn if a pointer is assigned a value that is not NULL, and then not used along any path. While the warnings can be insignificant, they do quite often uncover serious issues, if, for example, by a copy an paste error, the wrong pointer was used subsequently.

#	Check	
1	Dereference of possible NULL pointer	59
2	Comparison never holds	31
3	Comparison always holds	21
4	Uninitialized variable on some paths	15
5	Global variable access by `const` function	12
6	Dereference of possible NULL pointer by function	11
7	Call of function w/o `pure` by function with `pure` attribute	10
8	Unused variable on all paths	7
9	Uninitialized struct field	4
10	Store stack in a global	2

Table 1: The 10 checks with the most warnings for Dovecot, ranked by the number of warnings.

#	Check	1.2.0	1.2.9
1	Comparison never holds	281	282
2	Unused pointer value	54	55
3	NULL check after dereference	54	50
4	Uninitialized variable on some paths	35	35
5	Dereference of possible NULL pointer	35	34
6	Comparison always holds	33	35
7	Uninitialized struct field	10	10
8	Dereference of possible NULL pointer by function	8	8
9	Variable used in divisor before comparison with 0	6	6
10	Parameter checked before deref only on some paths	4	4

Table 2: The 10 checks with the most warnings for Wireshark 1.2.0 and 1.2.9, ranked by the number of warnings.

The number of warnings changed slightly from version 1.2.0 to 1.2.9, but not significantly. A warning that disappeared completely in revision 1.2.9 was a warning about the potential dereference of a definite NULL

#	Check	x.54	x.70
1	Dereference of possible NULL pointer	400	424
2	Comparison always holds	166	193
3	NULL check after dereference	110	112
4	Comparison never holds	92	102
5	Dereference of possible NULL pointer by function	54	63
6	Uninitialized variable on some paths	49	55
7	Paramter checked before derefence only on some paths	42	43
8	Unused variable on all paths	29	23
9	Switch case unreachable	21	21
10	Uninitialized variable on all paths	14	25

Table 3: The 10 checks with the most warnings for Chrome 5.0.375.54 and 5.0.375.70, ranked by the number of warnings.

pointer. There were only two warning of this particular check for Wireshark 1.2.0, both caused by the incorrect used of a macro `#define MATCH ((class == info->tclass) && (tag == info->tag))`. It was used on a possible paths when `info` was definitely NULL. In Wireshark 1.2.9 this macro was replaced by `#define MATCH (info && (class == info->tclass) && (tag == info->tag))`, i.e. it included a NULL check before the rest of the expression got evaluated. This effectively addressed the warning.

Table 3 lists the most common warning for Chrome. Also for this code bases most warning were concerned with trivial conditions, uninitialized variables, and potential NULL pointers. For Chrome there are significant changes between version 5.0.375.54 and 5.0.375.70, but this reflects that the later version also grew 11% in size. The next section will discuss a few warning that have been evaluated by NIST in detail.

6 Selected Warnings Explained

NIST selected as a part of SATE for each tool 30 warnings from the set of Dovecot warnings for evaluation. In addition NIST selected CVEs from Wireshark, that is known bugs the list of Common Vulnerabilities and Exposures maintained by MITRE. In this section we will discuss a few of these warnings and defects, since they illustrate nicely the types of analysis Goanna performs to detect potential defects.

Failed Error Handling Routine. The following warnings from the Dovecot code base deals with the correct use of semantic attributes; a checks which requires pattern matching only.

```
unichar.c:193: warning: Goanna[SEM-const-call]
Non-const function 'uint16_find' is called in
const function 'uni_ucs4_to_titlecase'
```

```
    unichar_t uni_ucs4_to_titlecase(unichar_t
        chr)
    { [...]
193   if (!uint16_find(titlecase16_keys,
          N_ELEMENTS(titlecase16_keys), chr, &
              idx))
            return chr;
          else [...]
```

The detected issue results from a wrong use of GNU semantic attribute `const`. These allow the use to define attributes of functions, which can then be used by the compiler to optimize code. In the above example function `uni_ucs4_to_titlecase` has the attribute `const`. The documentation on the GNU language extension says that "a function that calls a non-const function usually must not be const". This requirement is violated in the above example, since function `uint16_find` does not have the attribute `const`. To find this type of violation it is sufficient to check if all functions that are a called in a function with attribute `const`, have this attribute themselves. This can be achieved by combining two patterns on the AST, one to find function calls in `const` functions, and one to check the attribute of the called functions.

Failed Error Handling Routine. The following example, also from Dovecot, was found with a combination of summaries and abstract data tracking.

```
director-connection.c:655: warning: Goanna[RED-
cmp-never] Comparison never holds
```

```
655 if (str_array_length(args) != 2 ||
        director_args_parse_ip_port(...) < 0) {
          i_error( );
          return FALSE;
    }
```

The above code fragment is part of an error handler; the error routine `i_error` will be called if the output of `director_args_parse_ip_port` is negative. However, the output range of this function is in the interval $[0, 1]$ and will thus never be negative. Closer inspection showed that this code was refactored to return 0 if an error is detected rather than -1. Except for a few exceptions, the output of this function was treated like a Boolean. The mistake presumably entered since manual refactoring did not change all occurring tests in the error handler to use a Boolean condition. Abstract data tracking was used to determine the output range of `director_args_parse_ip_port`. This range was then part of the inter-procedural summary and subsequently used to detect unsatisfiable conditions in other functions.

Denial of Service. The following example, from the Wireshark code base, highlights a potential denial of service exploit. Goanna required its model checking capabilities to detect this security issue. The surrounding code is:

```
packet-smb.c:8211: warning: Goanna[PTR-param-
unchk-some] Parameter 'nti' is not checked
against NULL before it is dereferenced on some
paths, but on other paths it is
```

```
    case NT_TRANS_IOCTL: [...]
8211   dissect_smb2_ioctl_data(ioctl_tvb, pinfo,
           tree, top_tree, nti->ioctl_function,
           TRUE);
       [...]
    case NT_TRANS_SSD:
      if(nti){switch(nti->fid_type){ [...]
```

The detected issue resides in a longer switch statement. In one case `nti` is not checked to be not NULL before it is dereferenced, in the other case it is. This points to an inconsistency, which can lead to a NULL-pointer dereference. This particular example was reported as CVE-2010-2283 in the MITRE bug database. Finding this bug requires to compare behavior along two paths. In this example, the inconsistency happens within a few lines. In general inconsistent paths may contain conditional jumps and loops, such that an explicit path enumeration becomes infeasible. Model checking provides algorithms to checks this exhaustively and efficiently.

Unfortunately, this warning was not included in the Goanna report for Wireshark. In Goanna it is possible to set a timeout, after which the tool stop with the analysis of a file, and proceeds with the following file. For the SATE participation the timeout was set to 120 seconds, which in hindsight seems to be rather low. In this particular case only half of the file was analyzed, the function in line 8211 however was not. Increasing the timeout to a more reasonable five minutes would have revealed this bug.

Potential Program Crash. The final warning we like to discuss is also taken from the Dovecot code base. Goanna used model checking and interprocedural whole-program analysis to detect this defect.

```
director.c 180: warning: Goanna[PTR-null-assign
-fun-pos] Dereference of 'preferred_host' which
may be NULL
```

```
    *preferred_host =
        director_get_preferred_right_host(dir);
                  [...]
    if (cur_host != preferred_host)
180   (void)director_connect_host(dir,
          preferred_host);
    else {[...]}
```

The detected issues is based on the fact that the function `director_get_preferred_right_host` may return a NULL pointer and assigns it to `*preferred_host`. This is later used as parameter of `director_connect_host`. However, inter-procedural analysis shows that there exists a path in `director_connect_host` where this parameter will be dereferenced, without a prior check for being potentially NULL. The dereferencing of this NULL pointer can lead to an exception/crash that enables an attacker to potentially enter the system.

7 Conclusion

This report described the entry and results of the static analysis tool Goanna by Red Lizard Software for the

Static Analysis Tool Exposition, organized by NIST. Goanna is a novel type of tool that combines static analysis techniques such as pattern matching, data flow analysis and abstract data tracking with model checking to obtain a fast, scalable and precise solution to detect potential software defects. For SATE we selected the default checks of Goanna, and applied them to the five code bases.

The results show that Goanna is a competitive solution to find serious software defects in real life code. At the time of writing the final results of the evaluation by NIST evaluation are still unknown, and it would be premature to make a final comment. Intermediate results that were shared among participants, however, confirm that Goanna is at least on par with the other leading tools in this area when it comes to the fraction of serious security and quality issues detected, versus the fraction of insignificant and false warnings. This is what a tool should to deliver; actionable warnings that help with improving the code.

The intermediate feedback also gave us valuable feedback on how to further improve the tool. The feedback was in particular useful, since the manual evaluation of NIST went through the effort to describe the potential issue in detail. Cause for false positives were omitted corner cases, unused semantic information, incomplete semantic models, non-trivial invariants, and custom asserts. The first two causes can be directly addressed by refining the existing checks. Given that Goanna defines the checks separately from the analysis engine, this is an easy improvement. This type of issue that were revealed by the NIST evaluation have been addressed in the meanwhile.

Addressing an incomplete semantic model requires to improve the basic semantic models are already used by the tool; all that is required to refine these models. Non trivial invariants is a more fundamental problem of static analysis. For example, detecting one of the CVEs selected by NIST, CVE-2010-2286, would require to prove the absence of an appropriate loop invariant, for a loop that was realized by gotos that span approximately 2000 lines of code. This type of analysis is arguably outside of the scope of static analysis tools. Even more powerful techniques, such as automatic theorem provers, would have a hard time detecting such issues automatically, regardless of the problem that these solutions will currently also not scale to problems of this size. The final problem of custom asserts can be addressed by giving the user a way to redefine or refine checks, or give by giving them the means to add annotations or pragmas. However, from among users there exists some reluctancy to change the code to accommodate static analysis, which is expected to be fully automatic.

Red Lizard Software participated in 2010 for the first time in SATE. The challenge for the tools was to deal with code bases of different sizes and of different type. The evaluation by NIST focused mostly on the quality of the warnings, rather than on the speed of a solution or its ease of use. The exposition helped to shed a light on the strength and weaknesses of the tool, and confirmed that Goanna is a competitive solution for C/C++ analysis.

References

[BFK02] Michael Benedikt, Wenfei Fan, and Gabriel M. Kuper, *Structural properties of xpath fragments*, ICDT '03: Proceedings of the 9th International Conference on Database Theory (London, UK), Springer, 2002, pp. 79–95.

[CE82] Edmund M. Clarke and E. Allen Emerson, *Design and synthesis of synchronization skeletons for branching time temporal logic*, Logics of Programs Workshop, LNCS, vol. 131, Springer, 1982, pp. 52–71.

[Cou81] P. Cousot, *Semantic foundations of program analysis*, Program Flow Analysis: Theory and Applications, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981, pp. 303–342.

[FHJ+07] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch, *Model checking software at compile time*, Proc. TASE 2007, IEEE Computer Society, 2007.

[NNH99] F. Nielson, H. Riis Nielson, and C. L. Hankin, *Principles of program analysis*, Springer, 1999.

[QS82] Jean-Pierre Queille and Joseph Sifakis, *Specification and verification of concurrent systems in CESAR*, Proc. Intl. Symposium on Programming, Turin, April 6–8, 1982, Springer, 1982, pp. 337–350.

The use of machine learning with signal- and NLP processing of source code to fingerprint, detect, and classify vulnerabilities and weaknesses with MARFCAT

Serguei A. Mokhov
Concordia University
Montreal, QC, Canada
mokhov@cse.concordia.ca

Abstract

We present a machine learning approach to static code analysis and findgerprinting for weaknesses related to security, software engineering, and others using the open-source MARF framework and the MARFCAT application based on it for the NIST's SATE 2010 static analysis tool exposition workshop.

1 Introduction

This paper elaborates on the details of the methodology and the corresponding results of application of the machine learning techniques along with signal processing and NLP alike to static source code analysis in search for weaknesses and vulnerabilities in such a code. This work resulted in a proof-of-concept tool, code-named *MARFCAT*, a MARF-based Code Analysis Tool [14], presented at the Static Analysis Tool Exposition (SATE) workshop 2010 [21] collocated with the Software Assurance Forum on October 1, 2010.

At the core of the workshop there were C/C++-language and Java language tracks comprising CVE-selected cases as well as stand-alone cases. The CVE-selected cases had a vulnerable version of a software in question with a list of CVEs attached to it, as well as the most know fixed version within the minor revision number. One of the goals for the CVE-based cases is to detect the known weaknesses outlined in CVEs using static code analysis and also to verify if they were really fixed in the "fixed version" [21].
The test cases at the time included CVE-selected:

- C: Wireshark 1.2.0 (vulnerable) and Wireshark 1.2.9 (fixed)
- C++: Chrome 5.0.375.54 (vulnerable) and Chrome 5.0.375.70 (fixed)
- Java: Tomcat 5.5.13 (vulnerable) and Tomcat 5.5.29 (fixed)

and non-CVE selected:

- C: Dovecot 2.0-beta6
- Java: Pebble 2.5-M2

We develop MARFCAT to machine-learn from the CVE-based vulnerable cases and verify the fixed versions as well as non-CVE based cases from similar programming languages.

Organization

The related work, some of the present methodology is based on, is referenced in Section 2. The methodology summary is in Section 3. We present the results, most of which were reported at SATE2010, in Section 4. We then describe the machine learning aspects as well as mathematical estimates of functions of how to determine line numbers of unknown potentially weak code fragments in Section 3.5. (The latter is necessary since during the representation of the code a wave form (i.e. signal) with current processing techniques the line information is lost (e.g. filtered out as noise) making reports less informative, so we either machine-learn the line numbers or provide a mathematical estimate and that section describes the proposed methodology to do so, some of which was implemented.) Then we present a brief summary, description of the limitations of the current realization of the approach and concluding remarks in Section 5.

2 Related Work

Related work (to various degree of relevance) can be found below (this list is not exhaustive):

- Taxonomy of Linux kernel vulnerability solutions in terms of patches and source code as well as categories for both are found in [16].
- The core ideas and principles behind the MARF's pipeline and testing methodology for various algorithms in the pipeline adapted to this case are found in [11]. There also one can find the core options used to set the configuration for the pipeline in terms of algorithms used.
- A binary analysis using machine learning approach for quick scans for files of known types in a large collection of files is described in [15].
- The primary approach here is similar in a way that was done for DEFT2010 [13, 12] with the corresponding `DEFT2010App` and its predecessor `WriterIdentApp` [17].
- Tlili's 2009 PhD thesis covers topics on automatic detection of safety and security vulnerabilities in open source software [24].
- Statistical analysis, ranking, approximation, dealing with uncertainty, and specification inference in static code analysis are found in the works of Engler's team [8, 6, 7].
- Kong et al. further advance static analysis (using parsing, etc.) and specifications to eliminate human specification from the static code analysis in [5].
- Spectral techniques are used for pattern scanning in malware detection by Eto et al. in [1].
- Researchers propose a general data mining system for incident analysis with data mining engines in [4].
- Hanna et al. describe a synergy between static and dynamic analysis for the detection of software security vulnerabilities in [3] paving the way to unify the two analysis methods.
- The researchers propose a MEDUSA system for metamorphic malware dynamic analysis using API signatures in [18].

3 Methodology

Here we briefly outline the methodology of our approach to static source code analysis in its core principles in Section 3.1, the knowledge base in Section 3.2, machine learning categories in Section 3.3, and the high-level step-wise description in Section 3.4.

3.1 Core principles

The core methodology principles include:

- Machine learning
- Spectral and NLP techniques

We use signal processing techniques, i.e. presently we do not parse or otherwise work at the syntax and semantics levels. We treat the source code as a "signal", equivalent to binary, where each n-gram ($n = 2$ presently, i.e. two consecutive characters or, more generally, bytes) are used to construct a sample amplitude value in the signal.

We show the system examples of files with weaknesses and MARFCAT learns them by computing spectral signatures using signal processing techniques from CVE-selected test cases. When some of the mentioned techniques are applied (e.g. filters, silence/noise removal, other preprocessing and feature extraction techniques), the line number information is lost as a part of this process.

When we test, we compute how similar or distant each file is from the known trained-on weakness-laden files. In part, the methodology can approximately be seen as some signature-based antivirus or IDS software systems detect bad signature, except that with a large number of machine learning and signal processing algorithms, we test to find out which combination gives the highest precision and best run-time.

At the present, however, we are looking at the files overall instead of parsing the fine-grained details of patches and weak code fragments, which lowers the precision, but is fast to scan all the files.

3.2 CVEs – the "Knowledge Base"

The CVE-selected test cases serve as a source of the knowledge base to gather information of how known weak code "looks like" in the signal form, which we store as spectral signatures clustered per CVE or CWE. Thus, we:

- Teach the system from the CVE-based cases
- Test on the CVE-based cases
- Test on the non-CVE-based cases

3.3 Categories for Machine Learning

The tow primary groups of classes we train and test on include:

- CVEs [19, 20]
- CWEs [25] and/or our custom-made, e.g. per our classification methodology in [16]

The advantages of CVEs is the precision and the associated meta knowledge from [19, 20] can be all aggregated and used to scan successive versions of the the same software or derived products. CVEs are also generally uniquely mapped to CWEs. The CWEs as a primary class, however, offer broader categories, of kinds of weaknesses there may be, but are not yet well assigned and associated with CVEs, so we observe the loss of precision.

Since we do not parse, we generally cannot deduce weakness types or even simple-looking aspects like line numbers where the weak code may be. So we resort to the secondary categories, that are usually tied into the first two, which we also machine-learn along, shown below:

- Types (*sink*, *path*, *fix*)
- Line numbers

3.4 Basic Methodology

Algorithmically-speaking, MARFCAT performs the following steps to do its learning analysis:

1. Compile meta-XML files from the CVE reports (line numbers, CVE, CWE, fragment size, etc.). Partly done by a Perl script and partly manually. This becomes an index mapping CVEs to files and locations within files.

2. Train the system based on the meta files to build the knowledge base (learn). Presently in these experiments we use simple mean clusters of feature vectors per default MARF specification ([11, 23]).

3. Test on the training data for the same case (e.g. Tomcat 5.5.13 on Tomcat 5.5.13) with the same annotations to make sure the results make sense by being high and deduce the best algorithm combinations for the task.

4. Test on the testing data for the same case (e.g. Tomcat 5.5.13 on Tomcat 5.5.13) without the annotations as a sanity check.

5. Test on the testing data for the fixed case of the same software (e.g. Tomcat 5.5.13 on Tomcat 5.5.29).

6. Test on the testing data for the general non-CVE case (e.g. Tomcat 5.5.13 on Pebble).

3.5 Line Numbers

As was earlier mentioned, line number reporting with MARFCAT is an issue because the source text is essentially lost without line information preserved (filtered out as noise or silence or mixed in with another signal sample). Therefore, some conceptual ideas were put forward to either derive a heuristic, a function of a line number based on typical file attributes as described below, or learn the line numbers as a part of the machine learning process. While the methodology of the line numbers discussed more complete scenarios and examples, only and approximation subset was actually implemented in MARFCAT.

3.5.1 Line Number Estimation Methodology

Line number is a function of the file's dimensions in terms of line numbers, size in bytes, and words. The meaning of W may vary. The implementations of f may vary and can be purely mathematical or relativistic and with side effects. These dimensions were recorded in the meta XML files along with the other indexing information. This gives as the basic Equation 1.

$$l = f(L_T, B, W) \tag{1}$$

where

- L_T – number of lines of text in a file
- B – the size of the file in bytes
- W – number of *words* per `wc` [2], but can be any blank delimited printable character sequence; can also be an n-gram of n characters.

The function should be additive to allow certain components to be zero if the information is not available or not needed, in particular $f(B)$ and $f(W)$ may fall into this category. The ceiling $\lceil \ldots \rceil$ is required when functions return fractions, as shown in Equation 2.

$$f(L_T, B, W) = \lceil f(L_T) + f(B) + f(W) \rceil \tag{2}$$

Constraints on parameters:

- $l \in [1, \ldots, L_T]$ – the line number must be somewhere within the lines of text.
- $f(L_T) > 0$ – the component dependent on the the lines of text L_T should never be zero or less.
- $EOL = \{\texttt{\textbackslash n}, \texttt{\textbackslash r}, \texttt{\textbackslash r\textbackslash n}, \texttt{EOF}\}$. The inclusion of `EOF` accounts for the last line of text missing the traditional line endings, but is non-zero.
- $L_T > 0 \implies B > 0$
- $B > 0 \implies L_T > 0$ under the above definition of `EOL`; if `EOF` is excluded this implication would not be true
- $B = 0 \implies L_T = 0, W = 0$

Affine combination is in Equation 3:

$$f(L_T, B, W) = \lceil k_L \cdot f(L_T) + k_B \cdot f(B) + k_W \cdot f(W) \rceil \tag{3}$$

- $k_L + k_B + k_W < 1 \implies$ the line is within the triangle

Affine combination with context is in Equation 4:

$$f(L_T, B, W) = \lceil k_L \cdot f(L_T) + k_B \cdot f(B) + k_W \cdot f(W) \rceil \pm \Delta c \tag{4}$$

where $\pm\Delta c$ is the amount of context surrounding the line, like in `diff` [9]; with $c = 0$ we are back to the original affine combination.

Learning approach with matrices and probabilities from examples. This case of the line number determination must follow the preliminary positive test with some certainty that a give source code file contains weaknesses and vulnerabilities. This methodology in itself would be next to useless if this preliminary step is not performed.

In a simple case a line number is a cell in the 3D matrix M given the file dimensions alone, as in Equation 5. The matrix is sparse and unknown entries are 0 by default. Non-zero entries are learned from the examples of files with weaknesses. This matrix is capable of encoding a single line location per file of the same dimensions. As such it can't handle multiple locations per file or two or more distinct unrelated files with different line numbers for a single location. However, it serves as a starting point to develop a further and better model.

$$l = f(L_T, B, W) = M[L_T, B, W] \tag{5}$$

To allow multiple locations per file we either replace the W dimension with the locations dimension N if W is not needed, as e.g. in Equation 6, or make the matrix 4D by adding N to it, as in Equation 7. This will take care of the multiple locations issue mentioned earlier. N is not known at the classification stage, but the coordinates L_T, B, W will give a value in the 3D matrix, which is a vector of locations $\vec{n}$. At the reporting stage we simply report all of the elements in $\vec{n}$.

$$\vec{l} = f(L_T, B, W) = M[L_T, B, N] \tag{6}$$

$$\vec{l} = f(L_T, B, W) = M[L_T, B, W, N] \tag{7}$$

In the above matrices M, the returned values are either a line number l or a collection of line numbers $\vec{l}$ that were learned from examples for the files of those dimensions. However, if we discovered a file tested positive to contain a weakness, but we have never seen its dimensions (even taking into the account we can sometimes ignore W), we'll get a zero. This zero presents a problem: we can either (a) rely on one of the math functions described earlier to fill in that zero with a non-zero line number or (b) use probability values, and convert M to M_p, as shown in Equation 8.

The M_p matrix would contain a vector value $\vec{n_p}$ of probabilities a given line number is a line number of a weakness.

$$\vec{l_p} = f(L_T, B, W) = M_p[L_T, B, W, N] \tag{8}$$

We then select the most probable ones from the list with the highest probabilities. The index i within $\vec{l_p}$ represents the line number and the value at that index is the probability $p = \vec{l_p}[i]$.

Needless to say this 4D matrix is quite sparse and takes a while to learn. The learning is performed by counting occurrences of line numbers of weaknesses in the training data over total of entries. To be better usable for the unseen cases the matrix needs to be smoothed using any of the statistical estimators available, e.g. from NLP, such as add-delta, ELE, MLE, Good-Turing, etc. by spreading the probabilities over to the zero-value cells from the non-zero ones. This is promising to be the slowest but the most accurate method.

In MARF, M is implemented using `marf.util.Matrix`, a free-form matrix that grows upon the need lazily and allows querying beyond physical dimensions when needed.

4 Results

The preliminary results of application of our methodology are outlined in this section. We summarize the top precisions per test case using either signal-processing or NLP-processing of the CVE-based cases and their application to the general cases. Subsequent sections detail some of the findings and issues of MARFCAT's result releases with different versions.

4.1 Preliminary Results Summary

Current top precision at the SATE2010 timeframe:

- Wireshark:
  - CVEs (signal): 92.68%, CWEs (signal): 86.11%,
  - CVEs (NLP): 83.33%, CWEs (NLP): 58.33%
- Tomcat:
  - CVEs (signal): 83.72%, CWEs (signal): 81.82%,
  - CVEs (NLP): 87.88%, CWEs (NLP): 39.39%
- Chrome:
  - CVEs (signal): 90.91%, CWEs (signal): 100.00%,
  - CVEs (NLP): 100.00%, CWEs (NLP): 88.89%
- Dovecot:
  - 14 warnings; but it appears all quality or false positive
  - (very hard to follow the code, severely undocumented)
- Pebble:
  - none found during quick testing

What follows are some select statistical measurements of the precision in recognizing CVEs and CWEs under different configurations using the signal processing and NLP processing techniques.

"Second guess" statistics provided to see if the hypothesis that if our first estimate of a CVE/CWE is incorrect, the next one in line is probably the correct one. Both are counted if the first guess is correct.

4.2 Version SATE.4

4.2.1 Wireshark 1.2.0

Typical quick run on the enriched Wireshark 1.2.0 on CVEs is in Table 1. All 22 CVEs are reported. Pretty good precision for options `-diff` and `-cheb` (Diff and Chebyshev distance classifiers, respectively [11]). In Unigram, Add-Delta NLP results on Wireshark 1.2.0's training file for CVEs, the precision seems to be overall degraded compared to the classical signal processing pipeline. Only 20 out of 22 CVEs are reported, as shown in Table 2. CWE-based testing on

Wireshark 1.2.0 (also with some basic line heuristics that does not impact the precision) is in Table 3.

The following select reports are about Wireshark 1.2.0 using a small subset of algorithms. There are line numbers that were machine-learned from the `_train.xml` file. The two XML report files are the best ones we have chosen among several of them. Their precision rate using machine learning techniques is 92.68% after several bug corrections done. All CVEs are reported making recall 100%. The `stats-*.txt` files are there summarizing the evaluation precision. The results are as good as the training data given; if there are mistakes in the data selection and annotation XML files, then the results will also have mistakes accordingly.
The best reports are:
`report-noprepreprawfftcheb-wireshark-1.2.0-train.xml`
`report-noprepreprawfftdiff-wireshark-1.2.0-train.xml`

4.2.2 Wireshark 1.2.9

The following analysis reports are about Wireshark 1.2.9 using a small subset of MARF's algorithms. The system correctly does *not* report the fixed CVEs (currently, the primary class), so most of the reports come up empty (no noise). All example reports (one per configuration) validate with the schemas `sate_2010.xsd` and `sate_2010.pathcheck.xsd`.
The best (empty) reports are:
`report-noprepreprawfftcheb-wireshark-1.2.9-test.xml`
`report-noprepreprawfftdiff-wireshark-1.2.9-test.xml`
`report-noprepreprawffteucl-wireshark-1.2.9-test.xml`
`report-noprepreprawffthamming-wireshark-1.2.9-test.xml`

The below particular report shows the Minkowski distance classifier (`-mink`) was not perhaps the best choice, as it mistakingly reported a known CVE that was in fact fixed, this is an example of machine learning "red herring":
`report-noprepreprawfftmink-wireshark-1.2.9-test.xml`

4.2.3 Chrome 5.0.375.54

This version's CVE testing result of Chrome 5.0.375.54 (after updates and removal unrelated CVEs per SATE organizers) is in Table 4. The corresponding select reports produced below are about Chrome 5.0.375.54 using a small subset of algorithms. There are line numbers that were machine-learned from the `*_train.xml` file. The two `report-*.xml` files are ones of the best ones we have picked. Their precision rate using machine learning techniques is 90.91% after all the corrections done. The `stats-*.txt` file is there summarizing the evaluation precision in the end of that file. Again, the results are as good as the training data given; if there are mistakes in the data selection and annotation XML files, then the results will also have mistakes accordingly.
The best reports are:
`report-noprepreprawfftcheb-chrome-5.0.375.54-train.xml`
`report-noprepreprawfftdiff-chrome-5.0.375.54-train.xml`
Both validate with both sate2010 schemas.

4.2.4 Chrome 5.0.375.70

The following reports are about Chrome 5.0.375.70 using a small subset of algorithms. The system correctly does *not* report the fixed CVEs, so most of the reports come up empty (no

noise) as they are expected to be for known CVE-selected weaknesses. All example reports (one per configuration) validate with the schema `sate_2010.xsd` and `sate_2010.pathcheck.xsd`. The best (empty) reports are:
`report-noprepreprawfftcheb-chrome-5.0.375.70-test.xml`
`report-noprepreprawfftdiff-chrome-5.0.375.70-test.xml`
`report-noprepreprawffteucl-chrome-5.0.375.70-test.xml`
`report-noprepreprawfftthamming-chrome-5.0.375.70-test.xml`
`report-noprepreprawfftmink-chrome-5.0.375.70-test.xml`

4.3 Version SATE.5

4.3.1 Chrome 5.0.375.54

Here we complete the CVE results from the MARFCAT SATE.5 version by using Chrome 5.0.375.54 training on Chrome 5.0.375.54 with classical CWEs as opposed to CVEs. The result summary is in Table 5.

4.3.2 Tomcat 5.5.13

With this MARFCAT version we did first CVE-based testing on training for Tomcat 5.5.13. Classifiers corresponding to `-cheb` (Chebyshev distance) and `-diff` (Diff distance) continue to dominate as in the other test cases. An observation: for some reason, `-cos` (cosine similarity classifier) with the same settings as for the C/C++ projects (Wireshark and Chrome) actually preforms well and `*_report.xml` is not as noisy; in fact comparable to `-cheb` and `-diff`. These CVE-based results are summarized in Table 6. Further, we perform quick CWE-based testing on Tomcat 5.5.13. Reports are quite larger for `-cheb`, `-diff`, and `-cos`, but not for other classifiers. The precision results are illustrated in Table 7. Then, in SATE.5, quick Tomcat 5.5.13 CVE NLP testing shows higher precision of 87.88%, but the recall is poor, 25/31 – 6 CVEs are missing out (see Table 8). Subsequent, quick Tomcat 5.5.13 CWE NLP testing was surprisingly poor topping at 39.39% (see Table 9). The resulting select reports about this Apache Tomcat 5.5.13 test case using a small subset of algorithms are mentioned below with some commentary.

CVE-based training and reporting: As before, there are line numbers that were machine-learned from the `_train.xml` file as well as the types of locations and descriptions provided by the SATE organizers and incorporated into the reports via machine learning. This includes the types of locations, such as "fix", "sink", or "path" learned from the ogranizers-provided XML/spreadsheet as well as the source code files. Two of all the produced XML reports are the best ones. The macro precision rate in there using machine learning techniques is 83.72%. The `stats-*.txt` files are there summarizing the evaluation precision.
The best reports are:
`report-noprepreprawfftcheb-apache-tomcat-5.5.13-train-cve.xml`
`report-noprepreprawfftdiff-apache-tomcat-5.5.13-train-cve.xml`
(does not validate three tool-specific lines)
Other reports are, to a various degree of detail and noise:
`report-noprepreprawfftcos-apache-tomcat-5.5.13-train-cve.xml`
(does not validate two lines)
`report-noprepreprawffteucl-apache-tomcat-5.5.13-train-cve.xml`
(does not validate three tool-specific lines)
`report-noprepreprawfftthamming-apache-tomcat-5.5.13-train-cve.xml`

`report-noprepreprawfftmink-apache-tomcat-5.5.13-train-cve.xml`
`report-noprepreprepcharunigramadddelta-apache-tomcat-5.5.13-train-cve-nlp.xml`

The `--nlp` version reports use the NLP techniques with the machine learning instead of signal processing techniques. Those reports are largely comparable, but have smaller recall, i.e. some CVEs are completely missing out from the reports in this version. Some reports have problems with tool-specific ranks like: $4.199735736674989E-4$, which we will have to see how to reduce these.

CWE-based training and reporting: The CWE-based reports use the CWE as a primary class instead of CVE for training and reporting, and as such currently do not report on CVEs directly (i.e. no direct mapping from CWE to CVE exists unlike in the opposite direction); however, their recognition rates are not very low either in the same spots, types, etc. In the future version of MARFCAT the plan is to combine the two machine learning pipeline runs of CVE and CWE together to improve mutual classification, but right now it is not available. The CWE-based training is also used on the testing files say of Pebble to see if there are any similar weaknesses to that of Tomcat found, again e.g. in Pebble. CWEs, unlike CVEs for most projects, represent better cross-project classes as they are largely project-independent. Both CVE-based and CWE-base methods use the same data for training. CWEs are recognized correctly 81.82% for Tomcat. NLP-based CWE testing is not included as its precision was quite low ($\approx 39\%$).
The best reports are:
`report-cweidnoprepreprawfftcheb-apache-tomcat-5.5.13-train-cwe.xml`
(does not validate)
`report-cweidnoprepreprawfftdiff-apache-tomcat-5.5.13-train-cwe.xml`
(does not validate)
Other reports are, to a various degree of detail and noise:
`report-cweidnoprepreprawfftcos-apache-tomcat-5.5.13-train-cwe.xml`
`report-cweidnoprepreprawffteucl-apache-tomcat-5.5.13-train-cwe.xml`
(does not validate)
`report-cweidnoprepreprawffthamming-apache-tomcat-5.5.13-train-cwe.xml`
`report-cweidnoprepreprawfftmink-apache-tomcat-5.5.13-train-cwe.xml`

4.3.3 Pebble 2.5-M2

Using the machine learning approach of MARF by using the Tomcat 5.5.13 as a source of training on a Java project with known weaknesses, we used that (rather small) "knowledge base" to test if anything weak similar to the weaknesses in Tomcat are also present in the supplied version of Pebble 2.5-M2. The current result is that under the version of MARFCAT SATE.5 all reports come up empty under the current thresholding rules meaning the tool was not able to identify similar weaknesses in files in Pebble. The corresponding tool-specific log files are also provided if of interest, but the volume of data in them is typically large. It is planned to lower the thresholds after reviewing logs in detail to see if anything interesting comes up that we missed otherwise.

4.3.4 Tomcat and Pebble Testing Results Summary

- Tomcat 5.5.13 on Tomcat 5.5.29 classical CVE testing produced only report with `-cos` with 10 weaknesses, some correspond to the files in training. However, the line numbers reported are midline, so next to meaningless.

- Tomcat 5.5.13 on Tomcat 5.5.29 classical CWE testing also report with `-cos` with 2 weaknesses.
- Tomcat 5.5.13 on Tomcat 5.5.29 NLP CVE testing single report (quick testing only does add-delta, unigram) came up empty.
- Tomcat 5.5.13 on Tomcat 5.5.29 NLP CWE testing, also with a single report (quick testing only does add-delta, unigram) came up empty.
- Tomcat 5.5.13 on Pebble classical CVE reports are empty.
- Tomcat 5.5.13 on Pebble NLP CVE report is not empty, but reports wrongly on `blank.html` (empty HTML file) on multiple CVEs. The probability $P = 0.0$ for all in this case CVEs, not sure why it is at all reported. A red herring.
- Tomcat 5.5.13 on Pebble classical CWE reports are empty.
- Tomcat 5.5.13 on Pebble NLP CWE is similar to the Pebble NLP CVE report on `blank.html` entries, but fewer of them. All the other symptoms are the same.

4.4 Version SATE.6

4.4.1 Dovecot 2.0.beta6

This is a quick test and a report for Dovecot 2.0.beta6, with line numbers and other information. The report is 'raw', without our manual evaluation and generated as-is at this point.
The report of interest:
`report-cweidnoprepreprawfftcos-dovecot-2.0.beta6-wireshark-test-cwe.xml`
It appears though from the first glance most of the are warnings are 'bogus' or 'buggy', but could indicate potential presence of weaknesses in the flagged files. One thing is for sure the Dovecode's source code's main weakness is a near chronic lack of comments, which is also a weakness of a kind. Other reports came up empty. The source for learning was Wireshark 1.2.0.

4.4.2 Tomcat 5.5.29

This is another quick CVE-based evaluation of Tomcat 5.5.29, with line numbers, etc. They are 'raw', without our manual evaluation and generated as-is.
The reports of interest:
`report-noprepreprawfftcos-apache-tomcat-5.5.29-test-cve.xml`
`report-cweidnoprepreprawfftcos-apache-tomcat-5.5.29-test-cwe.xml`
As for the Dovecot case, it appears though from the first glance most of the warnings are either 'bogus' or 'buggy', but could indicate potential presence of weaknesses in the flagged files or fixed as such. Need more manual inspection to be sure. Other XML reports came up empty. The source for learning was Tomcat 5.5.13.

4.5 Version SATE.7

Up until this version NLP processing of Chrome was not successful. Errors related to the number of file descriptors opened and "mark invalid" for NLP processing of Chrome 5.0.375.54 for both CVEs and CWEs have been corrected, so we have produced the results for these cases. CVEs are reported in Table 10. CWEs are further reported in Table 11.

5 Conclusion

We review the current results of this experimental work, its current shortcomings, advantages, and practical implications. We also release MARFCAT Alpha version as open-source that can be found at [14]. This is following the open-source philosophy of greater good (MARF itself has been open-source from the very beginning [23]).

5.1 Shortcomings

The below is a list of most prominent issues with the presented approach. Some of them are more "permanent", while others are solvable and intended to be addressed in the future work. Specifically:

- Looking at a signal is less intuitive visually for code analysis by humans.
- Line numbers are a problem (easily "filtered out" as high-frequency "noise", etc.). A whole "relativistic" and machine learning methodology developed for the line numbers in Section 3.5 to compensate for that. Generally, when CVEs is the primary class, by accurately identifying the CVE number one can get all the other pertinent details from the CVE database, including patches and line numbers.
- Accuracy depends on the quality of the knowledge base (see Section 3.2) collected. "Garbage in – garbage out."
- To detect CVE or CWE signatures in non-CVE cases requires large knowledge bases (human-intensive to collect).
- No path tracing (since no parsing is present); no slicing, semantic annotations, context, locality of reference, etc. The "sink", "path", and "fix" results in the reports also have to be machine-learned.
- A lot of algorithms and their combinations to try (currently ≈ 1800 permutations) to get the best top N. This is, however, also an advantage of the approach as the underlying framework can quickly allow for such testing.
- File-level training vs. fragment-level training – presently the classes are trained based on the entire file where weaknesses are found instead of the known fragments from CVE-reported patches. The latter would be more fine-grained and precise than whole-file classification, but slower. However, overall the file-level processing is a man-hour limitation than a technological one.
- No nice GUI. Presently the application is script/command-line based.

5.2 Advantages

There are some key advantages of the approach presented. Some of them follow:

- Relatively fast (e.g. Wireshark's $\approx$ 2400 files train and test in about 3 minutes) on a now-commodity desktop.
- Language-independent (no parsing) – given enough examples can apply to any language, i.e. methodology is the same no matter C, C++, Java or any other source or binary languages (PHP, C#, VB, Perl, bytecode, assembly, etc.).

- Can automatically learn a large knowledge base to test on known and unknown cases.
- Can be used to quickly pre-scan projects for further analysis by humans and other tools that do in-depth semantic analysis.
- Can learn from other SATE'10 reports.
- Can learn from SATE'09 and SATE'08 reports.
- High precision in CVEs and CWE detection.
- Lots of algorithms and their combinations to select the best for a particular task or class (see Section 3.3).

5.3 Practical Implications

Most practical implications of all static code analyzers are obvious – to detect and report source code weaknesses and report them appropriately to the developers. We outline additional implications this approach brings to the arsenal below:

- The approach can be used on any target language without modifications to the methodology or knowing the syntax of the language. Thus, it scales to any popular and new language analysis with a very small amount of effort.
- The approach can nearly identically be transposed onto the compiled binaries and bytecode, detecting vulnerable deployments and installations – sort of like virus scanning of binaries, but instead scanning for infected binaries, one would scan for security-weak binaries on site deployments to alert system administrators to upgrade their packages.
- Can learn from binary signatures from other tools like Snort [22].

5.4 Future Work

There is a great number of possibilities in the future work. This includes improvements to the code base of MARFCAT as well as resolving unfinished scenarios and results, addressing shortcomings in Section 5.1, testing more algorithms and combinations from the related work, and moving onto other programming languages (e.g. PHP, ASP, C#). Furthermore, plan to conceive collaboration with vendors such as VeraCode, Coverity, and others who have vast data sets to test the full potential of the approach with the others and a community as a whole. Then move on to dynamic code analysis as well applying similar techniques there.

References

[1] Masashi Eto, Kotaro Sonoda, Daisuke Inoue, Katsunari Yoshioka, and Koji Nakao. A proposal of malware distinction method based on scan patterns using spectrum analysis. In *Proceedings of the 16th International Conference on Neural Information Processing: Part II*, ICONIP'09, pages 565–572, Berlin, Heidelberg, 2009. Springer-Verlag.

[2] Free Software Foundation, Inc. `wc` – print newline, word, and byte counts for each file. GNU coreutils 6.10, 2009. man 1 wc.

[3] Aiman Hanna, Hai Zhou Ling, Xiaochun Yang, and Mourad Debbabi. A synergy between static and dynamic analysis for the detection of software security vulnerabilities. In Robert Meersman, Tharam S. Dillon, and Pilar Herrero, editors, *OTM Conferences (2)*, volume 5871 of *Lecture Notes in Computer Science*, pages 815–832. Springer, 2009.

[4] Daisuke Inoue, Katsunari Yoshioka, Masashi Eto, Masaya Yamagata, Eisuke Nishino, Jun'ichi Takeuchi, Kazuya Ohkouchi, and Koji Nakao. An incident analysis system NICTER and its analysis engines based on data mining techniques. In *Proceedings of the 15th International Conference on Advances in Neuro-Information Processing – Volume Part I*, ICONIP'08, pages 579–586, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] Ying Kong, Yuqing Zhang, and Qixu Liu. Eliminating human specification in static analysis. In *Proceedings of the 13th international conference on Recent advances in intrusion detection*, RAID'10, pages 494–495, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *Foundations of Software Engineering (FSE)*, 2004.

[7] Ted Kremenek and Dawson Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *SAS 2003*, 2003.

[8] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th Symposium on Operating System Design and Implementation*, 2006.

[9] D. Mackenzie, P. Eggert, and R. Stallman. Comparing and merging files. [online], 2002. `http://www.gnu.org/software/diffutils/manual/ps/diff.ps.gz`.

[10] Serguei A. Mokhov. Introducing MARF: a modular audio recognition framework and its applications for scientific and software engineering research. In *Advances in Computer and Information Sciences and Engineering*, pages 473–478, University of Bridgeport, U.S.A., December 2007. Springer Netherlands. Proceedings of CISSE/SCSS'07.

[11] Serguei A. Mokhov. Study of best algorithm combinations for speech processing tasks in machine learning using median vs. mean clusters in MARF. In Bipin C. Desai, editor, *Proceedings of C3S2E'08*, pages 29–43, Montreal, Quebec, Canada, May 2008. ACM.

[12] Serguei A. Mokhov. Complete complimentary results report of the MARF's NLP approach to the DEFT 2010 competition. [online], June 2010. `http://arxiv.org/abs/1006.3787`.

[13] Serguei A. Mokhov. L'approche MARF à DEFT 2010: A MARF approach to DEFT 2010. In *Proceedings of TALN'10*, July 2010. To appear in DEFT 2010 System competition at TALN 2010.

[14] Serguei A. Mokhov. MARFCAT – MARF-based Code Analysis Tool. Published electronically within the MARF project, `http://sourceforge.net/projects/marf/files/Applications/MARFCAT/`, 2010–2011. Last viewed February 2011.

[15] Serguei A. Mokhov and Mourad Debbabi. File type analysis using signal processing techniques and machine learning vs. `file` unix utility for forensic analysis. In Oliver Goebel, Sandra Frings, Detlef Guenther, Jens Nedon, and Dirk Schadt, editors, *Proceedings of the IT Incident Management and IT Forensics (IMF'08)*, LNI140, pages 73–85. GI, September 2008.

[16] Serguei A. Mokhov, Marc-André Laverdière, and Djamel Benredjem. Taxonomy of linux kernel vulnerability solutions. In *Innovative Techniques in Instruction Technology, E-learning, E-assessment, and Education*, pages 485–493, University of Bridgeport, U.S.A., 2007. Proceedings of CISSE/SCSS'07.

[17] Serguei A. Mokhov, Miao Song, and Ching Y. Suen. Writer identification using inexpensive signal processing techniques. In Tarek Sobh and Khaled Elleithy, editors, *Innovations in Computing Sciences and Software Engineering; Proceedings of CISSE'09*, pages 437–441. Springer, December 2009. ISBN: 978-90-481-9111-6, online at: `http://arxiv.org/abs/0912.5502`.

[18] Vinod P. Nair, Harshit Jain, Yashwant K. Golecha, Manoj Singh Gaur, and Vijay Laxmi. MEDUSA: MEtamorphic malware dynamic analysis using signature from API. In *Proceedings of the 3rd International Conference on Security of Information and Networks*, SIN'10, pages 263–269, New York, NY, USA, 2010. ACM.

[19] NIST. National Vulnerability Database. [online], 2005–2011. http://nvd.nist.gov/.

[20] NIST. National Vulnerability Database statistics. [online], 2005–2011. http://web.nvd.nist.gov/view/vuln/statistics.

[21] Vadim Okun, Aurelien Delaitre, Paul E. Black, and NIST SAMATE. Static Analysis Tool Exposition (SATE) 2010. [online], 2010. See http://samate.nist.gov/SATE.html and http://samate.nist.gov/SATE2010Workshop.html.

[22] Sourcefire. Snort: Open-source network intrusion prevention and detection system (IDS/IPS). [online], 2010. http://www.snort.org/.

[23] The MARF Research and Development Group. The Modular Audio Recognition Framework and its Applications. [online], 2002–2011. http://marf.sf.net and http://arxiv.org/abs/0905.1235, last viewed April 2010.

[24] Syrine Tlili. *Automatic detection of safety and security vulnerabilities in open source software*. PhD thesis, Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Canada, 2009. ISBN: 9780494634165.

[25] Various contributors and MITRE. Common Weakness Enumeration (CWE) – a community-developed dictionary of software weakness types. [online], 2010. See http://cwe.mitre.org.

A Classification Result Tables

What follows are result tables with top classification results ranked from most precise at the top. This include the configuration settings for MARF by the means of options (the algorithm implementations are at their defaults [10]).

Table 1: CVE Stats for Wireshark 1.2.0, Quick Enriched, version SATE.4

guess	run	algorithms	good	bad	%
1st	1	`-nopreprep -raw -fft -diff`	38	3	92.68
1st	2	`-nopreprep -raw -fft -cheb`	38	3	92.68
1st	3	`-nopreprep -raw -fft -eucl`	29	12	70.73
1st	4	`-nopreprep -raw -fft -hamming`	26	15	63.41
1st	5	`-nopreprep -raw -fft -mink`	23	18	56.10
1st	6	`-nopreprep -raw -fft -cos`	37	51	42.05
2nd	1	`-nopreprep -raw -fft -diff`	39	2	95.12
2nd	2	`-nopreprep -raw -fft -cheb`	39	2	95.12
2nd	3	`-nopreprep -raw -fft -eucl`	34	7	82.93
2nd	4	`-nopreprep -raw -fft -hamming`	28	13	68.29
2nd	5	`-nopreprep -raw -fft -mink`	31	10	75.61
2nd	6	`-nopreprep -raw -fft -cos`	38	50	43.18
guess	run	class	good	bad	%
1st	1	CVE-2009-3829	6	0	100.00
1st	2	CVE-2009-2563	6	0	100.00
1st	3	CVE-2009-2562	6	0	100.00
1st	4	CVE-2009-4378	6	0	100.00
1st	5	CVE-2009-4376	6	0	100.00
1st	6	CVE-2010-0304	6	0	100.00
1st	7	CVE-2010-2286	6	0	100.00
1st	8	CVE-2010-2283	6	0	100.00
1st	9	CVE-2009-3551	6	0	100.00
1st	10	CVE-2009-3550	6	0	100.00
1st	11	CVE-2009-3549	6	0	100.00
1st	12	CVE-2009-3241	16	8	66.67
1st	13	CVE-2010-1455	34	20	62.96
1st	14	CVE-2009-3243	18	11	62.07
1st	15	CVE-2009-2560	8	6	57.14
1st	16	CVE-2009-2561	6	5	54.55
1st	17	CVE-2010-2285	6	5	54.55
1st	18	CVE-2009-2559	6	5	54.55
1st	19	CVE-2010-2287	6	6	50.00
1st	20	CVE-2009-4377	12	15	44.44
1st	21	CVE-2010-2284	6	9	40.00
1st	22	CVE-2009-3242	7	12	36.84
2nd	1	CVE-2009-3829	6	0	100.00
2nd	2	CVE-2009-2563	6	0	100.00
2nd	3	CVE-2009-2562	6	0	100.00
2nd	4	CVE-2009-4378	6	0	100.00
2nd	5	CVE-2009-4376	6	0	100.00
2nd	6	CVE-2010-0304	6	0	100.00
2nd	7	CVE-2010-2286	6	0	100.00
2nd	8	CVE-2010-2283	6	0	100.00
2nd	9	CVE-2009-3551	6	0	100.00
2nd	10	CVE-2009-3550	6	0	100.00
2nd	11	CVE-2009-3549	6	0	100.00
2nd	12	CVE-2009-3241	17	7	70.83
2nd	13	CVE-2010-1455	44	10	81.48
2nd	14	CVE-2009-3243	18	11	62.07
2nd	15	CVE-2009-2560	9	5	64.29
2nd	16	CVE-2009-2561	6	5	54.55
2nd	17	CVE-2010-2285	6	5	54.55
2nd	18	CVE-2009-2559	6	5	54.55
2nd	19	CVE-2010-2287	12	0	100.00
2nd	20	CVE-2009-4377	12	15	44.44
2nd	21	CVE-2010-2284	6	9	40.00
2nd	22	CVE-2009-3242	7	12	36.84

Table 2: CVE NLP Stats for Wireshark 1.2.0, Quick Enriched, version SATE.4

guess	run	algorithms	good	bad	%
1st	1	`-nopreprep -char -unigram` -add-delta	30	6	83.33
2nd	1	`-nopreprep -char -unigram` -add-delta	31	5	86.11
guess	run	class	good	bad	%
1st	1	CVE-2009-3829	1	0	100.00
1st	2	CVE-2009-2563	1	0	100.00
1st	3	CVE-2009-2562	1	0	100.00
1st	4	CVE-2009-4378	1	0	100.00
1st	5	CVE-2009-2561	1	0	100.00
1st	6	CVE-2009-4377	1	0	100.00
1st	7	CVE-2009-4376	1	0	100.00
1st	8	CVE-2010-2286	1	0	100.00
1st	9	CVE-2010-0304	1	0	100.00
1st	10	CVE-2010-2285	1	0	100.00
1st	11	CVE-2010-2284	1	0	100.00
1st	12	CVE-2010-2283	1	0	100.00
1st	13	CVE-2009-2559	1	0	100.00
1st	14	CVE-2009-3550	1	0	100.00
1st	15	CVE-2009-3549	1	0	100.00
1st	16	CVE-2010-1455	8	1	88.89
1st	17	CVE-2009-3243	3	1	75.00
1st	18	CVE-2009-3241	2	2	50.00
1st	19	CVE-2009-2560	1	1	50.00
1st	20	CVE-2009-3242	1	1	50.00
2nd	1	CVE-2009-3829	1	0	100.00
2nd	2	CVE-2009-2563	1	0	100.00
2nd	3	CVE-2009-2562	1	0	100.00
2nd	4	CVE-2009-4378	1	0	100.00
2nd	5	CVE-2009-2561	1	0	100.00
2nd	6	CVE-2009-4377	1	0	100.00
2nd	7	CVE-2009-4376	1	0	100.00
2nd	8	CVE-2010-2286	1	0	100.00
2nd	9	CVE-2010-0304	1	0	100.00
2nd	10	CVE-2010-2285	1	0	100.00
2nd	11	CVE-2010-2284	1	0	100.00
2nd	12	CVE-2010-2283	1	0	100.00
2nd	13	CVE-2009-2559	1	0	100.00
2nd	14	CVE-2009-3550	1	0	100.00
2nd	15	CVE-2009-3549	1	0	100.00
2nd	16	CVE-2010-1455	8	1	88.89
2nd	17	CVE-2009-3243	3	1	75.00
2nd	18	CVE-2009-3241	3	1	75.00
2nd	19	CVE-2009-2560	1	1	50.00
2nd	20	CVE-2009-3242	1	1	50.00

Table 3: CVE NLP Stats for Wireshark 1.2.0, Quick Enriched, version SATE.4

guess	run	algorithms	good	bad	%
1st	1	`-cweid -nopreprep -raw -fft -cheb`	31	5	86.11
1st	2	`-cweid -nopreprep -raw -fft -diff`	31	5	86.11
1st	3	`-cweid -nopreprep -raw -fft -eucl`	29	7	80.56
1st	4	`-cweid -nopreprep -raw -fft -hamming`	22	14	61.11
1st	5	`-cweid -nopreprep -raw -fft -cos`	33	25	56.90
1st	6	`-cweid -nopreprep -raw -fft -mink`	20	16	55.56
2nd	1	`-cweid -nopreprep -raw -fft -cheb`	33	3	91.67
2nd	2	`-cweid -nopreprep -raw -fft -diff`	33	3	91.67
2nd	3	`-cweid -nopreprep -raw -fft -eucl`	33	3	91.67
2nd	4	`-cweid -nopreprep -raw -fft -hamming`	27	9	75.00
2nd	5	`-cweid -nopreprep -raw -fft -cos`	41	17	70.69
2nd	6	`-cweid -nopreprep -raw -fft -mink`	22	14	61.11
guess	run	class	good	bad	%
1st	1	CWE-399	6	0	100.00
1st	2	NVD-CWE-Other	17	3	85.00
1st	3	CWE-20	50	10	83.33
1st	4	CWE-189	8	2	80.00
1st	5	NVD-CWE-noinfo	72	40	64.29
1st	6	CWE-119	13	17	43.33
2nd	1	CWE-399	6	0	100.00
2nd	2	NVD-CWE-Other	17	3	85.00
2nd	3	CWE-20	52	8	86.67
2nd	4	CWE-189	8	2	80.00
2nd	5	NVD-CWE-noinfo	83	29	74.11
2nd	6	CWE-119	23	7	76.67

Table 4: CVE Stats for Chrome 5.0.375.54, Quick Enriched, (clean CVEs) version SATE.4

guess	run	algorithms	good	bad	%
1st	1	`-nopreprep -raw -fft -eucl`	10	1	90.91
1st	2	`-nopreprep -raw -fft -cos`	10	1	90.91
1st	3	`-nopreprep -raw -fft -diff`	10	1	90.91
1st	4	`-nopreprep -raw -fft -cheb`	10	1	90.91
1st	5	`-nopreprep -raw -fft -mink`	9	2	81.82
1st	6	`-nopreprep -raw -fft -hamming`	9	2	81.82
2nd	1	`-nopreprep -raw -fft -eucl`	11	0	100.00
2nd	2	`-nopreprep -raw -fft -cos`	11	0	100.00
2nd	3	`-nopreprep -raw -fft -diff`	11	0	100.00
2nd	4	`-nopreprep -raw -fft -cheb`	11	0	100.00
2nd	5	`-nopreprep -raw -fft -mink`	10	1	90.91
2nd	6	`-nopreprep -raw -fft -hamming`	10	1	90.91
guess	run	class	good	bad	%
1st	1	CVE-2010-2301	6	0	100.00
1st	2	CVE-2010-2300	6	0	100.00
1st	3	CVE-2010-2299	6	0	100.00
1st	4	CVE-2010-2298	6	0	100.00
1st	5	CVE-2010-2297	6	0	100.00
1st	6	CVE-2010-2304	6	0	100.00
1st	7	CVE-2010-2303	6	0	100.00
1st	8	CVE-2010-2295	10	2	83.33
1st	9	CVE-2010-2302	6	6	50.00
2nd	1	CVE-2010-2301	6	0	100.00
2nd	2	CVE-2010-2300	6	0	100.00
2nd	3	CVE-2010-2299	6	0	100.00
2nd	4	CVE-2010-2298	6	0	100.00
2nd	5	CVE-2010-2297	6	0	100.00
2nd	6	CVE-2010-2304	6	0	100.00
2nd	7	CVE-2010-2303	6	0	100.00
2nd	8	CVE-2010-2295	10	2	83.33
2nd	9	CVE-2010-2302	12	0	100.00

Table 5: CWE Stats for Chrome 5.0.375.54, (clean CVEs) version SATE.5

guess	run	algorithms	good	bad	%
1st	1	`-cweid -nopreprep -raw -fft -cheb`	9	0	100.00
1st	2	`-cweid -nopreprep -raw -fft -cos`	9	0	100.00
1st	3	`-cweid -nopreprep -raw -fft -diff`	9	0	100.00
1st	4	`-cweid -nopreprep -raw -fft -eucl`	8	1	88.89
1st	5	`-cweid -nopreprep -raw -fft -hamming`	8	1	88.89
1st	6	`-cweid -nopreprep -raw -fft -mink`	6	3	66.67
2nd	1	`-cweid -nopreprep -raw -fft -cheb`	9	0	100.00
2nd	2	`-cweid -nopreprep -raw -fft -cos`	9	0	100.00
2nd	3	`-cweid -nopreprep -raw -fft -diff`	9	0	100.00
2nd	4	`-cweid -nopreprep -raw -fft -eucl`	8	1	88.89
2nd	5	`-cweid -nopreprep -raw -fft -hamming`	8	1	88.89
2nd	6	`-cweid -nopreprep -raw -fft -mink`	8	1	88.89
guess	run	class	good	bad	%
1st	1	CWE-79	6	0	100.00
1st	2	NVD-CWE-noinfo	6	0	100.00
1st	3	CWE-399	6	0	100.00
1st	4	CWE-119	6	0	100.00
1st	5	CWE-20	6	0	100.00
1st	6	NVD-CWE-Other	10	2	83.33
1st	7	CWE-94	9	3	75.00
2nd	1	CWE-79	6	0	100.00
2nd	2	NVD-CWE-noinfo	6	0	100.00
2nd	3	CWE-399	6	0	100.00
2nd	4	CWE-119	6	0	100.00
2nd	5	CWE-20	6	0	100.00
2nd	6	NVD-CWE-Other	11	1	91.67
2nd	7	CWE-94	10	2	83.33

Table 6: CVE Stats for Tomcat 5.5.13, version SATE.5

1st	1	-nopreprep -raw -fft -diff	36	7	83.72
1st	2	-nopreprep -raw -fft -cheb	36	7	83.72
1st	3	-nopreprep -raw -fft -cos	37	9	80.43
1st	4	-nopreprep -raw -fft -eucl	34	9	79.07
1st	5	-nopreprep -raw -fft -mink	28	15	65.12
1st	6	-nopreprep -raw -fft -hamming	26	17	60.47
2nd	1	-nopreprep -raw -fft -diff	40	3	93.02
2nd	2	-nopreprep -raw -fft -cheb	40	3	93.02
2nd	3	-nopreprep -raw -fft -cos	40	6	86.96
2nd	4	-nopreprep -raw -fft -eucl	36	7	83.72
2nd	5	-nopreprep -raw -fft -mink	31	12	72.09
2nd	6	-nopreprep -raw -fft -hamming	29	14	67.44
guess	run	algorithms	good	bad	%
1st	1	CVE-2006-7197	6	0	100.00
1st	2	CVE-2006-7196	6	0	100.00
1st	3	CVE-2006-7195	6	0	100.00
1st	4	CVE-2009-0033	6	0	100.00
1st	5	CVE-2007-3386	6	0	100.00
1st	6	CVE-2009-2901	3	0	100.00
1st	7	CVE-2007-3385	6	0	100.00
1st	8	CVE-2008-2938	6	0	100.00
1st	9	CVE-2007-3382	6	0	100.00
1st	10	CVE-2007-5461	6	0	100.00
1st	11	CVE-2007-6286	6	0	100.00
1st	12	CVE-2007-1858	6	0	100.00
1st	13	CVE-2008-0128	6	0	100.00
1st	14	CVE-2007-2450	6	0	100.00
1st	15	CVE-2009-3548	6	0	100.00
1st	16	CVE-2009-0580	6	0	100.00
1st	17	CVE-2007-1355	6	0	100.00
1st	18	CVE-2008-2370	6	0	100.00
1st	19	CVE-2008-4308	6	0	100.00
1st	20	CVE-2007-5342	6	0	100.00
1st	21	CVE-2008-5515	19	5	79.17
1st	22	CVE-2009-0783	11	4	73.33
1st	23	CVE-2008-1232	13	5	72.22
1st	24	CVE-2008-5519	6	6	50.00
1st	25	CVE-2007-5333	6	6	50.00
1st	26	CVE-2008-1947	6	6	50.00
1st	27	CVE-2009-0781	6	6	50.00
1st	28	CVE-2007-0450	5	7	41.67
1st	29	CVE-2007-2449	6	12	33.33
1st	30	CVE-2009-2693	2	6	25.00
1st	31	CVE-2009-2902	0	1	0.00
2nd	1	CVE-2006-7197	6	0	100.00
2nd	2	CVE-2006-7196	6	0	100.00
2nd	3	CVE-2006-7195	6	0	100.00
2nd	4	CVE-2009-0033	6	0	100.00
2nd	5	CVE-2007-3386	6	0	100.00
2nd	6	CVE-2009-2901	3	0	100.00
2nd	7	CVE-2007-3385	6	0	100.00
2nd	8	CVE-2008-2938	6	0	100.00
2nd	9	CVE-2007-3382	6	0	100.00
2nd	10	CVE-2007-5461	6	0	100.00
2nd	11	CVE-2007-6286	6	0	100.00
2nd	12	CVE-2007-1858	6	0	100.00
2nd	13	CVE-2008-0128	6	0	100.00
2nd	14	CVE-2007-2450	6	0	100.00
2nd	15	CVE-2009-3548	6	0	100.00
2nd	16	CVE-2009-0580	6	0	100.00
2nd	17	CVE-2007-1355	6	0	100.00
2nd	18	CVE-2008-2370	6	0	100.00
2nd	19	CVE-2008-4308	6	0	100.00
2nd	20	CVE-2007-5342	6	0	100.00
2nd	21	CVE-2008-5515	19	5	79.17
2nd	22	CVE-2009-0783	12	3	80.00
2nd	23	CVE-2008-1232	13	5	72.22
2nd	24	CVE-2008-5519	12	0	100.00
2nd	25	CVE-2007-5333	6	6	50.00
2nd	26	CVE-2008-1947	6	6	50.00
2nd	27	CVE-2009-0781	12	0	100.00
2nd	28	CVE-2007-0450	7	5	58.33
2nd	29	CVE-2007-2449	8	10	44.44
2nd	30	CVE-2009-2693	4	4	50.00
2nd	31	CVE-2009-2902	0	1	0.00

Table 7: CWE Stats for Tomcat 5.5.13, version SATE.5

guess	run	algorithms	good	bad	%
1st	1	`-cweid -nopreprep -raw -fft -cheb`	27	6	81.82
1st	2	`-cweid -nopreprep -raw -fft -diff`	27	6	81.82
1st	3	`-cweid -nopreprep -raw -fft -cos`	24	9	72.73
1st	4	`-cweid -nopreprep -raw -fft -eucl`	13	20	39.39
1st	5	`-cweid -nopreprep -raw -fft -hamming`	12	21	36.36
1st	6	`-cweid -nopreprep -raw -fft -mink`	9	24	27.27
2nd	1	`-cweid -nopreprep -raw -fft -cheb`	32	1	96.97
2nd	2	`-cweid -nopreprep -raw -fft -diff`	32	1	96.97
2nd	3	`-cweid -nopreprep -raw -fft -cos`	29	4	87.88
2nd	4	`-cweid -nopreprep -raw -fft -eucl`	17	16	51.52
2nd	5	`-cweid -nopreprep -raw -fft -hamming`	18	15	54.55
2nd	6	`-cweid -nopreprep -raw -fft -mink`	13	20	39.39
guess	run	class	good	bad	%
1st	1	CWE-264	7	0	100.00
1st	2	CWE-255	6	0	100.00
1st	3	CWE-16	6	0	100.00
1st	4	CWE-119	6	0	100.00
1st	5	CWE-20	6	0	100.00
1st	6	CWE-200	22	4	84.62
1st	7	CWE-79	24	21	53.33
1st	8	CWE-22	35	61	36.46
2nd	1	CWE-264	7	0	100.00
2nd	2	CWE-255	6	0	100.00
2nd	3	CWE-16	6	0	100.00
2nd	4	CWE-119	6	0	100.00
2nd	5	CWE-20	6	0	100.00
2nd	6	CWE-200	23	3	88.46
2nd	7	CWE-79	30	15	66.67
2nd	8	CWE-22	57	39	59.38

Table 8: CVE NLP Stats for Tomcat 5.5.13, version SATE.5

guess	run	algorithms	good	bad	%
1st	1	`-nopreprep -char -unigram` -add-delta	29	4	87.88
2nd	1	`-nopreprep -char -unigram` -add-delta	29	4	87.88
guess	run	class	good	bad	%
1st	1	CVE-2006-7197	1	0	100.00
1st	2	CVE-2006-7196	1	0	100.00
1st	3	CVE-2009-2901	1	0	100.00
1st	4	CVE-2006-7195	1	0	100.00
1st	5	CVE-2009-0033	1	0	100.00
1st	6	CVE-2007-1355	1	0	100.00
1st	7	CVE-2007-5342	1	0	100.00
1st	8	CVE-2009-2693	1	0	100.00
1st	9	CVE-2009-0783	1	0	100.00
1st	10	CVE-2008-2370	1	0	100.00
1st	11	CVE-2007-2450	1	0	100.00
1st	12	CVE-2008-2938	1	0	100.00
1st	13	CVE-2007-2449	3	0	100.00
1st	14	CVE-2007-1858	1	0	100.00
1st	15	CVE-2008-4308	1	0	100.00
1st	16	CVE-2008-0128	1	0	100.00
1st	17	CVE-2009-3548	1	0	100.00
1st	18	CVE-2007-5461	1	0	100.00
1st	19	CVE-2007-3382	1	0	100.00
1st	20	CVE-2007-0450	2	0	100.00
1st	21	CVE-2009-0580	1	0	100.00
1st	22	CVE-2007-6286	1	0	100.00
1st	23	CVE-2008-5515	3	1	75.00
1st	24	CVE-2008-1232	1	2	33.33
1st	25	CVE-2009-2902	0	1	0.00
2nd	1	CVE-2006-7197	1	0	100.00
2nd	2	CVE-2006-7196	1	0	100.00
2nd	3	CVE-2009-2901	1	0	100.00
2nd	4	CVE-2006-7195	1	0	100.00
2nd	5	CVE-2009-0033	1	0	100.00
2nd	6	CVE-2007-1355	1	0	100.00
2nd	7	CVE-2007-5342	1	0	100.00
2nd	8	CVE-2009-2693	1	0	100.00
2nd	9	CVE-2009-0783	1	0	100.00
2nd	10	CVE-2008-2370	1	0	100.00
2nd	11	CVE-2007-2450	1	0	100.00
2nd	12	CVE-2008-2938	1	0	100.00
2nd	13	CVE-2007-2449	3	0	100.00
2nd	14	CVE-2007-1858	1	0	100.00
2nd	15	CVE-2008-4308	1	0	100.00
2nd	16	CVE-2008-0128	1	0	100.00
2nd	17	CVE-2009-3548	1	0	100.00
2nd	18	CVE-2007-5461	1	0	100.00
2nd	19	CVE-2007-3382	1	0	100.00
2nd	20	CVE-2007-0450	2	0	100.00
2nd	21	CVE-2009-0580	1	0	100.00
2nd	22	CVE-2007-6286	1	0	100.00
2nd	23	CVE-2008-5515	3	1	75.00
2nd	24	CVE-2008-1232	1	2	33.33
2nd	25	CVE-2009-2902	0	1	0.00

Table 9: CWE NLP Stats for Tomcat 5.5.13, version SATE.5

guess	run	algorithms	good	bad	%
1st	1	-cweid -nopreprep -char -unigram -add-delta	13	20	39.39
2nd	1	-cweid -nopreprep -char -unigram -add-delta	17	16	51.52
guess	run	class	good	bad	%
1st	1	CWE-16	1	0	100.00
1st	2	CWE-255	1	0	100.00
1st	3	CWE-264	2	0	100.00
1st	4	CWE-119	1	0	100.00
1st	5	CWE-20	1	0	100.00
1st	6	CWE-200	3	1	75.00
1st	7	CWE-22	3	13	18.75
1st	8	CWE-79	1	6	14.29
2nd	1	CWE-16	1	0	100.00
2nd	2	CWE-255	1	0	100.00
2nd	3	CWE-264	2	0	100.00
2nd	4	CWE-119	1	0	100.00
2nd	5	CWE-20	1	0	100.00
2nd	6	CWE-200	4	0	100.00
2nd	7	CWE-22	5	11	31.25
2nd	8	CWE-79	2	5	28.57

Table 10: CVE NLP Stats for Chrome 5.0.375.54, version SATE.7

guess	run	algorithms	good	bad	%
1st	1	-nopreprep -char -unigram -add-delta	9	0	100.00
2nd	1	-nopreprep -char -unigram -add-delta	9	0	100.00
guess	run	class	good	bad	%
1st	1	CVE-2010-2304	1	0	100.00
1st	2	CVE-2010-2298	1	0	100.00
1st	3	CVE-2010-2301	1	0	100.00
1st	4	CVE-2010-2295	2	0	100.00
1st	5	CVE-2010-2300	1	0	100.00
1st	6	CVE-2010-2303	1	0	100.00
1st	7	CVE-2010-2297	1	0	100.00
1st	8	CVE-2010-2299	1	0	100.00
2nd	1	CVE-2010-2304	1	0	100.00
2nd	2	CVE-2010-2298	1	0	100.00
2nd	3	CVE-2010-2301	1	0	100.00
2nd	4	CVE-2010-2295	2	0	100.00
2nd	5	CVE-2010-2300	1	0	100.00
2nd	6	CVE-2010-2303	1	0	100.00
2nd	7	CVE-2010-2297	1	0	100.00
2nd	8	CVE-2010-2299	1	0	100.00

Table 11: CWE NLP Stats for Chrome 5.0.375.54, version SATE.7

guess	run	algorithms	good	bad	%
1st	1	-cweid -nopreprep -char -unigram -add-delta	8	1	88.89
2nd	1	-cweid -nopreprep -char -unigram -add-delta	8	1	88.89
guess	run	class	good	bad	%
1st	1	CWE-399	1	0	100.00
1st	2	NVD-CWE-noinfo	1	0	100.00
1st	3	CWE-79	1	0	100.00
1st	4	NVD-CWE-Other	2	0	100.00
1st	5	CWE-119	1	0	100.00
1st	6	CWE-20	1	0	100.00
1st	7	CWE-94	1	1	50.00
2nd	1	CWE-399	1	0	100.00
2nd	2	NVD-CWE-noinfo	1	0	100.00
2nd	3	CWE-79	1	0	100.00
2nd	4	NVD-CWE-Other	2	0	100.00
2nd	5	CWE-119	1	0	100.00
2nd	6	CWE-20	1	0	100.00
2nd	7	CWE-94	1	1	50.00