
Katerina Goseva-Popstajanova & Andrei Perhinschi
Lane Department of Computer Science and Electrical Engineering
West Virginia University
Morgantown, WV
Acknowledgements

This material is based upon work supported in part by NASA IV&V, Fairmont, WV

We thank Keenan Bowens, Travis Dawson, Roger Harris, Joelle Loretta, Jerry Sims and Christopher Williams for their valuable input and feedback.
- NASA develops, runs, and maintains many systems for which one or more security attributes (i.e. confidentiality, integrity, availability, authentication, authorization, and non-repudiation) are of vital importance

- Information assurance and cyber security have to be integrated in the traditional verification and validation process
Static code analysis

- Static analysis of source code provides a scalable method for code review
- Tools matured rapidly in the last decade
  - from simple lexical analysis to more complex and accurate techniques
- In general, static analysis problems are undecidable (i.e. it is impossible to construct an algorithm which always leads to a correct answer)
  - False negatives
  - False positives
To examine the ability of static code analysis tools to detect security vulnerabilities
Approach

- Surveyed the literature and vendor provided information on the state-of-the-art and practice of static code analysis tools
  - 15 commercial products
  - 8 tools licensed under some kind of open source license

- Selected three tools for detailed evaluation
  - To fully use the provided functionality all three tools require a build to be created or at least the software under test to be compiled

- Performance was evaluated using
  - Micro-benchmarking test suites for C/C++ and Java
  - Three open source programs with known vulnerabilities
EVALUATION BASED ON THE JULIET TEST SUITE
Juliet test suite

- Micro-benchmarking suite which covers large number of CWEs
  - Each CWE (Common Weakness Enumeration) represents a single vulnerability type
- Created by NSA and made publicly available at the NIST Web site
- C/C++ suite (version 1.1)
  - 119 CWEs
  - 57,099 test cases
- Java suite (version 1.1.1)
  - 113 CWEs
  - 23,957 test cases
Juliet test suite

- This presentation is focused on the CWEs covered by all three tools
  - 22 common C/C++ CWEs among the three tools (~21,000 test cases)
  - 19 common Java CWEs among the three tools (~7,500 test cases)
- Two of the tools covered significantly more CWEs
  - 90 C/C++ CWEs (~34,000 test cases)
  - 107 Java CWEs (~16,000 test cases)
  - Results were similar to the ones presented here
Automatic assessment

- Run each tool on the Juliet test suite
- Transform tool’s output in a common format
- Parse each CWE directory & assemble a list of test cases
- Parse the output & compute the confusion matrix
Confusion matrix & metrics

<table>
<thead>
<tr>
<th>Actual vulnerability</th>
<th>Reported vulnerability</th>
<th>No warning/error reported</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Positives (TP)</td>
<td>False Negatives (FN)</td>
<td></td>
</tr>
<tr>
<td>No vulnerability (good function/method)</td>
<td>False Positives (FP)</td>
<td>True Negatives (TN)</td>
</tr>
</tbody>
</table>

% of functions that are classified correctly  

\[ \text{Accuracy} = \frac{TN + TP}{TN + FN + FP + TP} \]

Probability of detecting a vulnerability (recall)  

\[ PD = \frac{TP}{TP + FN} \]

Probability of misclassifying a good function as a bad function (false alarm)  

\[ PF = \frac{FP}{TN + FP} \]

How close is the result to the ideal point \((pf, pd)=(0,1)\)  

\[ \text{Balance} = 1 - \frac{\sqrt{(0 - PF)^2 + (1 - PD)^2}}{\sqrt{2}} \]
The three tools have similar performance with Tool C performing slightly better.

Accuracy: C/C++ CWEs

**Tool A:** Range [0.27, 0.77], Average = 0.59, Median = 0.63  
**Tool B:** Range [0.50, 0.87], Average = 0.67, Median = 0.64  
**Tool C:** Range [0.41, 1], Average = 0.72, Median = 0.64
Recall: C/C++ CWEs

Each tool has 0% recall for some CWEs

For some CWEs (i.e., 197, 391, 478, 480, 482, 835) all three tools have 0% recall

Accuracy on its own is not a good metric for tools’ performance

Tool A: Range [0, 1], Average = 0.21, Median = 0.14
Tool B: Range [0, 0.87], Average = 0.26, Median = 0.10
Tool C: Range [0,1], Average = 0.39, Median = 0.42
Probability of false alarm: C/C++ CWEs

Tool C has noticeably lower false positive rate than Tools A and B.

Tool A: Range [0, 0.94], Average = 0.18, Median = 0.02
Tool B: Range [0, 0.52], Average = 0.09, Median = 0.01
Tool C: Range [0,0.94], Average = 0.07, Median = 0
Balance values for many CWEs were around 30%, which indicates poor overall performance.

Tool C performed slightly better than the other two tools.

Tool A: Range [0.28, 0.65], Average = 0.39, Median = 0.29
Tool B: Range [0.29, 0.87], Average = 0.46, Median = 0.36
Tool C: Range [0.29,1], Average = 0.53, Median = 0.46
ROC squares for C/C++ CWEs

Ideal result
\((pf, pd) = (0, 1)\)

Not many points are close to the ideal \((0,1)\) point
Tool C has noticeably lower false alarm rate
For each tool there are multiple CWEs at the \((0,0)\) point
Accuracy: Java CWEs

Accuracy values for Java CWEs vary somewhat more than those for C/C++ CWEs

All three tools attain a maximum accuracy value for several CWEs

Tool C seems to be performing slightly better than the other two tools

Tool A: Range [0.41,1], Average = 0.67, Median = 0.63
Tool B: Range [0,1], Average = 0.60, Median = 0.63
Tool C: Range [0.52,1], Average = 0.73, Median = 0.67
Recall: Java CWEs

Again, there were CWEs (i.e., 486 and 489) for which none of the tools correctly flagged any flawed constructs.

However, not as many as in case of C/C++ test suite.

Tool A performed slightly better than the other two tools.

- Tool A: Range [0,1], Average = 0.49, Median = 0.50
- Tool B: Range [0,1], Average = 0.35, Median = 0.18
- Tool C: Range [0,1], Average = 0.36, Median = 0.17
Probability of false alarm: Java CWEs

Similar trend as in case of the C/C++ false alarm values

Tool C performed better than the other two tools; Tool A performed slightly better than Tool B.

Tool A: Range [0,0.94], Average = 0.24, Median = 0
Tool B: Range [0,1], Average = 0.25, Median = 0.03
Tool C: Range [0,0.47], Average = 0.05, Median = 0
Balance: Java CWEs

Similar trend as in case of C/C++ balance values

For many CWEs balance values were around 30%, which is an indicator of overall poor performance

Tools A and C appear to perform slightly better than Tool B

Tool A: Range [0.29,1], Average = 0.50, Median = 0.34
Tool B: Range [0,1], Average = 0.43, Median = 0.34
Tool C: Range [0.29,1], Average = 0.52, Median = 0.41
ROC squares for Java CWEs

Ideal result
\((pf, pd) = (0, 1)\)

Not many points are close to the ideal \((0,1)\) point.
Tool C has noticeably lower false alarm rate.
For each tool there are multiple CWEs at the \((0,0)\) point.
C/C++

- **CWE 78 OS Command Injection**
  - Tool A had the highest recall (54%), but also very high probability of false alarm (89%)
  - Tools B and C performed poorly, with recall values around 4% and 0% respectively

- **CWE 134 Uncontrolled Format String**
  - Tool C was the most successful (with recall close to 79%), but with high probability of false alarm (i.e., 48%)
  - Tools A and B had lower recall values (i.e., around 30% and 38%, respectively)
Java

- CWE 190 Integer Overflow
  - Tool B had recall of around 27%, with relatively high false alarm rate of almost 22%
  - Neither Tool A nor Tool B detected CWE 190 (i.e. they had 0% recall)
EVALUATION BASED ON REAL PROGRAMS
Evaluation based on real software

- Three open-source software applications
  - Gzip
  - Dovecot
  - Apache Tomcat
- Older version with known vulnerabilities
- More recent version with the same vulnerabilities being fixed was used as an oracle
- A total of 44 known vulnerabilities in the three applications, mapped to 8 different CWEs
Gzip: Basic facts

- Popular open source archiving tool
- Written in C
- ~8,500 LOC
- Vulnerable version: 1.3.5 with 4 known vulnerabilities
- Version with fixed vulnerabilities: 1.3.6
# Gzip: Results

## Gzip-1.3.5 version with known vulnerabilities

<table>
<thead>
<tr>
<th>Tool</th>
<th>Warnings</th>
<th>Number of detected vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool A</td>
<td>112</td>
<td>1 out of 4</td>
</tr>
<tr>
<td>Tool B</td>
<td>36</td>
<td>0 out of 4</td>
</tr>
<tr>
<td>Tool C</td>
<td>119</td>
<td>1 out of 4</td>
</tr>
</tbody>
</table>

## Gzip-1.3.6 version with fixed vulnerabilities

<table>
<thead>
<tr>
<th>Tool</th>
<th>Warnings</th>
<th>Number of reported vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool A</td>
<td>206</td>
<td>1 out of 4</td>
</tr>
<tr>
<td>Tool B</td>
<td>125</td>
<td>0 out of 4</td>
</tr>
<tr>
<td>Tool C</td>
<td>374</td>
<td>1 out of 4</td>
</tr>
</tbody>
</table>

**True positive**

**False positive**
Dovecot: Basic facts

- IMAP/POP3 server for Unix-like operating systems
- Written in C
- ~280,000 LOC
- Vulnerable version: 1.2.0 with 8 known vulnerabilities
- Version with fixed vulnerabilities: 1.2.17
# Dovecot: Results

## Dovecot-1.2.0 version with known vulnerabilities

<table>
<thead>
<tr>
<th>Tool</th>
<th>Warnings</th>
<th>Number of detected vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool A</td>
<td>8,263</td>
<td>0 out of 8</td>
</tr>
<tr>
<td>Tool B</td>
<td>538</td>
<td>0 out of 8</td>
</tr>
<tr>
<td>Tool C</td>
<td>1,356</td>
<td>0 out of 8</td>
</tr>
</tbody>
</table>

## Dovecot-1.2.17 version with fixed vulnerabilities

<table>
<thead>
<tr>
<th>Tool</th>
<th>Warnings</th>
<th>Number of reported vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool A</td>
<td>8,655</td>
<td>0 out of 8</td>
</tr>
<tr>
<td>Tool B</td>
<td>539</td>
<td>0 out of 8</td>
</tr>
<tr>
<td>Tool C</td>
<td>1,293</td>
<td>0 out of 8</td>
</tr>
</tbody>
</table>
Tomcat: Basic facts

- Open source Java Servlet and JavaServer Pages implementation
- Written in Java
- ~4,800,000 LOC
- Vulnerable version: 5.5.13 with 32 known vulnerabilities
- Version with fixed vulnerabilities: 5.5.33
Tomcat: Basic facts

- Due to its much greater complexity, the majority of Tomcat’s vulnerabilities span several files and/or locations within each file
  - 4 out of 32 vulnerabilities occur at one location within one file
  - 9 out of 32 vulnerabilities occur at multiple locations within one file
  - 19 out of 32 vulnerabilities occur in multiple files
- We consider a true positive found if at least one of the file(s)/location(s) are matched by a tool
## Tomcat: Results

### Tomcat-5.5.13 version with known vulnerabilities

<table>
<thead>
<tr>
<th>Tool</th>
<th>Warnings</th>
<th>Number of detected vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool A</td>
<td>12,399</td>
<td>7 out of 32</td>
</tr>
<tr>
<td>Tool B</td>
<td>12,904</td>
<td>3 out of 32</td>
</tr>
<tr>
<td>Tool C</td>
<td>20,608</td>
<td>5 out of 32</td>
</tr>
</tbody>
</table>

### Tomcat-5.5.33 version with fixed vulnerabilities

<table>
<thead>
<tr>
<th>Tool</th>
<th>Warnings</th>
<th>Number of reported vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool A</td>
<td>167,837</td>
<td>2 out of 32</td>
</tr>
<tr>
<td>Tool B</td>
<td>13,129</td>
<td>0 out of 32</td>
</tr>
<tr>
<td>Tool C</td>
<td>21,128</td>
<td>1 out of 32</td>
</tr>
</tbody>
</table>
CONCLUDING REMARKS
None of the three tools produced very good results (i.e., high probability of detection (i.e., recall) and low probability of false alarm)

Tool C had the smallest false alarm rate among the three tools (mean value of 7% for the common C/C++ CWEs and 5% for the common Java CWEs)

Some CWEs were detected by all three tools, others by a combination of two tools or a single tool, while some CWEs were missed by all three tools
Conclusions

- The results of the evaluation with real open source programs were consistent with the evaluation based on the Juliet test suite
  - All three tools had high false negative rates (i.e. were not able to identify majority of the known vulnerabilities)
  - Tool A outperformed the other two tools on the application implemented in Java
- Static code analysis cannot be used as an assurance that the software is secure. Rather, it should be one of the techniques used, in addition to other complementary techniques