Gold Standard Method for Benchmarking Source Code Static Analysis Tools

Cyber Security Division
2012 Principal Investigators’ Meeting

October 11, 2012

Henny Sipma
Sr. Computer Scientist
Kestrel Technology, LLC
sipma@kestreltechnology.com
(650) 320 8474
Kestrel Technology

**Founded:** 2000

**Location:** Palo Alto, CA

**Core activity:** Sound static analysis (mathematical proof of safety)

**Tool:** CodeHawk

**Languages supported:** C, Java, x86 executables

**Underlying technology:** Abstract interpretation (Cousot, Cousot, 1977)

**Properties**

- Language-level properties: memory safety, null-dereference
  - **C:** (application-independent, mathematically well-defined properties)
    - CWEs: 119-127, 129, 131, 242, 252, 391, 466-469, 476, 682, 786-788, 805, 806, 824, 839

- **Java:** Taint analysis, loop-bound analysis (CWE606), integer overflow (CWE190/191)

- **X86:** Memory safety, information extraction
Our Tool: CodeHawk

- **Java byte code front end**: sound abstraction from Java byte code into CHIF
- **C source code front end**: sound abstraction from preprocessed CIL code into CHIF
- **x86 binary front end**: disassembly abstraction from x86 binary code into CHIF

Abstract domains:
- constants
- intervals
- strided intervals
- linear equalities
- polyhedra
- symbolic sets
- value sets
- taint

Iterators

- abstract interpretation engine
- our tool: CodeHawk
Proving safety of C programs

**Starting point:** Mathematically well-defined properties

Proving absence of:
CWEs: 119-127, 129, 131, 242, 252, 391, 466-469, 476, 682, 786-788, 805, 806, 824, 839

**Create proof obligations**
Safety conditions that state that the property holds at every relevant location in the program

**Generate invariants**
Assertions at all locations that are true for all inputs for all program executions

**Discharge proof obligations**
Location is safe from targeted vulnerabilities if invariants generated imply the safety conditions
Proving safety of C programs: feasible?

Not automatic: undecidable problem

Not easy:

Klein et al., Sel4: Formal Verification of an OS Kernel, SOSP 2009
Program size: 8700 lines of C, 600 lines of assembly code
Proof effort: 11 person-years

Our own experience with CodeHawk:

1100 small test programs (SAMATE): fully automatic, a few minutes
Larger benchmark programs (up to 1200 LOC): full verification: 1-2 person-weeks
Real-world applications (up to 100,000 LOC): 55-67% of proof obligations discharged automatically (~5 mins)
Perform a **full verification** of 6 real-world C applications
- ranging in size from 50,000 – 200,000 LOC,
- for CWEs 119, 120, 121, 122, 123, 124, 125, 126, 127, 129, 131, 134, 170, 242, 252, 391, 415, 416, 457, 466, 467, 468, 469, 476, 682, 786, 787, 788, 805, 806, 824, 839

**Ambitious?**

**Interesting Research?**

**Immediate Operational Capability**
Yes, a very important one,
but first a brief interlude on the current static analysis landscape
Current Reality:

Bug-finders are the first line of defense against software vulnerabilities

(but how effective are these bug-finders?)

Two important terms:

**False positive**: bug report that turns out not to be a bug

Makes software developers and managers very unhappy

**False negative**: bug that is not reported

Invisible, until exploited
Static Analysis Landscape: Bug-finders

**False positive:** bug report that turns out not to be a bug

Makes software developers and managers very unhappy

Naturally, software developers heavily favor bug-finders with **low false-positive rate**

**Reality:**

Low false positive rate is very easy to obtain for a bug-finder:

*Only report a bug if it is 99% likely to be a bug otherwise keep quiet*

Very high false negative rate (but that’s invisible to the developer)
The problem: Evaluating bug-finders

Tool A  Tool B

The hated false positives

The invisible false negatives
The problem: Evaluating bug-finders

The hated false positives

Tool A  Tool B

Developer will choose Tool A

Tool Vendor B is forced to lower its false positive rate (and the only cost-effective way to do so, is to increase false negatives) to stay in business

Strong economic incentive for tools that provide **less assurance**

Society suffers
The problem: Evaluating bug-finders

Maybe dealing with the extra false positives can be justified by knowing that a much higher level of assurance is achieved.

Make these false negatives visible!

Give Tool B due credit for its low false negative rate.
Evaluating Bug-finders

Problem has long been recognized by NIST: Dr. Paul Black

Small synthetic benchmarks

SAMATE (1163 test cases by MIT/Lincoln Labs), 2005
Juliet Test Suite v1.0 (45,324 test cases), Dec 2010
Juliet Test Suite v1.0 (57,099 test cases), Sep 2012

SATE competition: (real-world programs)

2008: Nagios, Lighttpd, Naim
2009: Irssi, Pvm
2010: Dovecot, ....
2011: ....
### SATE competition: participation

<table>
<thead>
<tr>
<th>vendor</th>
<th>Lighttpd</th>
<th>nagios</th>
<th>naim</th>
<th>Irssi</th>
<th>Pvm</th>
<th>Dovecot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkmarx</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Coverity</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CppCheck</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Flawfinder</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Fortify</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grammatech</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Klocwork</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>LDRA</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Marfcat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Redlizards</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Sparrow</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Veracode</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Evaluating Bug-finders

Problem has long been recognized by NIST: Dr. Paul Black

Small synthetic benchmarks
- SAMATE (1163 test cases by MIT/Lincoln Labs), 2005
- Juliet Test Suite v1.0 (45,324 test cases), Dec 2010
- Juliet Test Suite v1.0 (57,099 test cases), Sep 2012

SATE competition: (real-world programs)
- 2008: Nagios, Lighttpd, Naim
- 2009: Irssi, Pvm
- 2010: Dovecot, ....
- 2011: ....
Evaluating Bug-finders

Problem has long been recognized by NIST: **Dr. Paul Black**

Small synthetic benchmarks
- SAMATE (1163 test cases by MIT/Lincoln Labs), 2005
- Juliet Test Suite v1.0 (45,324 test cases), Dec 2010
- Juliet Test Suite v1.0 (57,099 test cases), Sep 2012

SATE competition: (real-world programs)
- 2008: **Nagios, Lighttpd, Naim**
- 2009: **Irssi, Pvm**
- 2010: **Dovecot, ....**
- 2011: ....

Gold standard + measurement tool

Measuring false positives and false negatives
Tasks

1. **Definition of metrics (02/13)**
   Redefine false positive rate / false negative rate relative to ground truth

2. **Develop analysis support for CWEs 457, 415/416, 170, and 134 (11/13)**
   Uninitialized variables, use-after-free, double-free, improper null-termination, uncontrolled format string

3. **Increase level of proof obligations proven automatically to > 80% (06/14)**
   Increase precision by improved pointer analysis, limited shape analysis, and higher degree of context sensitivity

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Proof obligations</th>
<th>Proven safe</th>
<th>Analysis time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naim</td>
<td>23,210</td>
<td>35,178</td>
<td>19,598 (56%)</td>
<td>102</td>
</tr>
<tr>
<td>Nagios (base)</td>
<td>48,321</td>
<td>42,591</td>
<td>28,677 (67%)</td>
<td>116</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>52,058</td>
<td>57,187</td>
<td>31,321 (55%)</td>
<td>410</td>
</tr>
<tr>
<td>Dovecot (part)</td>
<td>79,909</td>
<td>129,894</td>
<td>75,834 (58%)</td>
<td>121</td>
</tr>
</tbody>
</table>

Preliminary results for proving memory safety (2010)
4. Specialize the analyzer for the six applications to achieve 100% (06/14)
   E.g. introduce application-specific data structure invariants, environment assumptions

5. Develop measurement tool (08/14)

6. Apply measurement tool to published results and SWAMP tools (08/14)
Commercialization

- Software assurance centers (Lockheed Martin, GE, ….)
- Static analysis tool developers
- Developers of high-security software
- Evaluation kit, available Q1 2013

Government customers

- NIST
  - Provide prototype to NIST, early 2013
  - Incorporate feedback from NIST, throughout the program
  - Provide support for future SATE competitions
- Interested in other collaborations
Contribution to SWAMP

Evaluate tools for false positives and false negatives on benchmark programs

(SWAMP)

Other Resources (e.g., High Performance Computing Clusters)

Open Source Software (for starters) and potentially all government funded software
Operational Capability

Better assessment of vulnerabilities in critical cyber infrastructure

Measurement tool:
Provide advice which tool, or combination of tools is best suitable for each application area
Benefits to society

Create bigger market opportunity for high-quality static analysis tool vendors by enabling differentiating on false negatives

Let the market do its work

To create higher and higher levels of software assurance
Proposed Technical Approach

**Goal:** Provide “Gold Standard” benchmarking method for SWAMP-resident static analysis tools

**Tasks:**
- Define metrics as basis for measurement
- Develop analysis support for CWEs 134, 415/6, 457, 170
- Increase percentage of safety conditions proven automatically
- Full reference analysis of 6 NIST SATE benchmark applications
- Develop scoring tool to compare tool results to reference

**Current status of proposed technology:**
- Working prototype C analyzer based on abstract interpretation that automatically proves 55-67% of memory safety conditions for NIST benchmark applications

Operational Capability

**Quantitative performance targets:**
- Complete metrics definition for all targeted CWEs
- Increase percentage automation of proof to 80%
- 100% analysis of 6 NIST SATE applications for targeted CWEs

**Cost of ownership:** all deliverables provided under research contract with no additional cost

**Benefits to DHS and society at large**
- Allows measurement of false negatives of bug-finding tools
- Allows DHS to more accurately assess vulnerability exposure
- Feedback aids developers of bug-finders to improve their tools
- Advances the state of the art in scalable sound static analysis

Schedule and Milestones: 24 months period of performance

6: Definition of metrics
15: Analysis coverage for CWEs 457, 415/6, 134, 170
22: 80% level of automation on NIST benchmarks
22: 100% analysis of NIST benchmarks
24: Measurement tool, evaluation of SATE participants' tools

**Deliverables:**
- Gold Standard Analysis of NIST benchmarks
- Measurement tool to compare SWAMP results with reference
- Tool (as executable) with government rights

**Corporate Contact:** Doug Smith, smith@kestreltechnology.com
3260 Hillview Ave, Palo Alto, CA 94304, (650) 320 8474