Measuring the Effect of Code Complexity on Static Analysis Results

James Walden, Adam Messer, Alex Kuhl

Northern Kentucky University

February 5, 2009

Outline

1. Research Goals
2. Research Design
3. Results and Analysis
4. Conclusions and Future Work

Research Goals

1. Study how static analysis works on whole programs, not samples or synthetic benchmarks.
2. Determine if static analysis detection rates are correlated with code size or complexity.
3. Identify causes of failed vulnerability detection in static analysis tools.

Static Analysis Tools

Open Source lexing tools

- Flawfinder, ITS4, RATS
- High false positive rates.
- Purely local analysis.

Open Source parsing tools

- cqual, splint
- Can’t handle large C programs.

Commercial tools

- Coverity, Fortify, Klocwork, Polyspace
- Difficult to obtain.
- Older versions of gcc allow unsupported code.

Format String Vulnerabilities

Recent vulnerability

- Use %n specifier to write code to memory.
- September 1999.

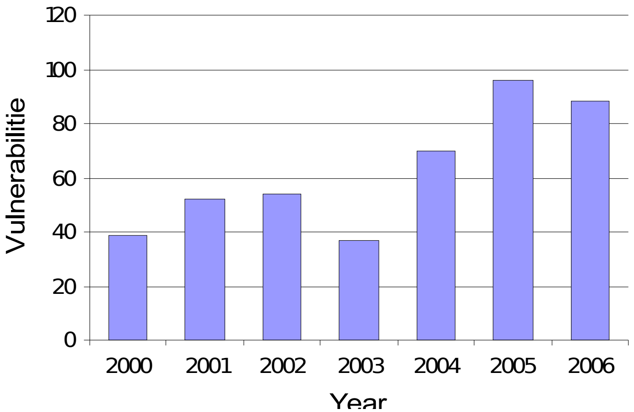
Small quantity

- 420 from 2000-2006.

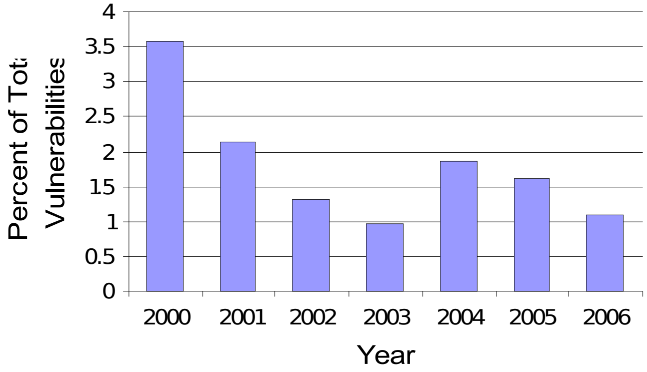
Easy to verify

- Does user input control the format specifier?

Format String Vulnerabilities



Percent of Format String Vulnerabilities



Metrics

Static Analysis Metrics

- Detection rate
- False positive rate
- Discrimination

Code Metrics

- Source Lines of Code (SLOC)
- Cyclomatic Complexity (CC)

Test Cases

35 format string vulnerabilities

- – Selected randomly from NVD 2000-2006.
- – Open source C/C++ code that compiles on Linux.
- – Each case has two versions of the code
  - One version has a format string vulnerability.
  - Other version is same program with vulnerability fixed.

Examples

- wu-ftpd
- screen
- stunnel
- gpg
- hylafax
- exim
- dhcpd
- squid
- Kerberos 5
- cdrtools
- gnats
- cvs
- socat
- ethereal
- openvpn

Results

Detections

- 22 of 35 (63%) flaws detected by SCA 4.5.

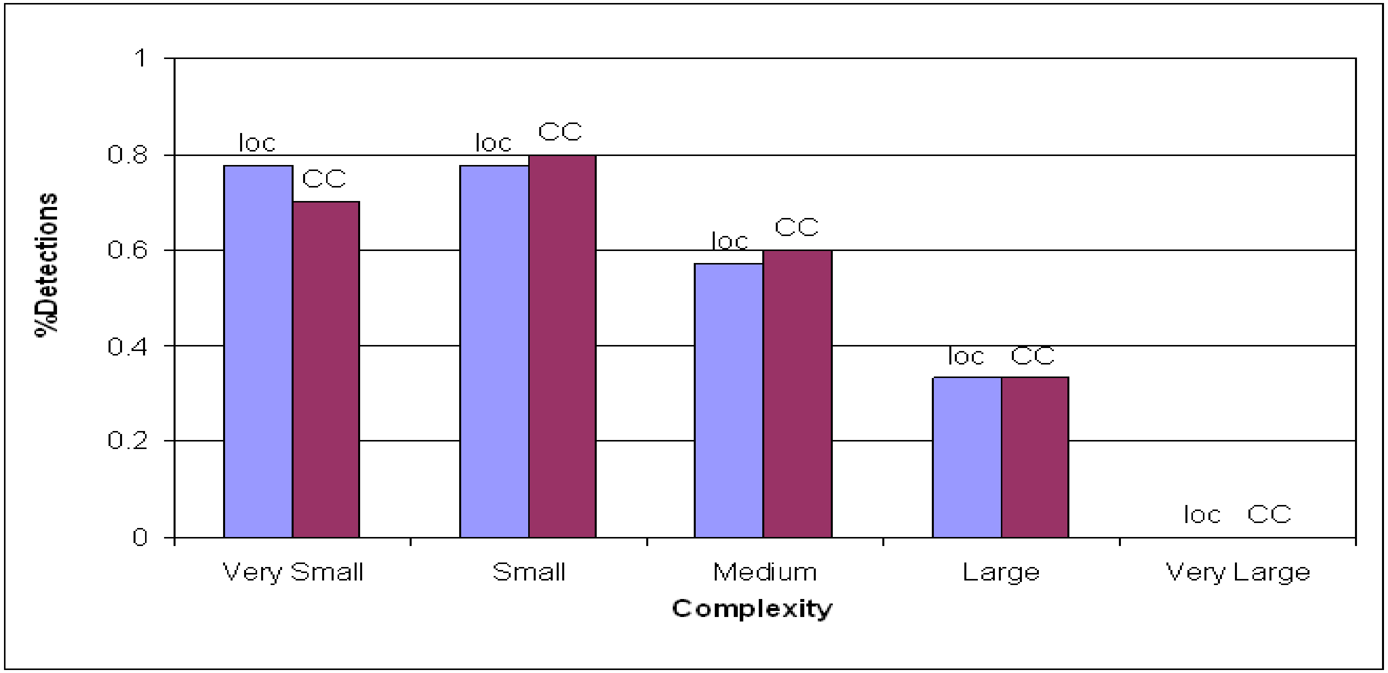
Detections by Complexity

- Divided samples into 5 complexity bins.
- No significant difference between SLOC and CC.

Discrimination:

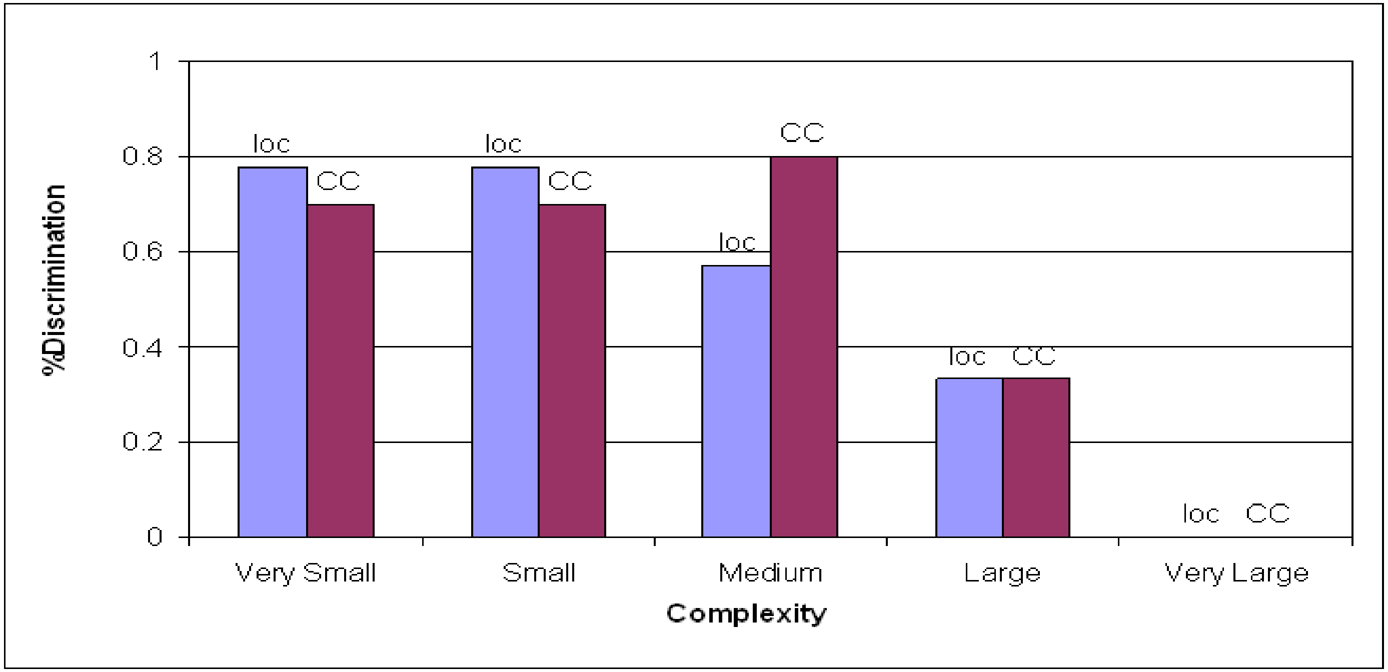
- Measure of how often analyzer passes fixed test cases when it also passes vulnerable case.
- Results almost identical to detection results since
- Only one false positive from 35 fixed samples.

Detections by Complexity Class



Class	Lines of Code	Samples	Cyclomatic	Samples
Very Small	< 5000	9	< 1000	10
Small	5000 – 25,000	9	1000 – 5000	10
Medium	25,000 – 50,000	7	5000 – 10,000	5
Large	50,000 – 100,000	6	10,000 – 25,000	6
Very Large	> 100,000	4	> 25,000	4

Discrimination by Complexity Class



Class	Lines of Code	Samples	Cyclomatic	Samples
Very Small	< 5000	9	< 1000	10
Small	5000 – 25,000	9	1000 – 5000	10
Medium	25,000 – 50,000	7	5000 – 10,000	5
Large	50,000 – 100,000	6	10,000 – 25,000	6
Very Large	> 100,000	4	> 25,000	4

Characteristics of Large Software

1. More complex control + data flow.
2. Participation of multiple developers.
3. Use of a broader set of language features.
4. Increased use of libraries that are not part of the C/C++ standard libraries.

Causes of 13 Failed Detections

Format string functions not in rule set.

- 4 of 13 (31%) failed from this cause.
- ex: ap_vsnprintf() from APR.
- Can be fixed by adding new rules.

Bug in varargs argument counting in SCA.

- 9 of 13 (69%) failed from this cause.
- Fixed in version 5 of Fortify SCA.

Generalizability of Results

Limits

- One static analysis tool studied.
- One class of vulnerabilities studied.

However, both causes apply to any vuln/tool:

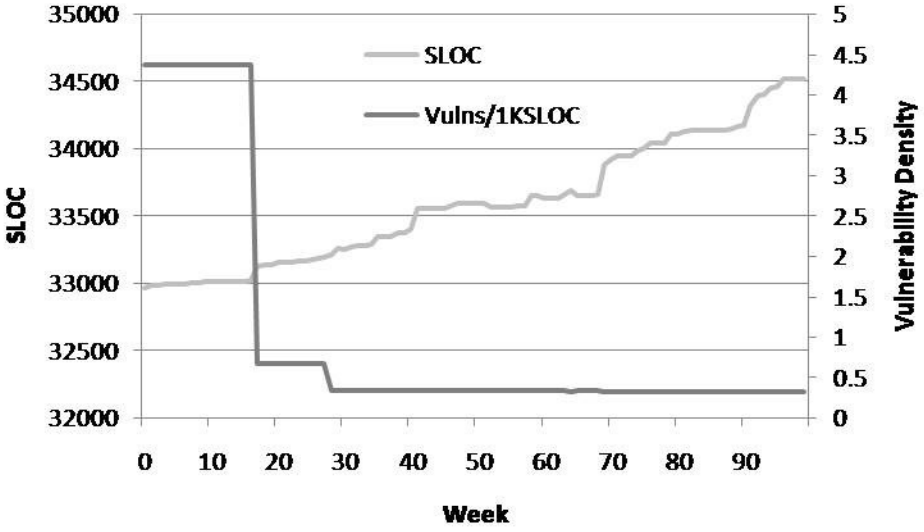
- Rule sets can’t include every dangerous sink.
- Static analysis software will have bugs.

How large are those effects?

- Do they vary by vulnerability type/language?

Future Work & Current Results

- How do static analysis results change with time? What happens after we remove all of the bugs that can be detected?
- How do code size and complexity metrics affect the number of vulnerabilities in a program over time? How does churn affect this?



Conclusions

35 Linux C programs with fmt string vulns.

- One version with a known vuln from NVD.
- One version where vuln was patched.

Static analysis detection rate of 63%.

- 31% of errors resulted from missing rules.
- 69% of errors resulted from bug in SCA.

Detection rate declines with code size/CC.

- Only 2 of 6 large projects had bugs detected.
- 0 of 4 very large projects had bugs detected.