Static Analysis methods and tools
An industrial study

Adj Prof Pär Emanuelsson – Ericsson and LiU
Prof Ulf Nilsson – LiU

ERICSSON
TAKING YOU FORWARD

Outline

- Why static analysis?
- What is it? Underlying technology
- An example
- Some tools (Coverity, KlocWork, PolySpace, …)
- Some case studies from Ericsson
- Conclusions

Method used

Tool comparision based on

- White papers
- Research reports from research groups behind tools
- Interviews with Ericsson staff
- Interviews with technical staff from tool vendors

What is SA and what can it be used for?

- Definition:
  - Analysis that does not actually run the code
- Our interest is:
  - Finding defects (preventing run-time errors)
  - Finding security vulnerabilities
- Other uses
  - Code optimization (e.g. removing run-time checks in safe languages)
  - Metrics
  - Impact analysis

Pros and cons of static analysis

- Pros
  - No test case design needed
  - No test-oracle needed
  - May detect hard-to-find bugs
  - Analyzed program need not be complete
  - Stub writing easier
- Cons
  - Potentially large number of ”false positives”
  - Does not relate to functional requirements
  - Takes programming competence to understand reports

Comparison to other techniques

- Compared to Testing
  - No test case design needed
  - No test-oracle needed
  - Can find defects that no amount of testing can do
- Compared to Formal proofs (e.g. model checking)
  - More lightweight
  - SA is much easier to use
  - SA does not need formal requirements

Software defects and errors

- *Software defect*: an anomaly in code that *might* manifest itself as an *error* at *run-time*
- Types of defects found by static analysis
  - Abrupt termination (e.g. division by zero)
  - Undefined behavior (e.g. array index out of bounds)
  - Performance degradation (e.g. memory leaks, dead code)
  - Security vulnerabilities (e.g. buffer overruns, tainted data)
- Defects not (easily) found with static analysis
  - Functional incorrectness
  - Infinite loops/non-termination
  - Errors in the environment

Examples of checkers (C-code)

- Null pointer dereference
- Uninitialized data
- Buffer/array overruns
- Dead code/unused data
- Bad return values
- Return pointers to local data
- Arithmetic operations with undefined result
- Arithmetic over-/underflow
- Parallel execution bugs
- (Non-termination)

Security vulnerabilities

- Unsafe system calls
- Weak encryption
- Access problems
- Unsafe string operations
- Buffer overruns
- Race conditions (Time-of-check, time-of-use)
- Command injections
- Tainted (untrusted) data

Buffer overflow

```
Char dst[256];
Char* s = read_string();
Strcpy(dst, s);
```

Imprecision of analyses

- Defects checked for by static analysis are *undecidable*
- *Analyses are necessarily imprecise*
- As a consequence
  - Code complained upon may be correct (false positives)
  - Code not complained upon may be defective (false negatives)
- Classic approaches to static analysis (sound analyses) report all defects checked for (no false negatives), but sometimes produce large amounts of false positives;
- Most industrial systems try to eliminate false positives but introduce false negatives as a consequence

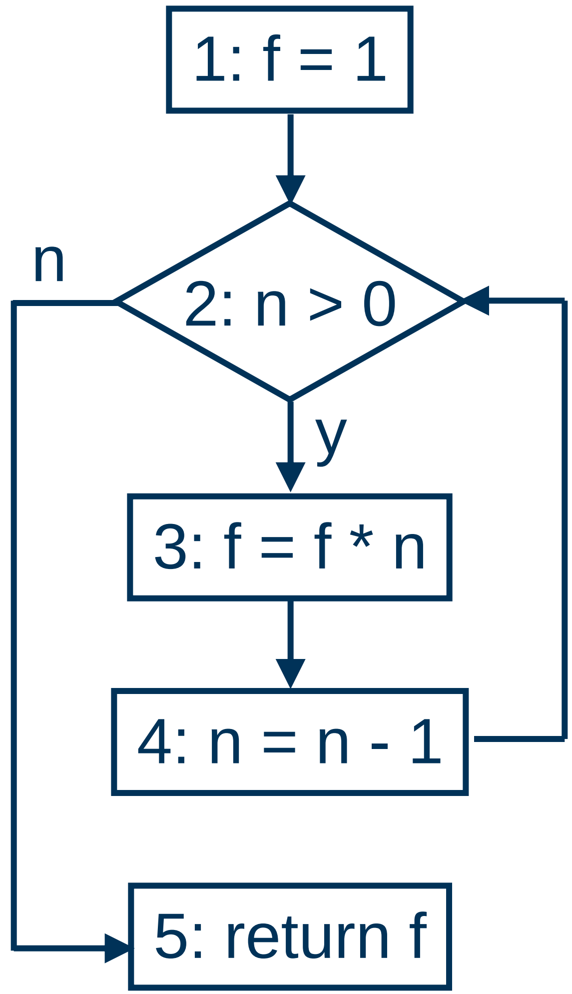
Imprecision vs analysis time

Precision depends heavily on analysis time

- Flow sensitive analysis
  - Takes program control flow into account
- Context sensitive analysis
  - Takes values of global variables and actual parameters of procedure calls into account
- Path sensitive analysis
  - Takes only valid execution paths into account
- Value analysis
  - Value ranges
  - Value dependencies

Example

```
fact(int n) {
1)      int f = 1;
2)      while( n > 0 ) {
3)              f = f * n;
4)              n = n – 1;
        }
5)      return f;
}
```



Control Flow Graph (CFG)

Program states (configurations)

- A program state is a mapping (function) from program variables to values. For example

$\sigma_1 = \{ n \rightarrow 1, f \rightarrow 0 \}$
$\sigma_2 = \{ n \rightarrow 3, f \rightarrow 0 \}$
$\sigma_3 = \{ n \rightarrow 5, f \rightarrow 0 \}$

Semantic equations

- We associate a set x_i of states with node i of the CFG (the set of states that can be observed upon reaching the node)

$x_1 = \{\{ n \to 1, f \to 0 \}, \{ n \to 3, f \to 0 \}\}$ % Example

$x_2 = \{ \sigma \mid \exists\sigma' \in x_1 \;\&\; \sigma(n)=\sigma'(n) \;\&\; \sigma(f)=1 \} \cup$

$\quad\quad \{ \sigma \mid \exists\sigma' \in x_4 \;\&\; \sigma(n)=\sigma'(n)-1 \;\&\; \sigma(f)= \sigma'(f) \}$

$x_3 = \{ \sigma \mid \sigma\in x_2 \;\&\; \sigma(n) > 0 \}$

$x_4 = \{ \sigma \mid \exists\sigma' \in x_3 \;\&\; \sigma(n)=\sigma'(n) \;\&\; \sigma(f)= \sigma'(f) * \sigma'(n) \}$

$x_5 = \{ \sigma \mid \sigma\in x_2 \;\&\; \sigma(n) \leq 0 \}$

Example run

Initially x1 = x2 = x3 = x4 = x5 = ∅

- x1 = {{n=1,f=0},{n=3,f=0}} given
- x2 = {{n=1,f=1},{n=3,f=1}} f=1
- x3 = {{n=1,f=1},{n=3,f=1}} n>0
- x4 = {{n=1,f=1},{n=3,f=3}} f=f*n
- x2 = {{n=0,f=1},{n=1,f=1},{n=2,f=3},{n=3,f=1}} f=1>2&4, n=n-1>1&3
- x3 = {{n=1,f=1},{n=2,f=3},{n=3,f=1}} n>0
- x4 = {{n=1,f=1},{n=2,f=6},{n=3,f=3}} f=f*n
- x2 = {{n=0,f=1},{n=1,f=1},{n=1,f=6},{n=2,f=3},{n=3,f=1}}
- x3 = {{n=1,f=1},{n=1,f=6},{n=2,f=3},{n=3,f=1}} n>0
- x4 = {{n=1,f=1},{n=1,f=6},{n=2,f=6},{n=3,f=3}} f=f*n
- x2 = {{n=0,f=1},{n=0,f=6},{n=1,f=1},{n=1,f=6},{n=2,f=3},{n=3,f=1}}
- x3 = {{n=1,f=1},{n=1,f=6},{n=2,f=3},{n=3,f=1}} n>0
- x5 = {{n=0,f=1},{n=0,f=6}} n<=0

Abstract descriptions of data

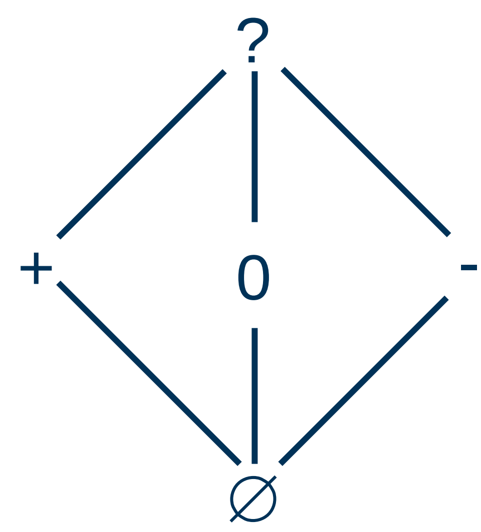
? = the set of all integers

+ = the set of all positive integers

0 = the set { 0 }

- = the set of all negative integers

∅ = the empty set (=unreachable)



Abstract operations

Abstract multiplication

Any integer

> 0

= 0

< 0

⊗	?	+	0	-
?	?	?	0	?
+	?	+	0	-
0	0	0	0	0
-	?	-	0	+

Abstract operations

Abstract subtraction

Any integer

> 0

= 0

< 0

⊖	?	+	0	-
?	?	?	?	?
+	?	?	+	+
0	?	-	0	+
-	?	-	-	?

Abstract semantic equations

$x_1 = \{ n = +, f = ? \}$

$x_2 = \{ n = lub^*(x_1(n), (x_4(n) \ominus +)), f = lub^*(+, x_4(f)) \}$

$x_3 = \{ n = +, f = x_2(f) \}$

$x_4 = \{ n = x_3(n), f = x_3(f) \otimes x_3(n)\}$

$x_5 = \{ n = ?, f = x_2(f) \}$

(*) lub(A,B) is the smallest description that contain both A and B (kind of set union)

Example abstract run

Initially x1 = x2 = x3 = x4 = x5 = { n= ∅, f= ∅ }

- x1 = { n=(+),f= ? } given
- x2 = { n=(+),f=(+) }
- x3 = { n=(+),f=(+) }
- x4 = { n=(+),f=(+) }
- x2 = { n= ?,f=(+) }
- x3 = { n=(+),f=(+) }
- x5 = { n= (+),f=(+) }

SA techniques

1. Pattern matching
2. Control flow analysis
3. Data flow analysis
4. Value analysis
   1. Intervals
   2. Aliasing analysis
   3. Variable dependencies
5. Abstract interpretation

Examples of dataflow analysis

- Reaching definitions (which definitions reach a point)
- Liveness (variables that are read before definition)
- Definite assignment (variable is always assigned before read)
- Available expressions (already computed expressions)
- Constant propagation (replace variable with value)

Aliasing

- x = 5
- y = 10
- = x

- x [i] = 5
- x [j] = 10
- = x[i]

Tool comparison

Tool	Coverity	Klocwork	Polyspace	Flexelint
Language	C/C++/Java	C/C++/Java	C/C++/ADA	C/C++
Program size	MLOC	MLOC	60KLOC	MLOC
Soundness	Unsound	Unsound	Sound	Unsound
False positives	few	few	many	many
Analysis	def,sec	def,sec,met	def	def
incrementality	yes	no	no	no

Coverity Prevent

- Company founded in 2002
- Originates from Dawson Engeler's research at Stanford
- Well documented through research papers
- Commonly viewed as market leading product
- Good results from Homeland Security's audit project
- Coverity Extend allows user-defined checks (Metal language)
- Good explanations of faults
- Good support for libraries
- Incremental

Klocwork K7

- Company founded by development group at Nortel 2001
- Similar to Coverity (in checkers provided)
- Besides finding defects: refactoring, code metrics, architecture analysis
- Easy to get started and use
- Good explanations of faults
- Good support for foreign libraries

Polyspace Verifier/Desktop

- French company co-founded by students of Patrick Cousot 1999. Aquired by Mathworks 2007.
- Claims to intercept 100% of the runtime errors checked for in C/C++/ADA programs.
- Customers in airline industry and the European space program (embedded software).
- Very thorough – especially on arithmatic
- Can be slow and produces many false positives
- Documentation hard to read
- Restricted support for security vulnerabilities and management of dynamic memory

Functionality	Coverity	KlocWork	PolySpace
Coding style	No	Some	No
Buffer overrun	Yes	Yes	Yes
Arithmetic over/underflow	No	No	Yes
Illegal shift operations	No	No	Yes
Undefined arithmetic operations	No	No	Yes
Bad return value	Yes	Yes	Yes
Memory/resource leaks	Yes	Yes	No
Use after free	Yes	Yes	No
Uninitialized variables	Yes	Yes	Yes
Size mismatch	Yes	Yes	Yes
Stack use	Yes	No	No
Dead code/data	Yes	Yes	Yes (code)
Null pointer dereference	Yes	Yes	Yes
STL checkers	Some	Some	No?
Uncaught exceptions	Beta (C++)	No	No
User assertions	No	No	Yes
Function pointers	No	No	Yes
Nontermination	No	No	Yes
Concurrency	Lock order	No	Shared data
Tainted data	Yes	Yes	No
Time-of-check Time-of-use	Yes	Yes	No
Unsafe system calls	Yes	Yes	No
MISRA support	No	No	Yes
Extensible	Yes	Some	No
Incremental analysis	Yes	No	No
False positives	Few	Few	Many
False negatives	Yes	Yes	No
Software metrics	No	Yes	No
Language support	C/C++	C/C++/Java	C/C++/Ada

Largest SA project?
Audit of open source projects

- Grant by Homeland Security in 2006
- Coverity, Klocwork and others
- More than 290 open source software projects analysed: Apache, FreeBSD, GTK, Linux, Mozilla, MySQL, PostgreSQL, and many more.
- +7000 defects fixed during first 18 months (50 000 up to now)
- See http://scan.coverity.com/

Other SA tools

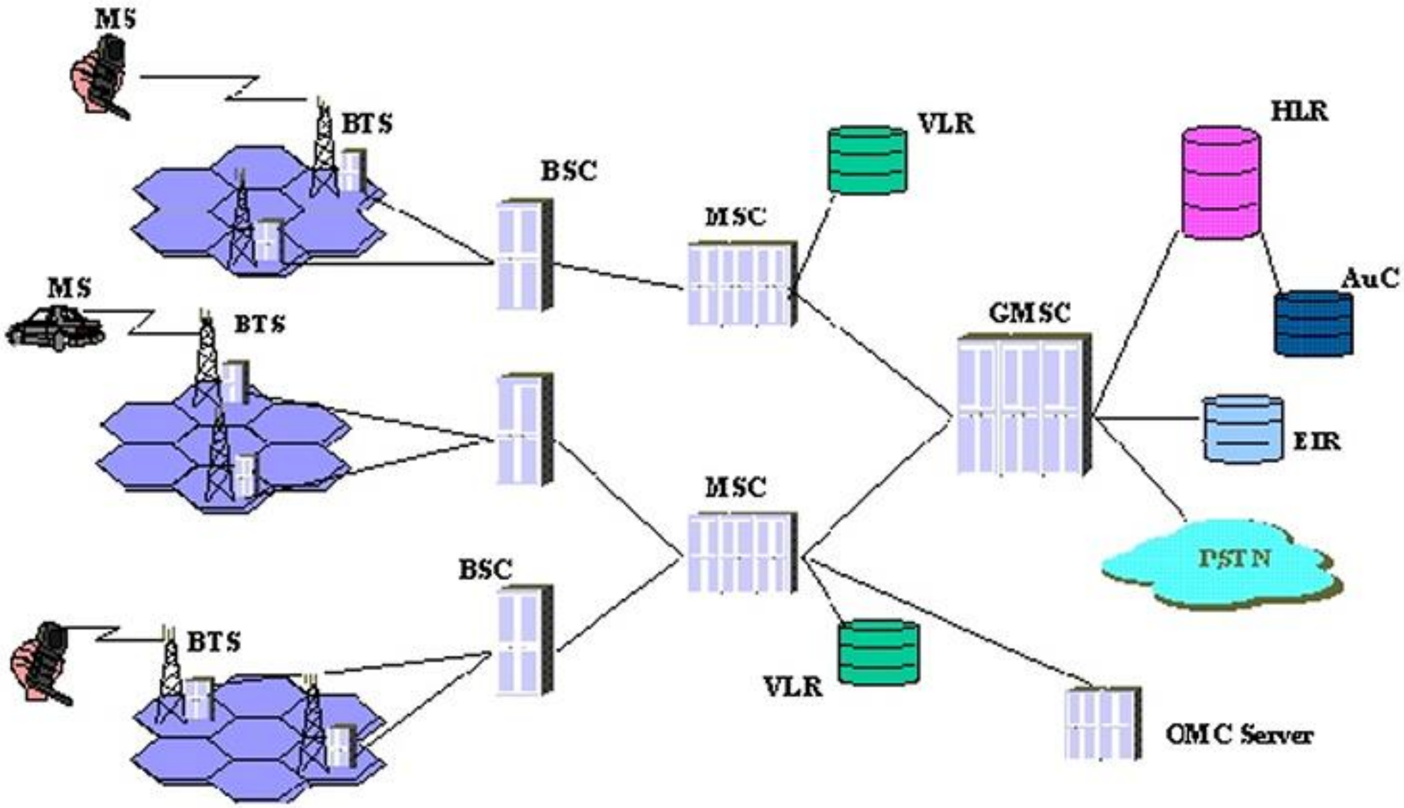
- Grammatech - Code sonar. Similar to Coverity and Klocwork. Co-founders Tom Reps and Tim Teitelbaum.
- Parasoft C++test – performs some static analysis (checks 700 coding standard rules).
- Purify focuses on memory-leaks, not defects in general. It is a dynamic tool – requires test cases.
- PREfast and PREfix – Microsoft proprietory.
- Astree – academic tool by Patric Cousot. Very thorough, works on C without recursion and dynamic memory.

Splint

- Open source
- C language
- Based on Lint
- Modified for security
- Annotations added
- Style warnings

Telecom system

Available 99.999%



Ericsson experiences 1 – Coverity - Flexelint

- Mature product that had been in use for several years and well tested
- FlexeLint 1 200 000 errors and warnings, could be reduced to 1 000 with a great deal of filtering work
- Coverity found 40 defects
- Had expected Coverity to find more defects and more serious ones
- Even if many of the defects found were not bugs that could cause a crash they were certainly things that should be corrected

Ericsson experiences 2 - Coverity

- 1,2 MLoC is analyzed in 3 hours
- Easy to install and use and no modifications to existing development environment needed
- Part of code was previously analyzed with Flexelint
- 1464 defects found
- 55% no real errors but bad style
- 2% false positives
- 38% bugs – 1% severe
- considerable number of severe defects were found although code is in PRA quality.

Ericsson experiences 3 – Coverity and Klocwork (43KLoC)

	Klocwork	False positives	Found by both tools	Coverity	False positives
Known memory leaks	0	0	0	0	0
Null-pointer defects	15	2	2	4	0
Found memory leaks	12	8	1	7	0
Unutilized variables	0	0	0	2	0
Freeing Non-Heap Memory	3	0	0	0	0
Buffer overruns	2	0	0	3	1
Total	**32**	**10**	**3**	**16**	**1**

Ericsson experiences 4 – Java. Coverity, Klocwork and CodePro

- A Java product with known faults was analyzed.
- Beta version of Coverity was used.
- Large difference in warnings:
  - Coverity 92, Klocwork 658, CodePro 8000.
- Coverity found many more faults and had far less false positives than Klocwork.
- Users seem to prefer Klocwork anyway (with filtering: only 19 warnings in the topmost 4 severity levels).
- CodePro is designed for interactive use.
- Interactivity of CodePro is appreciated, but possibility to save discovered defects is required.

Ericsson experiences summary

- Easy to get going and use - no big changes in processes needed.
- The tools discover many bugs that would not be found otherwise.
- Analysis time is acceptable and comparable to build time.
- Some users had expected the tools to find more defects and defects that were more severe
- Some users were surprised to find that several bugs were found in applications that had been in use for a long time.
- Many of the defects found would not cause a crash but after a small modification a serious crash could happen.
- Tools often discover different defects and often do not find known ones.
- Handling of third party libraries can make a big difference.
- Tools should be used throughout development
- Flexelint can be successful if applied from project start
- Coverity and Klocwork similar – but also very different results in some cases

Conclusions

- Good and useful tools
- Find bugs with little effort
- Some tools are mature
  - Can handle very large applications
  - Surprisingly few false positives
  - Easy to use
- Unclear how many defects that are *not* discovered

Litterature

- Mandatory
  - Emanuelsson, Nilsson: A Comparative Study pf Industrial static analysis tools
  - Example in Lecture
  - Livshitz, Lam: Finding Security Vulnerabilities in Java Applications with SA
- Non-mandatory
  - Balakrishnan,... WYSINWYX: What you see is not what you execute
  - Bessey, ...: A few billion lines of code later