Static Testing

Spring, 2023

Yi Xiang xiangyi@scut.edu.cn

Contents

- Static Testing
  - Code Review
  - Static Program Analysis

1. Static Testing

- Static testing is the process of carefully and methodically reviewing and analyzing the software for bugs without executing it.
  - Code Review
  - Static Program Analysis

- This form of testing is very valuable and has advantages over execution-based testing. Reported experience shows that a large number of faults can be found using static testing.

- This has benefits in terms of cost and productivity because faults are found (and corrected) early on and less time is required for execution-based tests.

Experience of Static Testing

- Raytheon
  - Reduced "rework" from 41% of cost to 20% of cost
  - Reduced effort to fix integration problems by 80%
- Paulk et al.: cost to fix a defect in space shuttle software
  - $1 if found in inspection
  - $13 during system test
  - $92 after delivery
- IBM
  - 1 hour of inspection saved 20 hours of testing
  - Saved 82 hours of rework if defects in released product
- IBM Santa Teresa Lab
  - 3.5 hours to find bug with inspection, 15-25 through testing
- C. Jones
  - Design/code inspections remove 50-70% of defects
  - Testing removes 35%
- R. Grady, efficiency data from HP
  - System use 0.21 defects/hour
  - Black box 0.28 defects/hour
  - White box 0.32 defects/hour
  - Reading/inspect 1.06 defects/hour
- Your mileage may vary
  - Studies give different answers
  - These results show what is possible

2. Code Review

Essential elements of code review

- Identify problems:
  - Find problems with the software such as missing items, mistakes, etc.
- Follow rules:
  - Amount of code to be reviewed, how much time will be spent, etc.
- Prepare:
  - Each participant should prepare in order to contribute to the review.
- Write a report:
  - Summarize the results of the review, make report available to the development team.

Informal Code Review

- Peer reviews
  - An informal small group of programmers and/or testers act as reviewers.
  - Participants should follow the 4 essential elements even through the review is informal.

- Walkthroughs
  - A more formal process in which the author of the code formally presents the code to a small group of programmers and/or testers.
  - The author reads the code line by line explaining what it does, reviewers listen and ask questions.
  - Participants should follow the 4 essential elements.

Formal Code Inspections

- An inspection is more comprehensive than a walk-through.
  - Code presenter is not the author of the code.
  - The other participants are the inspectors.
  - There is a moderator to assure that the rules are followed and the meeting runs smoothly.
- After the inspection a report is composed. The programmer then makes changes and a re-inspection occurs, if necessary.
- Formal code inspections are effective at finding bugs in code and designs and are gaining in popularity.

Formal Code Inspections – 5 steps

- Stage 1 - Overview
  - An overview document that details the product specifications/ design/code/plan is prepared by the programmer. The document is distributed to participants.

- Stage 2 - Preparation
  - The Tester must understand the document in detail. A checklist of fault types generally found in inspections ranked by frequency should be available to help concentrate their efforts.

- Stage 3 - Inspection
  - At the meeting there is a walk-through of the document to ensure that each item in the checklist is covered. Any faults found are simply documented for later correction.

Formal Code Inspections – 5 steps

- Stage 4 - Rework
  - After the meeting all faults and issues are resolved.

- Stage 5 - Follow-up
  - The leader of the code inspection group must finally ensure that every issue has been resolved, and should produce a final report. This will provide detail on items such as:
    – faults found categorized by their type
    – fault statistics (for example, the number of faults found compared to the number of faults found at same stage of development in other products)

✓ The report should also be able to recommend the redesign of a module or modules if too many faults were found. Other similar modules could be subjected to more rigorous testing when they are produced.

- Once the report is finalized the inspection can be declared to be complete.

Code review checklist: Data reference errors

- Is an un-initialized variable referenced?
- Are array subscripts integer values and are they within the array's bounds?
- Are there off-by-one errors in indexing operations or references to arrays?
- Is a variable used where a constant would work better?
- Is a variable assigned a value that's of a different type than the variable?
- Is memory allocated for referenced pointers?
- Are data structures that are referenced in different functions defined identically?

Code review checklist: Data declaration errors

- Are the variables assigned correct length, type, storage class?
  - E.g. should a variable be declared a string instead of an array of characters?
- If a variable is initialized at its declaration, is it properly initialized and consistent with its type?
- Are there any variable with similar names?
- Are there any variables declared that are never referenced or just referenced once (should be a constant)?
- Are all variables explicitly declared within a specific module?

Code review checklist: Computation errors

- Do any calculations that use variables have different data types?
  - E.g., add a floating-point number to an integer
- Do any calculations that use variables have the same data type but are different size?
  - E.g., add a long integer to a short integer
- Are the compiler's conversion rules for variables of inconsistent type or size understood?
- Is overflow or underflow in the middle of a numeric calculation possible?
- Is it ever possible for a divisor/modulus to be 0?
- Can a variable's value go outside its meaningful range?
  - E.g., can a probability be less than 0% or greater than 100%?
- Are parentheses needed to clarify operator presence rules?

Code review checklist: Comparison errors

- Are the comparisons correct?
  - E.g., < instead of <=
- Are there comparisons between floating-point values?
  - E.g., is 1.0000001 close enough to 1.0000002 to be equal?
- Are the operands of a Boolean operator Boolean?
  - E.g., in C 0 is false and non-0 is true

Code review checklist: Control flow errors

- Do the loops terminate? If not, is that by design?
- Does every **switch** statement have a **default** clause?
- Are there **switch** statements nested in loops?
  - E.g., careful because **break** statements in switch statements will not exit the loop … but break statements not in switch statements will exit the loop.
- Is it possible that a **loop** never executes? If it acceptable if it doesn't?
- Does the compiler support short-circuiting in expression evaluation?

Code review checklist: Subroutine parameter errors

- If constants are passed to the subroutine as arguments are they accidentally changed in the subroutine?
- Do the units of each parameter match the units of each corresponding argument?
  - E.g., English versus metric
  - This is especially pertinent for SOA components
- Do the types and sizes of the parameters received by a subroutine match those sent by the calling code?

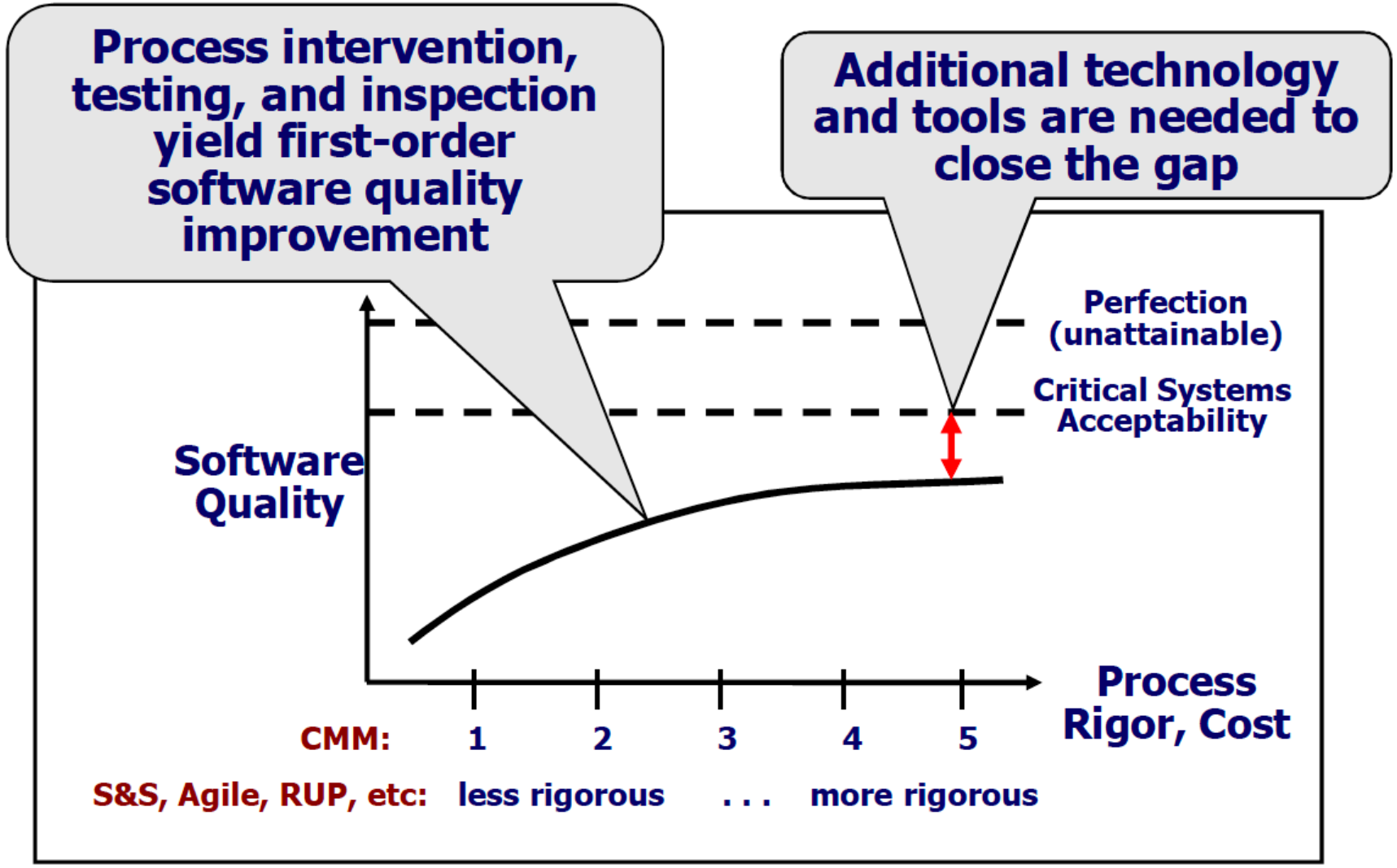
Code review checklist: Input/Output errors

- If the file or peripheral is not ready, is that error condition handled?
- Does the software handle the situation of the external device being disconnected?
- Have all error messages been checked for correctness, appropriateness, grammar, and spelling?
- Are all exceptions handled by some part of the code?
- Does the software adhere to the specified format of the date being read from or written to the external device?

Code review checklist: Other checks

- Does your code pass the **lint** test?
  - E.g., How about **gcc** compiler warnings?
- Is your code portable to other OS platforms?
- Does the code handle ASCII and Unicode?
- How about internationalization issues?
- Does your code rely on deprecated APIs?
- Will your code port to architectures with different byte orderings?
  - E.g., little (increasing numeric significance with increasing memory addresses) versus big (the opposite of little) endian?

3. Program Static Analysis



Program Static Analysis

Dynamic:

- Testing: Direct execution of code on test data in a controlled environment.

Static:

- Inspection: Human evaluation of code, design documents (specs and models), modifications.
- Analysis: Tools reasoning about the program without executing it.

➢ Try to discover issues by analyzing source code. No need to run.

- Defects of interest may be on uncommon or difficult-to-force execution paths for testing.
- What we really want to do is check the entire possible state space of the program for particular properties. .(e.g., race condition, buffer overflow, use after free)
- Static code analysis tools: Lint/ FindBugs/Coverity/Facebook Infer…

Defects Static Analysis can Catch

Defects that result from inconsistently following simple design rules.

- Security: Buffer overruns, improperly validated input.
- Memory safety: Null dereference, uninitialized data.
- Resource leaks: Memory, OS resources.
- API Protocols: Device drivers; real time libraries; GUI frameworks.
- Exceptions: Arithmetic/library/user-defined
- Encapsulation: Accessing internal data, calling private functions.
- Data races: Two threads access the same data without synchronization

The most common defect in open-source software

Coverity Scan Program (Stanford, 2006)

- Launched under a contract with the Department of Homeland Security to harden open source software which provides critical infrastructure for the Internet.
- Analyzed more than 290 open source projects, including Linux, Apache, PHP, and Android.

Defects	Frequency	Risk
Null pointer dereferences	27.60%	Medium
Resource leaks	23.19%	High
Incorrect expression	9.76%	Medium
Uninitialized variables	8.41%	High
Use after free	5.91%	High
Buffer overflows	5.52%	High

File: /smalivm/src/main/java/org/cf/smalivm/context/ExecutionNode.java

```
148         this.parent = parent;
```

```
< 1. Condition "parent != null", taking false branch
```

```
<< 2. Comparing "parent" to null implies that "parent" might be null.
```

```
149         if (parent != null) {
150             parent.addChild(this);
151         }
```

```
<<< CID 33615: Null pointer dereferences FORWARD_NULL
<<< 3. Calling a method on null object "parent".
```

```
152         getContext().setParent(parent.getContext());
153     }
```

Checker:
FORWARD_NULL

program Crash, exit, restart, execution of unauthorized code or command

File: /home/travis/build/kortemik/OpenTechBFG/neo/swf/SWF_Image.cpp

```
< 1. Condition "bitstream->Length() == 0", taking false branch

402         if( bitstream.Length() == 0 )
403         {
404                 // no clue why this happens
405                 return;
406         }
407         int width, height;

<< 2. Storage is returned from allocation function "Load".

<<< CID 35785: Resource leaks RESOURCE_LEAK
<<< 3. Ignoring storage allocated by "this->jpeg.Load(bitstream->ReadData(bitstream->Length()), bitstream->Le

408         jpeg.Load( bitstream.ReadData( bitstream.Length() ), bitstream.Length(), width, height );
409 }
410
```

DoS attacks, sensitive data leaks, resource consumption

Checker:
RESOURCE_LEAK

File: /home/travis/build/kortemik/OpenTechBFG/neo/renderer/OpenGL/gl_Image.cpp

```
75
76                  assert( x + width <= padW && y + height <= padH );
77                  // upload the non-aligned value, OpenGL understands that there
78                  // will be padding
79                  if( x + width > opts.width )
80                  {
```

<< "this->opts.width - x" looks like the original copy.

```
81                          width = opts.width - x;
82                  }
83                  if( y + height > opts.height )
84                  {
```

<<< CID 35766: Incorrect expression COPY_PASTE_ERROR
<<< "x" in "this->opts.height - x" looks like a copy-paste error.

< Should it say "y" instead?

```
85                          height = opts.height - x;
86                  }
87          }
```

Checker:
COPY_PASTE_ERROR

Unexpected output values, program logic errors, runtime errors

File: /home/travis/build/msoos/cryptominisat/python/pycryptosat.cpp

```
283         while ((lit = PyIter_Next(iterator)) != NULL) {
```

```
<< 3. Declaring variable "var" without initializer.
```

```
284             long var;
285             bool sign;
```

```
<<< CID 1306224: Uninitialized variables UNINIT
<<< 4. Using uninitialized value "var" when calling "convert_lit_to_sign_and_var".
```

```
286             int ret = convert_lit_to_sign_and_var(lit, var, sign);
287             Py_DECREF(lit);
```

Checker: UNINIT

Result in incorrect program logic, incorrect data, program crash

File: /wazuh_modules/wmodules.c

```
52  void wm_destroy() {
53      wmodule *cur_module;
54      wmodule *next_module;
55
```

```
< 1. Condition "cur_module", taking true branch

< 4. Condition "cur_module", taking true branch
```

```
57      for (cur_module = wmodules; cur_module; wmodules = next_module) {
```

```
<<< CID 117766: Memory - illegal accesses USE_AFTER_FREE
<<< 5. Dereferencing freed pointer "cur_module".
```

```
58          next_module = cur_module->next;
59          cur_module->context->destroy(cur_module->data);
```

```
<< 2. "free" frees "cur_module".
```

```
60          free(cur_module);
```

```
< 3. Jumping back to the beginning of the loop
```

```
61      }
```

Checker:
USE_AFTER_FREE

Memory – illegal accesses, Program Crash, exit, restart, resource consumption

File: /engines/tinsel/dialogs.cpp

```
< 13. Condition "Tinsel::g_objArray[i]", taking true branch

< 14. Condition "i < 21", taking true branch

< 16. Condition "Tinsel::g_objArray[i]", taking true branch

< 17. Condition "i < 21", taking true branch

<< 18. Checking "i < 21" implies that "i" may be up to 20 on the true branch.

<< 20. Incrementing "i". The value of "i" may now be up to 21.

<<< CID 1003944: Memory - illegal accesses OVERRUN
<<< 21. Overrunning array "Tinsel::g_objArray" of 21 8-byte elements at element index 21 (byte offset 1
```

```
3742                    for (i = 0; g_objArray[i] && i < MAX_WCOMP; i++) {
3743                        MultiMoveRelXY(g_objArray[i], x - center, deltay);
```

Checker: OVERRUN

Memory - illegal accesses, Program Crash, exit, restart, resource consumption

Empirical Results on Static Analysis

- Static analysis tools as early indicators of pre-release defect density （2005）
  - The defects identified by two different static analysis tools
  - Predict the actual pre-release defect density for Windows Server 2003
  - Identify fault-prone areas of code requiring further testing.

- On the Value of Static Analysis for Fault Detection in Software （2006）
  - Nortel Network/3 C/C++ projects/3 million LOC total/ large-scale industrial software
  - Early generation static analysis tools
  - Cost per fault of static analysis 61-72% compared to inspections
  - Effectively finds assignment, checking faults
  - Can be used to find potential security vulnerabilities

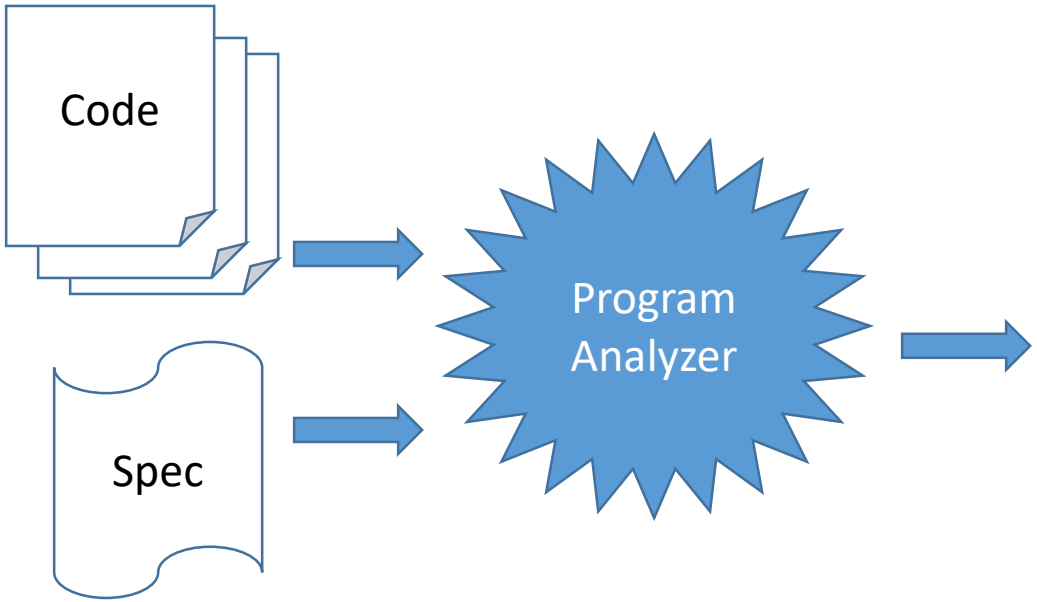
Results indicate static analysis tools are complementary to other fault-detection techniques for the economic production of a high-quality software product.

Quality assurance at Microsoft

- Original process: manual code inspection
  - Effective when system and team are small
  - Too many paths to consider as system grew

- Early 1990s: add massive system and unit testing
  - Tests took weeks to run
    - Diversity of platforms and configurations
    - Sheer volume of tests
  - Inefficient detection of common patterns, security holes
- Non-local, intermittent, uncommon path bugs was treading water in Windows Vista development
- Early 2000s: add static analysis

Program Analyzers



Report	Type	Line
1	mem leak	324
2	buffer oflow	4,353,245
3	sql injection	23,212
4	stack oflow	86,923
5	dang ptr	8,491
…	…	…
10,502	info leak	10,921

```
package com.google.devtools.staticanalysis;

public class Test {
```

▾ Lint — Missing a Javadoc comment.
Java
1:02 AM, Aug 21
Please fix — Not useful

```
  public boolean foo() {
    return getString() == "foo".toString();
```

▾ ErrorProne — String comparison using reference equality instead of value equality (see http://code.google.com/p/error-prone/wiki/StringEquality)
StringEquality
1:03 AM, Aug 21
Please fix
Suggested fix attached: show — Not useful

```
  }

  public String getString() {
    return new String("foo");
  }
}
```

//depot/google3/java/com/google/devtools/staticanalysis/Test.java

```
package com.google.devtools.staticanalysis;

public class Test {
  public boolean foo() {
    return getString() == "foo".toString();
  }

  public String getString() {
    return new String("foo");
  }
}
```

```
package com.google.devtools.staticanalysis;

import java.util.Objects;

public class Test {
  public boolean foo() {
    return Objects.equals(getString(), "foo".toString());
  }

  public String getString() {
    return new String("foo");
  }
}
```

Apply Cancel

Two fundamental concepts

Abstraction

- Elide details of a specific implementation.
- Capture semantically relevant details; ignore the rest.

Programs as data

- Programs are just trees/graphs!
- …and we know lots of ways to analyze trees/graphs, right?

Defining Static Analysis

- Systematic examination of an abstraction of program state space.
  - Does not execute code! (like code review)

- Abstraction: A representation of a program that is simpler to analyze.
  - Results in fewer states to explore; makes difficult problems tractable.

- Check if a particular property holds over the entire state space:
  - Liveness: “something good eventually happens.”
  - Safety: “this bad thing can’t ever happen.”
  - Compliance with mechanical design rules.

The Bad News: Rice's Theorem

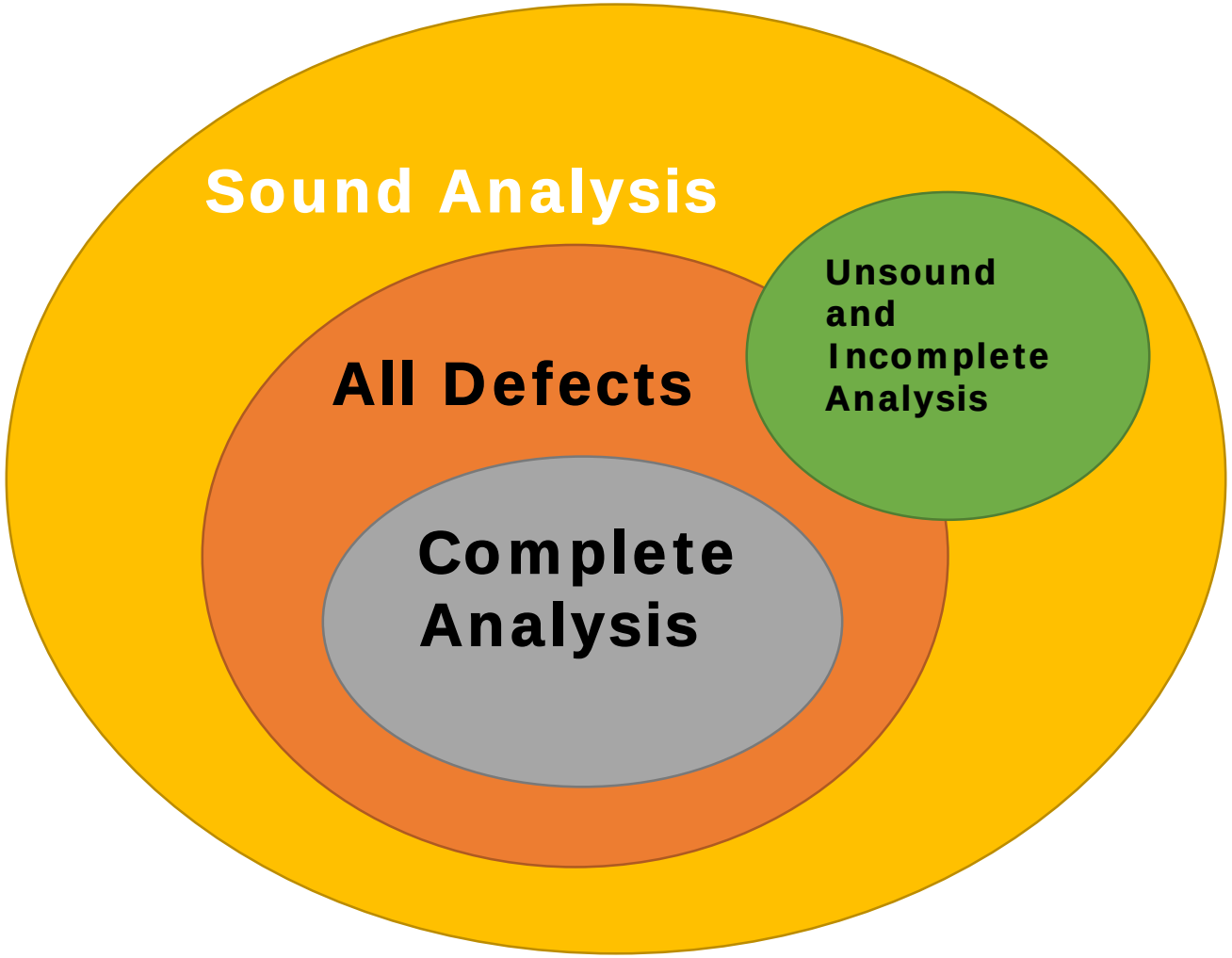
"Any nontrivial property about the language recognized by a Turing machine is undecidable.“

Henry Gordon Rice, 1953

Every static analysis is necessarily incomplete or unsound or undecidable (or multiple of these)

34

Results combined



	Error exists	No error exists
Error Reported	True positive (correct analysis result)	False positive
No Error Reported	False negative	True negative (correct analysis result)

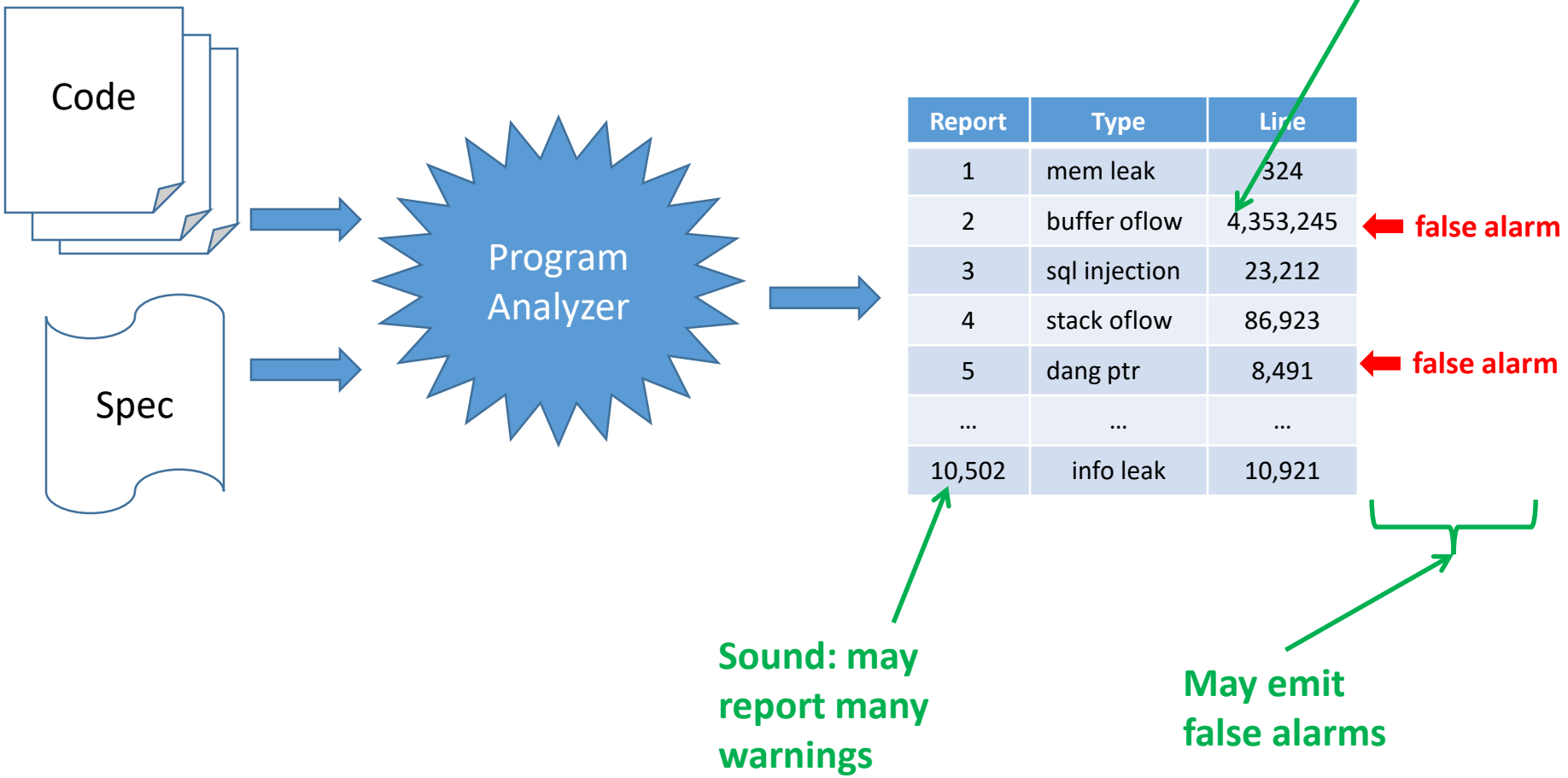
Sound Analysis:

reports all defects
-> no false negatives
typically overapproximated

Complete Analysis:

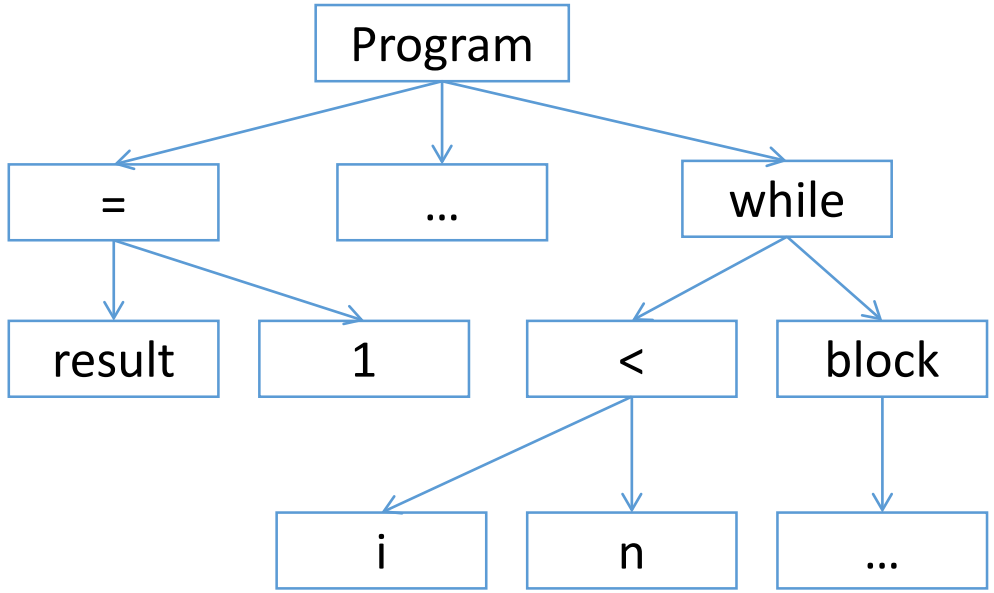
every reported defect is an actual defect
-> no false positives
typically underapproximated

Sound Program Analyzer



Abstract Syntax Trees

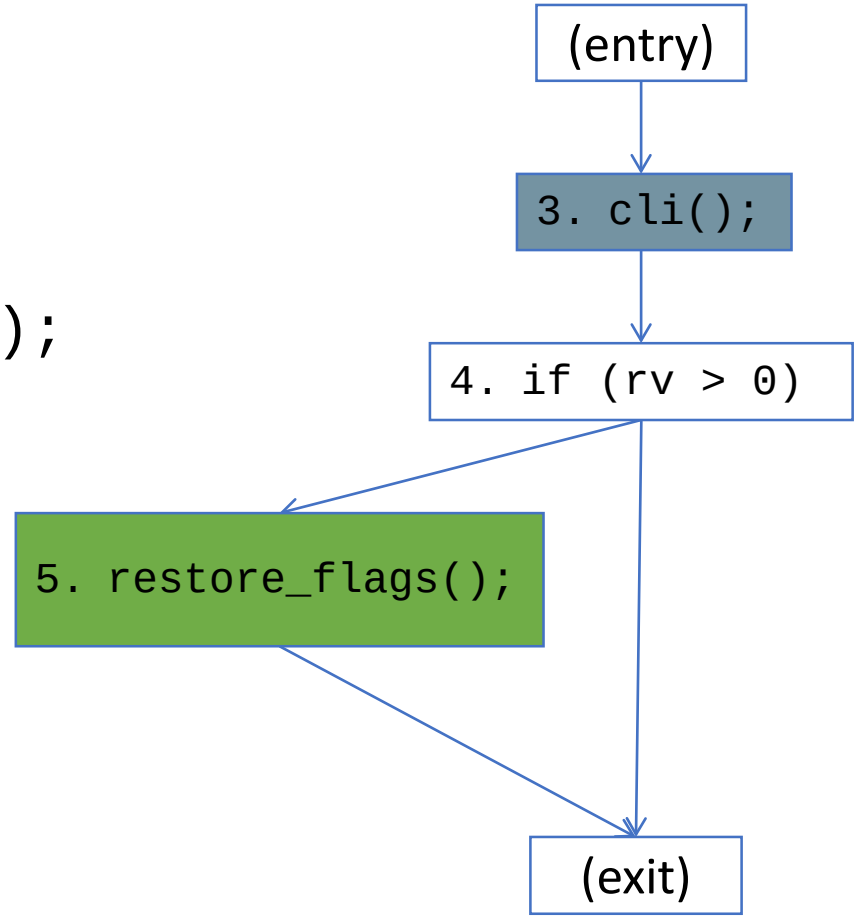
```
int result = 1;
int i = 2;
while (i < n) {
  result *= i;
  i++;
}
return result;
```



That is what your IDE and compiler are doing

Abstraction Control-Flow Graph

```
1.  void foo() {
2.      …
3.      cli();
4.    if (a) {
5.        restore_flags();
6.      }
7.  }
```



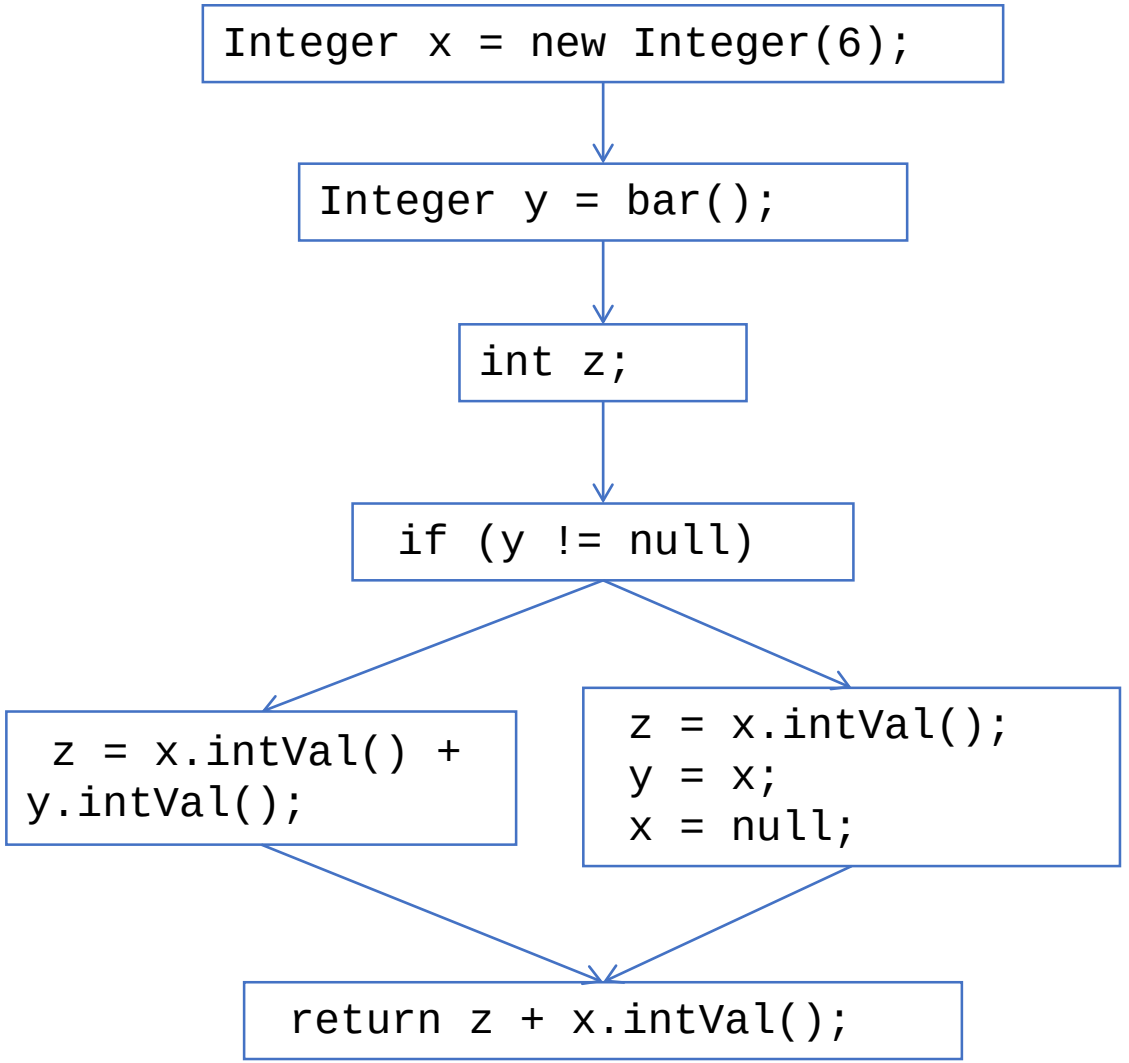
Data flow Analysis

```
1.  int foo() {
2.      Integer x = new Integer(6);
3.      Integer y = bar();
4.      int z;
5.      if (y != null)
6.          z = x.intVal() + y.intVal();
7.      else {
8.          z = x.intVal();
9.          y = x;
10.         x = null;
11.     }
12.     return z + x.intVal();
13. }
```

Are there any possible **null pointer exceptions** in this code?

In graph form . . .

```
1.int foo() {
2.    Integer x = new Integer(6);
3.    Integer y = bar();
4.    int z;
5.    if (y != null)
6.       z = x.intVal() + y.intVal();
7.    } else {
8.       z = x.intVal();
9.       y = x;
10.      x = null;
11.    }
12.    return z + x.intVal();
13.}
```



Null pointer analysis

- Track each variable in the program at all program points.

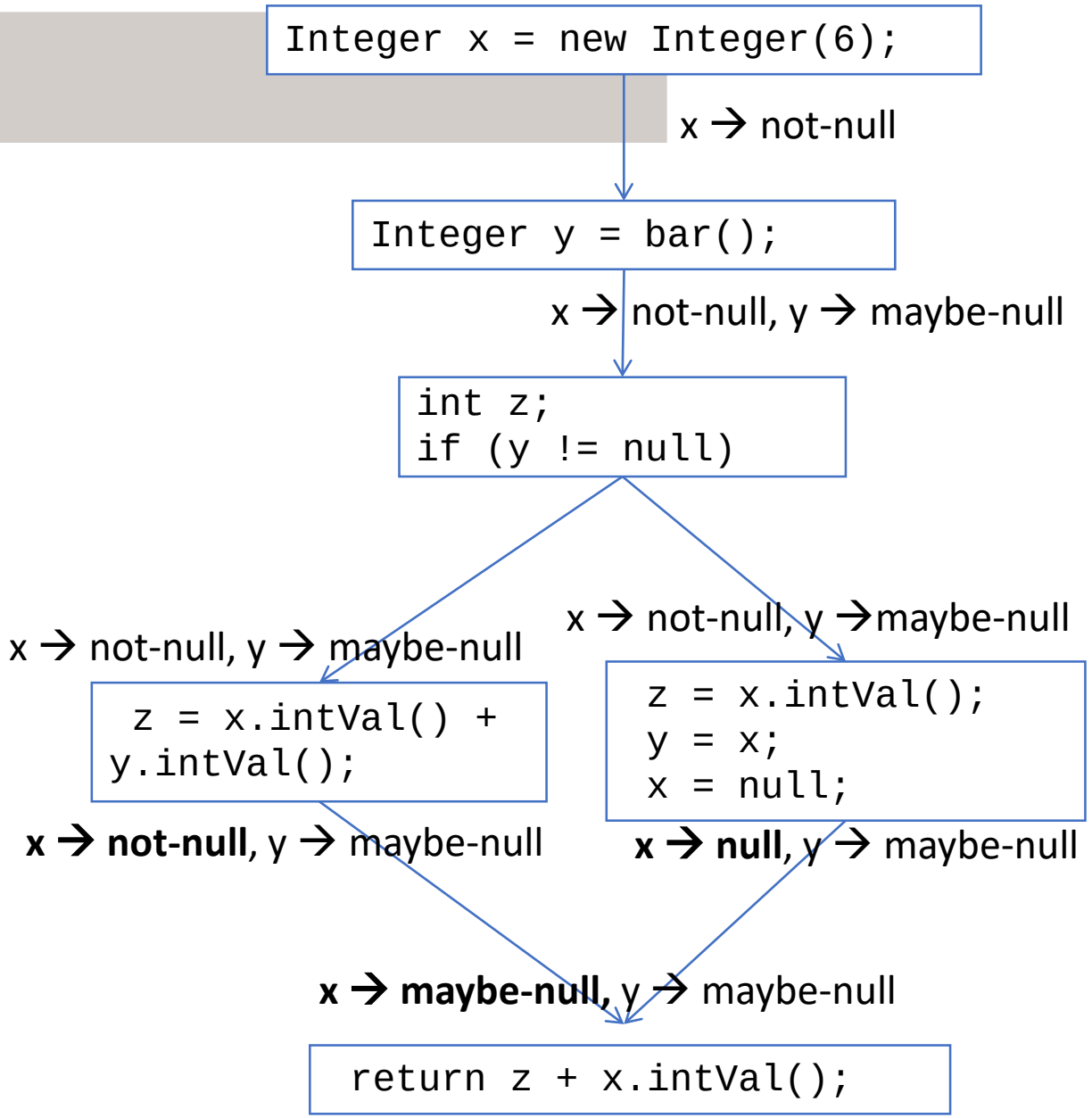
- Abstraction:
  - Program counter
  - 3 states for each variable: null, not-null, and maybe-null.

- Then check if, at each dereference, the analysis has identified whether the dereferenced variable is or might be null.

In graph form . . .

```
1.int foo() {
2.    Integer x = new Integer(6);
3.    Integer y = bar();
4.    int z;
5.    if (y != null)
6.       z = x.intVal() + y.intVal();
7.    } else {
8.       z = x.intVal();
9.       y = x;
10.      x = null;
11.    }
12.    return z + x.intVal();
13.}
```

Error: may have null pointer on line 12, because x may be null!



Examples of Data-Flow Analyses

- Null Analysis
  - Var -> {Null, NotNull, UNKNOWN}
- Zero Analysis
  - Var -> {Zero, NonZero, UNKNOWN}
- Sign Analysis
  - Var -> {-, +, 0, UNKNOWN}
- Range Analysis
  - Var -> {[0, 1], [1, 2], [0, 2], [2, 3], [0, 3], …, UNKNOWN}
- Constant Propagation
  - Var -> {1, 2, 3, …, UNKNOWN}
- File Analysis
  - File -> {Open, Close, UNKNOWN}
- Tons more!!!

Static Analysis vs. Testing

Which one to use when?

- Points in favor of Static Analysis
  - Don’t need to set up run environment, etc.
  - Can analyze functions/modules independently and in parallel
  - Don’t need to think of (or try to generate) program inputs

- Points in favor of Testing / Dynamic Analysis
  - Not deterred by complex program features
  - Can easily handle external libraries, platform-specific config, etc.
  - Ideally no false positives
  - Easier to debug when a failure is identified

facebook Engineering

Open Source | Platforms | Infrastructure Systems | Physical Infrastructure | Video Engineerin

POSTED ON MAY 2, 2018 TO DEVELOPER TOOLS, OPEN SOURCE

Sapienz: Intelligent automated software testing at scale

sapienz

By Ke Mao

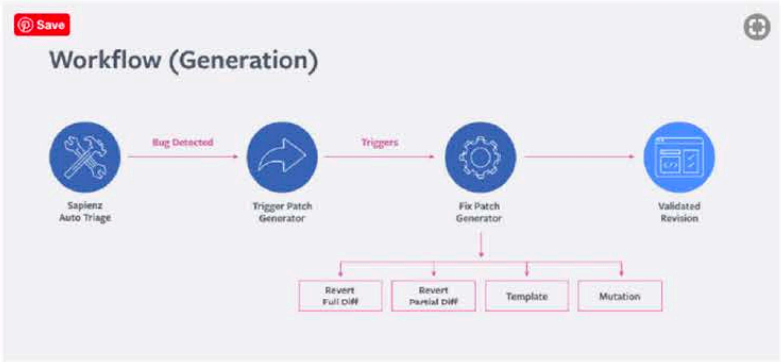
Sapienz technology leverages automated test design to make the testing process faster, more comprehensive, and more effective.

facebook Engineering

Open Source | Platforms | Infrastructure Systems | Physical Infrastructure | Video Engineering & AR/VR

POSTED ON SEP 13, 2018 TO AI RESEARCH, DEVELOPER TOOLS, OPEN SOURCE, PRODUCTION ENGINEERING

Finding and fixing software bugs automatically with SapFix and Sapienz



By Yue Jia Ke Mao Mark Harman

Debugging code is drudgery. But SapFix, a new AI hybrid tool created by Facebook engineers, can significantly reduce the amount of time engineers spend on debugging, while also speeding up the process of rolling out new software. SapFix can automatically generate fixes for specific bugs, and then propose them to engineers for approval and deployment to production.

SapFix has been used to accelerate the process of shipping robust, stable code updates to millions of devices using the Facebook Android app — the first such use of AI-powered testing and debugging tools in production at this scale. We intend to share SapFix with the engineering community, as it is the next step in the evolution of automating debugging, with the potential to boost the production and stability of new code for a wide range of companies and research organizations.

SapFix is designed to operate as an independent tool, able to run either with or without Sapienz, Facebook's intelligent automated software testing tool, which was announced at F8 and has already been deployed to production. In its current, proof-of-concept state,