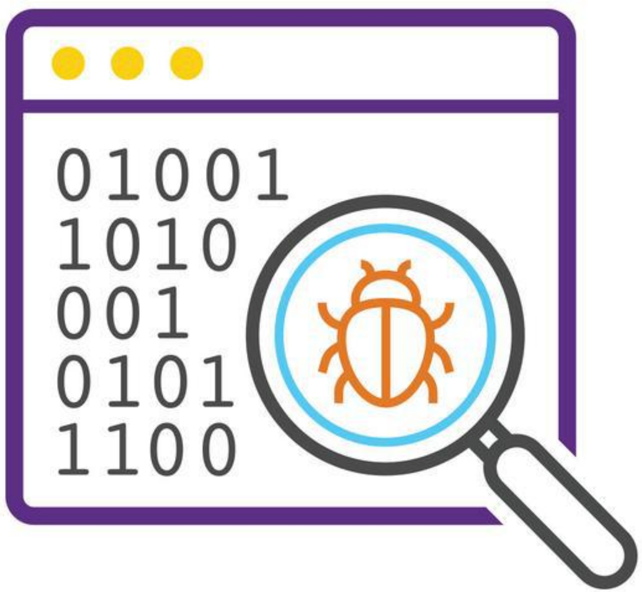
CS130: Software Engineering

Lecture 7 Static+Runtime Analysis



https://forms.gle/GTocHVJ1RuuM4M4TA

- A word: How’s life?
- A tweet: What might you want to know about a program *before* you run it? And why?

Assignment reminders

- Use same note sheet for all assignments
- TL should NOT be submitting any changes
- Project health is important!

Goals of this lecture

- Explain static and runtime analysis
- Discuss the pros and cons of the various approaches
- Show you how to use such analysis in your applications

Key skill: Be able to classify problems as being good candidates for static analysis vs runtime analysis

Static analysis

You’ve used static analysis before

Compiler warnings (ie, `-Wall`)

```
$ gcc -Wall foo.cc
foo.cc:10:2: error: 'my_var' defined but not used [-Werror=unused-variable]
```

Type checking in the compiler

```
$ gcc foo.cc
foo.cc:5:13: error: cannot convert ‘std::string {aka std::basic_string<char>}’ to ‘int’ in initialization
```

Linters

```
$ clang-tidy foo.cc
foo.cc:12:7: warning: this call will remove at most one item even when multiple items should be removed
```

Compiler warnings

First of all, you should use `-Wall` to enable lots of warnings and use `-Werror` to turn them into errors.

According to `man gcc`, `-Wall` turns on the following:

```
-Waddress -Warray-bounds (only with -O2) -Wc++0x-compat -Wchar-subscripts -Wenum-compare (in C/Objc; this is on by default in C++)
-Wimplicit-int (C and Objective-C only) -Wimplicit-function-declaration (C and Objective-C only) -Wcomment -Wformat -Wmain (only
for C/ObjC and unless -ffreestanding) -Wmissing-braces -Wnonnull -Wparentheses -Wpointer-sign -Wreorder -Wreturn-type
-Wsequence-point -Wsign-compare (only in C++) -Wstrict-aliasing -Wstrict-overflow=1 -Wswitch -Wtrigraphs -Wuninitialized
-Wunknown-pragmas -Wunused-function -Wunused-label -Wunused-value -Wunused-variable -Wvolatile-register-var
```

There are also tons of other warnings you could enable if you wanted to (eg, `-Wextra`):

```
-Wclobbered -Wempty-body -Wignored-qualifiers -Wmissing-field-initializers -Wmissing-parameter-type (C only)
-Wold-style-declaration (C only) -Woverride-init -Wsign-compare -Wtype-limits -Wuninitialized -Wunused-parameter
(only with -Wunused or -Wall) -Wunused-but-set-parameter (only with -Wunused or -Wall)
```

Static type checking

Compare:

Immutable

```
bool Sorted(const std::vector<int>& input) {
  const int* prev = nullptr;
  for (const auto& it : input) {
    if (prev && *prev > it) {
      return false;
    }
    prev = &it;
  }
  return true;
}
```

Infer type when not important

With:

Return type unknown

Immutable?

```
def Sorted(input):
  prev = None
  for it in input:
    if prev and prev > it:
      return false
    prev = it
  return true
```

- Static type checking gives you compile time errors about illegal operations
- This is opposed to runtime typed languages that only give you errors at runtime
- You can even do fancy things like type inference (`auto` in C++x11) at compile time

Linters

```
$ clang-tidy test.cc
```

- Lint was a tool originally developed alongside the C programming language.
- It was originally intended to help catch nonportable constructs.
- Often, you'll find a less pedantic linter built into compiler frontends.

Linters

```
// This is perfectly legal C++. Can you spot the bug?
{
  std::lock_guard<std::mutex>(&global_mutex);
  critical_section();
}

// Also legal, but is essentially a 'use-after-free'.
std::string str = "Hello, world!\n";
std::vector<std::string> messages;
messages.emplace_back(std::move(str));
std::cout << str;
```

- Linters may not seem like static analysis
- But, they correct more than style. Here are some examples:
  - Inaccurate erase/remove
  - Suspicious semicolon
  - Unused RAII
  - Use after move
- Many errors manifest as simple typos that are allowed by the compiler but are likely semantically wrong

What is static analysis?

- A process that inspects the code of your program without executing it directly
- Often will builds a control flow graph, though that isn't required
- Looks for patterns in that graph that represent likely problems

What is static analysis?

- As discussed, compiler warnings, type checking and linters are certainly forms of static analysis
- However, most people associate static analysis with a program (other than the compiler) that inspects you code for classes of bugs
- Often this checker is trying to disprove the existence of certain classes bugs in your code

Example: Use after free

```
// Allocate ‘a’ on the heap
int* a = new int;

// ‘a’ is still live, so this is OK
*a = 7;

// Free memory associated with ‘a’
delete a;

// Best case, this will trigger a SIGSEGV.
// Worst case, this will not trigger a SIGSEGV and silently
// do the wrong thing.
*a = 8;
```

- We can statically determine that ‘a’ isn’t live on the last line
- Value:
  - Helps us avoid SIGSEGVs
  - Helps avoid attacks that may be able to run malicious code by taking advantage of a use-after-free
- Yes, this example is super trivial

Example: Use after free

```
class Foo {
 public:
  Foo(int* a) a_(a) { … }
  set(int a) { *a_ = a; }
 private:
  int* a_;
};

Foo build() {
  int a;
  return Foo(&a);
}

void run() {
  // Best case, this will trigger a SIGSEGV.
  // Worst case, this silently do the wrong thing.
  build().set(7);
}
```

- … but this example is more complex
- In this case, **`int a`** isn't live at the point where it will be used in **`run()`**
- This is because **`Foo`** stored a pointer to **`int a`** as a member
- And then **`int a`** was freed when it went out of scope at the end of **`build()`**

Example: Buffer Overrun

```
// Wrong:
int a[10];
memset(a, 0, 100);
// This just stomped on 90*4 bytes past the end of ‘a’.

// Right:
int a[10];
memset(a, 0, sizeof(a));
```

- Reading or writing past the end of a buffer will produce undefined results
- Hopefully you run into a guard page and it causes a SIGSEGV
- Otherwise, it will just silently stomp on memory
- The compiler can often catch this when the size is known at compile time
- Much harder if it is dynamically sized

Example: Buffer Overrun

```
// Wrong:
int a[10];
memset(a, 0, 100);
// This just stomped on 90*4 bytes past the end of ‘a’.

// Right:
int a[10];
memset(a, 0, sizeof(a));

// Better:
int a[10] = {};
```

- Reading or writing past the end of a buffer will produce undefined results
- Hopefully you run into a guard page and it causes a SIGSEGV
- Otherwise, it will just silently stomp on memory
- The compiler can often catch this when the size is known at compile time
- Much harder if it is dynamically sized

An aside: How are Buffer Overruns Exploited?

```
// Wrong:
int a[10];
memset(a, 0, 100);
// This just stomped on 90*4 bytes past the end of 'a'.

// Right:
int a[10];
memset(a, 0, sizeof(a));

// More better:
int a[10] = {};
```

- Can you think of a way to use a buffer overrun to execute malicious code?
- Do you know of anything that can be done at runtime to defend against these sort of attacks?
- If interested in more, [Smashing the Stack for Fun and Profit]

Example: Deadlock Detection

```
Mutex mu1, mu2;

void foo() {
  mu1.Lock();
  mu2.Lock();
  mu2.Unlock();
  mu1.Unlock();
}

void bar() {
  mu2.Lock();
  mu1.Lock();
  mu1.Unlock();
  mu2.Unlock();
}

// It is not safe to run foo() and bar() concurrently
```

- Deadlock is when you have 2+ routines waiting on each other in a cycle.
- Order resource acquisition is one of a few ways to avoid deadlock (see also the famous dining philosophers problem)
- Static analyzers can detect if there is a reachable state where both **`foo()`** and **`bar()`** are both concurrently executing and waiting on each other

An aside: Lock Annotation

```
Mutex mu1, mu2;
int a GUARDED_BY(mu1);
int b GUARDED_BY(mu2);

void foo() REQUIRES(mu1, mu2) {
  a = 0;
  b = 0;
}

void test() {
  mu1.Lock();
  foo();         // Warning!  Requires mu2.
  mu1.Unlock();
}
```

- You can sometimes annotate your code to help the static analyzer better understand the intended behavior
- In the case of locks, you can explain what mutex guard which vars and then the static analyzer can check that invariant
- More about this in the threading lecture

Example: Uninitialized variable

```
// Wrong:
int a;
printf(“%d\n”, a);
// This printed random garbage from the stack.

// Right:
int a = 0;
printf(“%d\n”, a);
```

- Uninitialized vars can results in undefined behavior (the contents of that memory isn’t well defined).
- Assign-before-use is often easily identified by compilers or other static checks

Aside: Type checking `printf()`

```c
// Wrong:
int a;
printf("%d\n", a);
// This printed random garbage from the stack.

// Right:
int a = 0;
printf("%d\n", a);
```

- It is kinda crazy that the compiler will type check `printf()` for you.
- For one thing, it is a vararg function.
- It is part of stdlib, but checked by the compiler, which is at a different layer of abstraction.
- The compiler implementation needs to mirror stdlib implementation, which is annoying to keep in sync.
- That said, it is totally worth it because people always get it wrong!

Example: Dead code

```
const int kConstant = 7;

int foo = 5;
if (foo > kConstant) {
  // All this code is dead...
  [...]
}
```

- It is possible to prove that certain blocks are never reachable
- Here is a contrived example, but you can imagine this being arbitrarily complex
- In general, you can attempt to determine if a guard will always be false

How static analysis works

- In general, static analysis can be reduced to the halting problem, and therefore is undecidable in general.

- That is, the halting is a particular program property that you might want to compute statically, so by simple reduction the halting problem is a static analysis problem. Therefore, you can't compute all properties statically.

- But
  - We can produce approximates.
  - We can require assumptions or annotations to assist

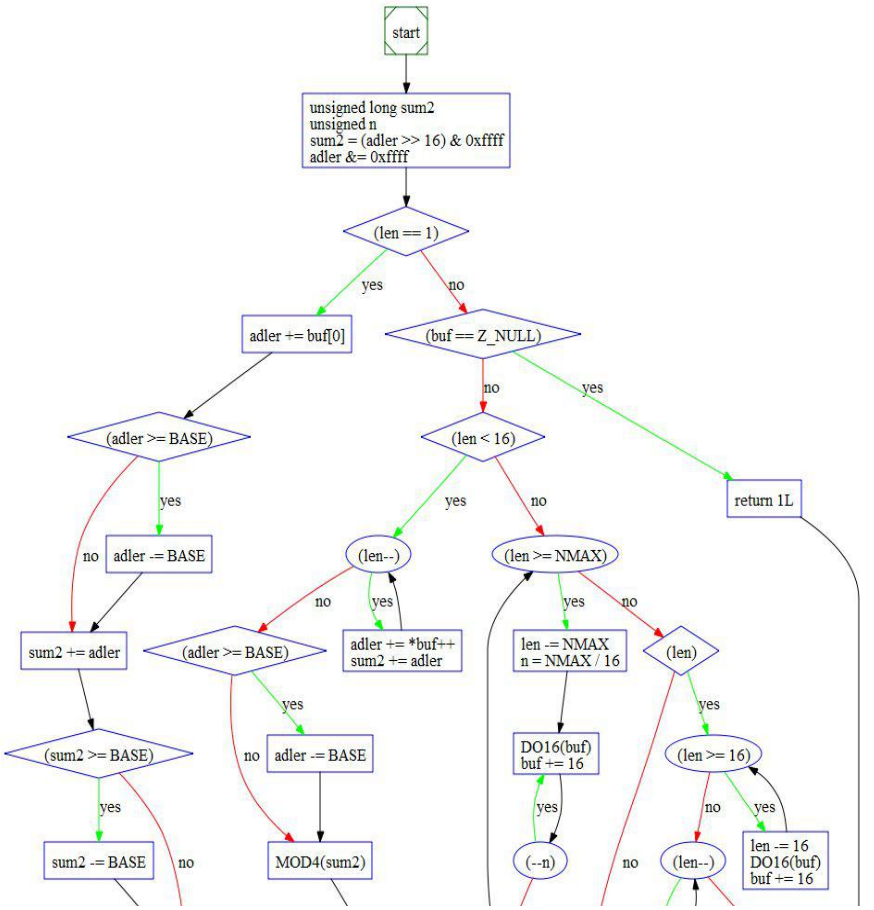
How static analysis works

Typical approaches:

- **Abstract interpretation:** model effect of statements on an abstract machine to identify mistakes

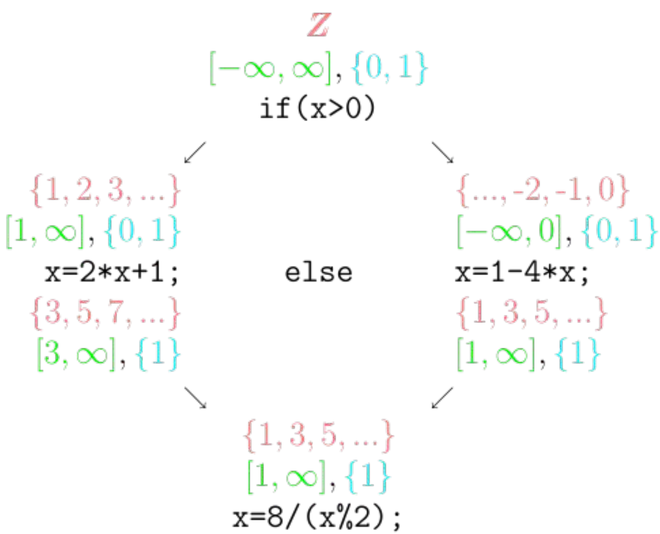
- **Data flow analysis:** attempt to determine possible input values based on the control flow graph (similar to general type inference in languages like ocaml)

Abstract Interpretation



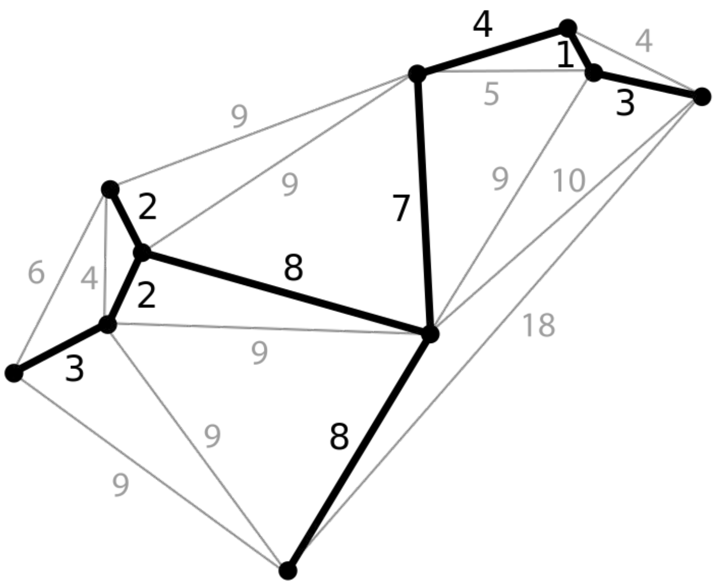
- Walk the control flow graph
- Keep track of things that are true when a given unit of code executes
- Determine if invariants are broken
- Example bugs you can catch:
  - Use after free
  - Uninitialized vars
  - Deadlock (if include concurrent execution)

Data Flow Analysis



- Keep track of the set of possible values a variable can take at a given point in the program
- Identify statements that break invariants for a possible value a variable could take on that point
- Example bugs you can catch:
  - buffer overrun
  - dead/unreachable code

Limitations of static analysis



- There are an exponential number of paths through a program
  → Keeping track for each requires an exponential amount of memory
- The range of possible inputs isn't limited much throughout the program
  → Results in false positives

Static Analysis: Overcoming limitations

- Since it's hard to prove/disprove things in static analysis, we often have to pick between low recall (too few bugs get caught) and low precision (correct code gets erroneously flagged)
- One solution: Extend the language

Static Analysis: Overcoming limitations

- Example: Java's `@VisibleForTesting` annotation
- In Java, best practices dictate that class methods should be private unless they need to be used outside of that class.
- Static analysis can flag when methods are non-private but also unused outside of the class context

```
// If no one outside of my
// class is using this method,
// generate a compile-time
// warning
public String getId() {
…
}
```

Static Analysis: Overcoming limitations

- What do you do if the only place you need access to the method is for unit testing purposes?
- Want to do 2 things:
  - Make the method public for use by the unit test
  - Maintain Java best practices and act like the method is private by ensuring it is not used anywhere else in the program

```
// Generate compile-time
// warning if this public
// method is used outside of
// either this class or unit
// tests
@VisibleForTesting
public String getId() {
…
}
```

Static Analysis: Overcoming limitations

- Another example: Typescript
- Javascript does not offer the syntax necessary to do static type checking
- What if you want to build an enterprise application using Javascript, but want the safety that static analysis can provide?

```
// p1, p2, and return types
// should be ints
function myFunction(p1, p2) {
  return p1 * p2;
}
```

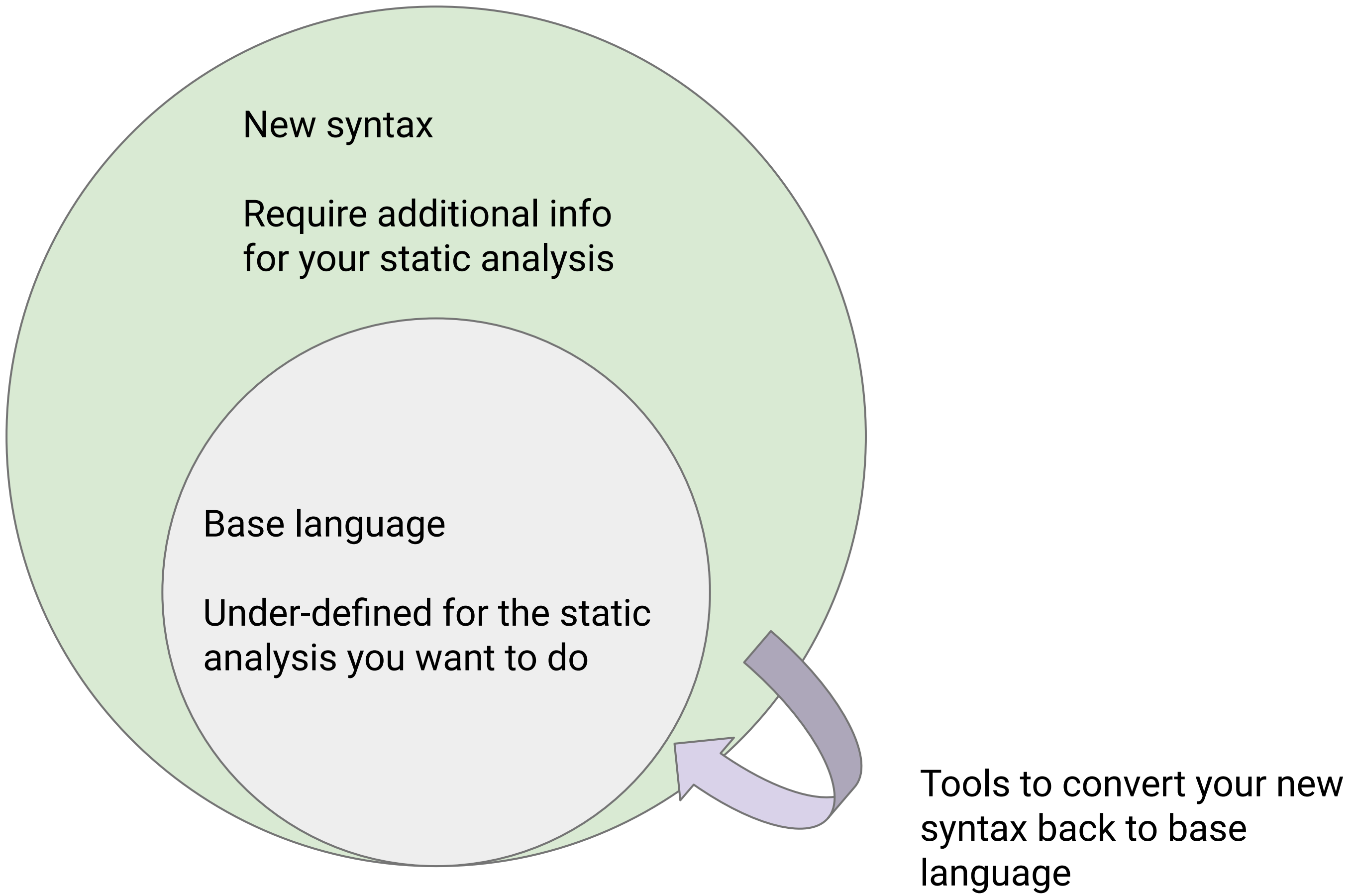
Static Analysis: Overcoming limitations

- Solution: Introduce new syntax
- Define “Typescript”, which is a superset of Javascript that can provide detailed typing information
- Then, provide tools:
    - Tool to perform static analysis
    - Tool to automatically generate Javascript from Typescript

In reality, this is the same tool:

The Typescript compiler

```
// p1, p2, and return types
// should be ints
function myFunction(
  p1: int,
  p2: int): int {
  return p1 * p2;
}
```


New syntax
Require additional info for your static analysis
Base language
Under-defined for the static analysis you want to do
Tools to convert your new syntax back to base language

Available static analysis tools

- type checking
- -Wall
- lint, clang-tidy
- gcc, clang
- findbugs
- coverity (not free)
- … and others

clang-format

- Uses LLVM's abstract syntax tree
- Allows automated reformatting of large swaths of code quickly and configurably
- Works with lots of languages
  - Java
  - C++
  - JavaScript
  - Python
  - ObjC

- Used by many large open/closed source projects
  - LLVM
  - Google
  - Chromium
  - Mozilla
  - Apple
  - WebKit
- How do you know this?
  - They implemented their own -styles :)

clang-format

- Doesn't really matter that much when you're the only one reading/writing the code. You might regret messy code later, but will probably still understand it.

- Does matter when **many many** people are all writing code in same project (possibly with differing opinions). Theory: code is read many more times than authored. Also, if your organization has gone through the effort making a style guide, it basically means the formatting matters / you might want to enforce it.

- Ends formatting arguments before they begin. "Just format it."

clang-format

[Live demo](#) ([SkittleParser](#))

FindBugs Demo

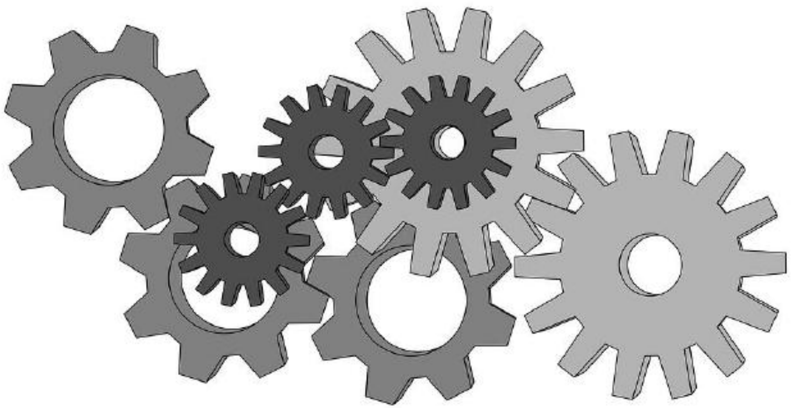
FindBugs (1.2.1-dev-20070506) Analysis for jdk1.7.0-b12

Bug Summary	Analysis Information	List bugs by bug category	List bugs by package

FindBugs Analysis generated at: Sun, 6 May 2007 03:12:12 -0400

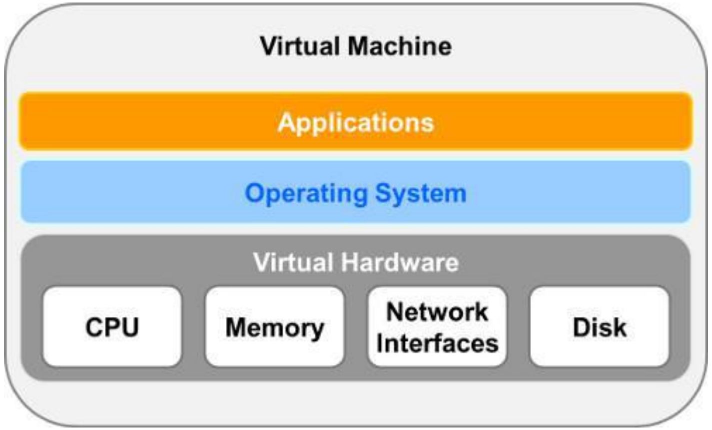
Package	Code Size	Bugs	Bugs p1	Bugs p2	Bugs p3	Bugs Exp.	Ratio
Overall (736 packages), (16445 classes)	963957	3901	259	3642			
com.sun.corba.se.impl.activation	1688	34	5	29			
com.sun.corba.se.impl.copyobject	71	1		1			
com.sun.corba.se.impl.corba	2118	33		33			
com.sun.corba.se.impl.dynamicany	2287	16	3	13			
com.sun.corba.se.impl.encoding	5652	55	1	54			
com.sun.corba.se.impl.interceptors	1979	41		41			
com.sun.corba.se.impl.io	3438	47	2	45			
com.sun.corba.se.impl.ior	1207	14	2	12			
com.sun.corba.se.impl.ior.iiop	457	4		4			
com.sun.corba.se.impl.javax.rmi.CORBA	337	3	1	2			
com.sun.corba.se.impl.logging	9374	8		8			
com.sun.corba.se.impl.naming.cosnaming	799	27	1	26			
com.sun.corba.se.impl.naming.pcosnaming	690	37	4	33			
com.sun.corba.se.impl.oa.poa	2102	31	1	30			
com.sun.corba.se.impl.orb	2324	46	2	44			
com.sun.corba.se.impl.orbutil	3795	25	3	22			
com.sun.corba.se.impl.orbutil.concurrent	320	4		4			
com.sun.corba.se.impl.orbutil.threadpool	357	8		8			
com.sun.corba.se.impl.presentation.rmi	1634	19	2	17			
com.sun.corba.se.impl.protocol	2133	15		15			
com.sun.corba.se.impl.protocol.giopmsgheaders	1861	13	1	12			
com.sun.corba.se.impl.resolver	299	1		1			
com.sun.corba.se.impl.transport	2266	24	1	23			

Runtime analysis



- Alternatively, you could just run the program in an instrumented runtime
- Then, just inspect what happened.
- Think of your brute force debugging sessions where you add `printf()`s until you find the issue.

How runtime analysis works

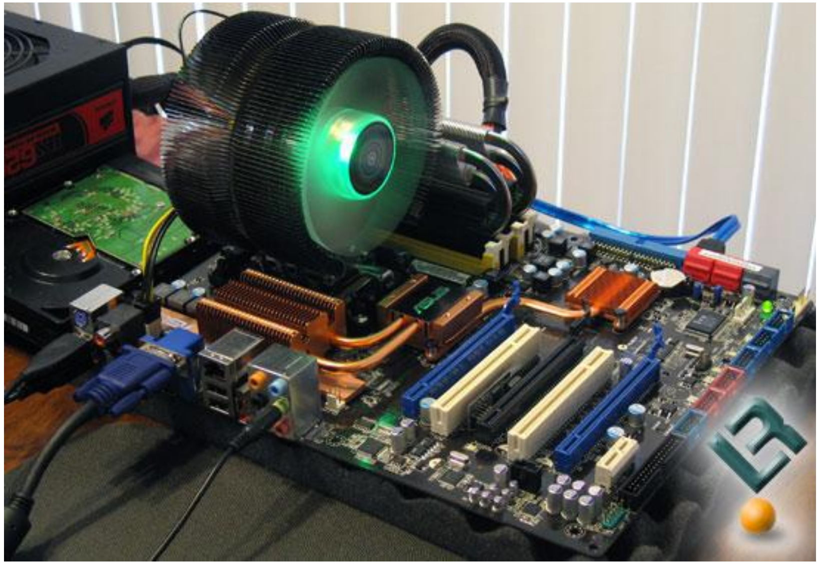


- Just a virtual machine that is checking for bad states (e.g., SIGSEGV, deadlock, etc.)
- Can keep track of real in-progress state, and pinpoint issues

Limitations of runtime analysis

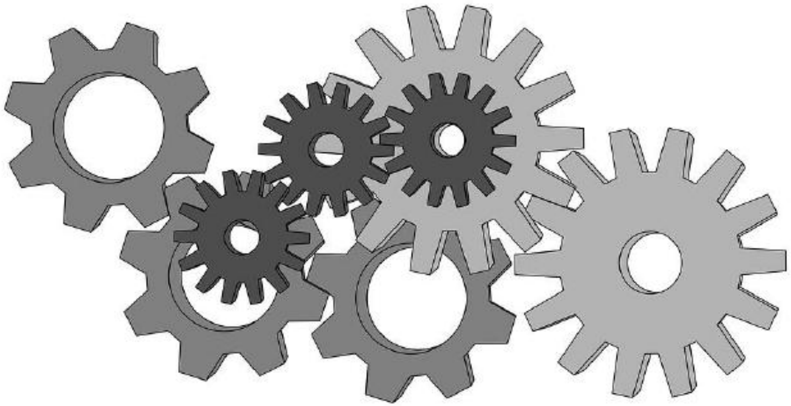
- Typically, this is slow
- That is, you can't run your code this way in prod (though not all that slow an absolute sense).
- Coverage is limited by test cases

Limitations of runtime analysis



- In some cases, hardware acceleration is available.
- For example, x86 has support for setting a breakpoint when a particular memory address is written to.
- This has virtually no impact on program execution speed.

Available runtime analysis tools



- gcov
- asan
- tsan
- gdb
- valgrind
- memcheck

Demo of gcov + lcov

```
int GreatestOfThree(int a,int b,int c) {
  if ((a > b) && (a > c)){ return a; }
  else if (b > c) { return b; }
  else { return c; }
  return 0;
}

TEST(GreaterTest,AisGreater){
  EXPECT_EQ(3, GreatestOfThree(3,1,2));
};

$ g++ -o main -fprofile-arcs
      -ftest-coverage main.cc
```

```
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from GreaterTest
[ RUN      ] GreaterTest.AisGreater
[       OK ] GreaterTest.AisGreater (0 ms)
[ RUN      ] GreaterTest.BisGreater
[       OK ] GreaterTest.BisGreater (0 ms)
[ RUN      ] GreaterTest.CisGreater
[       OK ] GreaterTest.CisGreater (0 ms)
[----------] 3 tests from GreaterTest (0 ms total)
[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 3 tests.
```

LCOV - code coverage report

Current view:	top level		Hit	Total	Coverage
Test:	new_coverage.info	**Lines:**	630	699	90.1 %
Date:	2017-09-26	**Functions:**	59	60	98.3 %

Directory	Line Coverage			Functions	
		90.1 %	630 / 699	98.3 %	59 / 60

Generated by: LCOV version 1.10

Reference: @naveen.maltesh on medium.com

Demo Video of gdb + Valgrind

```
$ valgrind ls
==211556== Memcheck, a memory error detector
==211556== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==211556== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==211556== Command: ls
==211556==
<normal output of ls>
==211556==
==211556== HEAP SUMMARY:
==211556==     in use at exit: 23,145 bytes in 18 blocks
==211556==   total heap usage: 55 allocs, 37 frees, 62,396 bytes allocated
==211556==
==211556== LEAK SUMMARY:
==211556==    definitely lost: 0 bytes in 0 blocks
==211556==    indirectly lost: 0 bytes in 0 blocks
==211556==      possibly lost: 0 bytes in 0 blocks
==211556==    still reachable: 23,145 bytes in 18 blocks
==211556==         suppressed: 0 bytes in 0 blocks
==211556== Rerun with --leak-check=full to see details of leaked memory
==211556==
==211556== For counts of detected and suppressed errors, rerun with: -v
==211556== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
$ valgrind ls
==211556== Memcheck, a memory error detector
==211556== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==211556== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==211556== Command: ls
==211556==
<normal output of ls>
==211556==
==211556== HEAP SUMMARY:
==211556==     in use at exit: 23,145 bytes in 18 blocks
==211556==   total heap usage: 55 allocs, 37 frees, 62,396 bytes allocated
==211556==
==211556== LEAK SUMMARY:
==211556==    definitely lost: 0 bytes in 0 blocks
==211556==    indirectly lost: 0 bytes in 0 blocks
==211556==      possibly lost: 0 bytes in 0 blocks
==211556==    still reachable: 23,145 bytes in 18 blocks
==211556==         suppressed: 0 bytes in 0 blocks
==211556== Rerun with --leak-check=full to see details of leaked memory
==211556==
==211556== For counts of detected and suppressed errors, rerun with: -v
==211556== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Static Analysis

- **Coverage:** Can cover a somewhat broad class of bugs.
- **Correctness:** Variable false positive rate, depending on bug in question.
- **Issues:** Inherently limited because it is difficult to check for new classes of bugs.

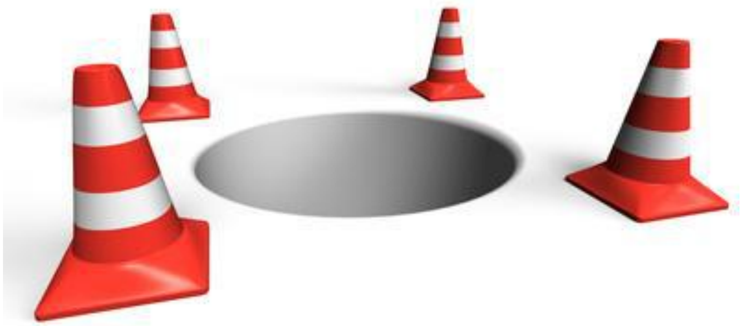
Runtime Analysis

- **Coverage:** Limited by your test coverage.
- **Correctness:** Detects problems that actually happened, not those that might happen.
- **Issues:** Often slow and cannot prove the absence of a bug or class of bugs.

The modern editor

- Modern editors have static analysis built in.
- They essentially have a compiler front end running in the background all the time.
- Can use this analysis to note compile errors (sorta handy).
- More importantly, they can even automate refactors and run static analysis.
- (Note: you can still get this stuff even if you prefer a shell-based editor)

Pitfalls



- Static analysis is no substitute for good coding practice.
- For example:
  - RAII (unique_ptr or scoped_lock) for acquiring resources.
  - Lock acquisition order to avoid deadlock
  - Initialize your variables
  - Code defensively

Let's Check Out:

A word: We can't even solve
The halting problem!
Is static analysis worth it?
A tweet: What tool interested
you the most? Why?