CMPT 473
Software Quality Assurance

Program Analysis Tools

Nick Sumner
Fixing bugs is costly

Why?
Fixing bugs is costly

- The longer broken code exists, the more code depends upon it.
Fixing bugs is costly

- The longer broken code exists, the more code depends upon it.
- Once developers have moved on, finding the root cause of a bug is difficult
Fixing bugs is costly

- The longer broken code exists, the more code depends upon it.
- Once developers have moved on, finding the root cause of a bug is difficult.
- Bugs that escape into the wild have real world impact
  - Unintended car acceleration
  - Spacecraft crashes
  - Security leaks
  - ...
Fixing bugs is costly

- Strategy so far:
  - Test to ensure that expected behaviors seem okay
Fixing bugs is costly

● Strategy so far:
  – Test to ensure that expected behaviors seem okay

Why do we still have bugs?
Fixing bugs is costly

• Strategy so far:
  – Test to ensure that expected behaviors seem okay
  – But we have seen that testing alone is a best effort process: no panacea in adequacy criteria
Fixing bugs is costly

• Strategy so far:
  – Test to ensure that expected behaviors seem okay
  – But we have seen that testing alone is a best effort process: no panacea in adequacy criteria

• Instead we can be proactive:
  – Explicitly search for certain known classes of bugs
Fixing bugs is costly

- **Strategy so far:**
  - Test to ensure that expected behaviors seem okay
  - But we have seen that testing alone is a best effort process: no panacea in adequacy criteria

- **Instead we can be proactive:**
  - Explicitly search for certain known classes of bugs
  - Guard against certain classes of bugs
Fixing bugs is costly

• Strategy so far:
  – Test to ensure that expected behaviors seem okay
  – But we have seen that testing alone is a best effort process: no panacea in adequacy criteria

• Instead we can be proactive:
  – Explicitly search for certain known classes of bugs
  – Guard against certain classes of bugs
  – Even prove that certain bugs are not present
Fixing bugs is costly

• Strategy so far:
  – Test to ensure that expected behaviors seem okay
  – But we have seen that testing alone is a best effort process: no panacea in adequacy criteria

• Instead we can be proactive:
  – Explicitly search for certain known classes of bugs
  – Guard against certain classes of bugs
  – Even prove that certain bugs are not present
  – Identify bad styles that may lead to bugs
How can we do this?

- Increasingly pervasive approach is to use program analysis
How can we do this?

- Increasingly pervasive approach is to use *program analysis*
  - Set of tools/techniques that allow computers to automatically reason about the behavior of programs
How can we do this?

- Increasingly pervasive approach is to use *program analysis*
  - Set of tools/techniques that allow computers to automatically reason about the behavior of programs

- Push the burden of understanding programs onto computers
How can we do this?

- Increasingly pervasive approach is to use program analysis
  - Set of tools/techniques that allow computers to automatically reason about the behavior of programs

- Push the burden of understanding programs onto computers
  - People have trouble with repetitive, subtle behavior
How can we do this?

- Increasingly pervasive approach is to use *program analysis*
  - Set of tools/techniques that allow computers to automatically reason about the behavior of programs
- Push the burden of understanding programs onto computers
  - People have trouble with repetitive, subtle behavior
  - Computers excel at it
For example

```c
if ((err = update(&ctx, &server)) != 0)
goto fail;
if ((err = update(&ctx, &params)) != 0)
goto fail;
goto fail;
if ((err = final(&ctx, &hashOut)) != 0)
goto fail;
```
For example

```c
if ((err = update(&ctx, &server)) != 0)
goto fail;
if ((err = update(&ctx, &params)) != 0)
goto fail;
goto fail;
if ((err = final(&ctx, &hashOut)) != 0)
goto fail;
```

Why is this difficult for people to catch?
For example

```c
if ((err = update(&ctx, &server)) != 0)
goto fail;
if ((err = update(&ctx, &params)) != 0)
goto fail;
goto fail;
if ((err = final(&ctx, &hashOut)) != 0)
goto fail;
```

Why is this difficult for people to catch?

Why should a computer be able to find it?
For example

```c
if ((err = update(&ctx, &server)) != 0)
goto fail;
if ((err = update(&ctx, &params)) != 0)
goto fail;
    goto fail;
if ((err = final(&ctx, &hashOut)) != 0)
goto fail;
```

- There are bugs that people can miss but that computers can easily find.
  - *Rules* can determine what is buggy or not
For example

```c
if ((err = update(&ctx, &server)) != 0)
goto fail;
if ((err = update(&ctx, &params)) != 0)
goto fail;
goto fail;
if ((err = final(&ctx, &hashOut)) != 0)
goto fail;
```

- There are bugs that people can miss but that computers can easily find.
  - **Rules** can determine what is buggy or not

**BUG:** Both branches of the if statement have the same target
Two main categories of tools

- *Dynamic analysis* tools
  - Run the program and reason about that *single execution*
Two main categories of tools

- **Dynamic analysis** tools
  - Run the program and reason about that *single execution*
  - Best at helping explain bugs that are already occurring
Two main categories of tools

- **Dynamic analysis** tools
  - Run the program and reason about that single execution
  - Best at helping explain bugs that are already occurring

- **Static analysis** tools
  - Examine the source code or binary and reason about all possible executions
Two main categories of tools

- **Dynamic analysis** tools
  - Run the program and reason about that single execution
  - Best at helping explain bugs that are already occurring

- **Static analysis** tools
  - Examine the source code or binary and reason about all possible executions
  - Best at identifying bugs that haven't struck yet but might in the future
Two main categories of tools

- Neither approach is perfect
Two main categories of tools

- Neither approach is perfect

What are the limitations of dynamic approaches?
Two main categories of tools

• Neither approach is perfect

What are the limitations of dynamic approaches?

What are the limitations of static approaches?

This one is tougher....
Two main categories of tools

- Neither approach is perfect
  - Dynamic approaches require a test case to analyze
Two main categories of tools

• Neither approach is perfect
  – Dynamic approaches require a test case to analyze
  – Static approaches are limited by the halting problem

  The halting problem strikes *again*....
Two main categories of tools

- Neither approach is perfect
  - Dynamic approaches require a test case to analyze
  - Static approaches are limited by the halting problem

- The results are imperfect
  - **False positives** – Warnings about bugs that don't actually exist
Two main categories of tools

- Neither approach is perfect
  - Dynamic approaches require a test case to analyze
  - Static approaches are limited by the halting problem
- The results are imperfect
  - False positives – Warnings about bugs that don't actually exist
  - False negatives – Missing warnings for bugs that do exist
Two main categories of tools

- Neither approach is perfect
  - Dynamic approaches require a test case to analyze
  - Static approaches are limited by the halting problem
- The results are imperfect
  - False positives – Warnings about bugs that don't actually exist
  - False negatives – Missing warnings for bugs that do exist
- Learning how to use these tools effectively can take practice
But what can they actually do?

- Many tools are freely available:
  - *Lint
  - FindBugs
  - Clang Static Analyzer
  - ESC/Java
  - Valgrind
  - Clang Sanitizers
  - ... (and more on the course web page)
Taking a look at Valgrind

- Valgrind
  - Uses *dynamic binary instrumentation*
Taking a look at Valgrind

- Valgrind
  - Uses dynamic binary instrumentation
  - Modifies an already compiled binary to check for errors
Taking a look at Valgrind

- Valgrind
  - Uses dynamic binary instrumentation
  - Modifies an already compiled binary to check for errors
  - Many built in tools
    - Memcheck – memory safety analyses
    - Cachegrind – performance analyses
    - Helgrind & DRD – Thread safety analyses
Taking a look at Valgrind

- Valgrind
  - Uses dynamic binary instrumentation
  - Modifies an already compiled binary to check for errors
  - Many built in tools
    - Memcheck – memory safety analyses
    - CacheGrind – performance analyses
    - Helgrind & DRD – Thread safety analyses
  - Used extensively in the real world
    - http://valgrind.org/gallery/
Taking a look at Valgrind

- Valgrind
  - Uses dynamic binary instrumentation
  - Modifies an already compiled binary to check for errors
  - Many built in tools
    - Memcheck – memory safety analyses
    - CacheGrind – performance analyses
    - Helgrind & DRD – Thread safety analyses
  - Used extensively in the real world
    - http://valgrind.org/gallery/

Does not work for Java or Python by default. Why?!
Taking a look at clang sanitizers

- Clang sanitizers
  - Use *compile time instrumentation*
Taking a look at clang sanitizers

• Clang sanitizers
  – Use compile time instrumentation
  – Rewrites the program once to perform analyses every time it executes
  – Able to exploit source level information
Taking a look at clang sanitizers

- **Clang sanitizers**
  - Use compile time instrumentation
  - Rewrites the program once to perform analyses every time it executes
  - Able to exploit source level information
  - Many built in tools
    - AddressSanitizer – Address safety analysis
    - MemorySanitizer – Defined value analysis
    - ThreadSanitizer – Thread safety analysis
    - Undefined Behavior – Just what it sounds like (which is?)
Taking a look at clang sanitizers

- Clang sanitizers
  - Use compile time instrumentation
  - Rewrites the program once to perform analyses every time it executes
  - Able to exploit source level information
  - Many built in tools
    - AddressSanitizer – Address safety analysis
    - MemorySanitizer – Defined value analysis
    - ThreadSanitizer – Thread safety analysis
    - Undefined Behavior – Just what it sounds like
  - Used extensively at google (chrome, ...)

So far...

- We've looked at dynamic analysis tools.
  - *False positives* are less common
  - *False negatives* are inherent
So far...

- We've looked at dynamic analysis tools.
  - *False positives* are less common
  - *False negatives* are inherent
- What about the static analysis tools?
Clang static analyzer

- 'scan-build'
  - Integrates into the build process
Clang static analyzer

- 'scan-build'
  - Integrates into the build process
  - Uses abstract interpretation to simulate many different paths through the program at once
Clang static analyzer

• 'scan-build'
  – Integrates into the build process
  – Uses *abstract interpretation* to simulate many different paths through the program at once
  – Generates summaries showing exactly how errors *may* occur
Clang static analyzer

• 'scan-build'
  – Integrates into the build process
  – Uses *abstract interpretation* to simulate many different paths through the program at once
  – Generates summaries showing exactly how errors *may* occur
  – Many automatically recognized bugs
    • And a plug-in system for recognizing new ones.
Clang static analyzer

- 'scan-build'
  - Integrates into the build process
  - Uses *abstract interpretation* to simulate many different paths through the program at once
  - Generates summaries showing exactly how errors *may* occur
  - Many automatically recognized bugs
    - And a plug-in system for recognizing new ones.
  - Poorly organized & asserted code yields many errors
FindBugs

- FindBugs
  - Static analysis of Java bytecode
FindBugs

- Static analysis of Java bytecode
- Uses several techniques to balance speed, precision, false positives, and false negatives
FindBugs

- Static analysis of Java bytecode
- Uses several techniques to balance speed, precision, false positives, and false negatives
- Emphasis on pragmatic, actionable results
- Massive variety of bugs & code smells detected
FindBugs

- Static analysis of Java bytecode
- Uses several techniques to balance speed, precision, false positives, and false negatives
- Emphasis on pragmatic, actionable results
- Massive variety of bugs & code smells detected

I would argue that a Java project not using FindBugs is a broken project!
Dealing With False Information

- False *negatives* are unfortunate, but no extra burden
Dealing With False Information

- False *negatives* are unfortunate, but no extra burden
- False *positives* can waste developer time
  - Like chasing ghosts through the source code

You must eventually figure out that the ghost isn't real
Dealing With False Information

- False *negatives* are unfortunate, but no extra burden
- False *positives* can waste developer time
  - Like chasing ghosts through the source code
  - Want to determine whether warnings are real

This takes a lot of work & happens every time. Can we do better?
Dealing With False Information

- False *negatives* are unfortunate, but no extra burden
- False *positives* can waste developer time
  - Like chasing ghosts through the source code
  - Want to determine whether warnings are real
  - Avoid chasing this same ghost in the future!
Dealing With False Information

- False *negatives* are unfortunate, but no extra burden

- False *positives* can waste developer time
  - Like chasing ghosts through the source code
  - Want to determine whether warnings are real
  - Avoid chasing this same ghost in the future!

Blacklisting & suppression allows us to “remember” false positives & prevent them in the future....

[DEMO]
Verification

• The tools so far try to look for bugs
  – They can still miss them [Clang SA DEMO]
Verification

- The tools so far try to look for bugs
  - They can still miss them [Clang SA DEMO]
- In contrast, we can try to use verification to prove the absence of (certain types of) bugs.

Have you seen / heard of such tools before?
Verification

- The tools so far try to look for bugs
  - They can still miss them [Clang SA DEMO]
- In contrast, we can try to use verification to prove the absence of (certain types of) bugs.
  - [CBMC DEMO]
Verification

- The tools so far try to look for bugs
  - They can still miss them [Clang SA DEMO]
- In contrast, we can try to use *verification* to *prove the absence* of (certain types of) bugs.
  - [CBMC DEMO]
- Why didn't we just do this from the beginning?

Any ideas?
Verification

- The tools so far try to look for bugs
  - They can still miss them [Clang SA DEMO]
- In contrast, we can try to use verification to prove the absence of (certain types of) bugs.
  - [CBMC DEMO]
- Why didn't we just do this from the beginning?
  - Historically more difficult to use
Verification

• The tools so far try to look for bugs
  – They can still miss them [Clang SA DEMO]

• In contrast, we can try to use verification to prove the absence of (certain types of) bugs.
  – [CBMC DEMO]

• Why didn't we just do this from the beginning?
  – Historically more difficult to use
  – Historically more complex → more overhead
Verification

- The tools so far try to look for bugs
  - They can still miss them [Clang SA DEMO]
- In contrast, we can try to use verification to **prove the absence** of (certain types of) bugs.
  - [CBMC DEMO]
- Why didn't we just do this from the beginning?
  - Historically more difficult to use
  - Historically more complex → more overhead
  - Still approximate, at some level (time, space, ...)
    - They'll still miss bugs in the end
Verification

- The tools so far try to look for bugs
  - They can still miss them [Clang SA DEMO]
- In contrast, we can try to use **verification** to prove the absence of (certain types of) bugs.
  - [CBMC DEMO]
- Why didn't we just do this from the beginning?
  - Historically more difficult to use
  - Historically more complex → more overhead
  - Still approximate, at some level (time, space, ...)
    - But they are getting better!
    - Used extensively in safety critical systems.
  - They'll still miss bugs in the end