QA: Static Analysis

Michael Hilton and **Rohan Padhye**

Administrativia

- HW4 Part D is due tonight
- HW5 will be released soon
  - HW5 Part A is based on static/dynamic analysis tools and CI
  - HW5 Part B is based on ML explainability (covered Thursday)

Learning goals

- Give a one sentence definition of static analysis. Explain what types of bugs static analysis targets.
- Give an example of syntactic or structural static analysis.
- Construct basic control flow graphs for small examples by hand.
- Give a high-level description of dataflow analysis and cite some example analyses.
- Explain at a high level why static analyses cannot be sound, complete, and terminating; assess tradeoffs in analysis design.
- Characterize and choose between tools that perform static analyses.
- Contrast static analysis tools with software testing and dynamic analysis tools as a means of catching bugs.

What smells?

```java
class Foo {
    int a; int b;

    public boolean equals(Object other) {
        Foo foo = (Foo) other;
        if (foo != null)
          if (foo.a != this.a)
            return false;
          if (foo.b == this.b)
            return true;
          else return false;
    }

    public int a() {
        return this.a();
    }

    public int b() {
        return this.b();
    }
}
```

What smells?

```c
int dtls1_process_heartbeat(SSL *s)
    {
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */

    /* Read type and payload length first */
    hbtype = *p++;
    n2s(p, payload);
    pl = p;

    if (s->msg_callback)
        s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
            &s->s3->rrec.data[0], s->s3->rrec.length,
            s, s->msg_callback_arg);

    if (hbtype == TLS1_HB_REQUEST)
        {
        unsigned char *buffer, *bp;
        int r;

        /* Allocate memory for the response, size is 1 byte
         * message type, plus 2 bytes payload length, plus
         * payload, plus padding
         */
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;

        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);
        memcpy(bp, pl, payload);
        bp += payload;
        /* Random padding */
        RAND_pseudo_bytes(bp, padding);

        r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
```

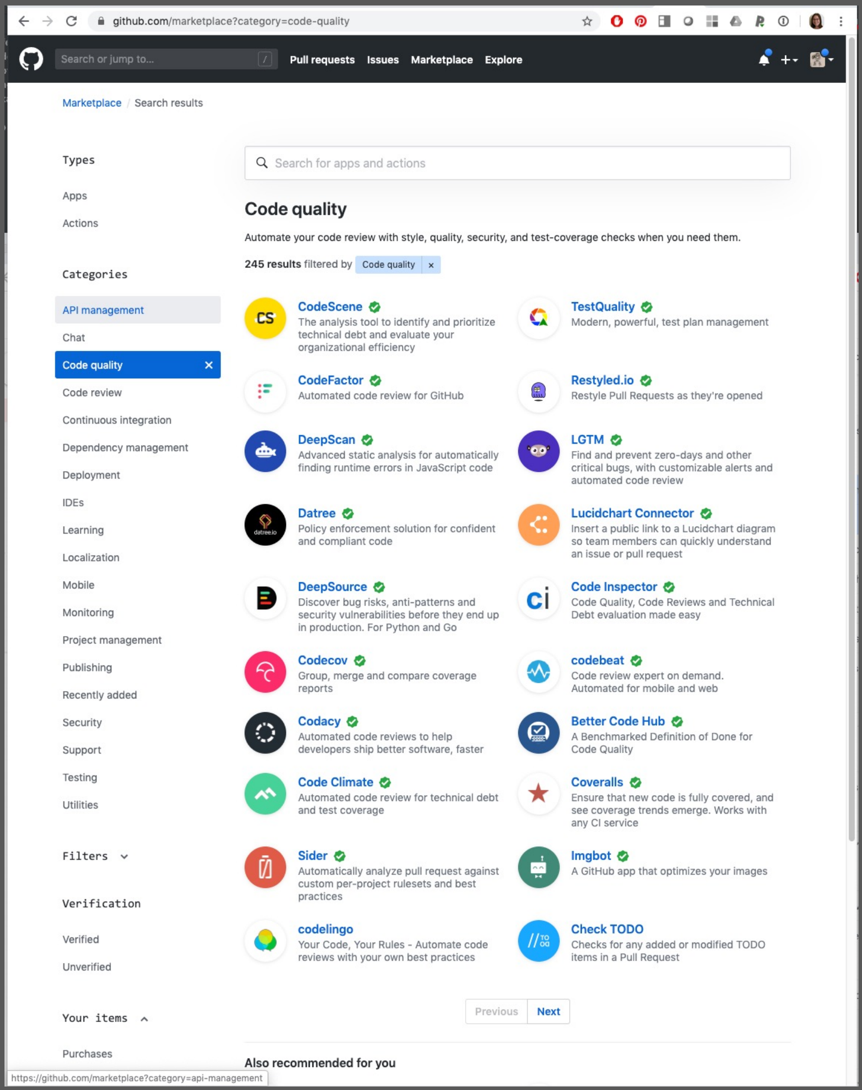
Static Analysis

- 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.

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

Key: check compliance to simple, mechanical design rules



https://github.com/marketplace?category=code-quality

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

public class Test {
```

Lint — Java — 1:02 AM, Aug 21

Missing a Javadoc comment.

Please fix   Not useful

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

ErrorProne — StringEquality — 1:03 AM, Aug 21

String comparison using reference equality instead of value equality (see http://code.google.com/p/error-prone/wiki/StringEquality)

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;	package com.google.devtools.staticanalysis;
	import java.util.Objects;
public class Test {	public class Test {
public boolean foo() {	public boolean foo() {
return getString() == "foo".toString();	return Objects.equals(getString(), "foo".toString());
}	}
public String getString() {	public String getString() {
return new String("foo");	return new String("foo");
}	}
}	}

Apply   Cancel

How do they work?

```java
class Foo {
    int a; int b;

    public boolean equals(Ob
        Foo foo = (Foo) othe
        if (foo != null)
            if (foo.a != this.a
                return false;
            if (foo.b == this.
                return true;
            else return false;
    }

    public int a() {
        return this.a();
    }

    public int b() {
        return this.b();
    }
}
```

```c
int dtls1_process_heartbeat(SSL *s)
    {
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */

    /* Read type and payload length first */
    hbtype = *p++;
    n2s(p, payload);
    pl = p;

    if (s->msg_callback)
        s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
            &s->s3->rrec.data[0], s->s3->rrec.length,
            s, s->msg_callback_arg);

    if (hbtype == TLS1_HB_REQUEST)
        {
        unsigned char *buffer, *bp;
        int r;

        /* Allocate memory for the response, size is 1 byte
         * message type, plus 2 bytes payload length, plus
         * payload, plus padding
         */
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;

        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);
        memcpy(bp, pl, payload);
        bp += payload;
        /* Random padding */
        RAND_pseudo_bytes(bp, padding);

        r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
```

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)

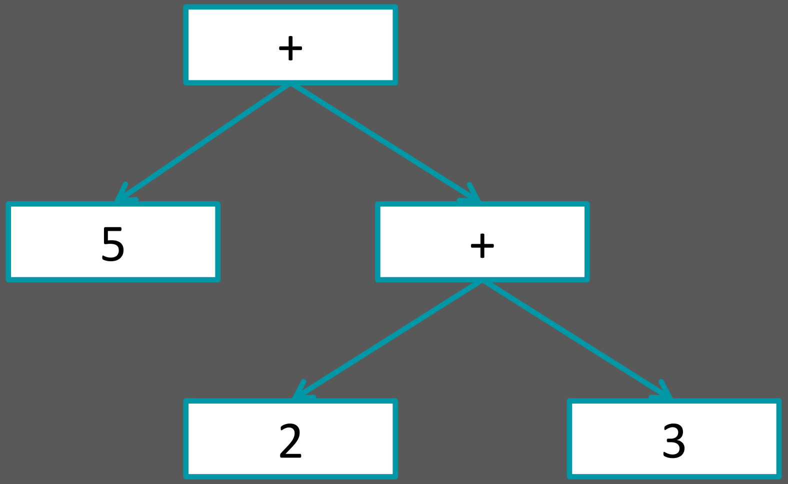
SIMPLE SYNTACTIC AND STRUCTURAL ANALYSES

Type Analysis

```java
public void foo() {
    int a = computeSomething();

    if (a == "5")
        doMoreStuff();
}
```

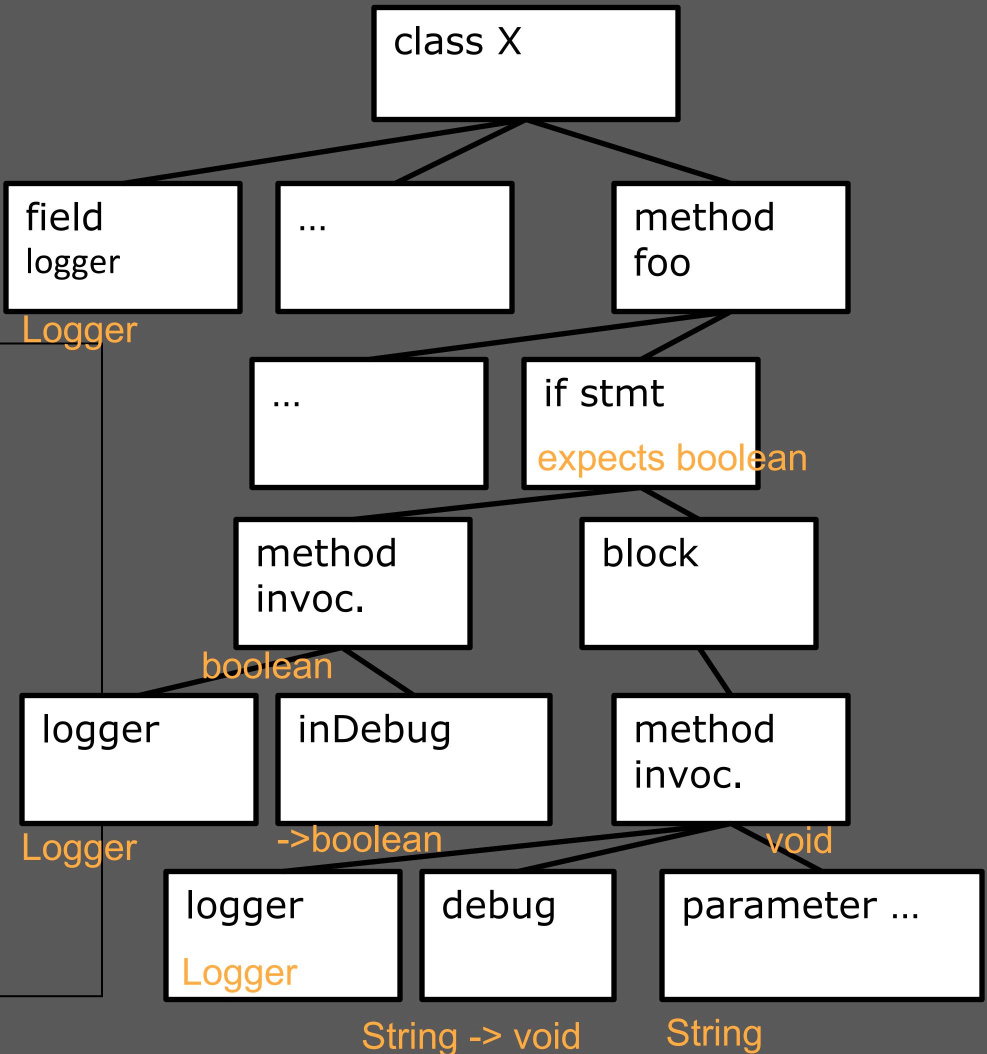
Abstraction: abstract syntax tree

- Tree representation of the syntactic structure of source code.
  - Parsers convert concrete syntax into abstract syntax, and deal with resulting ambiguities.
- Records only the semantically relevant information.
  - Abstract: doesn't represent every detail (like parentheses); these can be inferred from the structure.
- (How to build one? Take compilers!)



Type checking

```
class X {
  Logger logger;
  public void foo() {
    …
    if (logger.inDebug()) {
      logger.debug("We have " + conn +
"connections.");
    }
  }
}
class Logger {
   boolean inDebug() {…}
   void debug(String msg) {…}
}
```



Syntactic Analysis

Find every occurrence of this pattern:

```
public foo() {
  …
   logger.debug(“We have ” + conn + “connections.”);
}
```

```
public foo() {
  …
  if (logger.inDebug()) {
    logger.debug(“We have ” + conn + “connections.”);
  }
}
```

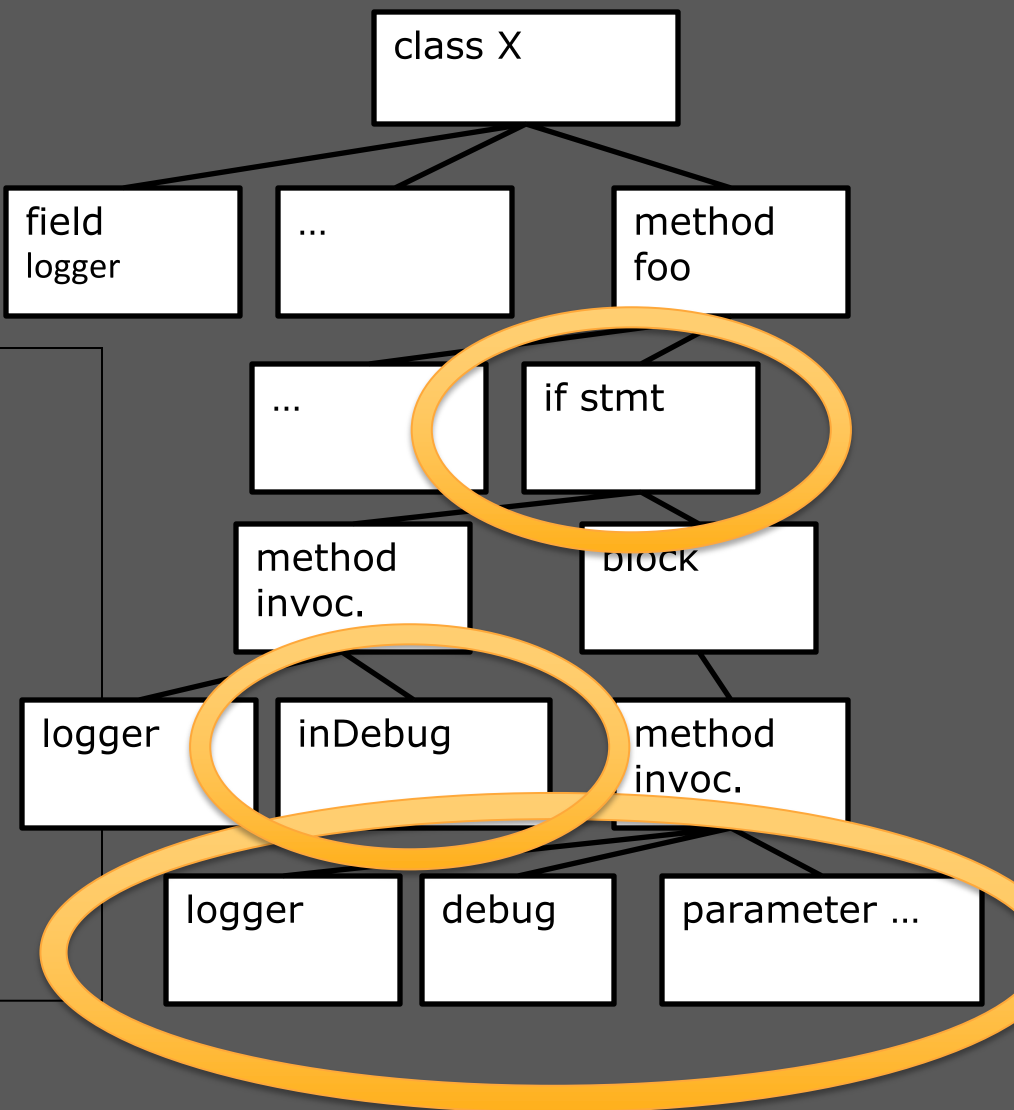
grep "if \(logger\.inDebug" . -r

Abstract syntax tree walker

- Check that we don't create strings outside of a `Logger.inDebug` check
- Abstraction:
  - Look only for calls to `Logger.debug()`
  - Make sure they're all surrounded by if (`Logger.inDebug()`)
- Systematic: Checks all the code
- Known as an Abstract Syntax Tree (AST) walker
  - Treats the code as a structured tree
  - Ignores control flow, variable values, and the heap
  - Code style checkers work the same way

Structural Analysis

```
class X {
  Logger logger;
  public void foo() {
    …
    if (logger.inDebug()) {
      logger.debug("We have " + conn + 
"connections.");
    }
  }
}
class Logger {
   boolean inDebug() {…}
   void debug(String msg) {…}
}
```



Structural analysis for possible NPEs?

```
if (foo != null)
    foo.a();
foo.b();
```

Which of these should be flagged for NPE?

Surely safe? Surely bad? Suspicious? // Limitations of structural analysis

A

```
if (foo != null)
    foo.a();
foo.b();
```

B

```
if (foo == null)
    foo = new Foo();
foo.b();
```

C

```
if (foo != null)
    foo.a();
else
    foo = new Foo();

foo.b();
```

D

```
if (foo != null)
    foo.a();
else
    foo.b();
```

CONTROL-FLOW AND DATA-FLOW ANALYSIS

Control/Dataflow analysis

- **Reason** about all possible executions, via paths through a *control flow graph*.
  - Track information relevant to a property of interest at every *program point*.
- Define an **abstract domain** that captures only the values/states relevant to the property of interest.
- **Track** the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.

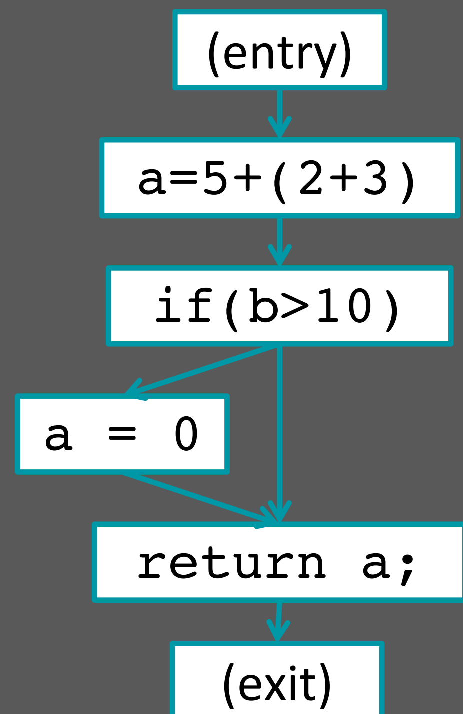
Control/Dataflow analysis

- **Reason** about all possible executions, via paths through a *control flow graph*.
  - Track information relevant to a property of interest at every *program point*.
- Define an **abstract domain** that captures only the values/states relevant to the property of interest.
- **Track** the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.

Control flow graphs

- A tree/graph-based representation of the flow of control through the program.
  - Captures all possible execution paths.
- Each node is a basic block: no jumps in or out.
- Edges represent control flow options between nodes.
- *Intra-procedural*: within one function.
  - cf. inter-procedural

```
a = 5 + (2 + 3)
if (b > 10) {
   a = 0;
}
return a;
```



How can CFG be used to identify this issue?

```java
public int foo() {
    doStuff();

    return 3;

    doMoreStuff();
    return 4;
}
```

Control/Dataflow analysis

- **Reason** about all possible executions, via paths through a *control flow graph*.
  - Track information relevant to a property of interest at every program point.
- Define an **abstract domain** that captures only the values/states relevant to the property of interest.
- **Track** the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.

NPE analysis revisited

A

```
if (foo != null)
    foo.a();
foo.b();
```

B

```
if (foo == null)
    foo = new Foo();
foo.b();
```

C

```
if (foo != null)
    foo.a();
else
    foo = new Foo();

foo.b();
```

D

```
if (foo != null)
    foo.a();
else
    foo.b();
```

Abstract Domain for NPE Analysis

- Map of `Var -> {Null, NotNull, Unknown}`

- For example:
```
foo -> Null
bar -> NonNull
baz -> Unknown
```

- Mapping tracked at every program point (before/after each CFG node). Updated across nodes and edges.

- // let's say `foo -> Null` and `bar->Null`
  `foo = new Foo();`
  // at this point, we have `foo -> NotNull` and `bar -> Null`

NPE analysis revisited

A

```
if (foo != null)
    foo.a();
foo.b();
```

B

```
if (foo == null)
    foo = new Foo();
foo.b();
```

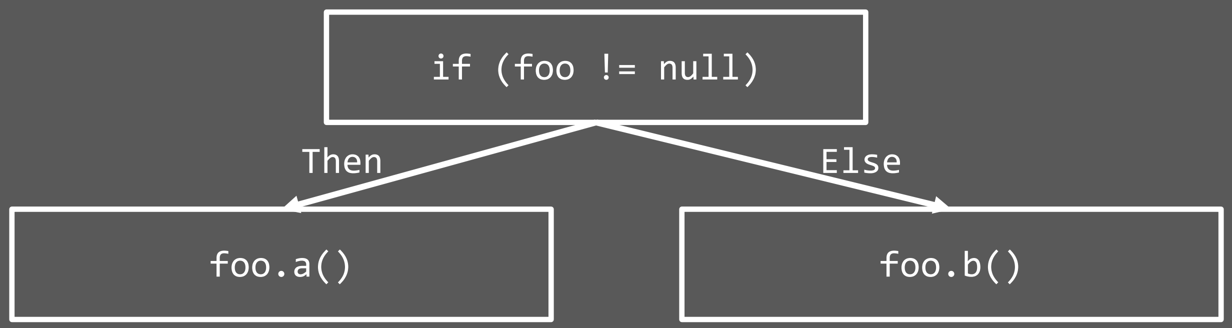
C

```
if (foo != null)
    foo.a();
else
    foo = new Foo();

foo.b();
```

D

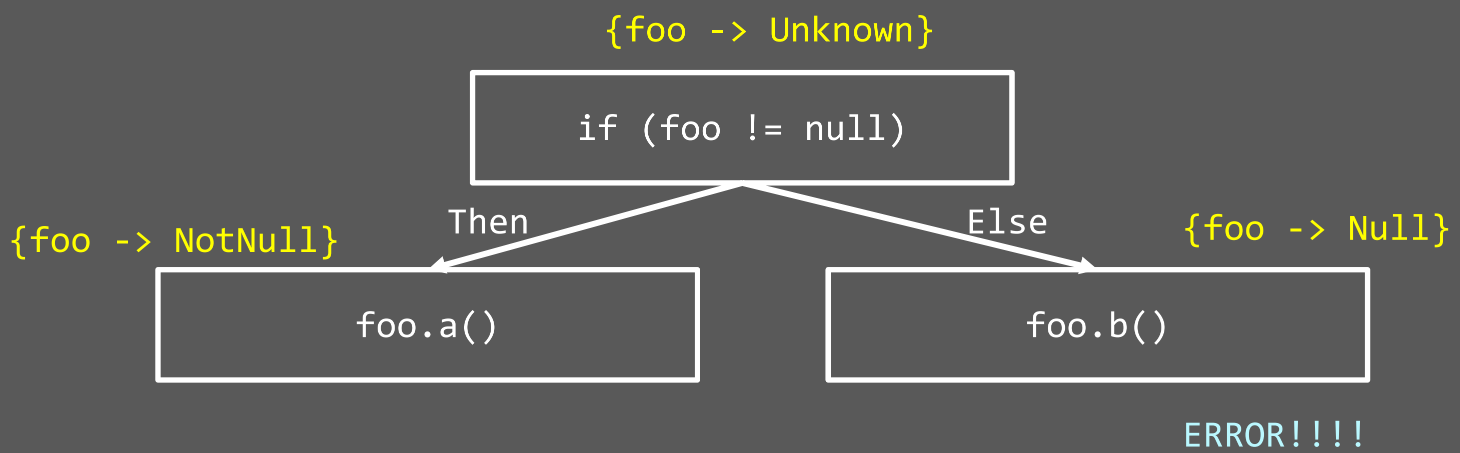
```
if (foo != null)
    foo.a();
else
    foo.b();
```

Data-Flow Analysis Examples



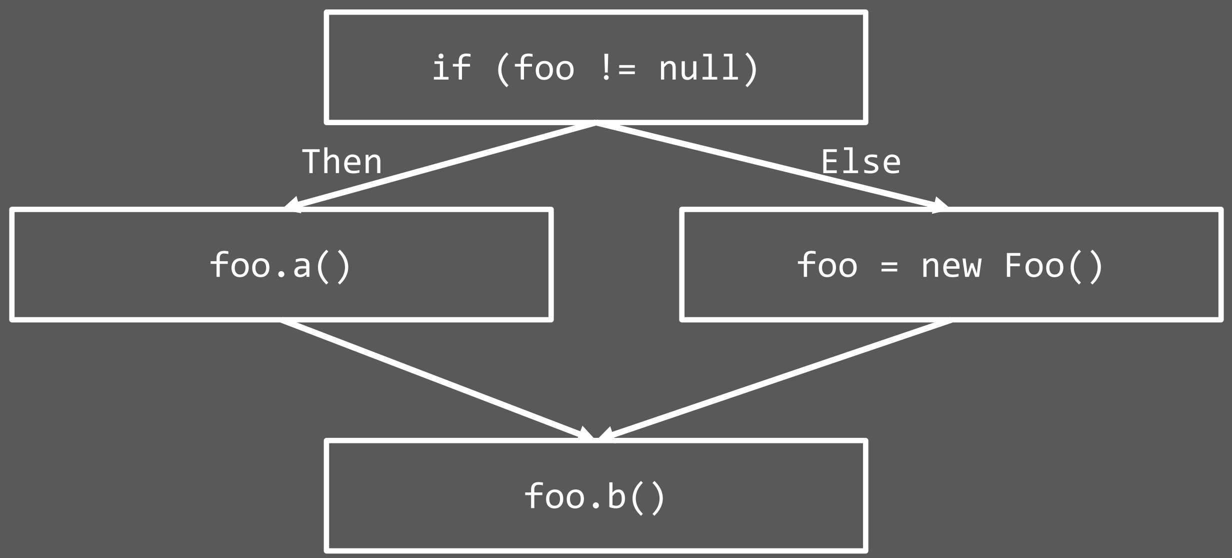

```
if (foo != null)
    foo.a();
else
    foo.b();
```

Data-Flow Analysis Examples



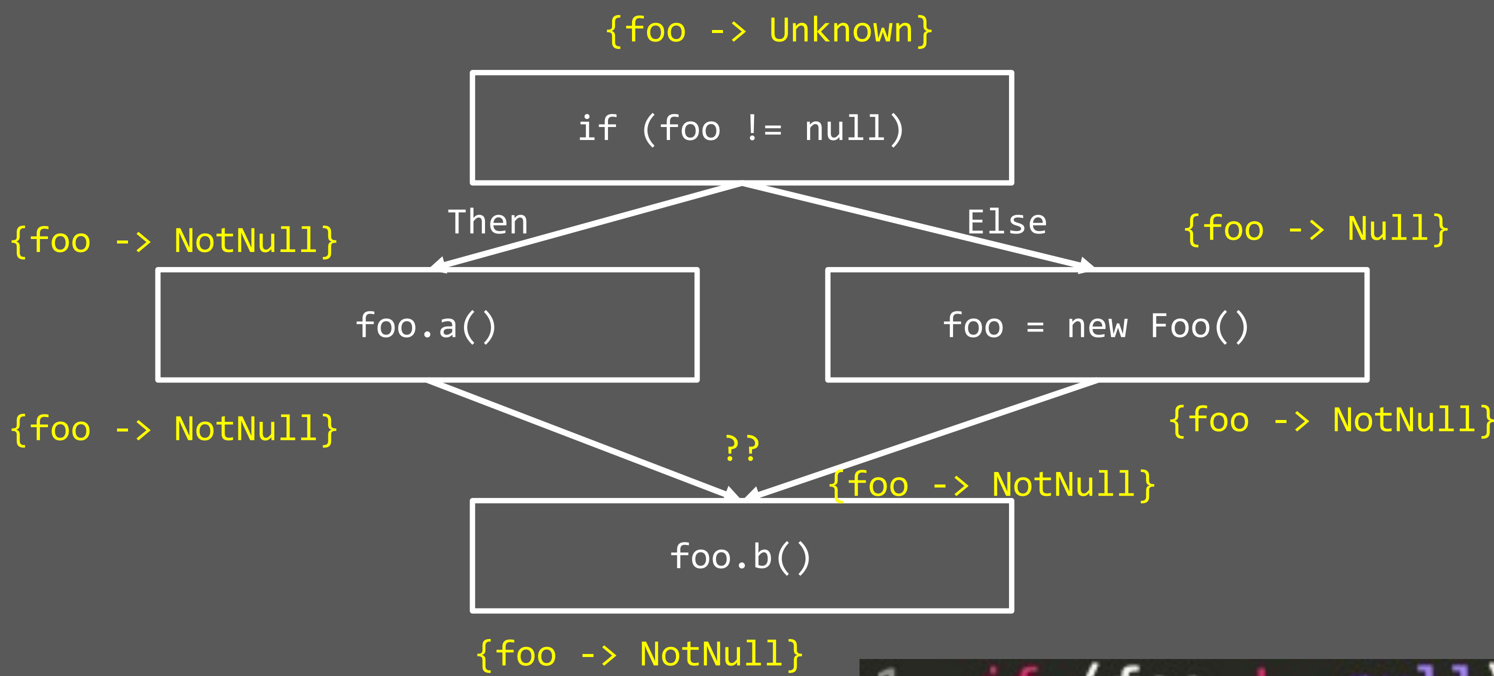

```
if (foo != null)
    foo.a();
else
    foo.b();
```

Data-Flow Analysis Examples



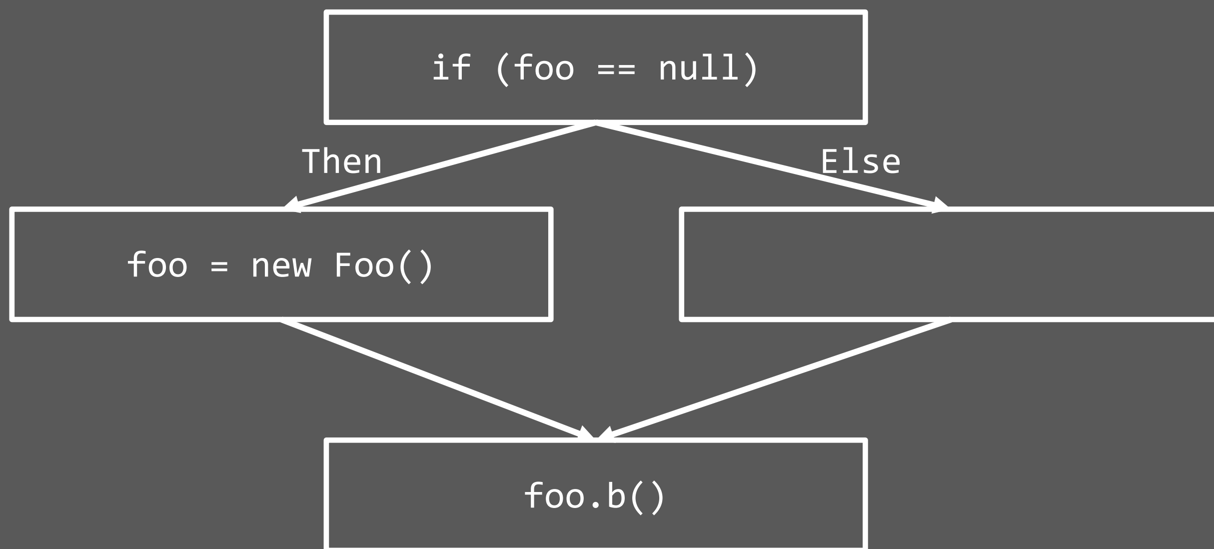

```
if (foo != null)
    foo.a();
else
    foo = new Foo();

foo.b();
```

Data-Flow Analysis Examples



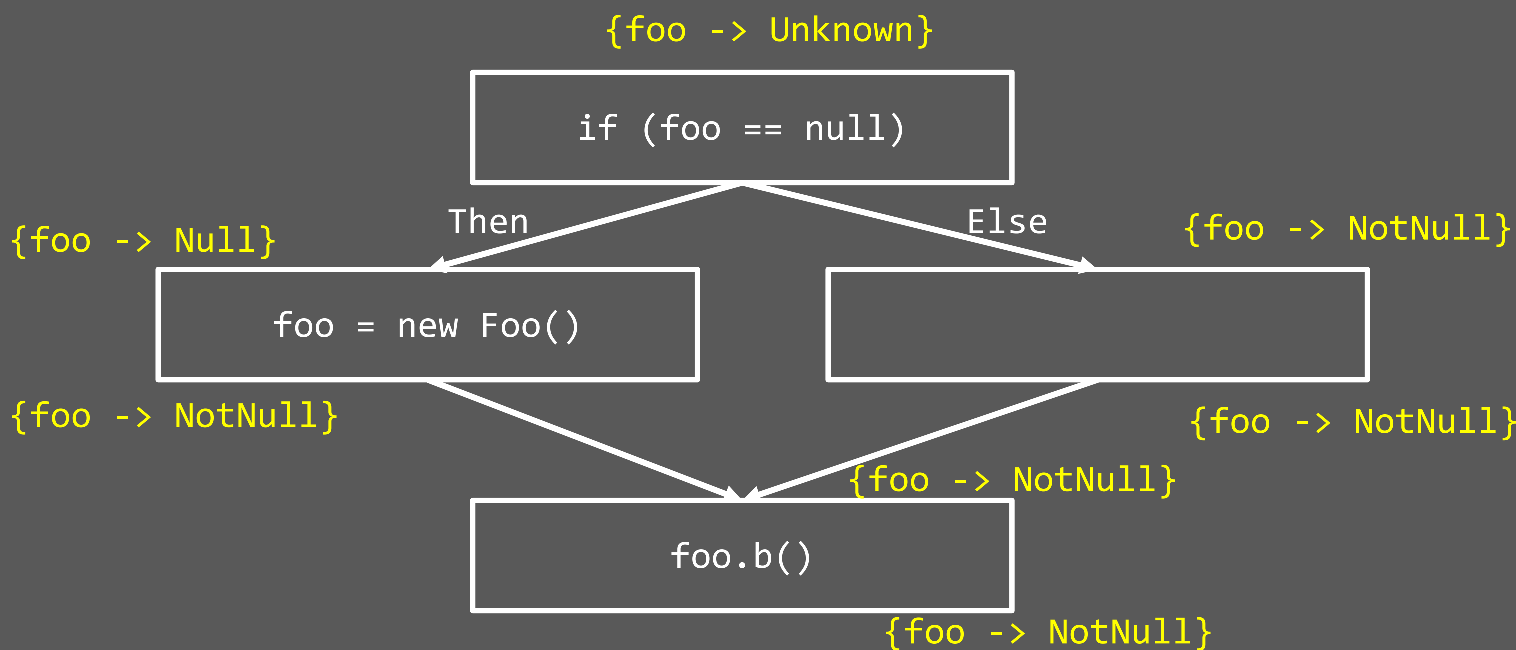

```
if (foo != null)
    foo.a();
else
    foo = new Foo();

foo.b();
```

Data-Flow Analysis Examples



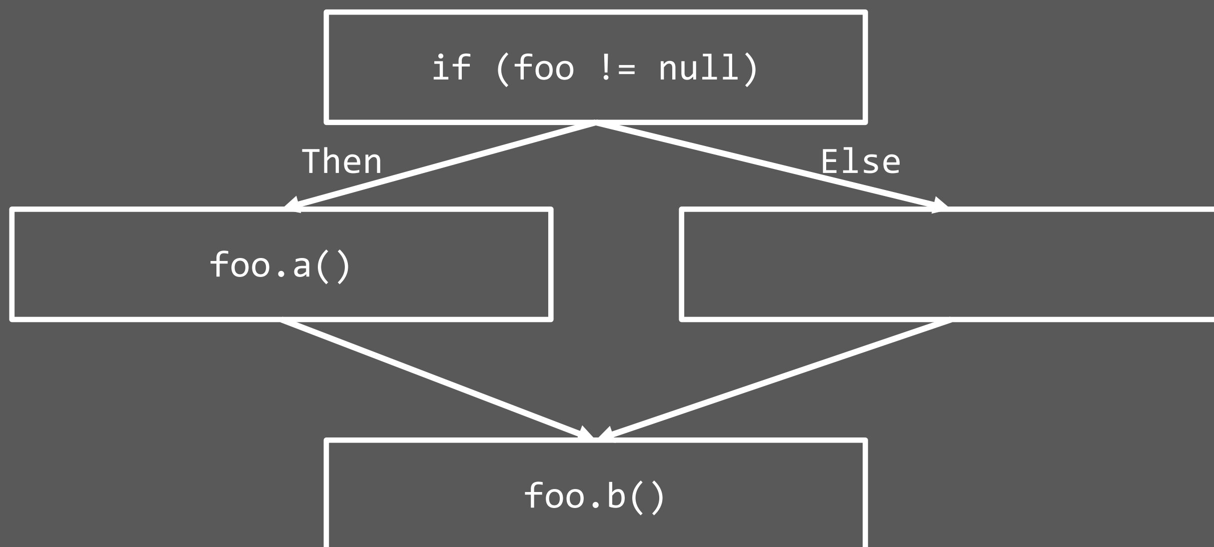

```
if (foo == null)
    foo = new Foo();
foo.b();
```

Data-Flow Analysis Examples



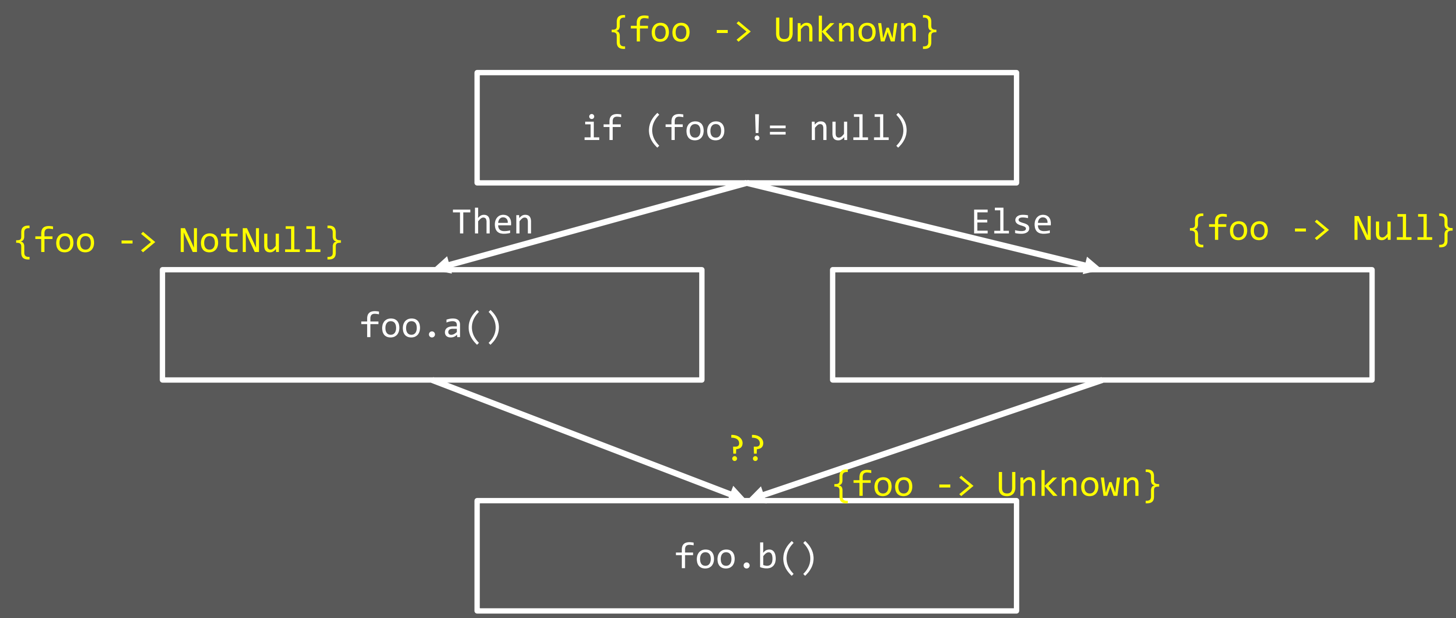

```
if (foo == null)
    foo = new Foo();
foo.b();
```

Data-Flow Analysis Examples




```
if (foo != null)
    foo.a();
foo.b();
```

Data-Flow Analysis Examples



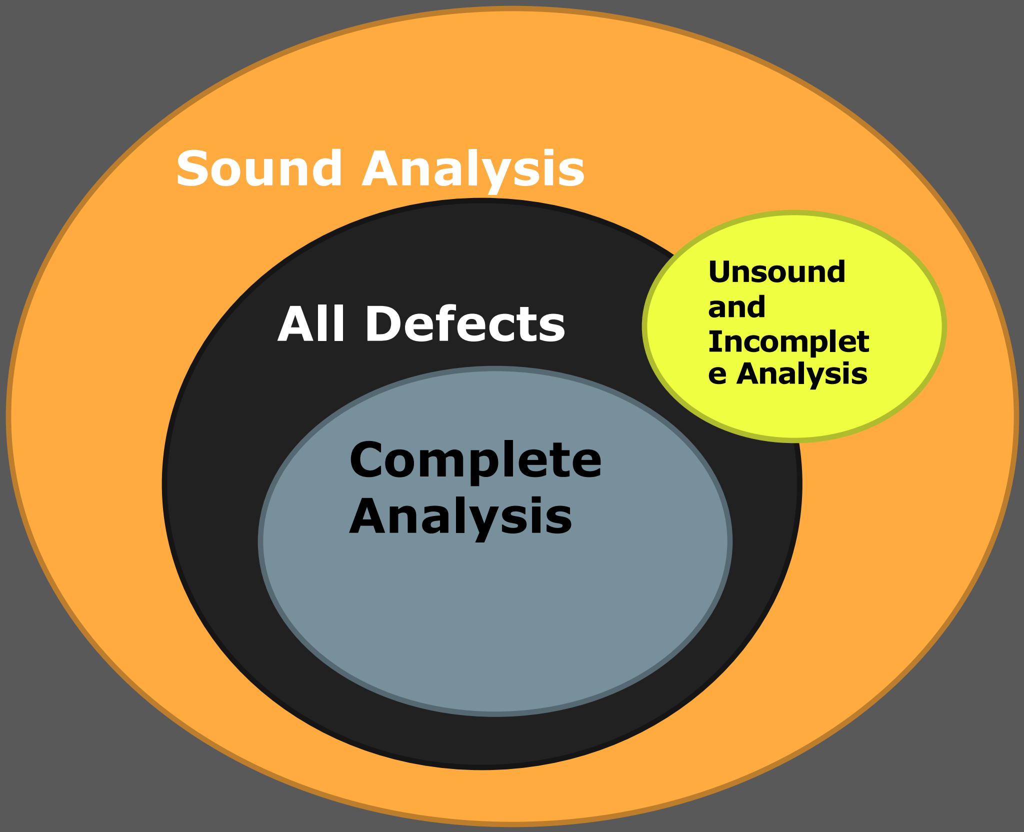

```
if (foo != null)
    foo.a();
foo.b();
```

Interpreting abstract states

- "Null" means "must be NULL at this point, regardless of path taken"
- "NotNull" is similar
- "Unknown" means "may be NULL or not null depending on the path taken"

- Unknown must be dealt with due to Rice's theorem
    - Can make analysis smarter (at the cost of more algorithmic complexity) to reduce Unknowns, but can't get rid of them completely

- Whether to raise a flag on UNKNOWN access depends on usability/soundness.
    - False positives if warning on UNKNOWN
    - False negatives if no warning on UNKNOWN


Sound Analysis
All Defects
Unsound and Incomplete Analysis
Complete Analysis

Examples of Data-Flow Anlayses

- 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!!!

Data-Flow Analysis: Challenges

- Loops
  - Fixed-point algorithms guarantee termination at the cost of losing information (“Unknown”)
- Functions
  - Analyze them separately or analyze whole program at once
  - “Context-sensitive” analyses specialize on call sites (think: duplicate function body for every call site via inlining)
- Recursion
  - Makes context-sensitive analyses explode (cf. loops)
- Object-oriented programming
- Heap memory
  - Need to abstract mapping keys not just values
- Exceptions

Shameless Plug

17-355/17-665/17-819 Program Analysis

- Offered in Spring 2022 (by Rohan Padhye). Sign up now if interested!
- Topics include:
  - Program semantics
  - Dataflow analysis
  - Pointer Analysis
  - Control flow analysis for functional programs
  - Model checking
  - SMT solvers
  - Program verification
  - Symbolic Execution
  - Race Detection for Concurrent Programs
- Beautiful theory 😍 + building automatic bug-finding tools 💻
- Check out https://cmu-program-analysis.github.io (2020 website)

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