Software Foundations of Security & Privacy 15315 Spring 2017 Lecture 3: Testing

Matt Fredrikson
mfredrik@cs.cmu.edu

January 24, 2016

Why test?

Why test?

- Find bugs
- Determine whether program matches specification
- Check that performance/resource use isn't an issue
- Make sure changes don't break existing functionality
- Validate the specification

Why test?

- Find bugs
- Determine whether program matches specification
- Check that performance/resource use isn't an issue
- Make sure changes don't break existing functionality
- Validate the specification

In general, testing can help us

1. **uncover problems**,
2. and **increase our confidence in the program's correctness**

Testing: Pros and Cons

Done correctly, testing is good at **finding bugs**

Testing: Pros and Cons

Done correctly, testing is good at **finding bugs**

- Systematic way of aligning specification with implementation

Testing: Pros and Cons

Done correctly, testing is good at **finding bugs**

- Systematic way of aligning specification with implementation
- Practical, straightforward to implement, and widely-used in industry

Testing: Pros and Cons

Done correctly, testing is good at **finding bugs**

- Systematic way of aligning specification with implementation
- Practical, straightforward to implement, and widely-used in industry
- Significantly less expensive than static analysis, formal verification

Testing: Pros and Cons

Done correctly, testing is good at **finding bugs**

- Systematic way of aligning specification with implementation
- Practical, straightforward to implement, and widely-used in industry
- Significantly less expensive than static analysis, formal verification
- Has proven effective at prevent important, potentially expensive, classes of bugs

Testing: Pros and Cons

Done correctly, testing is good at **finding bugs**

- Systematic way of aligning specification with implementation
- Practical, straightforward to implement, and widely-used in industry
- Significantly less expensive than static analysis, formal verification
- Has proven effective at prevent important, potentially expensive, classes of bugs

Testing can never show the **absence of bugs**

Testing: Pros and Cons

Done correctly, testing is good at **finding bugs**

- Systematic way of aligning specification with implementation
- Practical, straightforward to implement, and widely-used in industry
- Significantly less expensive than static analysis, formal verification
- Has proven effective at prevent important, potentially expensive, classes of bugs

Testing can never show the **absence of bugs**

- Testing is **not** verification!

Testing: Pros and Cons

Done correctly, testing is good at **finding bugs**

- Systematic way of aligning specification with implementation
- Practical, straightforward to implement, and widely-used in industry
- Significantly less expensive than static analysis, formal verification
- Has proven effective at prevent important, potentially expensive, classes of bugs

Testing can never show the **absence of bugs**

- Testing is **not** verification!
- Realistically, it only guarantees that the program works on the tests you give it

Testing: Pros and Cons

Done correctly, testing is good at **finding bugs**

- Systematic way of aligning specification with implementation
- Practical, straightforward to implement, and widely-used in industry
- Significantly less expensive than static analysis, formal verification
- Has proven effective at prevent important, potentially expensive, classes of bugs

Testing can never show the **absence of bugs**

- Testing is **not** verification!
- Realistically, it only guarantees that the program works on the tests you give it
- Extrapolating coverage guarantees from test cases is dangerous!

Testing for security

Attackers are always looking for bugs

- Manually analyzing open-source software
- Blackbox “fuzz” testing (more on this later)

Testing for security

Attackers are always looking for bugs

- Manually analyzing open-source software
- Blackbox "fuzz" testing (more on this later)

Fuzzing is used heavily in security testing

- Mandated in Microsoft's development lifecycle
- Uncovered numerous "million dollar bugs" in real systems

Testing for security

Attackers are always looking for bugs

- Manually analyzing open-source software
- Blackbox "fuzz" testing (more on this later)

Fuzzing is used heavily in security testing

- Mandated in Microsoft's development lifecycle
- Uncovered numerous "million dollar bugs" in real systems

Good, principled testing throughout development is important

- The more bugs you find, the harder an attacker's job
- Forces programmer to think about corner cases, requirements

Starting small: unit testing

A **unit test** exercises an individual component in the program

- Functions, modules, classes, etc.

Starting small: unit testing

A **unit test** exercises an individual component in the program

- Functions, modules, classes, etc.

The goal is to require that units meet their specifications in isolation

- Best practice: annotate each unit with a **contract**
- **requires** specify what's needed to use the unit
- **ensures** specify what the unit provides

```
void sort(string[] A, int lower, int upper)
  //@requires 0 <= lower && lower <= upper && upper <= \length(A);
  //@ensures is_sorted(A, lower, upper);
```

Unit tests are designed to show that these obligations are met

Working up: integration testing

Unit tests aren't enough to gain confidence in most cases

Working up: integration testing

Unit tests aren't enough to gain confidence in most cases

Integration tests stitch units together

Two basic approaches for integration testing:

Working up: integration testing

Unit tests aren't enough to gain confidence in most cases

Integration tests stitch units together

Two basic approaches for integration testing:

- **Bottom-up**: Start with units that have no dependencies, and work upwards through the dependency graph.

Working up: integration testing

Unit tests aren't enough to gain confidence in most cases

Integration tests stitch units together

Two basic approaches for integration testing:

- **Bottom-up**: Start with units that have no dependencies, and work upwards through the dependency graph.
- **Top-down**: Start at the top of the dependency graph, and work down. This often requires providing "stub" functions that implement just enough functionality to pass certain tests, or provide diagnostic information.

In practice, it's common to apply both strategies in some combination.

Building a test suite: basic “random” testing

Consider a function `maximum` that returns the largest `int` in a list

Suppose we run the following tests:

Input	Output	Correct?
$[1, 3, 5, 7, 9, 10]$	10	*yes*
$[2, 4, 6, 8, 10, 12]$	12	*yes*
$[4, 3, 2, 1]$	4	*yes*
$[0, 1, 0, 1, 0, 1]$	1	*yes*
$[1, 2, 4, 8, 16, 32, 64]$	64	*yes*
$[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]$	34	*yes*

Building a test suite: basic "random" testing

Consider a function `maximum` that returns the largest `int` in a list

Suppose we run the following tests:

Input	**Output**	**Correct?**
$[1, 3, 5, 7, 9, 10]$	10	*yes*
$[2, 4, 6, 8, 10, 12]$	12	*yes*
$[4, 3, 2, 1]$	4	*yes*
$[0, 1, 0, 1, 0, 1]$	1	*yes*
$[1, 2, 4, 8, 16, 32, 64]$	64	*yes*
$[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]$	34	*yes*

Do we assume that it works?

Random testing: counterexample

```
let maximum l = List.fold_left max 0 l
```

Random testing: counterexample

```
let maximum l = List.fold_left max 0 l
```

What is `(maximum [])`?

Random testing: counterexample

```
let maximum l = List.fold_left max 0 l
```

What is `(maximum [])`?

Our specification wasn't precise enough

Random testing: counterexample

```
let maximum l = List.fold_left max 0 l
```

What is `(maximum [])`?

Our specification wasn't precise enough

Probably want to raise an exception, or return `None`, on empty list

Random testing: counterexample

```
let maximum l =
  match l with
      [] -> invalid_arg "empty list"
    | _ -> List.fold_left max 0 l
```

Random testing: counterexample

```
let maximum l =
  match l with
      [] -> invalid_arg "empty list"
    | _ -> List.fold_left max 0 l
```

What is `(maximum [-3, -2, -1])`?

Random testing: counterexample

```
let maximum l =
  match l with
      [] -> invalid_arg "empty list"
    | _ -> List.fold_left max 0 l
```

What is `(maximum [-3, -2, -1])`?

Our tests failed to cover half the number line!

Random testing: counterexample

```
let maximum l =
  match l with
      [] -> invalid_arg "empty list"
    | _ -> List.fold_left max 0 l
```

What is `(maximum [-3, -2, -1])`?

Our tests failed to cover half the number line!

Even with a simple specification, one might overlook major cases

Systematic testing

We need to be systematic in how we go about testing

This requires good answers to the following questions:

- Which inputs do we choose?
- How do we check the outputs?
- How do we know when we've tested enough?

Partitioning: which inputs to choose

We can't test all possible inputs, and "random" testing doesn't work

Partitioning: which inputs to choose

We can't test all possible inputs, and "random" testing doesn't work

We want to find a set of tests that:

1. is small enough to feasibly run
2. is likely to catch most of the bugs we care about

Partitioning: which inputs to choose

We can't test all possible inputs, and "random" testing doesn't work

We want to find a set of tests that:

1. is small enough to feasibly run
2. is likely to catch most of the bugs we care about

Intuition: although the input space is very large, the program has limited functionality

Partitioning: which inputs to choose

We can’t test all possible inputs, and “random” testing doesn’t work

We want to find a set of tests that:

1. is small enough to feasibly run
2. is likely to catch most of the bugs we care about

Intuition: although the input space is very large, the program has limited functionality

- program behavior must be “similar” on many different inputs

Partitioning: which inputs to choose

We can't test all possible inputs, and "random" testing doesn't work

We want to find a set of tests that:

1. is small enough to feasibly run
2. is likely to catch most of the bugs we care about

Intuition: although the input space is very large, the program has limited functionality

- program behavior must be "similar" on many different inputs
- identify inputs yielding similar behavior, pick a representative test

Partitioning: which inputs to choose

We can't test all possible inputs, and "random" testing doesn't work

We want to find a set of tests that:

1. is small enough to feasibly run
2. is likely to catch most of the bugs we care about

Intuition: although the input space is very large, the program has limited functionality

- program behavior must be "similar" on many different inputs
- identify inputs yielding similar behavior, pick a representative test
- make sure each "input partition" is covered by a test

Partitioning: which inputs to choose

We can't test all possible inputs, and "random" testing doesn't work

We want to find a set of tests that:

1. is small enough to feasibly run
2. is likely to catch most of the bugs we care about

Intuition: although the input space is very large, the program has limited functionality

- program behavior must be "similar" on many different inputs
- identify inputs yielding similar behavior, pick a representative test
- make sure each "input partition" is covered by a test

This is called **partitioning**

Identifying good partitions

Partitions should correspond to relevant properties of the input space

- Test suite will vary these properties
- “Completeness” follows from correspondence between partitions and program behavior

Two basic approaches: **black-box** and **white-box**

Identifying good partitions

Partitions should correspond to relevant properties of the input space

- Test suite will vary these properties
- "Completeness" follows from correspondence between partitions and program behavior

Two basic approaches: **black-box** and **white-box**

- Black-box: as the name suggests, view the program as an opaque function and test to the specification

Identifying good partitions

Partitions should correspond to relevant properties of the input space

- Test suite will vary these properties
- "Completeness" follows from correspondence between partitions and program behavior

Two basic approaches: **black-box** and **white-box**

- Black-box: as the name suggests, view the program as an opaque function and test to the specification
- White-box: use knowledge of implementation & code to generate representative tests and coverage metrics

Black-box testing

Basic idea: partition inputs according to the specification

- Maybe you have the code, but pretend that you don't

Black-box testing

Basic idea: partition inputs according to the specification

- Maybe you have the code, but pretend that you don't

Black-box testing has several advantages

Black-box testing

Basic idea: partition inputs according to the specification

- Maybe you have the code, but pretend that you don't

Black-box testing has several advantages

1. Tests are independent of the code. Why is this good?

Black-box testing

Basic idea: partition inputs according to the specification

- Maybe you have the code, but pretend that you don't

Black-box testing has several advantages

1. Tests are independent of the code. Why is this good?
2. Language-independent

Black-box testing

Basic idea: partition inputs according to the specification

- Maybe you have the code, but pretend that you don't

Black-box testing has several advantages

1. Tests are independent of the code. Why is this good?
2. Language-independent
3. Aligns with user's point of view

Black-box testing

Basic idea: partition inputs according to the specification

- Maybe you have the code, but pretend that you don't

Black-box testing has several advantages

1. Tests are independent of the code. Why is this good?
2. Language-independent
3. Aligns with user's point of view

...and a few limitations

Black-box testing

Basic idea: partition inputs according to the specification

- Maybe you have the code, but pretend that you don't

Black-box testing has several advantages

1. Tests are independent of the code. Why is this good?
2. Language-independent
3. Aligns with user's point of view

...and a few limitations

1. Requires a complete, unambiguous specification

Black-box testing

Basic idea: partition inputs according to the specification

- Maybe you have the code, but pretend that you don't

Black-box testing has several advantages

1. Tests are independent of the code. Why is this good?
2. Language-independent
3. Aligns with user's point of view

...and a few limitations

1. Requires a complete, unambiguous specification
2. Usually hard to know if coverage is sufficient

Black-box testing

Basic idea: partition inputs according to the specification

- Maybe you have the code, but pretend that you don't

Black-box testing has several advantages

1. Tests are independent of the code. Why is this good?
2. Language-independent
3. Aligns with user's point of view

...and a few limitations

1. Requires a complete, unambiguous specification
2. Usually hard to know if coverage is sufficient
3. Difficult to automate

Black-box selection strategies

Enumerate "paths" through the specification

- Use `requires`, `ensures`, failure/exception cases
- Test each valid combination to cover all intended cases
- Also: make sure the spec doesn't "miss" any possible inputs

Black-box selection strategies

Enumerate "paths" through the specification

- Use `requires`, `ensures`, failure/exception cases
- Test each valid combination to cover all intended cases
- Also: make sure the spec doesn't "miss" any possible inputs

Test boundary/extremal values

- Identify input ranges, choose test values close to low and high-ends
- e.g., integer range, buffer size, …
- Good exercise to find holes in the specification

Black-box selection strategies

Enumerate "paths" through the specification

- Use `requires`, `ensures`, failure/exception cases
- Test each valid combination to cover all intended cases
- Also: make sure the spec doesn't "miss" any possible inputs

Test boundary/extremal values

- Identify input ranges, choose test values close to low and high-ends
- e.g., integer range, buffer size, …
- Good exercise to find holes in the specification

Off-nominal values

- Identify invalid inputs, choose values that test each one
- For example, break invariants (recall: Heartbleed) and violate assumptions

Example

What are the relevant features of the `maximum` function?

```
(* if l is non-empty, returns the greatest element
   if l is empty, returns None *)
let maximum (l : int list) : int option =
  ...
```

Example

What are the relevant features of the `maximum` function?

```
(* if l is non-empty, returns the greatest element
   if l is empty, returns None *)
let maximum (l : int list) : int option =
  ...
```

- The size of the list (0, 1, 2, large, very large)

Example

What are the relevant features of the `maximum` function?

```
(* if l is non-empty, returns the greatest element
   if l is empty, returns None *)
let maximum (l : int list) : int option =
  ...
```

- The size of the list (0, 1, 2, large, very large)
- Position of maximum value (beginning, middle, end)

Example

What are the relevant features of the `maximum` function?

```
(* if l is non-empty, returns the greatest element
   if l is empty, returns None *)
let maximum (l : int list) : int option =
  ...
```

- The size of the list (0, 1, 2, large, very large)
- Position of maximum value (beginning, middle, end)
- Range of values (negative, positive, max/min values)

Example

What are the relevant features of the `maximum` function?

```
(* if l is non-empty, returns the greatest element
   if l is empty, returns None *)
let maximum (l : int list) : int option =
  ...
```

- The size of the list (0, 1, 2, large, very large)
- Position of maximum value (beginning, middle, end)
- Range of values (negative, positive, max/min values)
- Existence of duplicate values

Example

What are the relevant features of the `maximum` function?

```
(* if l is non-empty, returns the greatest element
   if l is empty, returns None *)
let maximum (l : int list) : int option =
  ...
```

- The size of the list (0, 1, 2, large, very large)
- Position of maximum value (beginning, middle, end)
- Range of values (negative, positive, max/min values)
- Existence of duplicate values
- Ordering of elements (ascending, descending, “random”)

White-box testing

Use details of the implementation to design and evaluate tests

- Develop partitions to maximize *code coverage*
- Test internal features like caching, domain-splitting, etc.

White-box testing

Use details of the implementation to design and evaluate tests

- Develop partitions to maximize *code coverage*
- Test internal features like caching, domain-splitting, etc.

Whitebox testing has its own advantages

White-box testing

Use details of the implementation to design and evaluate tests

- Develop partitions to maximize *code coverage*
- Test internal features like caching, domain-splitting, etc.

Whitebox testing has its own advantages

1. Tests are tailored to the code, easier to find certain bugs

White-box testing

Use details of the implementation to design and evaluate tests

- Develop partitions to maximize *code coverage*
- Test internal features like caching, domain-splitting, etc.

Whitebox testing has its own advantages

1. Tests are tailored to the code, easier to find certain bugs
2. Possible to know when we've "covered" the implementation

White-box testing

Use details of the implementation to design and evaluate tests

- Develop partitions to maximize *code coverage*
- Test internal features like caching, domain-splitting, etc.

Whitebox testing has its own advantages

1. Tests are tailored to the code, easier to find certain bugs
2. Possible to know when we've "covered" the implementation
3. Can oftentimes be automated

White-box testing

Use details of the implementation to design and evaluate tests

- Develop partitions to maximize *code coverage*
- Test internal features like caching, domain-splitting, etc.

Whitebox testing has its own advantages

1. Tests are tailored to the code, easier to find certain bugs
2. Possible to know when we've "covered" the implementation
3. Can oftentimes be automated

Disadvantages:

White-box testing

Use details of the implementation to design and evaluate tests

- Develop partitions to maximize *code coverage*
- Test internal features like caching, domain-splitting, etc.

Whitebox testing has its own advantages

1. Tests are tailored to the code, easier to find certain bugs
2. Possible to know when we've "covered" the implementation
3. Can oftentimes be automated

Disadvantages:

1. Expensive, and still not verification

White-box testing

Use details of the implementation to design and evaluate tests

- Develop partitions to maximize *code coverage*
- Test internal features like caching, domain-splitting, etc.

Whitebox testing has its own advantages

1. Tests are tailored to the code, easier to find certain bugs
2. Possible to know when we've "covered" the implementation
3. Can oftentimes be automated

Disadvantages:

1. Expensive, and still not verification
2. Automatic tools are language-dependent, rely on heuristics

Coverage metrics: when to stop testing

We mentioned that whitebox testing enables better "coverage" measurement

Coverage metrics: when to stop testing

We mentioned that whitebox testing enables better “coverage” measurement

The basic goal is to make sure tests cover all the relevant code

Coverage metrics: when to stop testing

We mentioned that whitebox testing enables better "coverage" measurement

The basic goal is to make sure tests cover all the relevant code

There are several ways to measure this

Coverage metrics: when to stop testing

We mentioned that whitebox testing enables better “coverage” measurement

The basic goal is to make sure tests cover all the relevant code

There are several ways to measure this

- Statements
- Branches
- Paths
- Traces
- ...

Coverage metrics: when to stop testing

We mentioned that whitebox testing enables better “coverage” measurement

The basic goal is to make sure tests cover all the relevant code

There are several ways to measure this

- Statements
- Branches
- Paths
- Traces
- ...

Each offers a different tradeoff between cost and completeness

Coverage criteria: statements

Goal: Design a test set such that each *primitive statement* is executed at least once

Coverage criteria: statements

Goal: Design a test set such that each *primitive statement* is executed at least once

A primitive statement contains no nested statements

- Assignments, function calls are examples of primitive statements
- Loops, conditionals are examples of *compound statements*

Coverage criteria: statements

Goal: Design a test set such that each *primitive statement* is executed at least once

A primitive statement contains no nested statements

- Assignments, function calls are examples of primitive statements
- Loops, conditionals are examples of *compound statements*

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: statements

What test set achieves statement coverage?

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: statements

What test set achieves statement coverage?

$(n = 101, c = 1)$?

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: statements

What test set achieves statement coverage?

$(n = 101, c = 1)$?
no

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: statements

What test set achieves statement coverage?

$(n = 101, c = 1)$?

no

$(n = 101, c = 1)$,
$(n = 100, c = 1)$?

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: statements

What test set achieves statement coverage?

$(n = 101, c = 1)$?
no

$(n = 101, c = 1)$,
$(n = 100, c = 1)$?
yes

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: statements

What test set achieves statement coverage?

$(n = 101, c = 1)$?
no

$(n = 101, c = 1)$,
$(n = 100, c = 1)$?
yes

$(n = 101, c = 2)$?

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: statements

What test set achieves statement coverage?

$(n = 101, c = 1)$?
no

$(n = 101, c = 1)$,
$(n = 100, c = 1)$?
yes

$(n = 101, c = 2)$?
yes

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: branches

Goal: Design a test set such that each *branch* is executed at least once

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: branches

Goal: Design a test set such that each *branch* is executed at least once

Branching comes from several constructs:

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: branches

Goal: Design a test set such that each *branch* is executed at least once

Branching comes from several constructs:

- conditional (if-then-else)
- `match/case`
- loops

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: branches

What tests give us branch coverage?

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: branches

What tests give us branch coverage?

Same as before:
$(n = 101, c = 1)$, $(n = 100, c = 1)$
$(n = 101, c = 2)$

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: branches

Is branch coverage the same
as statement coverage?

Coverage criteria: branches

Is branch coverage the same as statement coverage?

No

Coverage criteria: branches

Is branch coverage the same as statement coverage?

No

```
if c1 then
  f1();
if c2 then
  f2();
fail_if_f2_not_called();
```

Coverage criteria: branches

Is branch coverage the same as statement coverage?

No

In this example:

- `c1 = true, c2 = true` covers all statements

```
if c1 then
  f1();
if c2 then
  f2();
fail_if_f2_not_called();
```

Coverage criteria: branches

Is branch coverage the same as statement coverage?

No

In this example:

- `c1 = true, c2 = true` covers all statements
- `c1 = false, c2 = false` hits already-covered statements, but fails

```
if c1 then
  f1();
if c2 then
  f2();
fail_if_f2_not_called();
```

Coverage criteria: branches

Is branch coverage the same as statement coverage?

No

In this example:

- `c1 = true, c2 = true` covers all statements
- `c1 = false, c2 = false` hits already-covered statements, but fails

```
if c1 then
  f1();
if c2 then
  f2();
fail_if_f2_not_called();
```

Statement coverage:
`c1 = true, c2 = true`

Branch coverage:
`c1 = true, c2 = true`
`c1 = false, c2 = false`
need both tests!

Coverage criteria: conditions

Goal: Design a test set such that each *condition* evaluates to both values

```
if c1 then
  f1();
if c2 then
  f2();
fail_if_f2_not_called();
```

Coverage criteria: conditions

Goal: Design a test set such that each *condition* evaluates to both values

A condition is a Boolean expression appearing in a guarded statement

- i.e., `if`, `while`, …

```
if c1 then
  f1();
if c2 then
  f2();
fail_if_f2_not_called();
```

Coverage criteria: conditions

Goal: Design a test set such that each *condition* evaluates to both values

A condition is a Boolean expression appearing in a guarded statement

- i.e., `if`, `while`, ...

Want to ensure that each Boolean *subexpression* evaluates to *true* and *false*

```
if c1 then
  f1();
if c2 then
  f2();
fail_if_f2_not_called();
```

Coverage criteria: conditions

Goal: Design a test set such that each *condition* evaluates to both values

A condition is a Boolean expression appearing in a guarded statement

- i.e., `if`, `while`, ...

Want to ensure that each Boolean *subexpression* evaluates to *true* and *false*

```
if c1 then
  f1();
if c2 then
  f2();
fail_if_f2_not_called();
```

In this case, branch and condition coverage are equivalent

Both are given by the tests:

`c1 = true, c2 = true`

`c1 = false, c2 = false`

Coverage criteria: conditions

Are branch and condition coverage always the same?

```
if c1 then
  f1();
if c2 then
  f2();
fail_if_f2_not_called();
```

Coverage criteria: conditions

Are branch and condition coverage always the same?

No

```
if c1 then
  f1();
if c2 then
  f2();
fail_if_f2_not_called();
```

Coverage criteria: conditions

Are branch and condition coverage always the same?

No

```
if (c1 || c2) then
  f1();
if c3 then
  f2();
fail_if_c2_is_true();
```

Coverage criteria: conditions

Are branch and condition coverage always the same?

No

In this example,

- Branch coverage:
  `(true, false, true)`
  `(false, false, false)`
- Condition coverage:
  `(true, false, true)`
  `(false, true, false)`
  `(false, false, true)`

```
if (c1 || c2) then
  f1();
if c3 then
  f2();
fail_if_c2_is_true();
```

Coverage criteria: paths

Goal: Design a test set such that each *path* is executed

A path is a sequence of statements in the program:

- that takes it from an entry point to termination
- and follows the control-flow structure

Coverage criteria: paths

Goal: Design a test set such that each *path* is executed

A path is a sequence of statements in the program:

- that takes it from an entry point to termination
- and follows the control-flow structure

How many paths are in this program?

```
if c1 then
  if c2 then
    f1();
  else
    f2();
if c3 then
  f3();
if c4 then
  f4();
```

Coverage criteria: paths

Goal: Design a test set such that each *path* is executed

A path is a sequence of statements in the program:

- that takes it from an entry point to termination
- and follows the control-flow structure

```
if c1 then
  if c2 then
    f1();
  else
    f2();
if c3 then
  f3();
if c4 then
  f4();
```

How many paths are in this program?

12: 2^4 - $\{$duplicates from $\texttt{c1} = 0\}$

Coverage criteria: paths

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

How many paths are in this program?

Coverage criteria: paths

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

How many paths are in this program?

Too many to test

Coverage criteria: paths

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

How many paths are in this program?

Too many to test

- Bounded by width of machine integer, squared

Coverage criteria: paths

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

How many paths are in this program?

Too many to test

- Bounded by width of machine integer, squared
- This “bound” isn’t any better than exhaustive testing

Coverage criteria: paths

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

How many paths are in this program?

Too many to test

- Bounded by width of machine integer, squared
- This “bound” isn’t any better than exhaustive testing

Loops & recursion make exhaustive path coverage infeasible

Weaknesses of coverage criteria

We can roughly order criteria by level of "confidence":

statement $<$ branch $<$ condition $<$ path

Weaknesses of coverage criteria

We can roughly order criteria by level of "confidence":

statement $<$ branch $<$ condition $<$ path

But testing is inherently an incomplete, heuristic measure

Consider this flawed "median" function

Good tests require careful thought, discipline, and a well-conceived specification

```
let median x y z =
  z
```

Weaknesses of coverage criteria

We can roughly order criteria by level of "confidence":

statement $<$ branch $<$ condition $<$ path

But testing is inherently an incomplete, heuristic measure

Consider this flawed "median" function

Good tests require careful thought, discipline, and a well-conceived specification

```
let median x y z =
  z
```

We can achieve all-paths and still fail to test

`x = 3, y = 1, z = 2`

Fuzz testing

Fuzz testing

> Sitting in my apartment in Madison in the Fall of 1988, there was a wild midwest thunderstorm pouring rain and lighting up the late night sky. That night, I was logged on to the Unix system in my office via a dial-up phone line over a 1200 baud modem. With the heavy rain, there was noise on the line and that noise was interfering with my ability to type sensible commands to the shell and programs that I was running ... What did surprise me was the fact that the noise seemed to be causing programs to crash.
>
> — Prof. Bart Miller

Fuzzing vs. Regression testing

Regression tests run the program on “normal” (i.e., expected) inputs and look for bad things. This is usually incorporated into a build/release framework to make sure that changes don’t break previous functionality.

Fuzzing vs. Regression testing

Regression tests run the program on “normal” (i.e., expected) inputs and look for bad things. This is usually incorporated into a build/release framework to make sure that changes don’t break previous functionality.

Fuzz testing runs the program on a large number of “abnormal” inputs, and look for **very bad** things. This mimics an attacker, who will look for unexpected ways of running the program that might lead to exploitable flaws.

Fuzzing: what it's good for

Simple idea: feed random inputs to the program, look for crashes/exceptions

- Works in blackbox, whitebox settings
- Can be mostly random, or heavily influenced by existing tests or program internals
- In either case, it's automated: lots of inputs, no regard for norms

Fuzzing: what it's good for

Simple idea: feed random inputs to the program, look for crashes/exceptions

- Works in blackbox, whitebox settings
- Can be mostly random, or heavily influenced by existing tests or program internals
- In either case, it's automated: lots of inputs, no regard for norms

Why is this an effective technique?

- Random processes don't share our assumptions, biases
- Oftentimes, crashes give an entry point for exploits
- In practice, it works: original fuzzers found bugs in 33% of Unix utility programs

Black-Box fuzzing: simplest approach

Given a program:

- Identify all inputs that could be controlled by attacker
- Generate random values for those inputs
- See if the program crashes on any values

Black-Box fuzzing: simplest approach

Given a program:

- Identify all inputs that could be controlled by attacker
- Generate random values for those inputs
- See if the program crashes on any values

This has obvious limitations, from what we've seen earlier

Black-Box fuzzing: simplest approach

Given a program:

- Identify all inputs that could be controlled by attacker
- Generate random values for those inputs
- See if the program crashes on any values

This has obvious limitations, from what we've seen earlier

Why might it work better than manual "random" testing?

Black-Box fuzzing: simplest approach

Given a program:

- Identify all inputs that could be controlled by attacker
- Generate random values for those inputs
- See if the program crashes on any values

This has obvious limitations, from what we've seen earlier

Why might it work better than manual "random" testing?

1. Scale: computer can generate and test many more random values than we're willing to write

Black-Box fuzzing: simplest approach

Given a program:

- Identify all inputs that could be controlled by attacker
- Generate random values for those inputs
- See if the program crashes on any values

This has obvious limitations, from what we've seen earlier

Why might it work better than manual "random" testing?

1. Scale: computer can generate and test many more random values than we're willing to write
2. True (pseudo)randomness: humans are terrible at generating random data

Black-Box mechanics

Inputs come from many sources

- Files, standard input, network, signals, devices, ...

Black-Box mechanics

Inputs come from many sources

- Files, standard input, network, signals, devices, ...

Common strategy: intercept syscalls

- `open`, `read`, `send`, `ioctl` ...

Black-Box mechanics

Inputs come from many sources

- Files, standard input, network, signals, devices, ...

Common strategy: intercept syscalls

- `open`, `read`, `send`, `ioctl` ...

What about coverage?

Black-Box mechanics

Inputs come from many sources

- Files, standard input, network, signals, devices, ...

Common strategy: intercept syscalls

- `open`, `read`, `send`, `ioctl` ...

What about coverage?

- Many fuzzers *instrument* the target program

Black-Box mechanics

Inputs come from many sources

- Files, standard input, network, signals, devices, ...

Common strategy: intercept syscalls

- `open`, `read`, `send`, `ioctl` ...

What about coverage?

- Many fuzzers *instrument* the target program
- Insert bookkeeping instructions that count which instructions visited

Black-Box mechanics

Inputs come from many sources

- Files, standard input, network, signals, devices, ...

Common strategy: intercept syscalls

- `open`, `read`, `send`, `ioctl` ...

What about coverage?

- Many fuzzers *instrument* the target program
- Insert bookkeeping instructions that count which instructions visited
- Best to rely on compiler for this, but often possible on binaries

Black-Box fuzzing: perturbation-based

Given a program and an existing test:

- Perturb (randomly change) the test in various ways
- E.g., randomly flip bits, delete/append data
- See if the program crashes on any values

Black-Box fuzzing: perturbation-based

Given a program and an existing test:

- Perturb (randomly change) the test in various ways
- E.g., randomly flip bits, delete/append data
- See if the program crashes on any values

This approach can be extended with heuristics.

- Which tests to use as seeds?
- Strategies for perturbing seeds, ignoring certain types of input

Black-Box fuzzing: perturbation-based

Given a program and an existing test:

- Perturb (randomly change) the test in various ways
- E.g., randomly flip bits, delete/append data
- See if the program crashes on any values

This approach can be extended with heuristics.

- Which tests to use as seeds?
- Strategies for perturbing seeds, ignoring certain types of input

What are the strengths and weaknesses?

Black-Box fuzzing: perturbation-based

Given a program and an existing test:

- Perturb (randomly change) the test in various ways
- E.g., randomly flip bits, delete/append data
- See if the program crashes on any values

This approach can be extended with heuristics.

- Which tests to use as seeds?
- Strategies for perturbing seeds, ignoring certain types of input

What are the strengths and weaknesses?

1. Still very easy to use, often finds egregious bugs

Black-Box fuzzing: perturbation-based

Given a program and an existing test:

- Perturb (randomly change) the test in various ways
- E.g., randomly flip bits, delete/append data
- See if the program crashes on any values

This approach can be extended with heuristics.

- Which tests to use as seeds?
- Strategies for perturbing seeds, ignoring certain types of input

What are the strengths and weaknesses?

1. Still very easy to use, often finds egregious bugs
2. Test seeds can guide search towards less-random inputs

Black-Box fuzzing: perturbation-based

Given a program and an existing test:

- Perturb (randomly change) the test in various ways
- E.g., randomly flip bits, delete/append data
- See if the program crashes on any values

This approach can be extended with heuristics.

- Which tests to use as seeds?
- Strategies for perturbing seeds, ignoring certain types of input

What are the strengths and weaknesses?

1. Still very easy to use, often finds egregious bugs
2. Test seeds can guide search towards less-random inputs
3. Seeds can also limit the search, bias towards assumptions

Black-Box fuzzing: perturbation-based

Given a program and an existing test:

- Perturb (randomly change) the test in various ways
- E.g., randomly flip bits, delete/append data
- See if the program crashes on any values

This approach can be extended with heuristics.

- Which tests to use as seeds?
- Strategies for perturbing seeds, ignoring certain types of input

What are the strengths and weaknesses?

1. Still very easy to use, often finds egregious bugs
2. Test seeds can guide search towards less-random inputs
3. Seeds can also limit the search, bias towards assumptions
4. Doesn't work well with checksums, intricate grammars/protocols

Black-Box fuzzing: generation-based

Given a program and a format description:

- Use the format to generate valid inputs
- Iteratively perturb each location in the format
- See if the program crashes on any values

Black-Box fuzzing: generation-based

Given a program and a format description:

- Use the format to generate valid inputs
- Iteratively perturb each location in the format
- See if the program crashes on any values

This is a “smarter” way of inserting perturbations

- Adhering (mostly) to the format ensures that early consistency/syntax checks are passed
- Easier to achieve coverage, requires fewer test cases

Black-Box fuzzing: generation-based

Given a program and a format description:

- Use the format to generate valid inputs
- Iteratively perturb each location in the format
- See if the program crashes on any values

This is a “smarter” way of inserting perturbations

- Adhering (mostly) to the format ensures that early consistency/syntax checks are passed
- Easier to achieve coverage, requires fewer test cases

What are the weaknesses?

Black-Box fuzzing: generation-based

Given a program and a format description:

- Use the format to generate valid inputs
- Iteratively perturb each location in the format
- See if the program crashes on any values

This is a “smarter” way of inserting perturbations

- Adhering (mostly) to the format ensures that early consistency/syntax checks are passed
- Easier to achieve coverage, requires fewer test cases

What are the weaknesses?

1. Writing a mechanized format description, generator is labor-intensive

Black-Box fuzzing: generation-based

Given a program and a format description:

- Use the format to generate valid inputs
- Iteratively perturb each location in the format
- See if the program crashes on any values

This is a “smarter” way of inserting perturbations

- Adhering (mostly) to the format ensures that early consistency/syntax checks are passed
- Easier to achieve coverage, requires fewer test cases

What are the weaknesses?

1. Writing a mechanized format description, generator is labor-intensive
2. Format might not match the code, lead to missed bugs

White-box fuzzing

Problem statement

Given a program and a set of inputs, generate a test set that maximizes code coverage.

White-box fuzzing

Problem statement

Given a program and a set of inputs, generate a test set that maximizes code coverage.

Main idea: Use the code itself to generate random inputs

1. Generate **constraints** that reflect the program's control flow
2. Solve the constraints, map solution to corresponding inputs
3. Run program on these inputs, look for crashes or exceptions

This idea was developed by Patrice Godefroid at Microsoft, ca. 2005-present

Static test generation

We want to generate tests that exercise all paths

Static test generation

We want to generate tests that exercise all paths

```
if x < 2 then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Static test generation

We want to generate tests that exercise all paths

Basic approach: generate constraints

```
if x < 2 then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Static test generation

We want to generate tests that exercise all paths

```
if x < 2 then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Basic approach: generate constraints

First path: execute `f1, f3, f4`

Static test generation

We want to generate tests that exercise all paths

```
if x < 2 then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Basic approach: generate constraints

First path: execute `f1, f3, f4`

$$x < 2 \land y > 3 \land x < y \land y > 3 \land x \geq y$$

Static test generation

We want to generate tests that exercise all paths

```
if x < 2 then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Basic approach: generate constraints

First path: execute `f1, f3, f4`

$$x < 2 \wedge y > 3 \wedge x < y \wedge y > 3 \wedge x \geq y$$

Infeasible!

Static test generation

We want to generate tests that exercise all paths

```
if x < 2 then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Basic approach: generate constraints

First path: execute `f1, f3, f4`

$$x < 2 \wedge y > 3 \wedge x < y \wedge y > 3 \wedge x \geq y$$

Infeasible!

Second path: execute `f1, f3`

Static test generation

We want to generate tests that exercise all paths

```
if x < 2 then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Basic approach: generate constraints

First path: execute `f1, f3, f4`

$$x < 2 \land y > 3 \land x < y \land y > 3 \land x \geq y$$

Infeasible!

Second path: execute `f1, f3`

$$x < 2 \land y > 3 \land x < y \land \neg(y > 3 \land x \geq y)$$

Static test generation

We want to generate tests that exercise all paths

```
if x < 2 then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Basic approach: generate constraints

First path: execute `f1, f3, f4`

$$x < 2 \land y > 3 \land x < y \land y > 3 \land x \geq y$$

Infeasible!

Second path: execute `f1, f3`

$$x < 2 \land y > 3 \land x < y \land \neg(y > 3 \land x \geq y)$$

`x = 1, y = 4`

Static test generation

We want to generate tests that exercise all paths

```
if x < 2 then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Basic approach: generate constraints

First path: execute `f1, f3, f4`

$$x < 2 \wedge y > 3 \wedge x < y \wedge y > 3 \wedge x \geq y$$

Infeasible!

Second path: execute `f1, f3`

$$x < 2 \wedge y > 3 \wedge x < y \wedge \neg(y > 3 \wedge x \geq y)$$

`x = 1, y = 4`

...and so forth

Static test generation

This isn't always possible

Static test generation

This isn't always possible

```
if x = SHA1(...) then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Second path: execute `f1, f3`

Static test generation

This isn't always possible

```
if x = SHA1(...) then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Second path: execute `f1, f3`

$$x = \texttt{SHA1}(\ldots) \wedge y > 3 \wedge x < y \wedge \ldots$$

Static test generation

This isn't always possible

```
if x = SHA1(...) then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Second path: execute `f1, f3`

$$x = \texttt{SHA1}(\ldots) \wedge y > 3 \wedge x < y \wedge \ldots$$

Solving requires finding `SHA` pre-image

Dynamic symbolic test generation

Think: generation-based testing++

Dynamic symbolic test generation

Think: generation-based testing++

Given a program and test case:

1. Run the test case, and collect constraints along the tested path
2. Modify constraints by negating selected *literals*
3. Solve new constraints, generate corresponding inputs
4. Repeat until all assertions are reached [Korel 1990, ...]
5. Or, generate inputs for all feasible paths [Godefroid et al 2005]

Dynamic symbolic test generation

Think: generation-based testing++

Given a program and test case:

1. Run the test case, and collect constraints along the tested path
2. Modify constraints by negating selected *literals*
3. Solve new constraints, generate corresponding inputs
4. Repeat until all assertions are reached [Korel 1990, ...]
5. Or, generate inputs for all feasible paths [Godefroid et al 2005]

This approach is called DART (**D**irected **A**utomated **R**andom **T**esting)

Example

Start with `x = 5, y = 4, z = 0`

Assume that $\texttt{SHA1}(0) = 5$

```
if x = SHA1(z) then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Example

```
if x = SHA1(z) then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Start with `x = 5, y = 4, z = 0`

Assume that $\texttt{SHA1}(0) = 5$

This yields the path:

$$\texttt{assume}(x = \texttt{SHA1}(z))$$
$$\texttt{assume}(y > 3)$$
$$\texttt{f1}();$$
$$\texttt{assume}(\neg(x < y))$$
$$\texttt{assume}(y > 3 \text{ \&\& } x \geq y)$$
$$\texttt{f4}()$$

Example

```
if x = SHA1(z) then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Start with `x = 5, y = 4, z = 0`

Assume that $\texttt{SHA1}(0) = 5$

This yields the path:

$$\begin{aligned}
&\texttt{assume}(x = \texttt{SHA1}(z))\\
&\texttt{assume}(y > 3)\\
&\texttt{f1}();\\
&\texttt{assume}(\neg(x < y))\\
&\texttt{assume}(y > 3 \ \&\& \ x \geq y)\\
&\texttt{f4}()
\end{aligned}$$

We can still explore `f2, f3` by changing y

We fix x and z, change other literals

Example

```
if x = SHA1(z) then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Start with `x = 5, y = 4, z = 0`

Assume that $\texttt{SHA1}(0) = 5$

This yields the path:

$$\begin{aligned}&\texttt{assume}(x = \texttt{SHA1}(z))\\&\texttt{assume}(y > 3)\\&\texttt{f1}();\\&\texttt{assume}(\neg(x < y))\\&\texttt{assume}(y > 3 \;\&\&\; x \geq y)\\&\texttt{f4}()\end{aligned}$$

We can still explore `f2, f3` by changing y

We fix x and z, change other literals

$$x = 5 \land z = 0 \land \neg(y > 3) \land \neg(x < y) \land \ldots$$

More intelligent search

Start with a well-formed seed test

More intelligent search

Start with a well-formed seed test

Generate the path constraint

- Negate each literal independently
- Generate a new test for each negation, add to test set
- Repeat until resources run out, or we have path coverage

More intelligent search

Start with a well-formed seed test

Generate the path constraint

- Negate each literal independently
- Generate a new test for each negation, add to test set
- Repeat until resources run out, or we have path coverage

This approach tests many “layers” of the program early

Contrast with classic depth-first approach

Static test generation

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt >= 4) crash();
}
```

input = "good"

Path constraint:

I_0 != 'b' → I_0 = 'b'   bood

I_1 != 'a' → I_1 = 'a'   gaod

I_2 != 'd' → I_2 = 'd'   godd

I_3 != '!' → I_3 = '!'   goo!

good

Gen 1

Negate each constraint in path constraint

Solve new constraint → new input

Example from Patrice Godefroid

DART implementations

This approach has been used in many tools

- EXE (Stanford), concurrently with Godefroid's original work
- CUTE (Bell Labs), concurrently with original work
- SAGE (Microsoft Research)
- PEX (Microsoft Research)
- YOGI (Microsoft Research)
- Vigilante (Microsoft Research)
- BitScope (CMU/Berkeley)
- CatchConv (Berkeley)
- Splat (UCLA)
- Apollo (MIT/IBM)

Next Lecture

Introducing **enforceable security policies**

Techniques for proving these policies