Vulnerability Analysis (II): Symbolic Execution

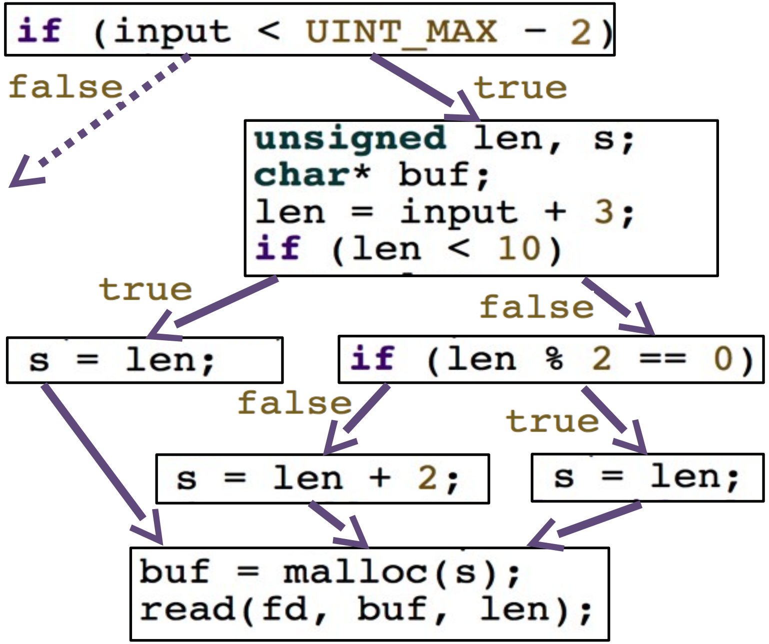
Slide credit: Vijay D'Silva

1	Efficiency of Fuzzing
2	Symbolic Reasoning
3	Path Predicates
4	Bug Finding

1	Efficiency of Fuzzing
2	Symbolic Reasoning
3	Path Predicates
4	Bug Finding

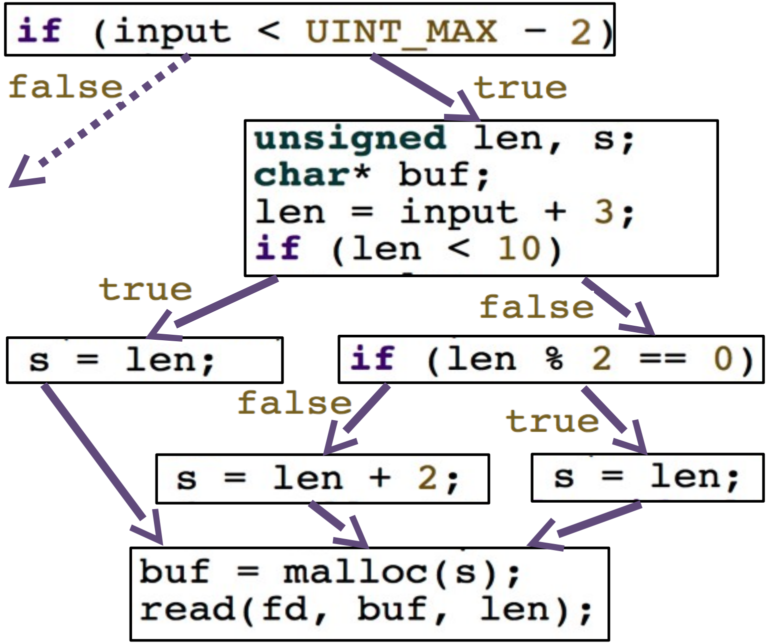
Quiz: Coverage

```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else
      s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
     ....
  }
}
```



Quiz: Coverage

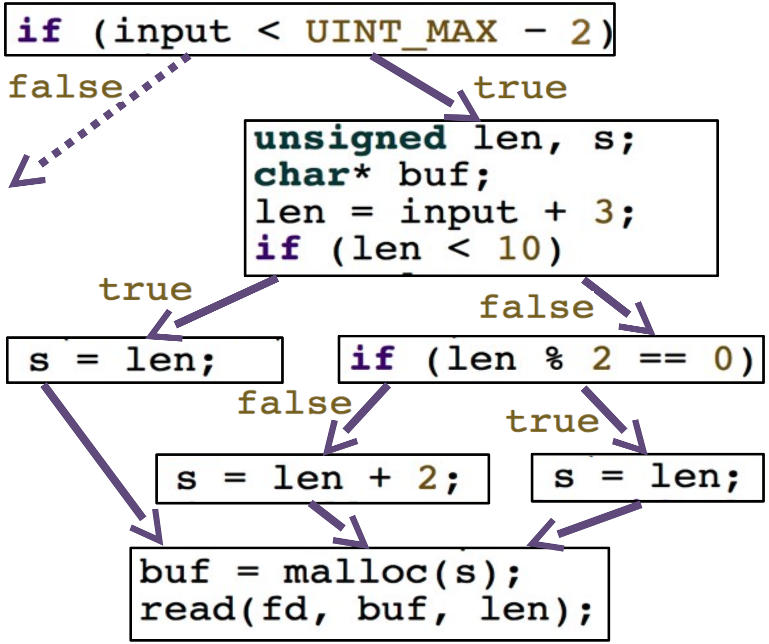
```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else
      s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
     ....
  }
}
```



	Lines	Branches	Paths
# of			
# of inputs for full			

Quiz: Coverage

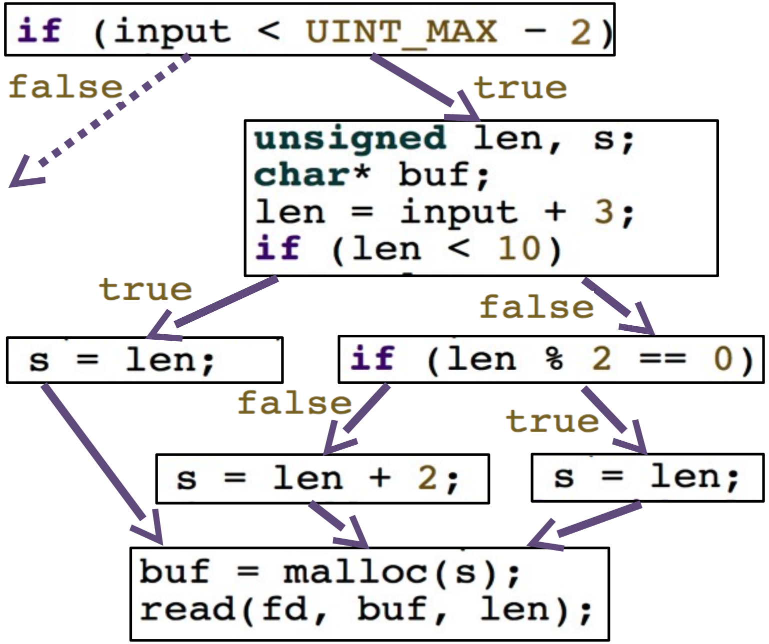
```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else
      s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
     ....
  }
}
```



	Lines	Branches	Paths
# of	10	3	3
# of inputs for full			

Quiz: Coverage

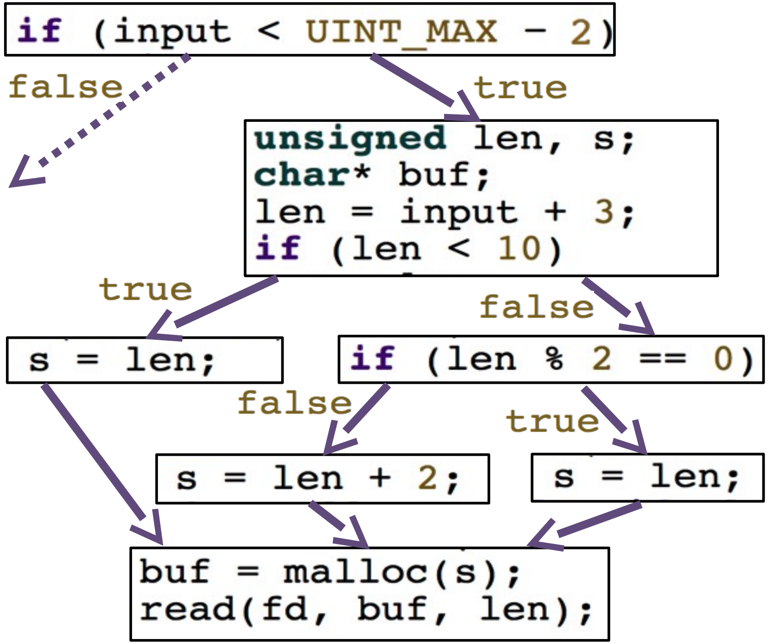
```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else
      s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
     ....
  }
}
```



	Lines	Branches	Paths
# of	10	3	3
# of inputs for full coverage	3	3	3

Quiz: Coverage

```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else
      s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
     ....
  }
}
```



What is the expected number of inputs required to cover the highlighted line, using random test-case generation? Assuming unsigned is 32 bits.

Efficiency of Test-Case Generation

We can evaluate the efficiency of a test-case generation technique with respect to a coverage metric by comparing

minimum # of inputs vs. *expected # of inputs*

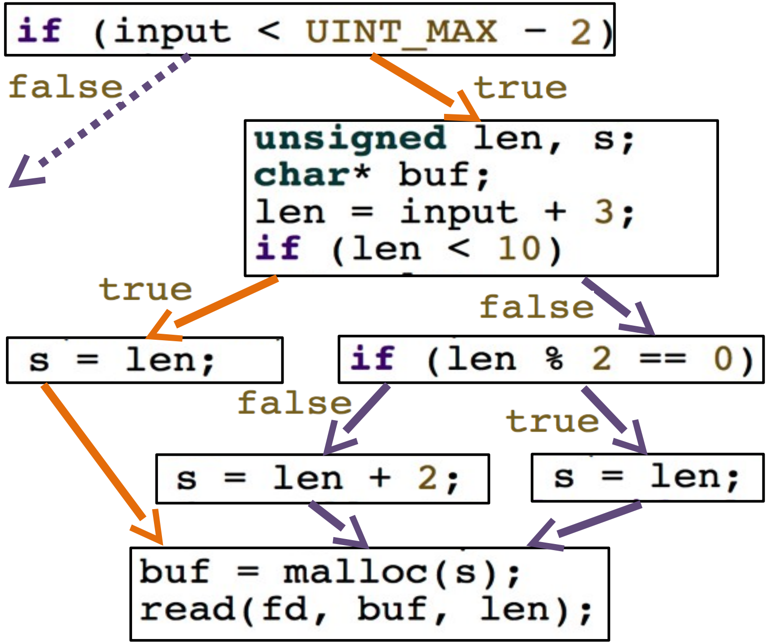
required for full coverage using that metric

A technique is

- *efficient* if the minimum value is close to expected value
- *not efficient* if minimum << expected value

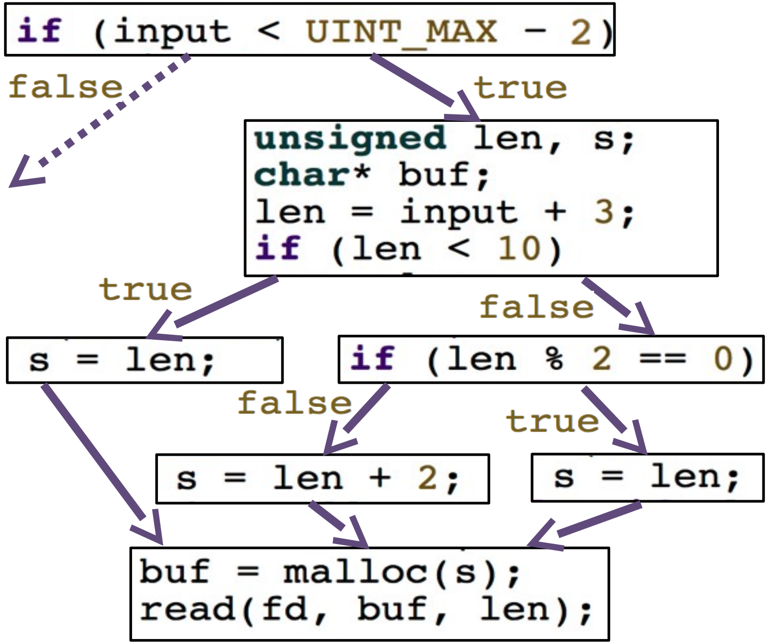
Inputs and Paths

input
3
6
4



Inputs and Paths

input
3
6
4

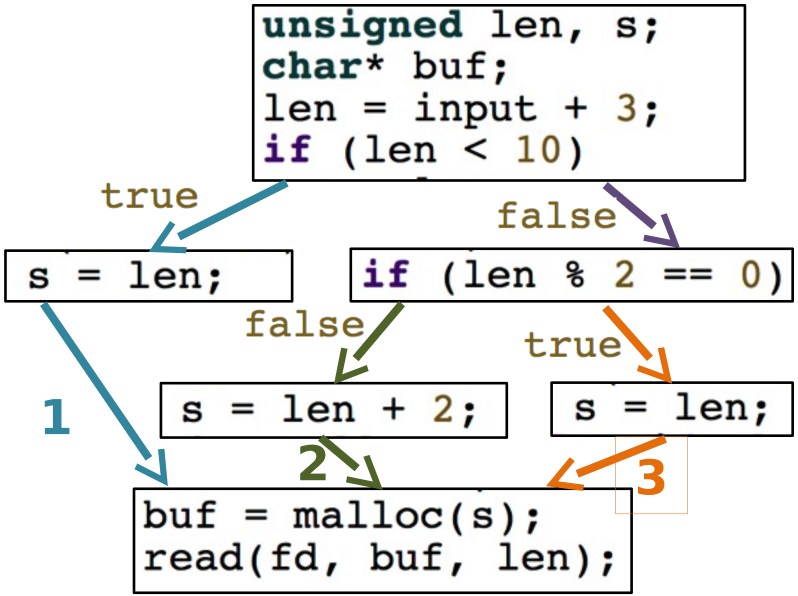


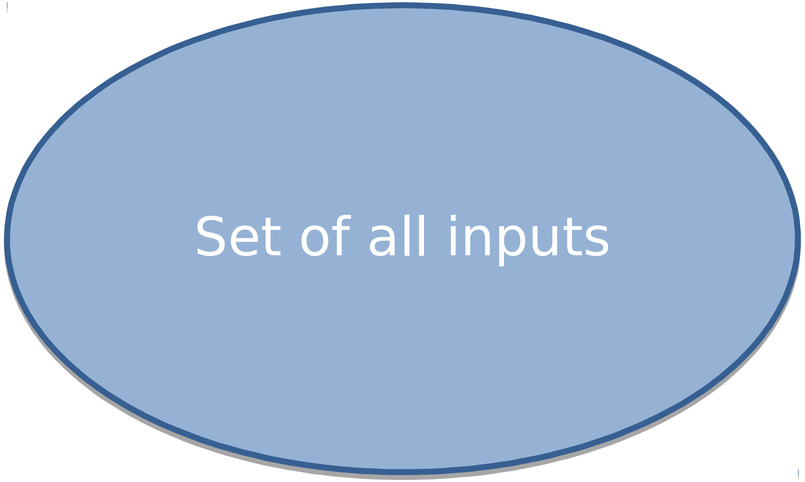
There are many examples where
minimum # << expected #
of inputs for random fuzzing.

Can we do better if we take program structure into account?

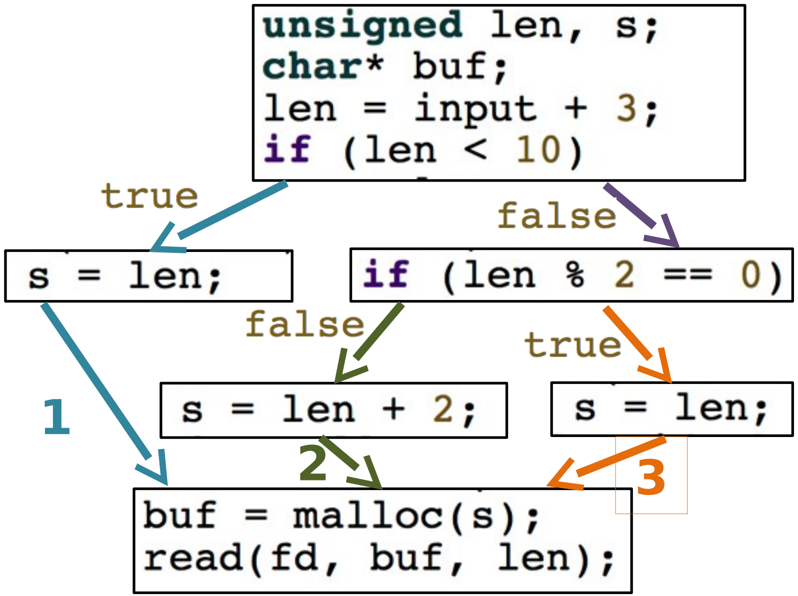
1	Efficiency of Fuzzing
2	Symbolic Reasoning
3	Path Predicates
4	Bug Finding

Focus on Sets of Values

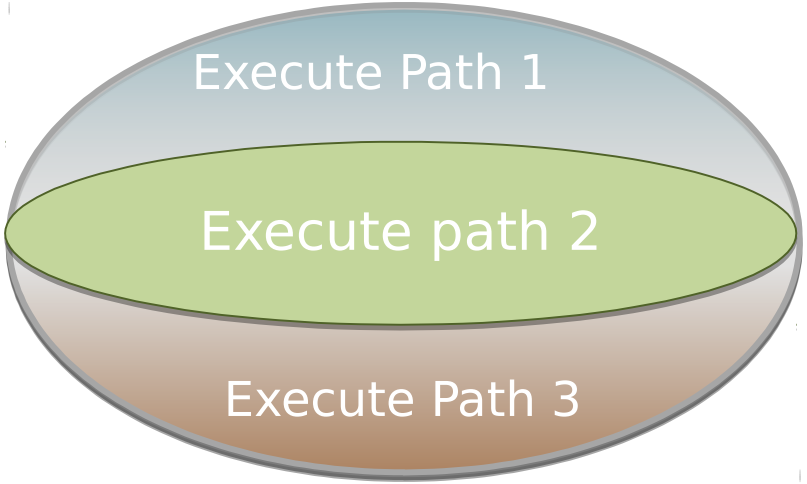




Focus on Sets of Values



Inputs that



Goal: find one element of each set

Symbolic analysis provides a way to directly manipulate sets

Symbolic vs. Explicit Representation

Explicit representation

x	-3	-1	1	3
y	0	2	4	6

x	-7	-5	-3	-1	1	3	5	7
y	-4	-2	0	2	4	6	8	10

x	...	-5	-3	-1	1	3	5	...
y	...	-2	0	2	4	6	8	...

Symbolic representation

x > -4 && x< 4
&& x % 2 == 1 && y == x + 3

x > -8 && x < 8
&& x % 2 == 1 && y == x + 3

x % 2 == 1 && y == x + 3

Symbolic Representation

A symbolic representation encodes a set of values in terms of properties of those values.

Representation	Example	Set Represented
Formula	x > 8 && x%4 = 0 && x < 24	8, 12, 16, 20
Regular expression	report_*[012].pdf	report_0.pdf, report0.pdf, report_1.pdf,...

Tradeoff of Symbolic Representation

Advantages

- Can be exponentially smaller than explicit representation of finite sets
- Can represent infinite sets (e.g. regular expressions)
- Generic algorithms (e.g. same algorithms for a certain type of formulas)

Tradeoff

- Performing basic operations may be expensive
- Specialized algorithms are required
- Difficult to predict size of representation

Satisfiability

A formula is *satisfiable* if there is a way to assign values to variables and make the formula.

(x > 0 && x < 20 && x == y + y) is *satisfied* by (x:10,y:5)

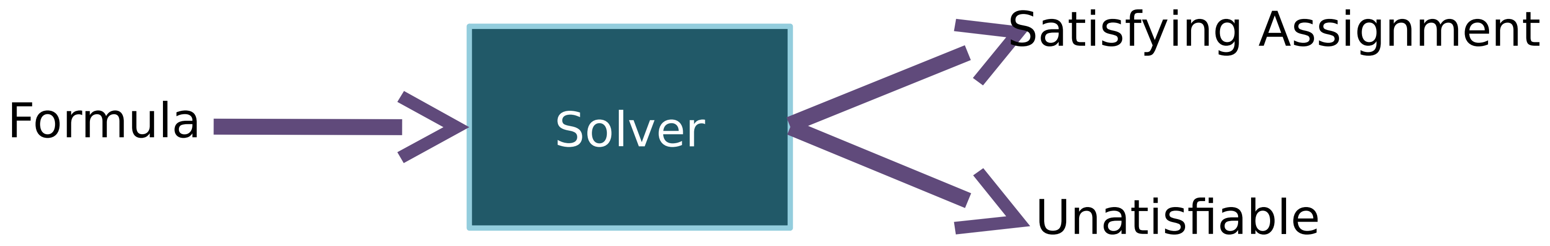
(x > 0 && x < 20 && x == y + y) is *not satisfied* by (x:13,y:6)

A formula is satisfied by a *satisfying assignment.*

A formula is *unsatisfiable* if every assignment of values to variables makes the formula false

(x > 0 && x < 20 && x == y + y && x%2 == 1) is *unsatisfiable*

Solvers



A *solver* determines if a formula is satisfiable.

- A SAT solver is a solver for propositional logic
- An SMT solver is a solver for formulas in a first-order logic

Theories

A *theory* specifies the meaning of special symbols.

Theory	Symbols	Operations
Natural numbers	0,1,2, +, - , …	Standard
Bit-Vectors	0,1,2,+,-, ^, &, | , …	Bitwise operations, machine arithmetic
Strings	a,b,c, a.b, e*, …	Concatenation, Kleene-star, etc.
Arrays	a, a[x], <=, a[x] +4, …	Indexing, reading, comparison

Examples of Solvers for Specific Theories

STP	Bit-vectors and arrays https://sites.google.com/site/stpfastprover/
Hampi	Strings, Perl-like regular expressions http://people.csail.mit.edu/akiezun/hampi/
Kaluza	String expressions http://webblaze.cs.berkeley.edu/2010/kaluza/
Beaver	Bit-vectors http://uclid.eecs.berkeley.edu/jha/beaver-dist/beaver.html

Examples of Solvers for Multiple Theories

Z3	Equality, inear, non-linear arithmetic, arrays, bit-vectors, etc. http://z3.codeplex.com/
CVC4	Equality, linear arithmetic, arrays, bit-vectors, strings, etc. http://cvc4.cs.nyu.edu/web/
YICES	Equality, linear arithmetic, bit-vectors, arrays, lambda expressions http://yices.csl.sri.com/
MATHSAT	Linear arithmetic, bit-vectors, floating-point http://mathsat.fbk.eu/

1	Efficiency of Fuzzing
2	Symbolic Reasoning
3	Path Predicates
4	Bug Finding

SELECT--A FORMAL SYSTEM FOR
TESTING AND DEBUGGING PROGRAMS
BY SYMBOLIC EXECUTION*

Robert S. Boyer
Bernard Elspas
Karl N. Levitt
Computer Science Group
Stanford Research Institute
Menlo Park, California 94025

ACM 1975

A PROGRAM TESTING SYSTEM*

Lori A. Clarke
Computer and Information Science Dept.
University of Massachusetts
Amherst, Massachusetts 01002

ACM 1976

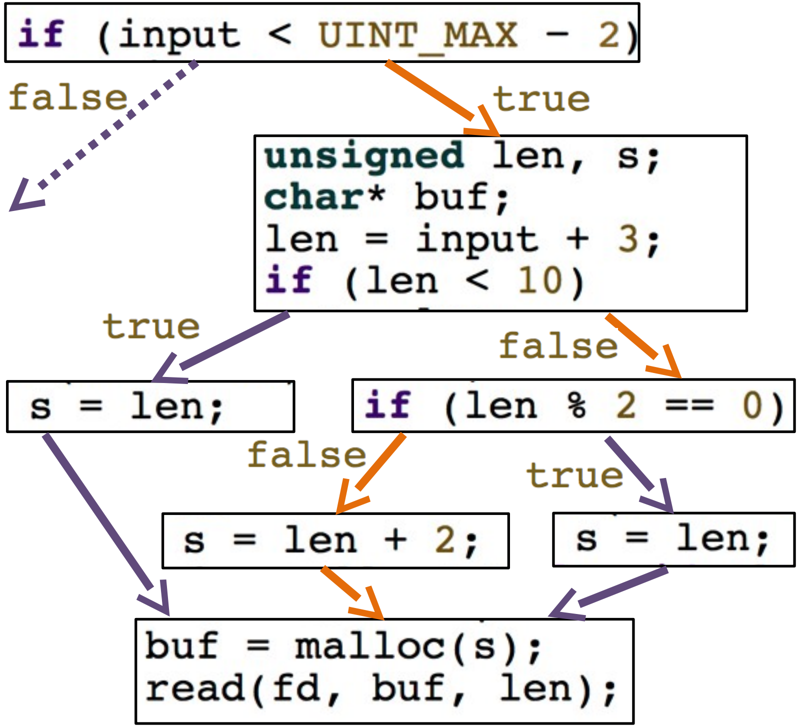
Programming Languages — B. Wegbreit, Editor

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

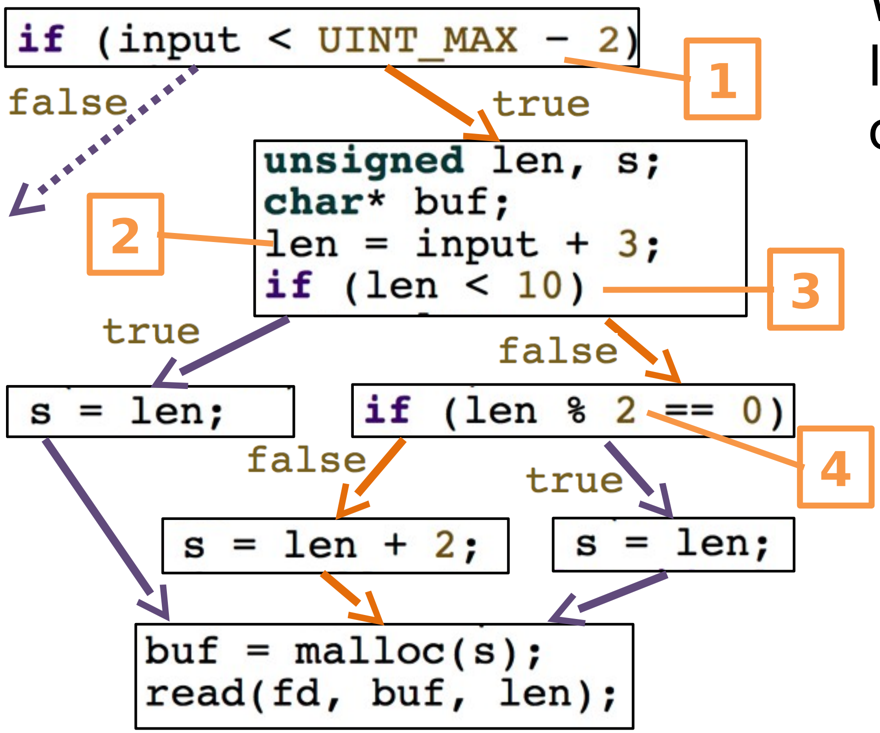
CACM 1976

Paths as Formulas



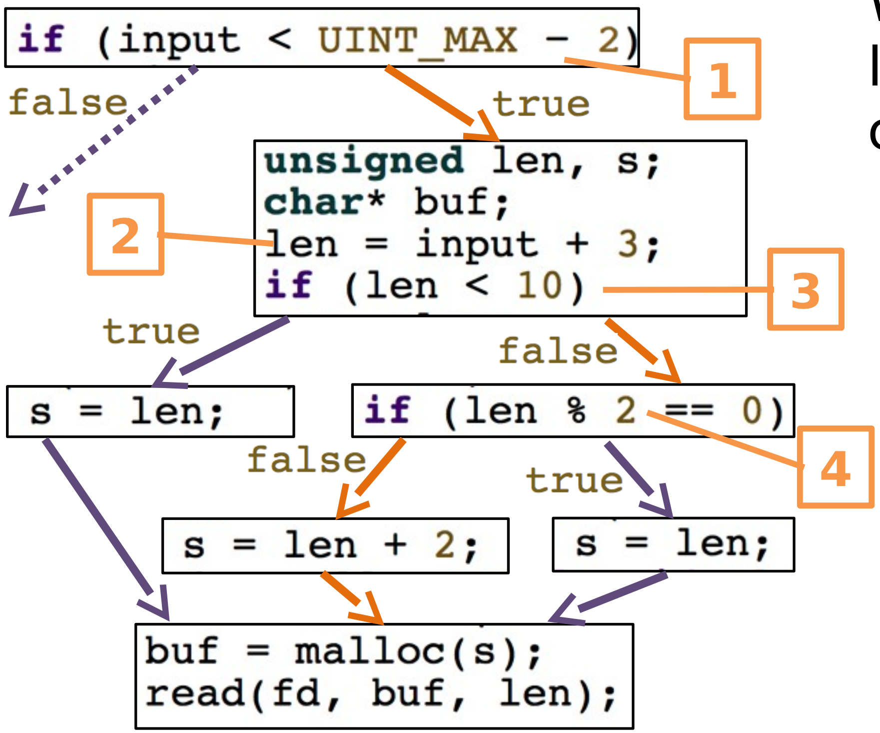
Write a formula for the values of len and input that execute the colored path.

Paths as Formulas



Write a formula for the values of len and input that execute the colored path.

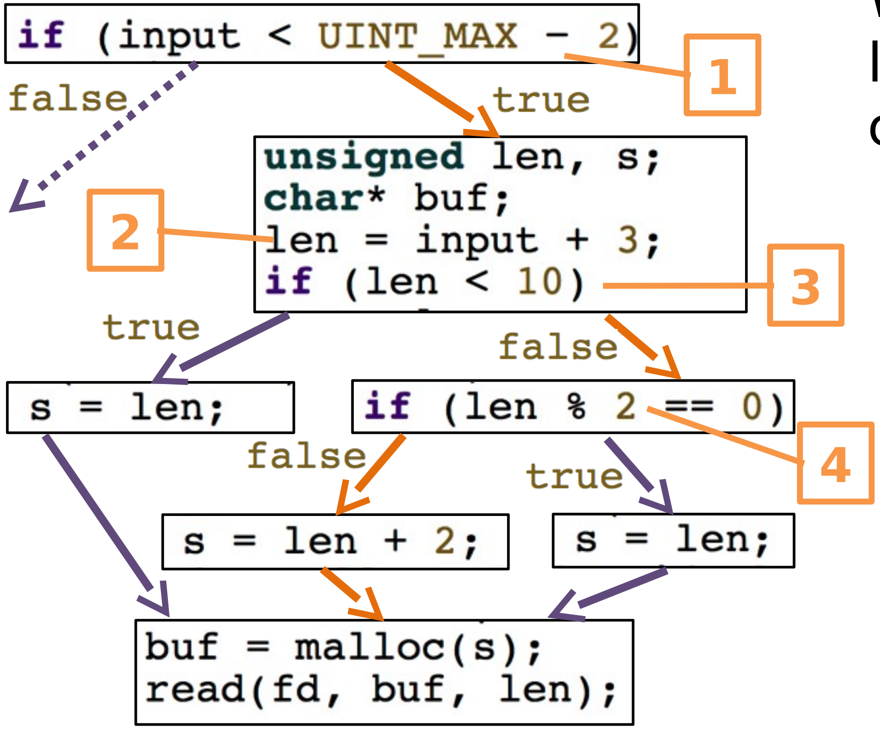
Paths as Formulas



Write a formula for the values of len and input that execute the colored path.

1	input < UINT_MAX - 2

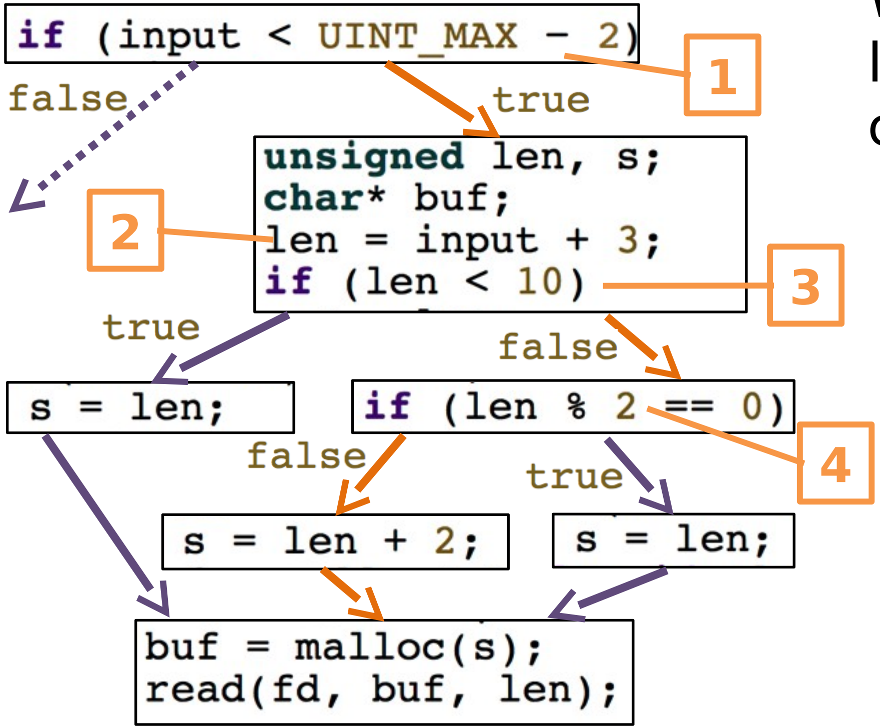
Paths as Formulas



Write a formula for the values of len and input that execute the colored path.

1		input < UINT_MAX - 2
2	&&	len == input + 3

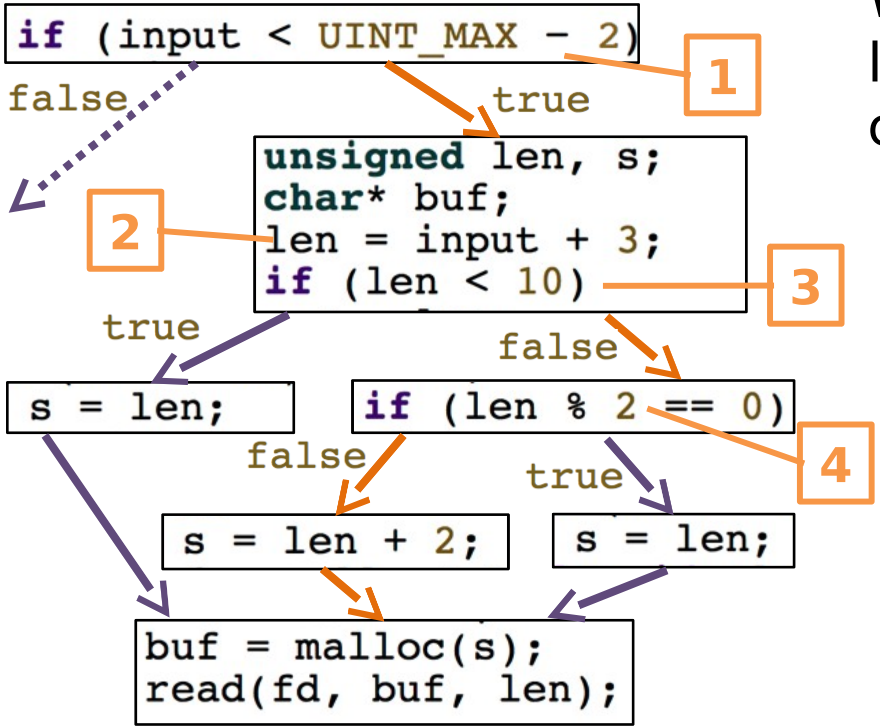
Paths as Formulas



Write a formula for the values of len and input that execute the colored path.

1		input < UINT_MAX - 2
2	&&	len == input + 3
3	&&	! (len < 10)

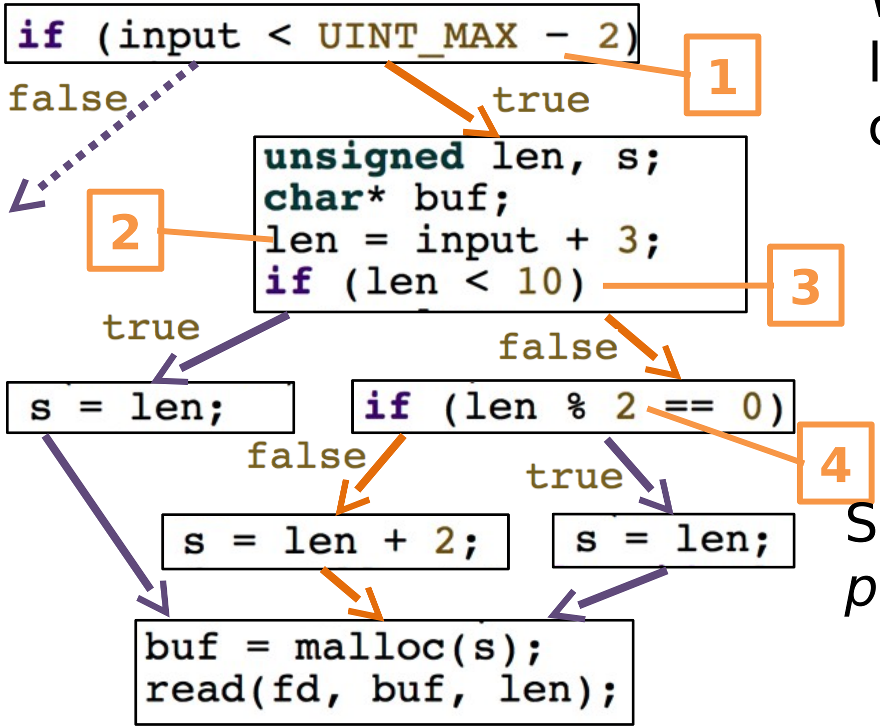
Paths as Formulas



Write a formula for the values of len and input that execute the colored path.

1		input < UINT_MAX - 2
2	&&	len == input + 3
3	&&	! (len < 10)
4	&&	! (len % 2 == 0)

Paths as Formulas



Write a formula for the values of len and input that execute the colored path.

1		input < UINT_MAX - 2
2	&&	len == input + 3
3	&&	! (len < 10)
4	&&	! (len % 2 == 0)

Satisfying assignments to the *path predicate*:

input	8	10	12	14	16	18	…
len	11	13	15	17	19	21	…

Path Predicates

A *path predicate* encodes the constraints that must be satisfied for a program path to be executed.

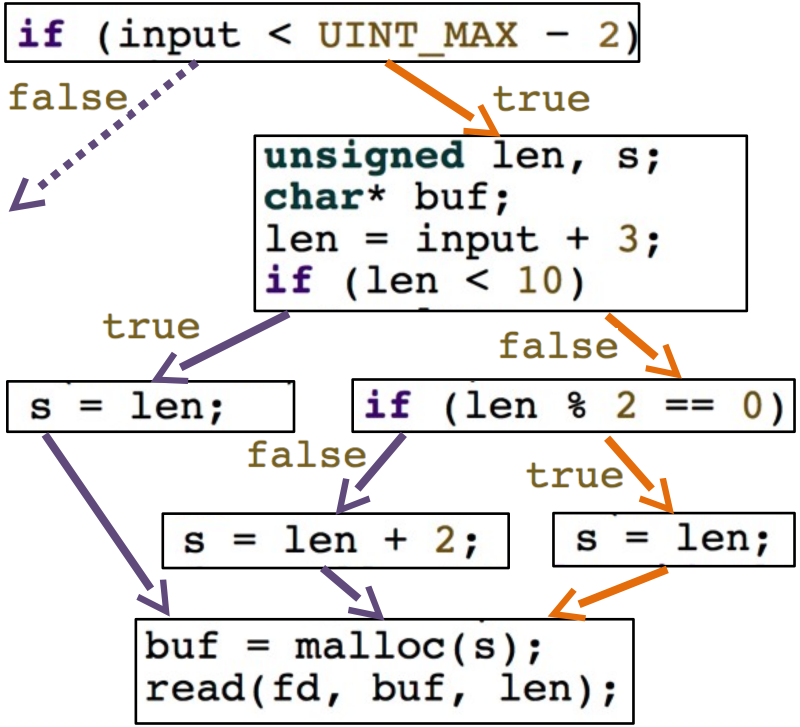
It symbolically represents all inputs for executing the path.

To construct a path predicate

- Rename variables to have unique occurrences
- Assignments become equalities
- Branches are themselves, or negated
- Sequence is conjunction

Theory used should support a proper model of program statements and memory model

Quiz: Path Predicates



Write a formula for the values of len and input that execute the colored path.

1	Efficiency of Fuzzing
2	Symbolic Reasoning
3	Path Predicates
4	Bug Finding

Quiz: Spot the Bug

Can you spot the bug involving the integer variables?

```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else
      s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
     ....
  }
}
```

Quiz: Spot the Bug

Can you spot the bug involving the integer variables?

```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else
      s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
     ....
  }
}
```

Quiz: Spot the Bug

Can you add an assertion to catch the bug?

```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else
      s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
     ....
  }
}
```

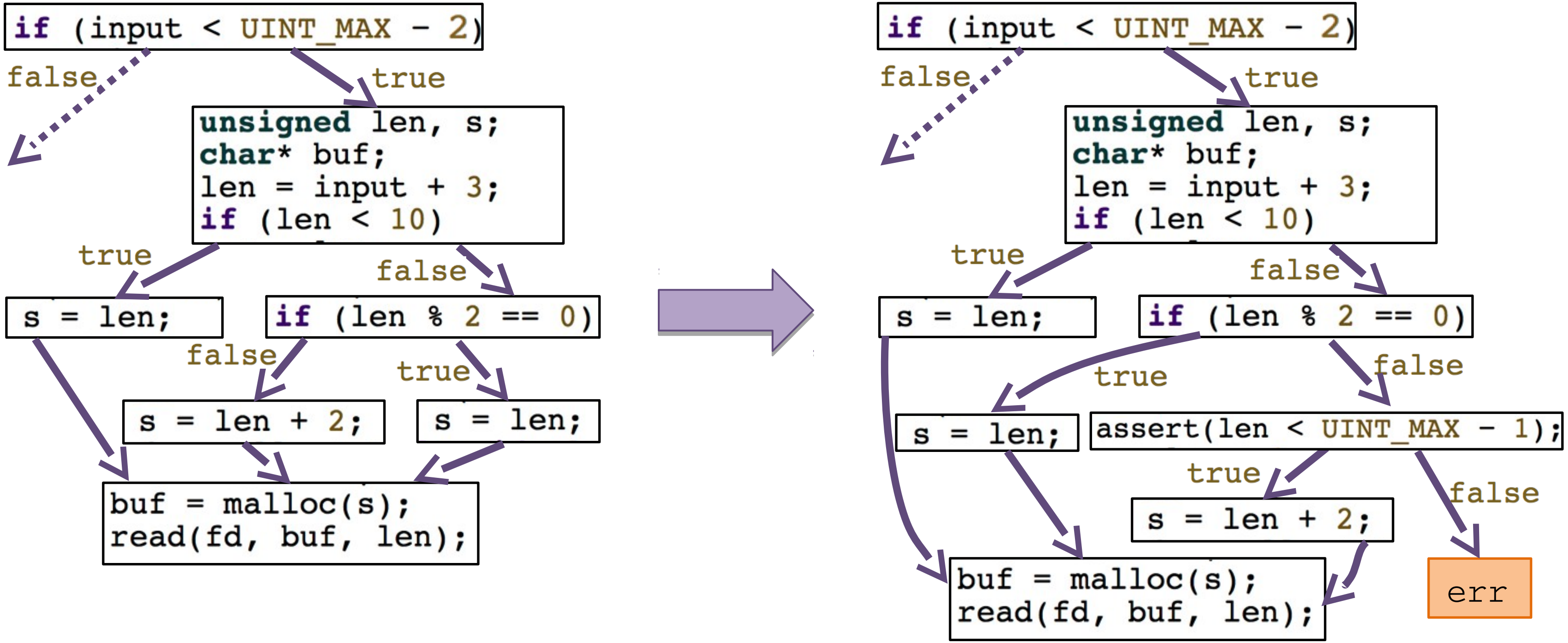
Quiz: Spot the Bug

Can you add an assertion to catch the bug?

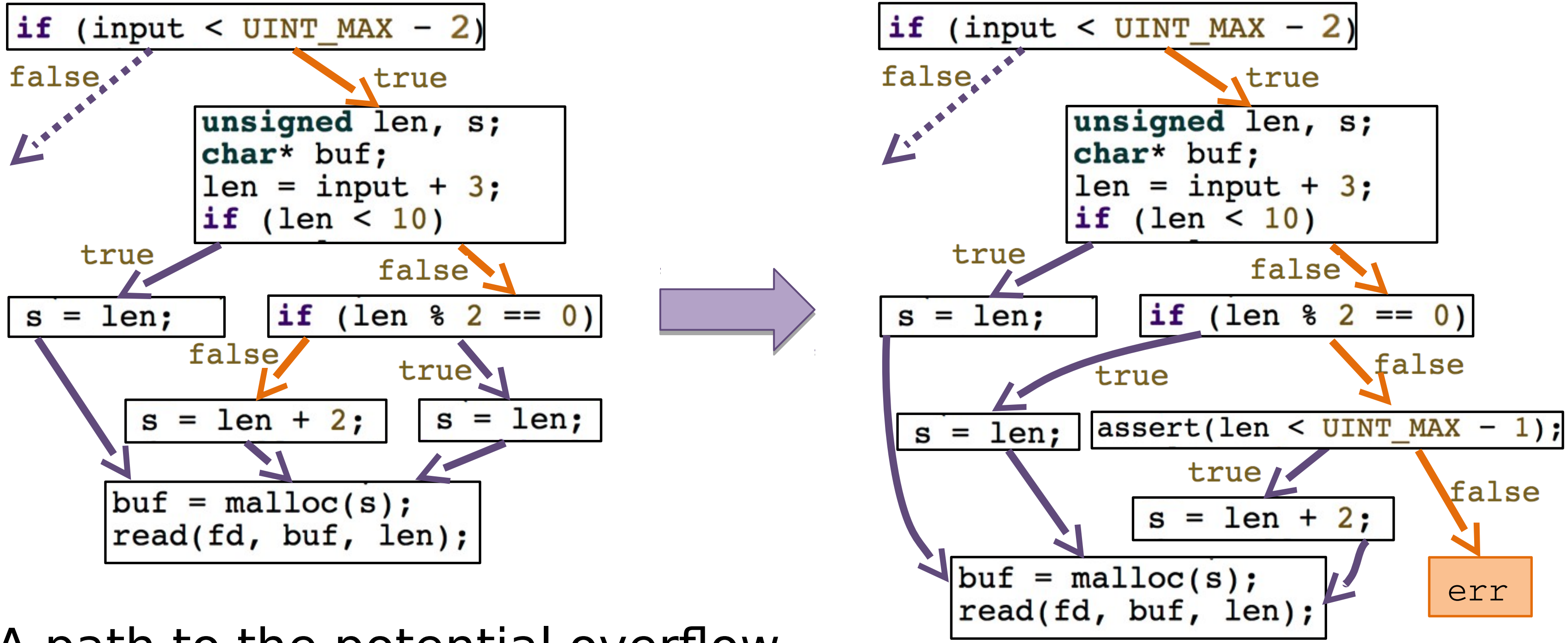
```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else
      s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
     ....
  }
}
```

```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else {
      assert(len < UINT_MAX - 1);
      s = len + 2;
    }
    buf = malloc(s);
    read(fd, buf, len);
     ....
  }
}
```

Adding Assertion to the CFG



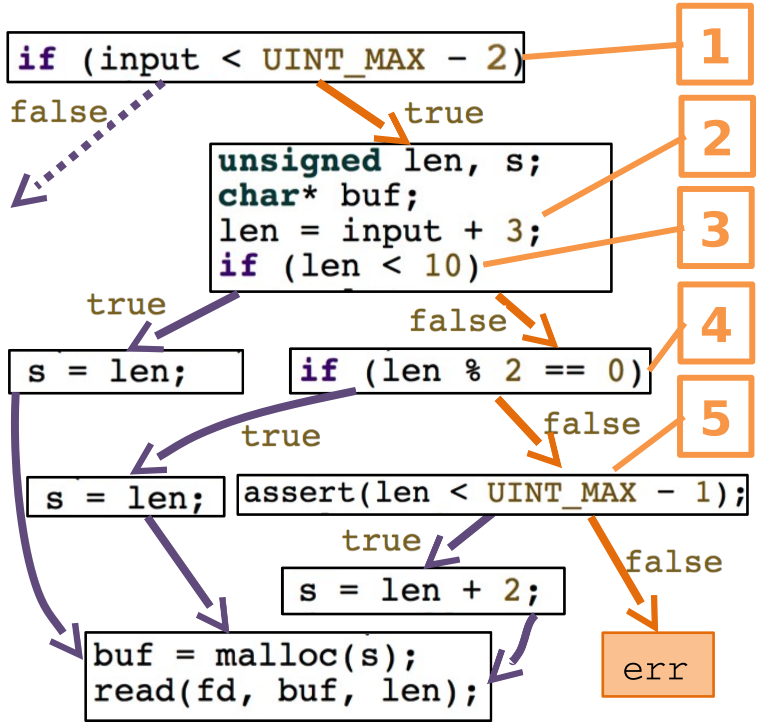
Adding Assertion to the CFG



A path to the potential overflow becomes a path to a potential assertion violation.

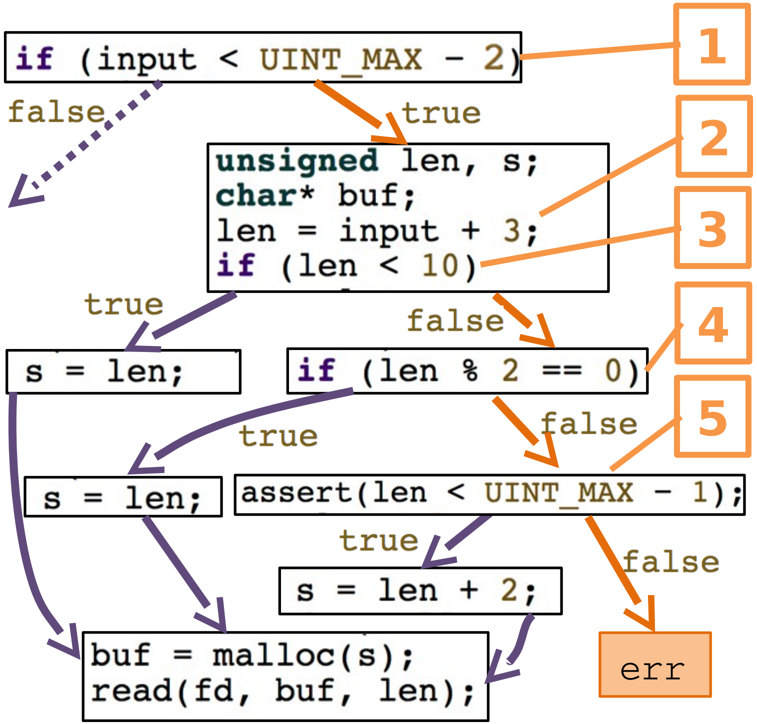
Dawn Song

Path Predicate for Assertion Violation



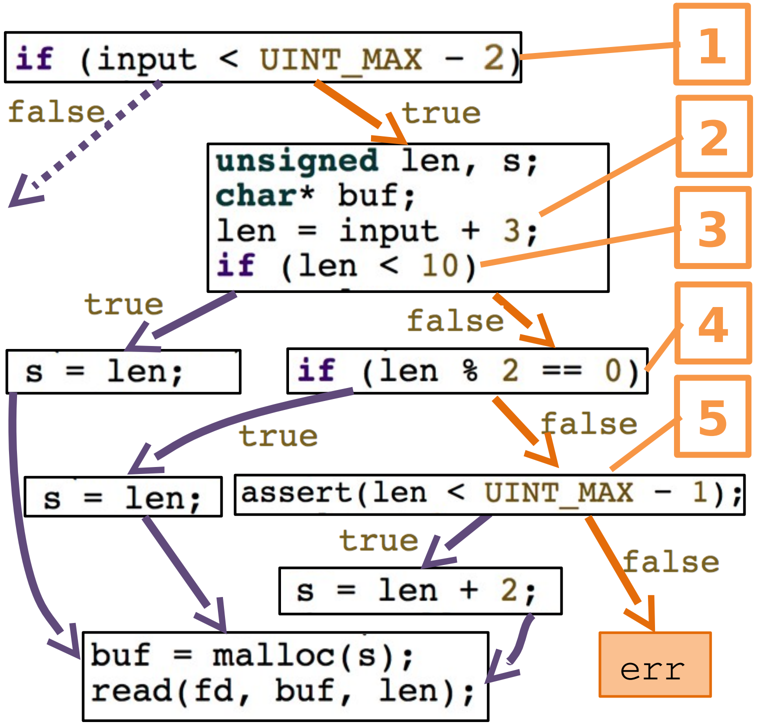
Path Predicate for Assertion Violation

1	input < UINT_MAX - 2
2	&& len == input + 3
3	&& ! (len < 10)
4	&& ! (len % 2 == 0)



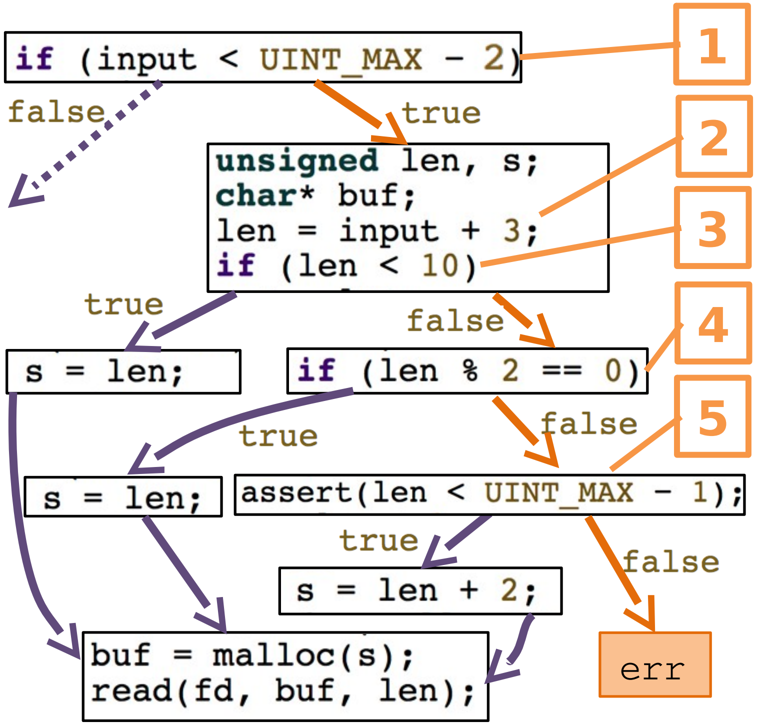
Path Predicate for Assertion Violation

1		input < UINT_MAX - 2
2	&&	len == input + 3
3	&&	! (len < 10)
4	&&	! (len % 2 == 0)
5	&&	!(len < UINT_MAX – 1)



Path Predicate for Assertion Violation

1		input < UINT_MAX - 2
2	&&	len == input + 3
3	&&	! (len < 10)
4	&&	! (len % 2 == 0)
5	&&	!(len < UINT_MAX – 1)



In a theory that correctly encodes the program's semantics, this formula is satisfiable if and only if the assertion can be violated

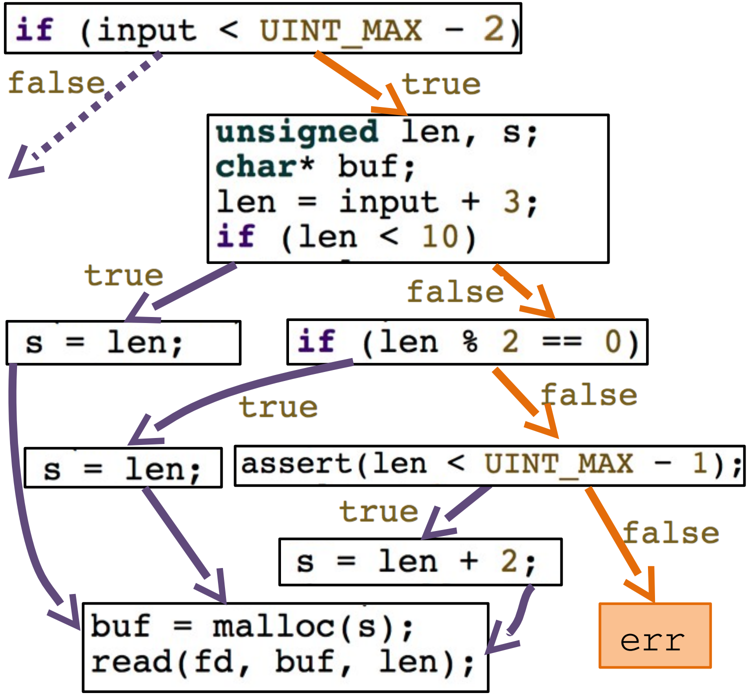
Assertion Violation as Satisfiability

In the appropriate theory, the formula

	input < UINT_MAX - 2
&&	len == input + 3
&&	! (len < 10)
&&	! (len % 2 == 0)
&&	!(len < UINT_MAX - 1)

is satisfied by the assignment

input	UINT_MAX - 3
len	UINT_MAX



Constraint-Based Automatic Test Case Generation

- Look inside the box
  - Use the code itself to guide the fuzzing
- Encode security/safety properties as assertions
- Explore program paths on which assertions occur
- Steps involved
  1. Find inputs going down different execution paths
  2. For a given path, check if there are inputs that cause a violation of the security property

Articles about Current Symbolic Execution Tools

DART	*DART: Directed Automated Random Testing*, Godefroid, Klarlund, Sen, PLDI 2005 http://dl.acm.org/citation.cfm?id=1065036
CUTE	*CUTE: A Concolic Unit Testing Engine for C,* Sen, Marinov, Agha, ESEC/FSE 2005 http://dl.acm.org/citation.cfm?id=1081750
KLEE	*KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems* Programs, Cadar, Dunbar, Engler, OSDI 2008 https://www.usenix.org/legacy/event/osdi08/tech/full_papers/cadar/cadar_html/

Articles about Symbolic Execution for Security

BitBlaze	*BitBlaze: A New Approach to Computer Security via Binary Analysis*, Song, Brumley, Yin, Caballero, Jager, Kang, Liang, Newsome, Poosankam, Saxena, ICISS 2008 http://bitblaze.cs.berkeley.edu/papers/bitblaze_iciss08.pdf
BAP	*BAP: A Binary Analysis Platform*, Brumley, Jager, Avgerinos, Schwartz, CAV 2011 http://www.ece.cmu.edu/~ejschwar/papers/cav11.pdf
S2E	*S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems,* Chipounov, Kuznetsov, Candea, ASPLOS 2011 http://dslab.epfl.ch/pubs/s2e.pdf?attredirects=0
SAGE	*SAGE: Automated Whitebox Fuzzing for Security Testing*, Godefroid, Levin, Molnar, CACM 2012 http://dl.acm.org/citation.cfm?id=2093564

Summary of Symbolic Execution for Bug Finding

- Augment a program with appropriate assertions
- Symbolically execute a path
  - Create formula representing path constraint and assertion failure
  - Solve constraints with a solver
  - A satisfying assignment, if found, is an input triggering a bug
- Reverse a branch condition to explore a different path
  - Give solver the new constraint
  - If the constraint is satisfiable
    - The path is feasible
    - There is an input going down a *different path*