IS593: Language-based Security

4. Concepts in Language-based Security

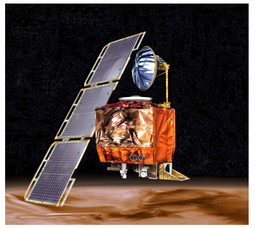
Kihong Heo

KAIST

Impact of Poor Software Quality

The Patriot Missile (1991)
Floating-point roundoff
28 soldiers died

The Ariane-5 Rocket (1996)
Integer Overflow
$100M



NASA's Mars Climate Orbiter (1999)
Meters-Inches Miscalculation
$125M

Cost of Software Quality Assurance

*“We have as **many testers** as we have developers.*
*And testers spend **all their time testing**, and developers spend **half their time testing**. We're more of a testing, a quality software organization than we’re a software organization”*
- Bill Gates

Q: What is the solution to improve software quality at low cost?

A: Program analysis

What to Analyze?

CWE Definitions

Sort Results By : CWE Number Vulnerability Count

Total number of cwe definitions : 668 Page : 1 (This Page) 2 3 4 5 6 7 8 9 10 11 12 13 14

Select Select&Copy

CWE Number	Name	Number Of Related Vulnerabilities
119	Failure to Constrain Operations within the Bounds of a Memory Buffer	12328
79	Failure to Preserve Web Page Structure ('Cross-site Scripting')	11807
20	Improper Input Validation	7669
200	Information Exposure	6316
89	Improper Sanitization of Special Elements used in an SQL Command ('SQL Injection')	5643
22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	2968
94	Failure to Control Generation of Code ('Code Injection')	2400
125	Out-of-bounds Read	2122
287	Improper Authentication	1746
284	Access Control (Authorization) Issues	1627
416	Use After Free	1256
190	Integer Overflow or Wraparound	1113
476	NULL Pointer Dereference	900
78	Improper Sanitization of Special Elements used in an OS Command ('OS Command Injection')	788
787	Out-of-bounds Write	737
362	Race Condition	615
59	Improper Link Resolution Before File Access ('Link Following')	518
77	Improper Sanitization of Special Elements used in a Command ('Command Injection')	489
400	Uncontrolled Resource Consumption ('Resource Exhaustion')	463
611	Information Leak Through XML External Entity File Disclosure	393
434	Unrestricted Upload of File with Dangerous Type	385
732	Incorrect Permission Assignment for Critical Resource	350
74	Failure to Sanitize Data into a Different Plane ('Injection')	327
798	Use of Hard-coded Credentials	319
772	Missing Release of Resource after Effective Lifetime	306
269	Improper Privilege Management	305
601	URL Redirection to Untrusted Site ('Open Redirect')	265
502	Deserialization of Untrusted Data	257
134	Uncontrolled Format String	216
704	Incorrect Type Conversion or Cast	180
415	Double Free	173

Heartbleed, 2019
OpenSSL
CVE-2014-0160

Shellshock, 2014
Bash
CVE-2014-6271

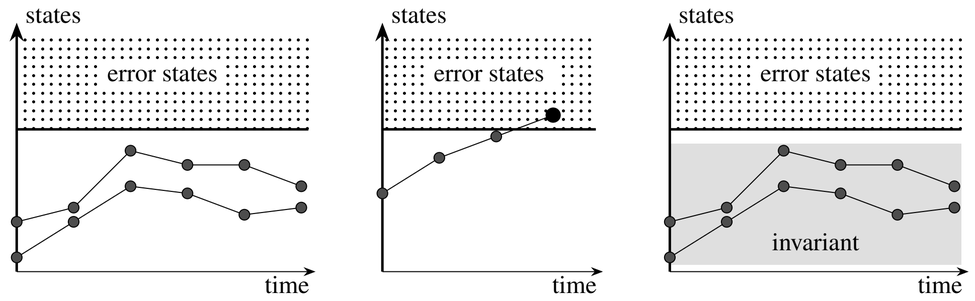
goto fail, 2014
MacOS / iOS
CVE-2014-1266

Properties

- Points of interest in programs
  - for verification, bug detection, optimization, understanding, etc
- In this lecture
  - safety properties
  - liveness properties
  - information-flow properties

Safety Property

- A program **never** exhibit a behavior observable within **finite time**
  - “Bad things will never occur”
  - If false, then there exists a **finite counterexample**
- Bad things: integer overflow, buffer overrun, deadlock, etc
- To prove: all executions never reach error states



(a) Correct executions (b) An incorrect execution (c) Proof by invariance

Invariant

- Assertions supposed to be **always true**
  - Starting from a state in the invariant: any computation step also leads to another state in the invariant (i.e., fixed point!)
  - E.g., “x has an int value during the execution”, “y is larger than 1 at line 5”
- Loop invariant: assertion to be true at the beginning of every loop iteration

```
x = 0;
while (x < 10) {
  x = x + 1;
}
```

Loop invariant 1: “x is an integer”

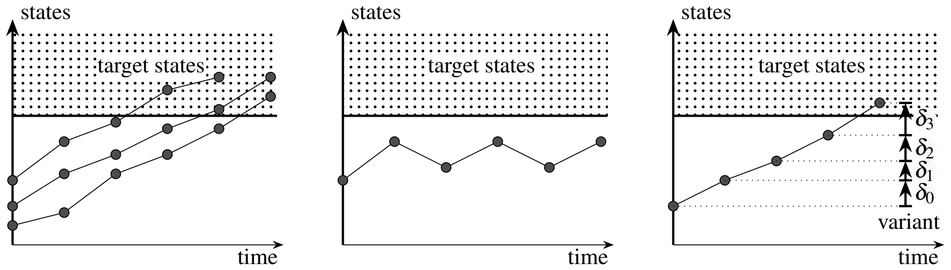
Loop invariant 2: “0 <= x < 10”

Example: Division-by-Zero

```
1: int main(){
2:   int x = input();      // True
3:   x = 2 * x - 1;        // x is an odd number
4:   while (x > 0) {       // x is a positive odd number
5:     x = x - 2;
6:   }                     // x is an odd number
7:   assert(x != 0);
8:   return 10 / x;
9: }
```

Liveness Property

- A program will **never** exhibit a behavior observable only after **infinite time**
  - “Good things will eventually occur”
  - If false then there exists an **infinite counterexample**
- Good things: termination, fairness, etc
- To prove: all executions eventually reach target states



(a) Correct executions (b) An incorrect execution (c) Proof by variance

Variant

- A quantity that **evolves towards** the set of target states
  (so guarantee any execution eventually reach the set)
- Usually, a value that is strictly decreasing for some well-founded order relation
  - Well-founded order: there exists a minimal element
  - E.g.) an expression of integer type that always takes a positive value and strictly decreasing

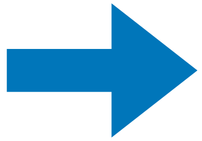
```
x = pos_int();
while (x > 0) {
  x = x - 1;
}
```

x is always a positive integer $\wedge$ **x is strictly decreasing** $\Rightarrow$ **The program terminates**

Example: Termination

- Introduce variable c that stores the value of “step counter”
  - Initially, c is equal to zero
  - Each program execution step increments c by one

```
// A factorial program
i = n;
r = 1;
while (i > 0) {
  r = r * i;
  i = i - 1;
}
```



```
// An instrumented program
i = n;
r = 1;
c = 2;
while (i > 0) {
  r = r * i;
  i = i - 1;
  c = c + 3;
}
```

c <= 3n + 2

0 <= 3n + 2 - c /\ 3n + 2 - c is strictly decreasing ⇒ termination

Trace Properties

- A semantic property $\mathscr{P}$ that can be defined by a **set of execution traces** that satisfies $\mathscr{P}$
  - Safety and liveness properties are trace properties

$$[\![P]\!] \subseteq T_{ok}$$

- State properties: defined by a set of states (so, obviously trace properties)
  - E.g., division-by-zero, integer overflow
- Any trace property: the conjunction of a safety and a liveness property

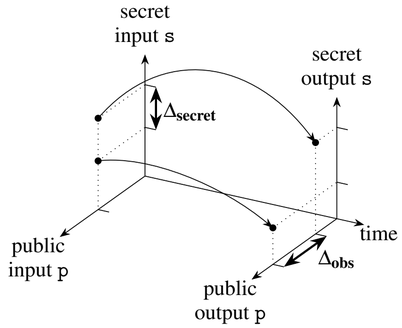
Example

- Correctness of a sorting algorithm as trace property

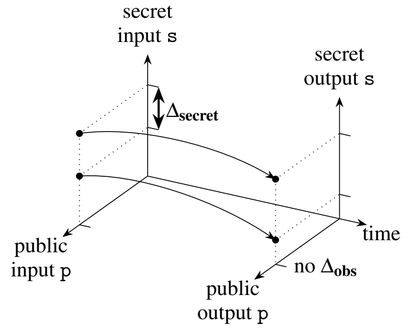
	Safety or Liveness?	State?
Should not fail with a run-time error	Safety	O
Should terminate	Liveness	-
Should return a sorted array	Safety	O
Should return an array with the same elements and multiplicity	Safety	X

Information Flow Properties

- Properties stating the absence of dependence between **pairs of executions**
  - Beyond trace properties: so called **hyperproperties**
- Mostly used for security purposes:
  - e.g.) multiple executions with public data should not derive private data



A pair of executions with insecure information flow



A pair of executions without insecure information flow

Example

- Assume that variables s (secret) and p (public) take only 0 and 1

```
// Program 0
p_out := p_in
```

Input		Output
p	s	p
0	0	0
0	1	0
1	0	1
1	1	1

```
// Program 1
p_out := s * p_in
```

Input		Output
p	s	p
0	0	0
0	1	0
1	0	0
1	1	1

```
// Program 2
p_out := |rand(p_in) - s|
```

Input		Output
p	s	p
0	0	0 or 1
0	1	0 or 1
1	0	0 or 1
1	1	0 or 1

A Hard Limit: Undecidability

Theorem (Rice's theorem). Any non-trivial semantic properties are undecidable.

- Non-trivial property: worth the effort of designing a program analyzer for
  - trivial: true or false for all programs
- Undecidable? If decidable, it can solves the Halting problem
  - An analyzer **A** for a property: "This program always prints 1 and finishes"
  - Given a program **P**, generate "**P**; print 1;"
  - **A** says "Yes": **P** halts, **A** says "No": **P** does not halt

Toward Computability

Undecidable

⇒ Automatic, terminating, and exact reasoning is impossible

⇒ If we give up one of them, it is computable!

- **Manual** rather than **automatic**: assisted proving
  - require expertise and manual effort
- **Possibly nonterminating** rather than **terminating**: model checking, testing
  - require stopping mechanisms such as timeout
- **Approximate** rather than **exact**: static analysis
  - report spurious results

Soundness and Completeness

- Given a semantic property $\mathscr{P}$, and an analysis tool **A**
- If **A** were perfectly accurate,

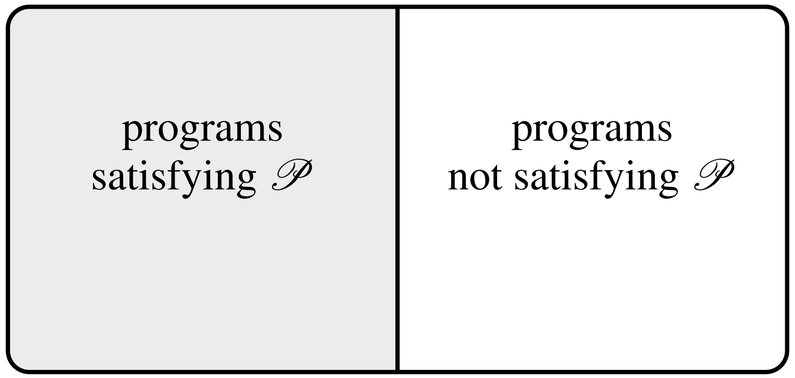
$$\text{For all program p, } \mathbf{A}(\text{p}) = \mathbf{true} \iff \text{p satisfies } \mathscr{P}$$

which consists of

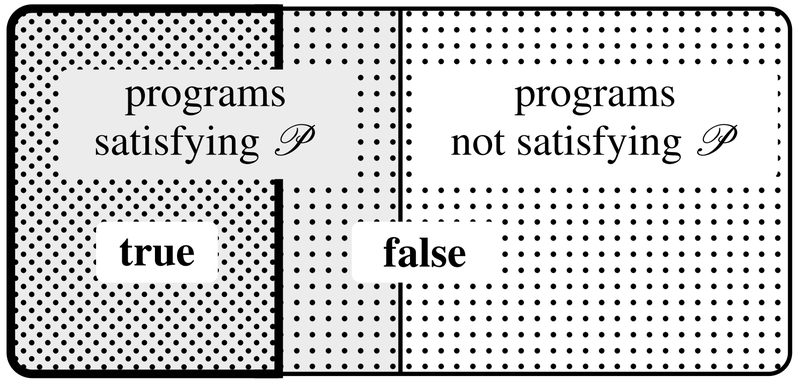
$$\text{For all program p, } \mathbf{A}(\text{p}) = \mathbf{true} \Rightarrow \text{p satisfies } \mathscr{P} \qquad \textbf{(soundness)}$$

$$\text{For all program p, } \mathbf{A}(\text{p}) = \mathbf{true} \Leftarrow \text{p satisfies } \mathscr{P} \qquad \textbf{(completeness)}$$

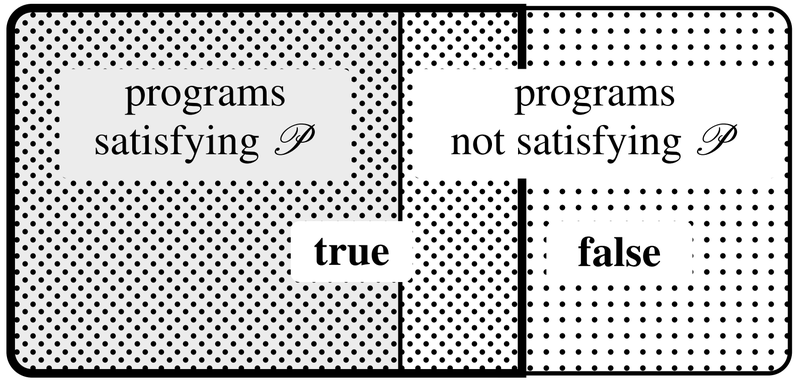
Soundness and Completeness



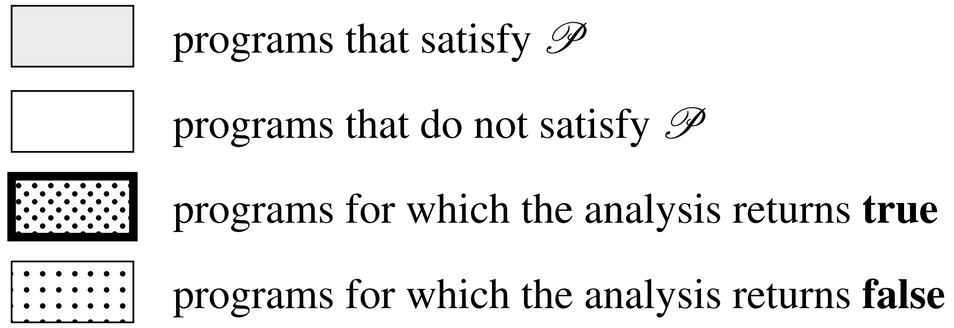
(a) Programs



(b) Sound, incomplete analysis



(c) Unsound, complete analysis



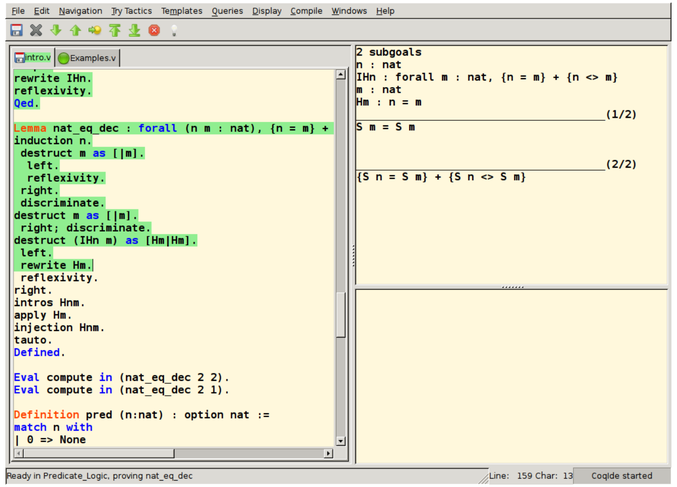
(d) Legend

Testing

- Check a set of **finite executions**
  - e.g., random testing, concolic (**conc**rete + symb**olic**) testing
- In general, **unsound yet complete**
  - Unsound: cannot prove the absence of errors
  - Complete: produce counterexamples (i.e., erroneous inputs)
- Example: Google's oss-fuzz (https://github.com/google/oss-fuzz)

Assisted Proving

- Machine-assisted proof techniques
  - Relying on user-provide invariants
  - Using proof assistants (e.g., Coq, Isabelle/HOL)

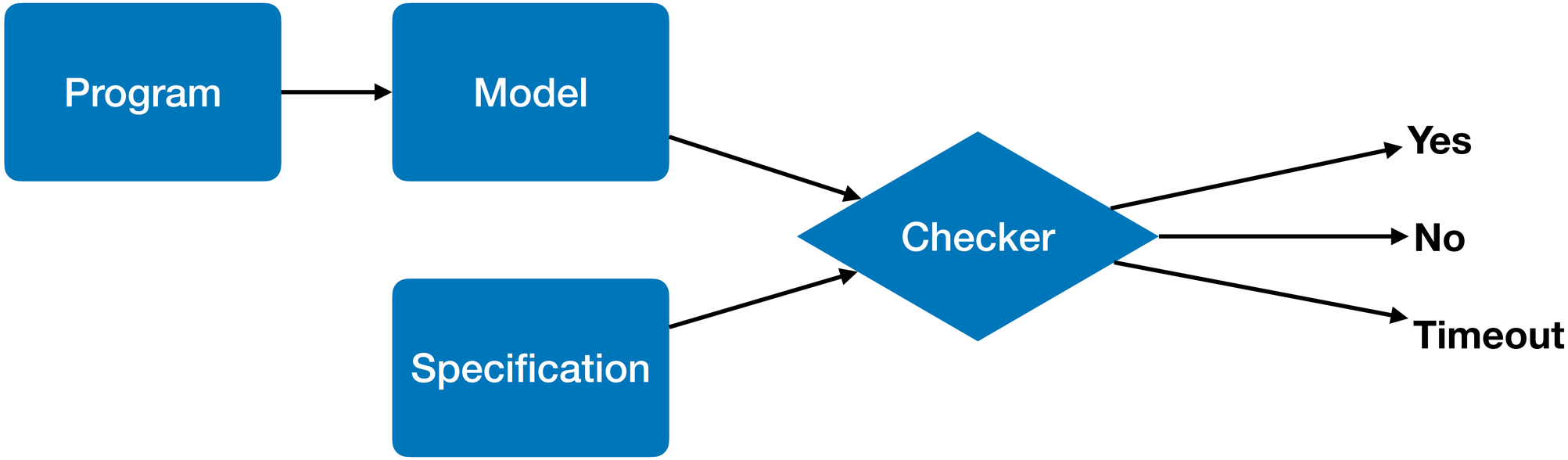


- **Sound and complete (up to the ability of the proof assistant)**
  - require manual effort / expertise
- Example: CompCert (verified C compiler), seL4 (verified microkernel)

Model Checking

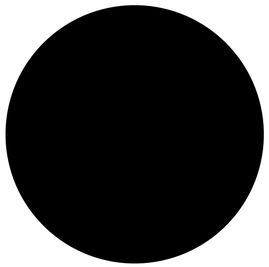
- Automatic technique to verify if a model satisfies a specification
  - Model of the target program (finite automata)
  - Specification written in logical formula
  - Verification via exhaustive search of the state space (graph reachability)
- **Sound and complete with respect to the model**
  - May incur infinite model refinement steps
- Example: SLAM (MS Windows device driver verifier)

Model Checking Overview

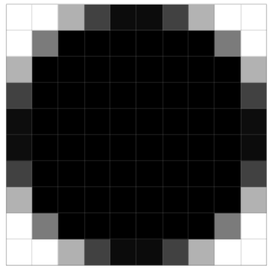


Model

- Finite state machines constructed manually or by some automatic tools
- Gap between models (finite systems) and programs (infinite systems)
  - either unsound or incomplete with respect to the target program
- Techniques to automatically refine the model on demand
  - may continue indefinitely so stopping mechanisms are required

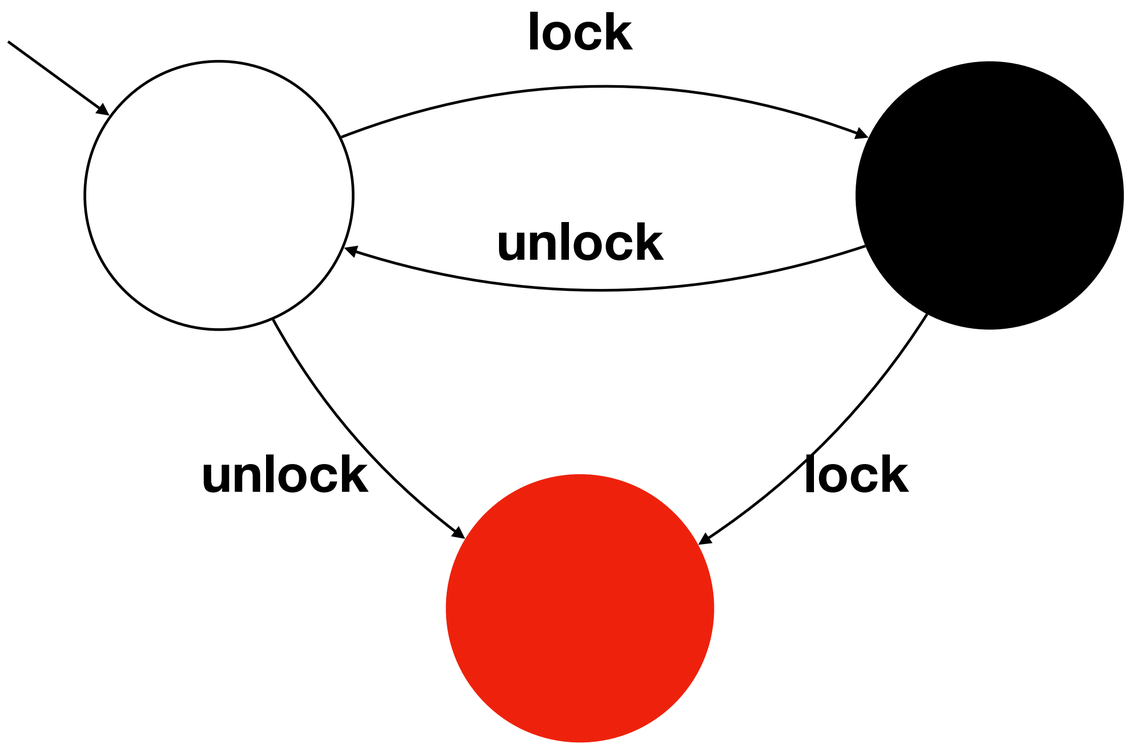


Program



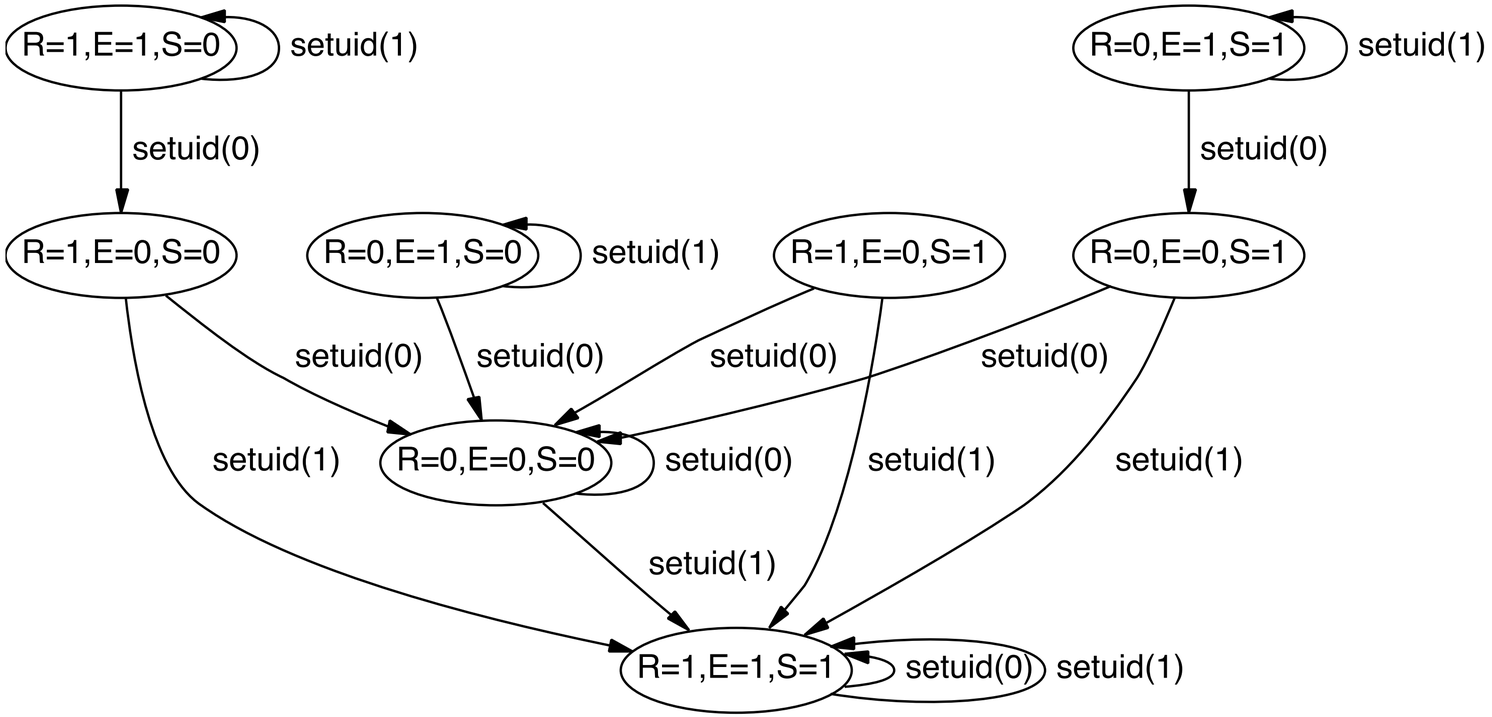
Model

Example 1: Double Locking



Calls to lock and unlock must alternate

Example: Drop Root Privilege



“User applications must not run with root privilege”

When exec is called, must have suid ≠ 0

*Hao Chen, David Wagner, and Drew Dean. Setuid Demystified, *USENIX Security Symposium*, 2002

Specification

- Written in a formal language: modal logic
  - Modal logic = propositional logic + {necessarily, possibly}
  - Esp., truth values of assertions vary with time (temporal logic)
  - E.g., LTL (linear temporal logic), CTL (computational tree logic)
- Describe assertions on program properties
  - “x is always positive”, “x can be positive”,
    “x remains positive until y is negative”, “x is positive after state s”, …

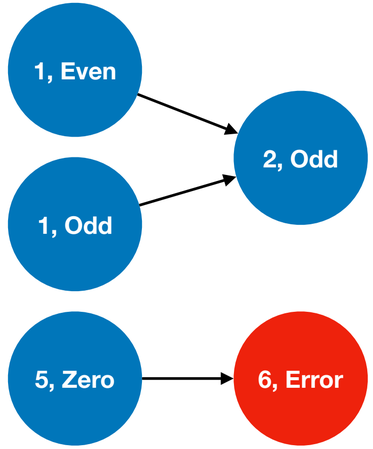
Example: Model & Specification

- State = Label × {Even, Odd, Zero, Error} : finite
- Specification: “The error state is unreachable from the initial states”
  - Initial states: {<1, Even>, <1, Odd>}

Target Program

```
  int main(){
1:  int x = input();
2:  x = 2 * x - 1;
3:  while (x > 0) {
4:    x = x - 2;
5:  }
6:  assert(x != 0);
7:  return 10 / x;
  }
```

Example transitions

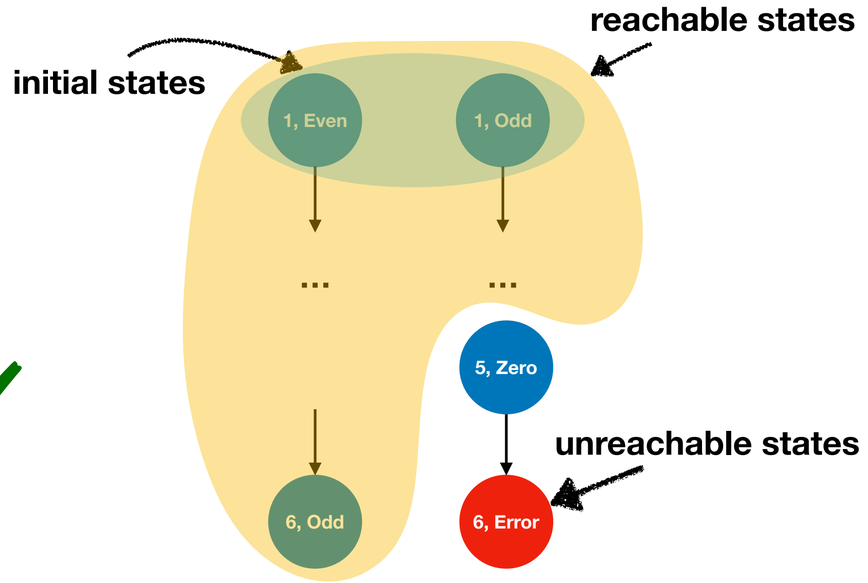


Example: Reachability Check

- Check the reachability of the error state from the initial states
  - Unreachable: verified
  - Reachable and counter example: real bug or spurious warning (why?)

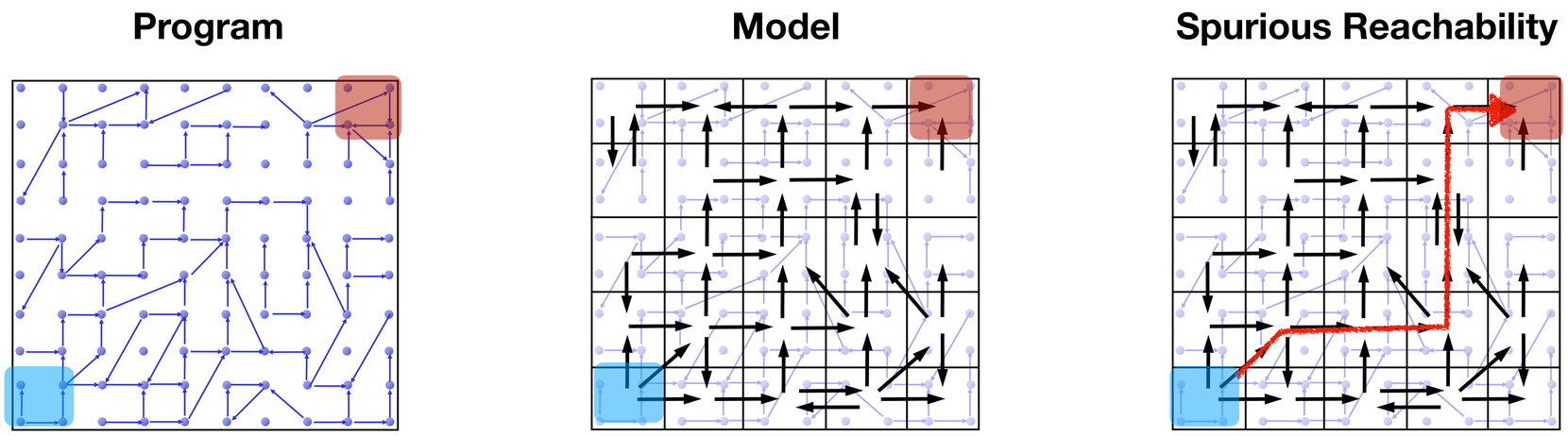
Target Program

```
  int main(){
1:  int x = input();
2:  x = 2 * x - 1;
3:  while (x > 0) {
4:    x = x - 2;
5:  }
6:  assert(x != 0);
7:  return 10 / x;
  }
```



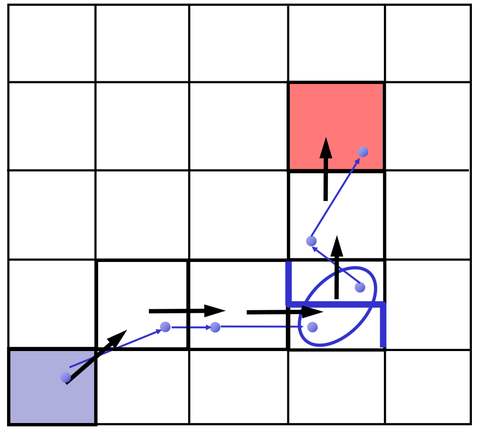
Spurious Reachability

- (Finite) Model is an abstraction of the (infinite) target program



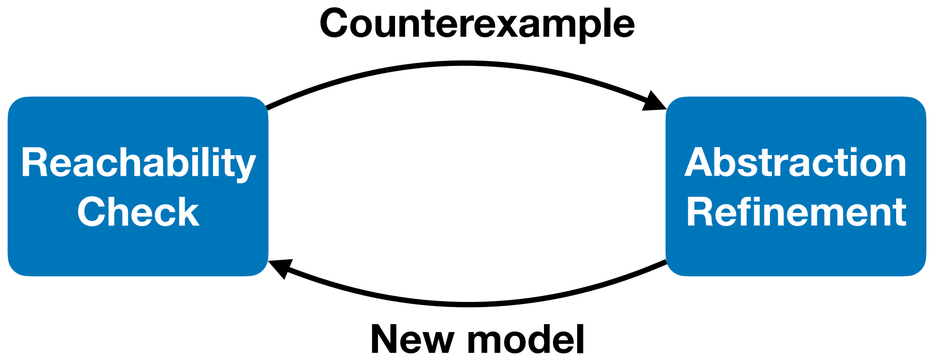
Abstraction Refinement

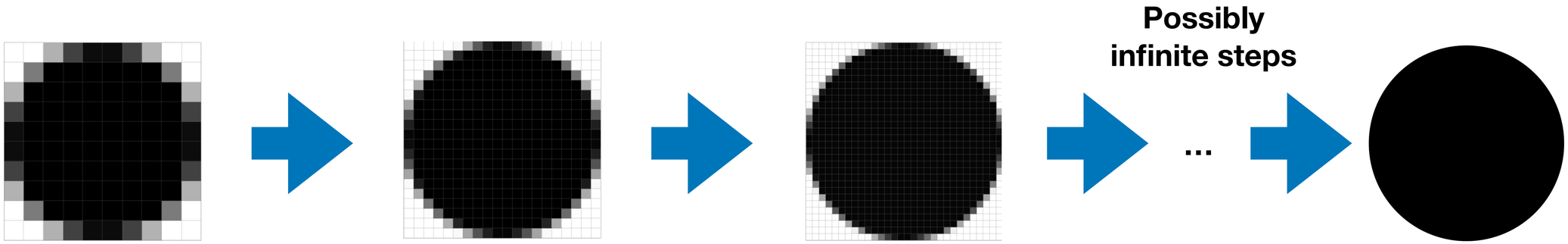
- Automatically refine the model when a spurious counterexample is found
  - New model: to conclude the spurious error is infeasible
  - Until a real counterexample is found or a proof is completed
- May not terminate



Iterative Abstraction Refinement

- CEGAR: CounterExample-Guided Abstraction Refinement





Summary of Model Checking

- Model (FSM) + Specification (Modal logic) + Verification (Reachability check)
- Theoretical characteristics:
  - If a model checker says “Yes”, the property is guaranteed to hold (**Sound**)
  - If a model checker says “No”,
    - the counterexample is either a real bug or a spurious warning
  - (refinement; verification)$^{+}$ until “Yes”, a real bug found, or timeout

- Further reading:
  Model Checking, E. M. Clarke, O. Grumberg, D. Kroening, D. Peled and H. Veith, 2018

Static Analysis

- **Over-approximate** (not exact) the set of all program behavior
- In general, **sound and automatic, but incomplete**
  - May have spurious results
- Based on a foundational theory : Abstract interpretation
- Variants:
  - under-approximating static analysis: automatic, complete, unsound
  - bug finder: automatic, unsound, incomplete, and heuristics
- Example: type systems, ASTREE, Facebook Infer, Sparrow, etc

Example

```
1:  static char *curfinal = “HDACB  FE”;
2:
3:  keysym = read_from_input();
4:
5:  if ((KeySym)(keysym) >= 0xFF9987)
6:  {
7:      unparseputc((char)(keysym – 0xFF91 + ‘P’), pty);
8:      key = 1;
9:  }
10: else if (keysym >= 0)
11: {
12:     if (keysym < 16)
13:     {
14:         if (read_from_input())
15:         {
16:             if (keysym >= 10) return;
17:             curfinal[keysym] = 1;
18:         }
19:         else
20:         {
21:             curfinal[keysym] = 2;
22:         }
23:     }
24:     if (keysym < 10)
25:         unparseput(curfinal[keysym], pty);
26: }
```

curfinal: buffer of size 10

keysym : any integer

keysym: [0, 15]

SAFE

keysym: [0, 9]

Buffer-overrun

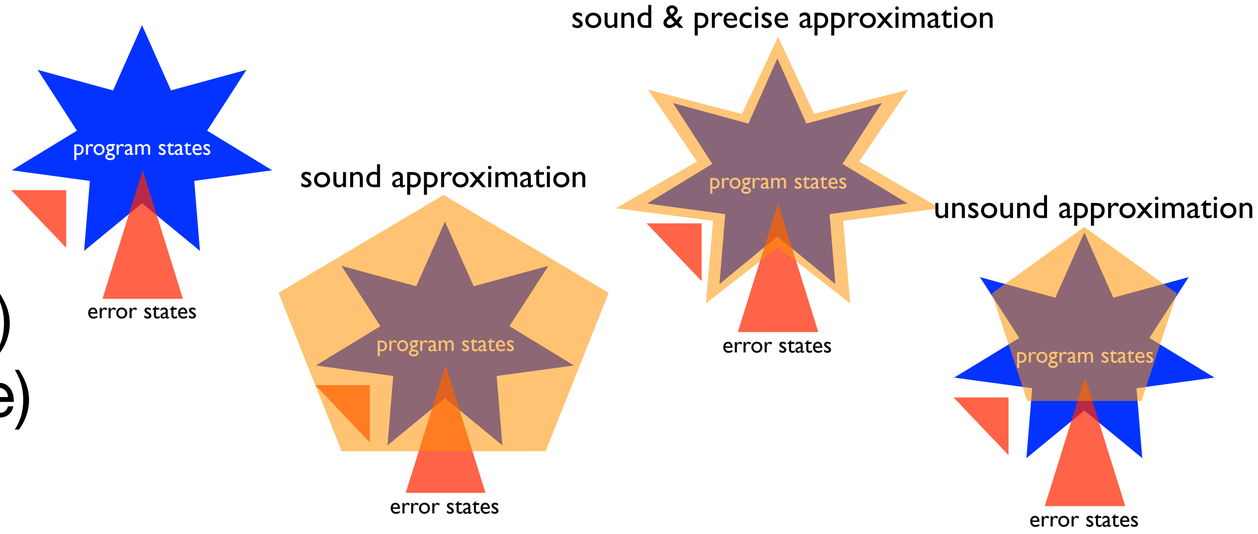
size of curfinal: [10, 10]
keysym: [0, 15]

keysym: [0, 9]

SAFE

Approximation

- Compute approximated (inaccurate) semantics instead of exact semantics
  - Inaccurate ≠ incorrect
  - E.g., reality: {2, 4, 6, 8, …}
    answer 1: “even” (exact)
    answer 2: “positive” (conservative)
    answer 3: “multiple of 4” (omissive)
    answer 4: “odd” (wrong)



- Given a program and property, the analysis may answer “Yes”, “No”, or “Don’t know” because of approximation

- Key point: choosing a right approximation to prove a given target property

Principle of Static Analysis

- How to design a sound approximation of real executions?
- How to guarantee the termination of static analysis?

A: Abstract Interpretation

Summary

- Different techniques for program reasoning due to the **computability barrier**
- Each program reasoning technique has its own pros and cons

	Automatic	Sound	Complete	Object	When
Testing	Yes	No	Yes	Program	Dynamic
Assisted Proving	No	Yes	Yes/No	Model	Static
Model Checking of finite-state model	Yes	Yes	Yes	Finite Model	Static
Model checking at program level	Yes	Yes	No	Program	Static
Conservative Static Analysis	Yes	Yes	No	Program	Static
Bug Finding	Yes	No	No	Program	Static