[OPLSS: Static Analysis]

Concepts in Static Analysis

Sukyoung Ryu

[Courtesy by Prof. Kihong Heo]

July 3, 2023

OPLSS: Static Analysis

(1) **Concepts in Static Analysis**

(2) **Operational / Denotational Semantics**

(3) **Abstract Interpretation**

(4) **Automatic Derivation of Static Analysis**

OPLSS: Static Analysis

- Reference
  - Xavier Rival and Kwangkeun Yi,

    Introduction to Static Analysis: an Abstract Interpretation Perspective,

    MIT Press, 2020

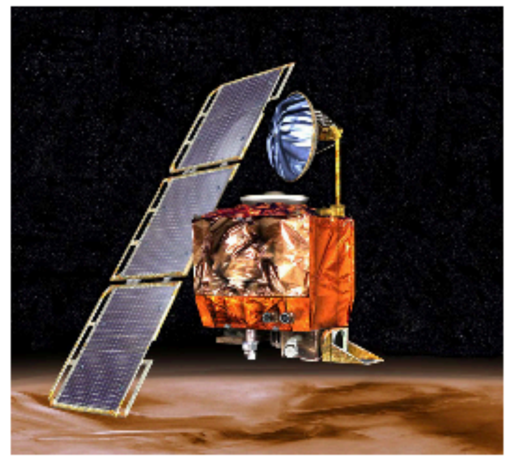
    https://mitpress.mit.edu/9780262043410/

Software Bugs: A Persistent Problem

- Back in the 90's

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

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



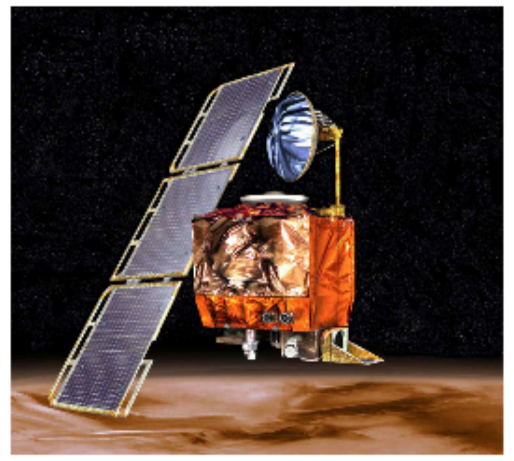
NASA's Mars Climate Orbiter (1999)
Meters-Inches Miscalculation
$125M

Software Bugs: A Persistent Problem

- Back in the 90's

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

- And now

Software Bugs: A Persistent Problem

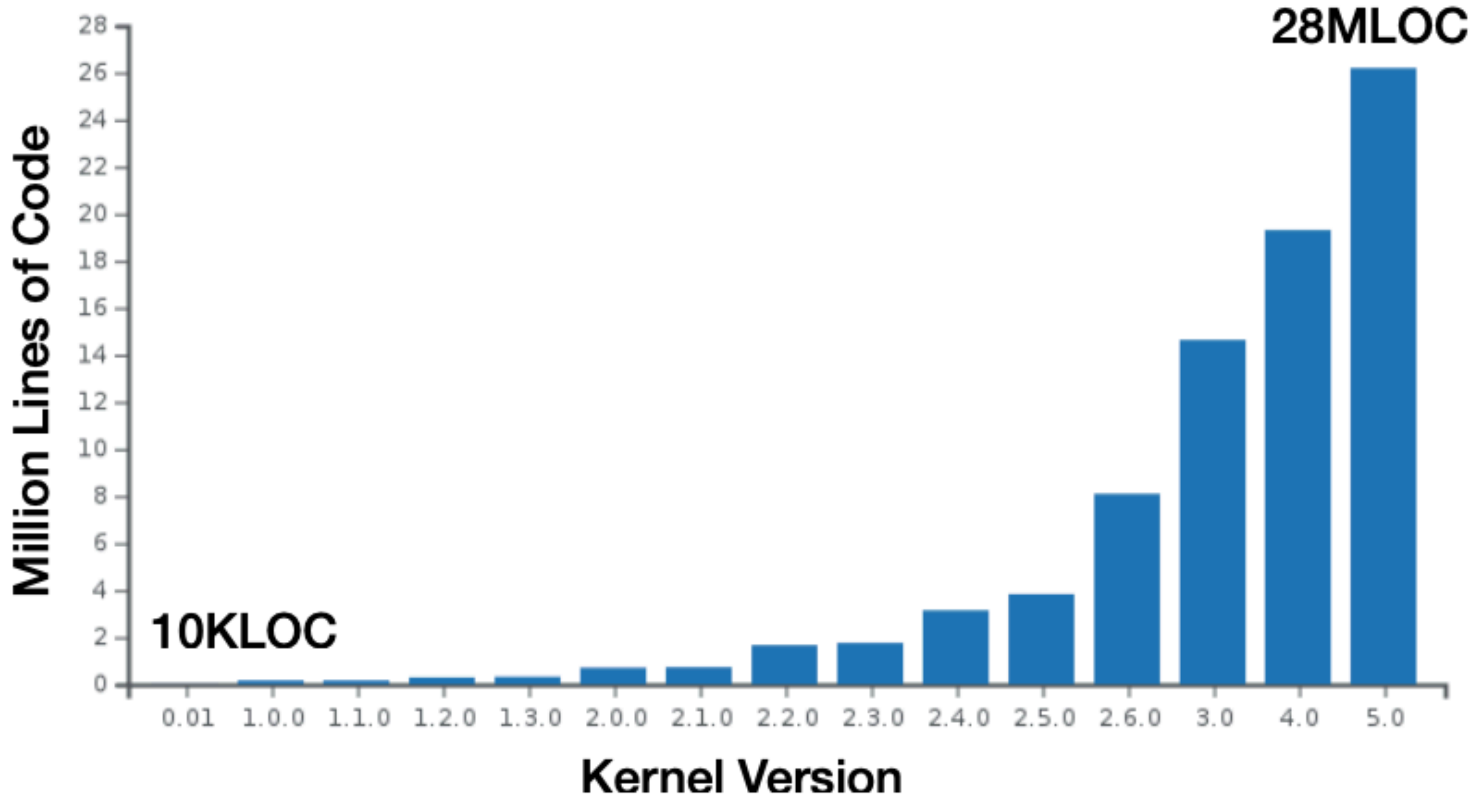
COST OF A SOFTWARE BUG

$100	$1,500	$10,000
If found in Gathering Requirements phase	If found in QA testing phase	If found in Production

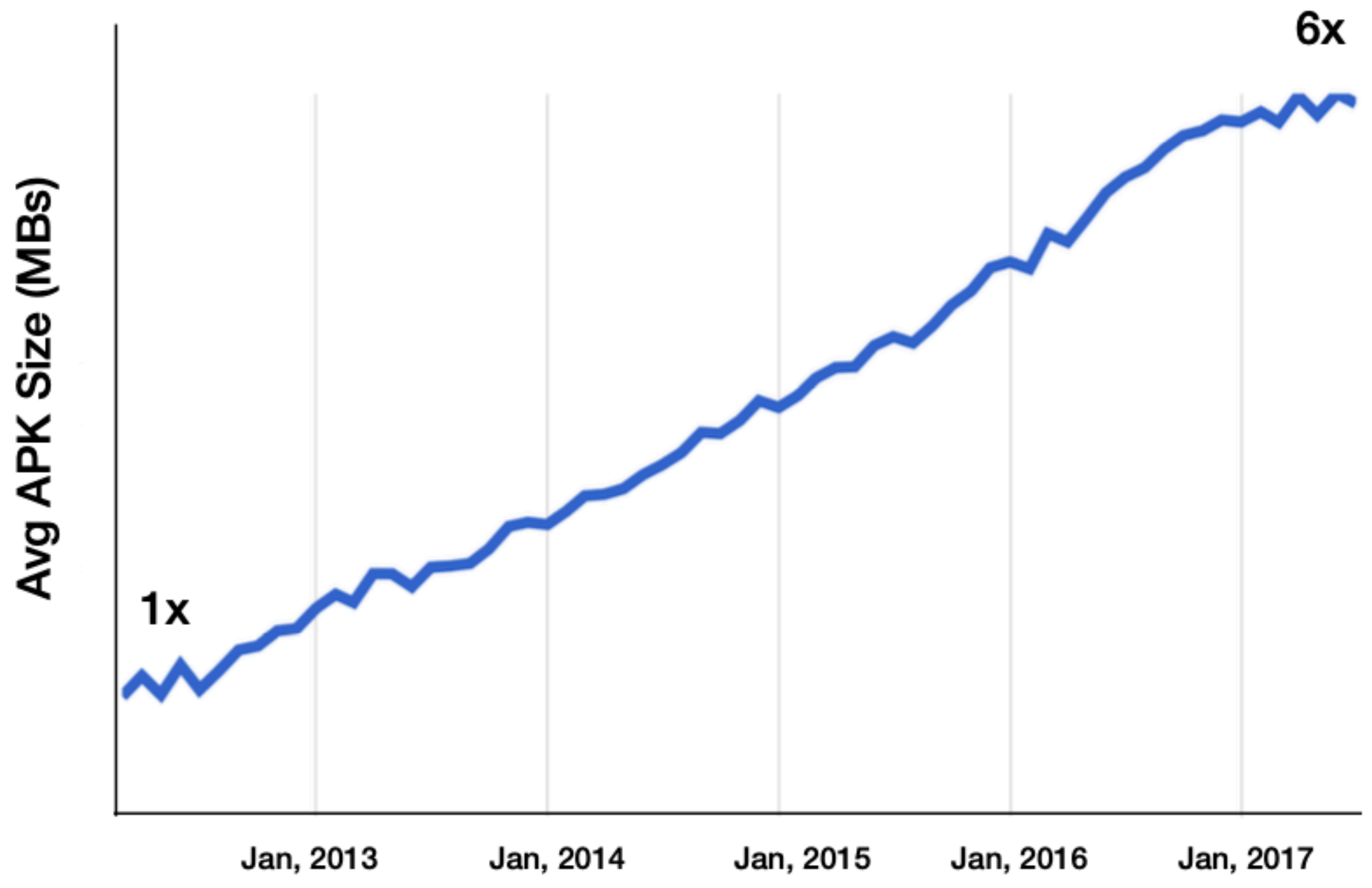
- IBM Systems Sciences Institute, 2015

Why Software Still Fails?

Size of Linux Kernel



Avg. Size of Android Apps



X

GitHub

10M+ New Developers
44M+ New Repositories
87M+ New Pull Requests
in 2019

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

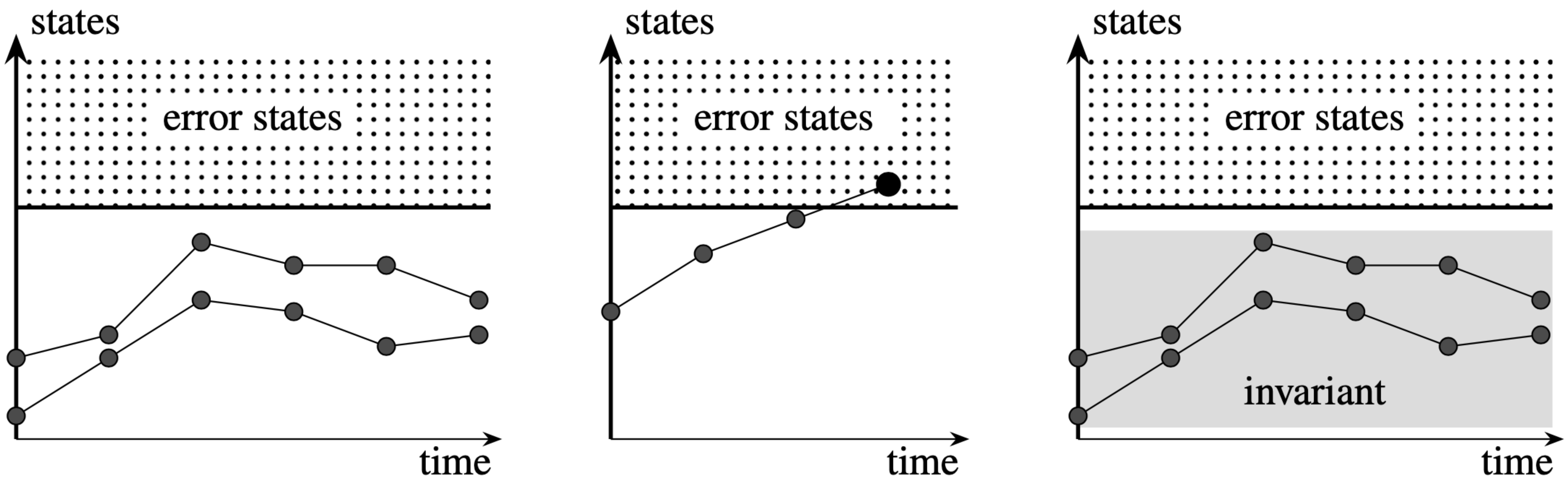
- Points of interest in programs
  - for verification, bug detection, optimization, understanding, etc
- In this lecture
  - safety properties
  - liveness properties
  - information-flow properties

Safety Property

- A program **never** exhibits 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

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
  - 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;
}
```

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
  - 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”

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
  - 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

```
int main(){
  int x = input();
  x = 2 * x - 1;
  while (x > 0) {
    x = x - 2;
  }
  assert(x != 0);
  return 10 / x;
}
```

Example: Division-by-Zero

```
int main(){
  int x = input();    // True
  x = 2 * x - 1;
  while (x > 0) {
    x = x - 2;
  }
  assert(x != 0);
  return 10 / x;
}
```

Example: Division-by-Zero

```
int main(){
  int x = input();      // True
  x = 2 * x - 1;        // x is an odd number
  while (x > 0) {
    x = x - 2;
  }
  assert(x != 0);
  return 10 / x;
}
```

Example: Division-by-Zero

```
int main(){
  int x = input();    // True
  x = 2 * x - 1;      // x is an odd number
  while (x > 0) {     // x is a positive odd number
    x = x - 2;
  }
  assert(x != 0);
  return 10 / x;
}
```

Example: Division-by-Zero

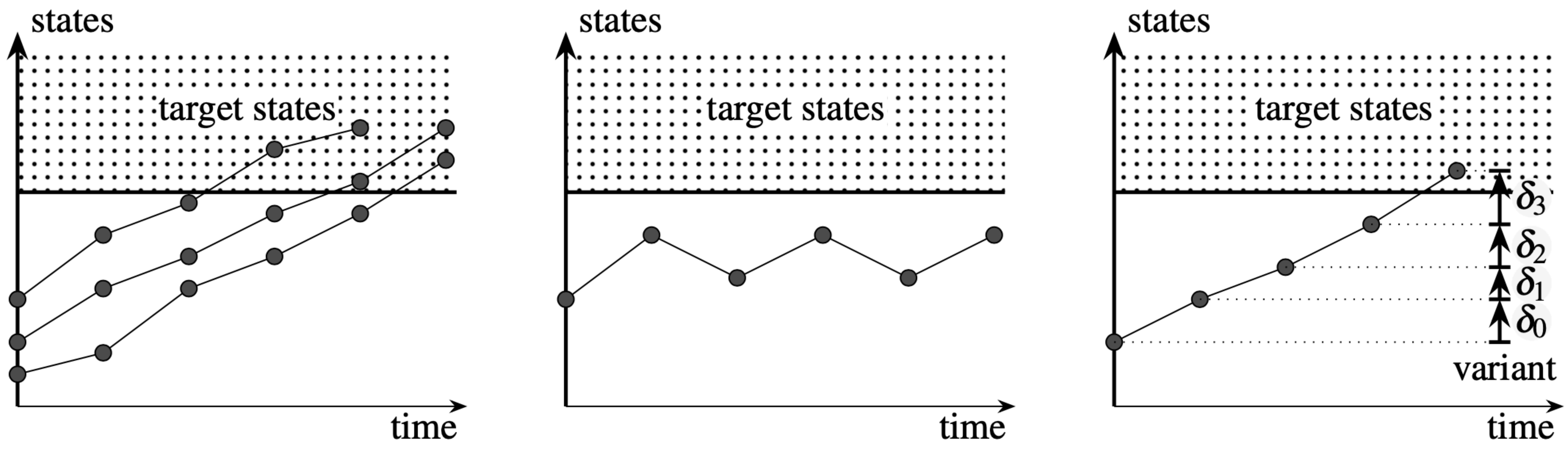
```
int main(){
  int x = input();          // True
  x = 2 * x - 1;            // x is an odd number
  while (x > 0) {           // x is a positive odd number
    x = x - 2;
  }                         // x is an odd number
  assert(x != 0);
  return 10 / x;
}
```

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

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;
}
```

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

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**

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**

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

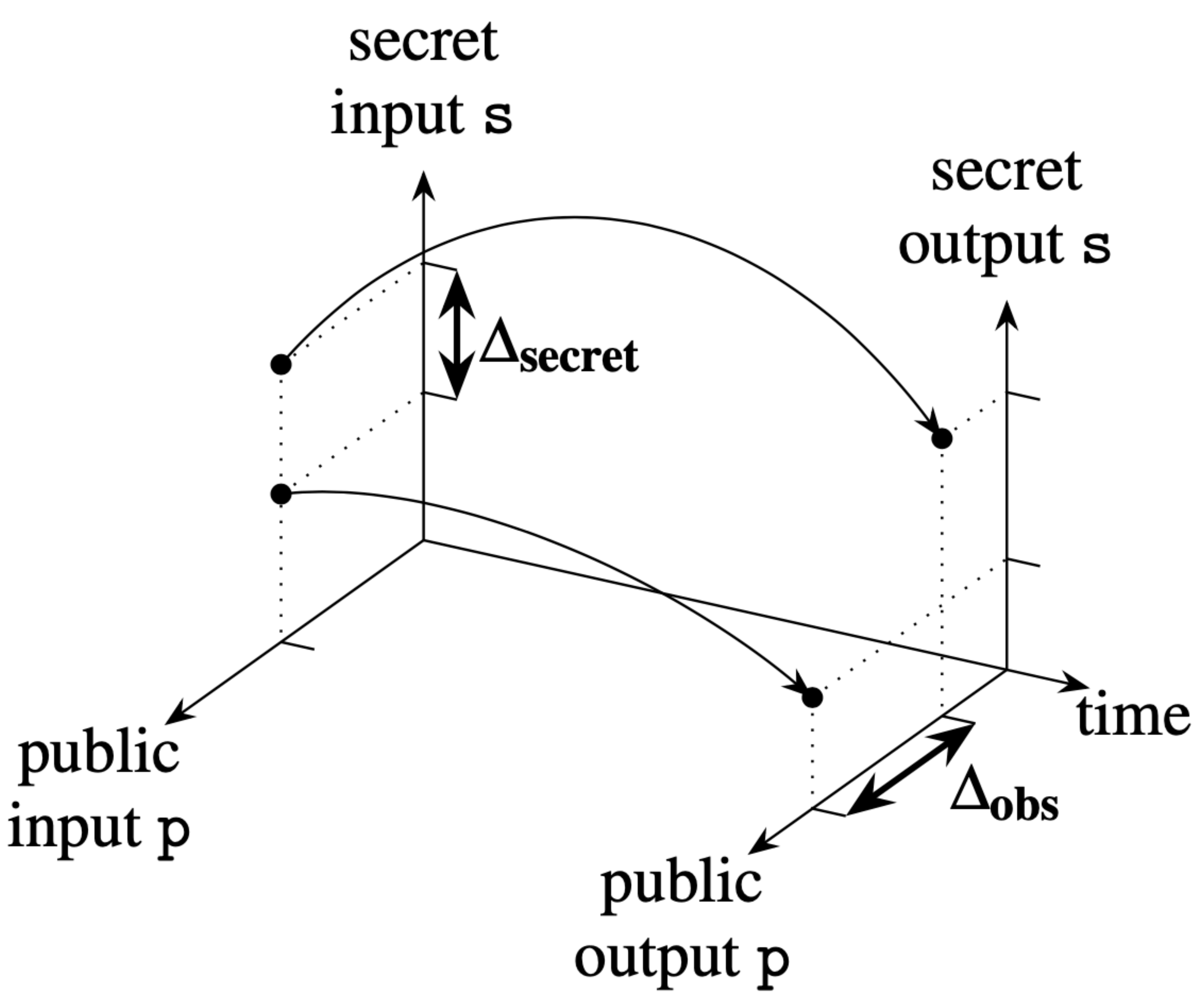
	Safety or Liveness?	State?
Should not fail with a run-time error	Safety	O
Should terminate	Liveness	-
Should return a sorted array	Safety	O
Should return an array with the same elements and multiplicity	Safety	X

Information Flow Properties

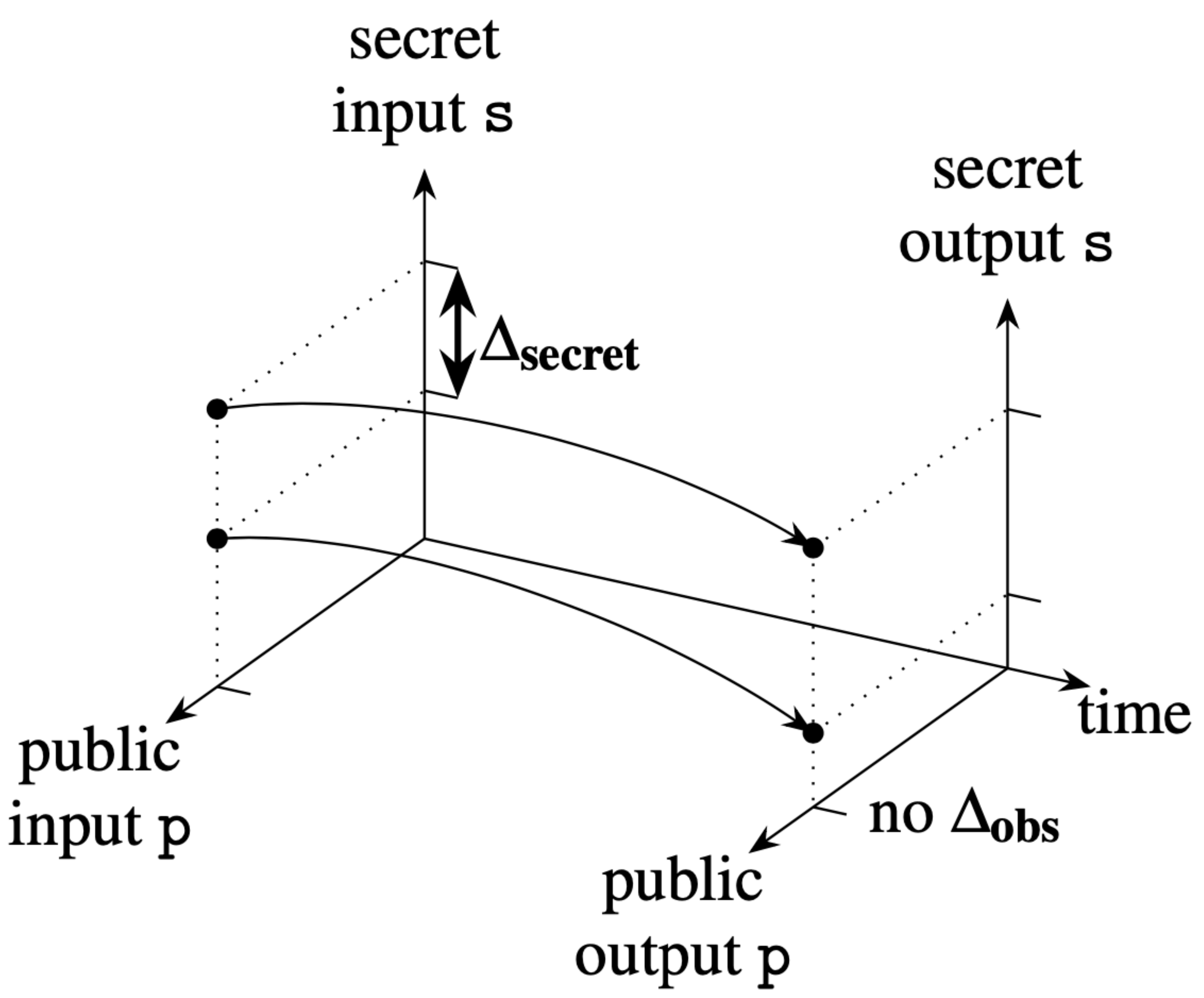
- Properties stating the absence of dependence between **pairs of executions**
  - Beyond trace properties: so called **hyperproperties**
- Mostly used for security purposes:
  - e.g.) multiple executions with public data should not derive private data

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
```

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

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

Example

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

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

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

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

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

Example

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

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

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

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

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

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

Example

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

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

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

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

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

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

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.

Undecidable

⇒ Automatic, terminating, and exact reasoning is impossible

Toward Computability

Undecidable

⇒ Automatic, terminating, and exact reasoning is impossible

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

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**: testing, model checking
  - 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,

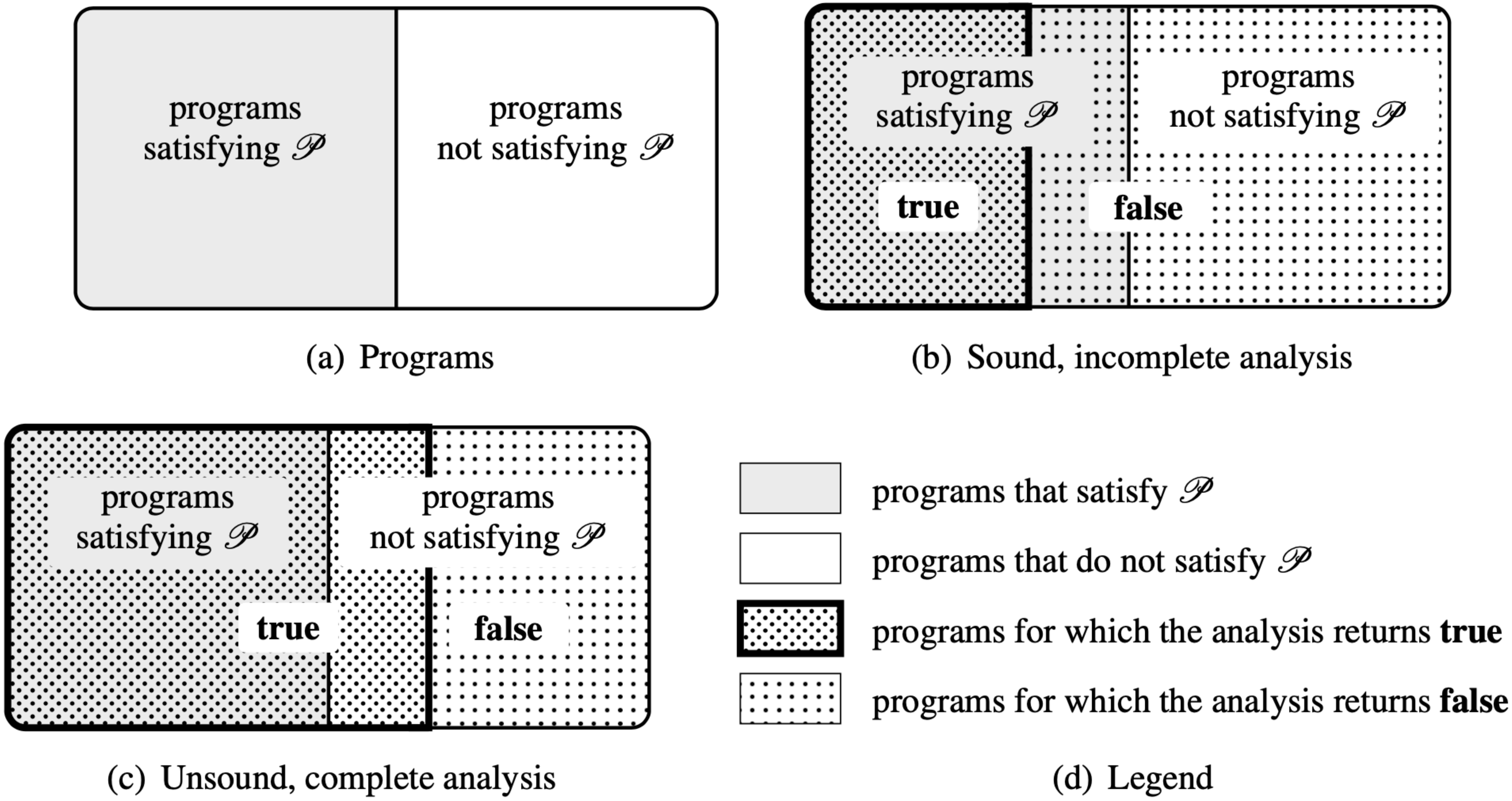
$$\text{For all program p, } \mathbf{A}(\text{p}) = \mathbf{true} \iff \text{p satisfies } \mathscr{P}$$

which consists of

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

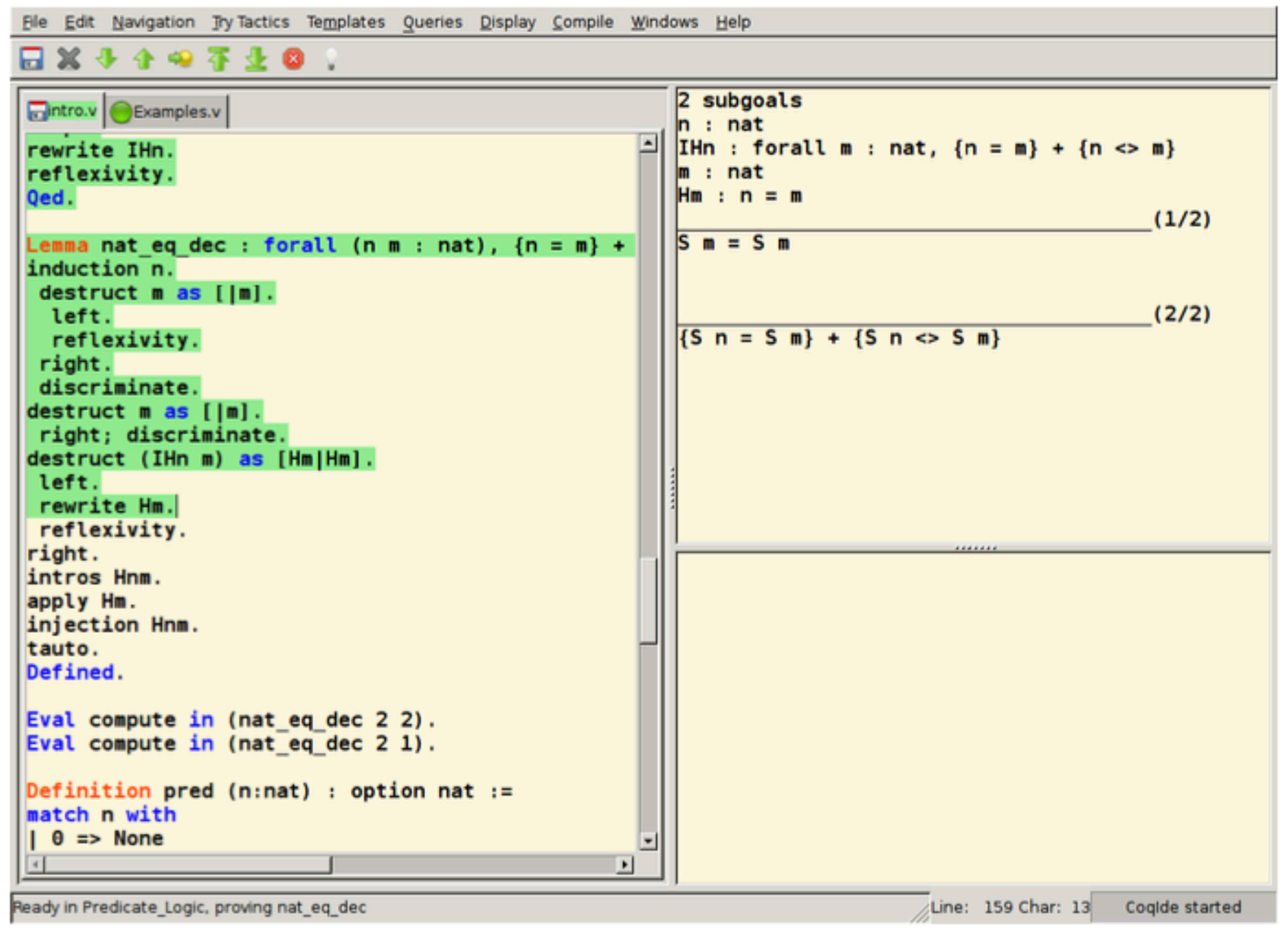
$$\text{For all program p, } \mathbf{A}(\text{p}) = \mathbf{true} \Leftarrow \text{p satisfies } \mathscr{P} \qquad \textbf{(completeness)}$$

Soundness and Completeness



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)

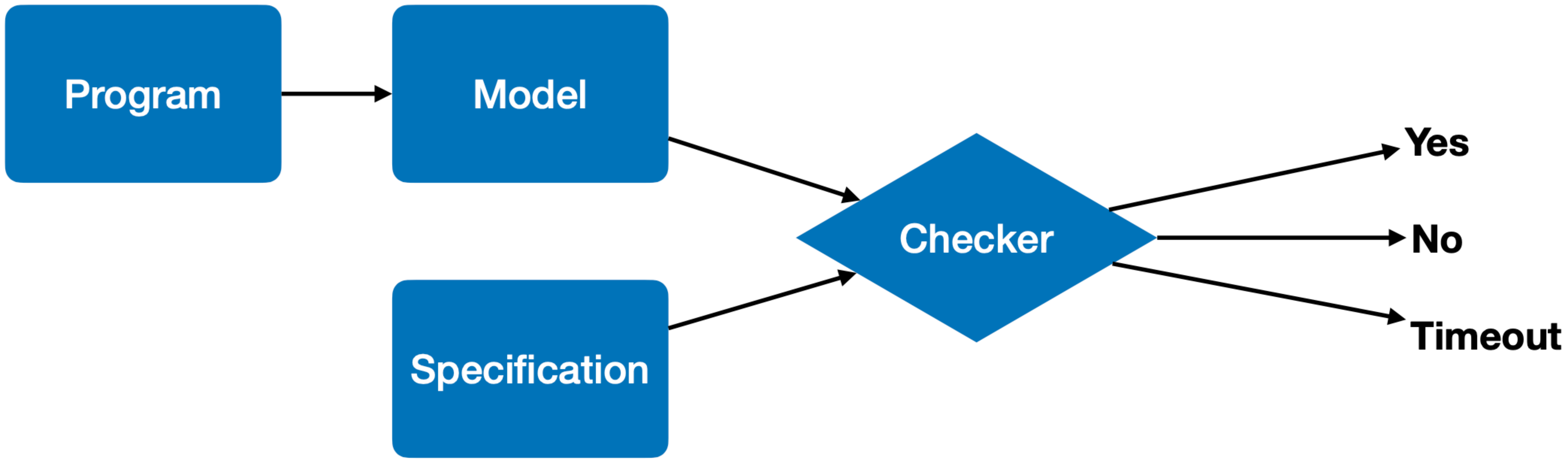
Testing

- 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)
- Further reading:
  Introduction to Software Testing, P. Ammann and J. Offutt, 2016

Model Checking

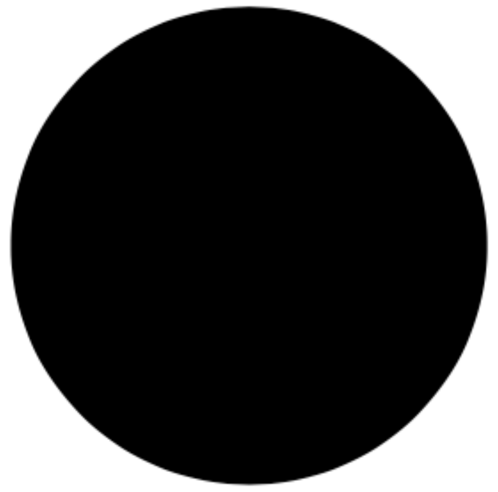
- Automatic technique to verify if a model satisfies a specification
  - Model of the target program (finite automata)
  - Specification written in logical formula
  - Verification via exhaustive search of the state space (graph reachability)
- **Sound and complete with respect to the model**
  - May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

Model Checking Overview

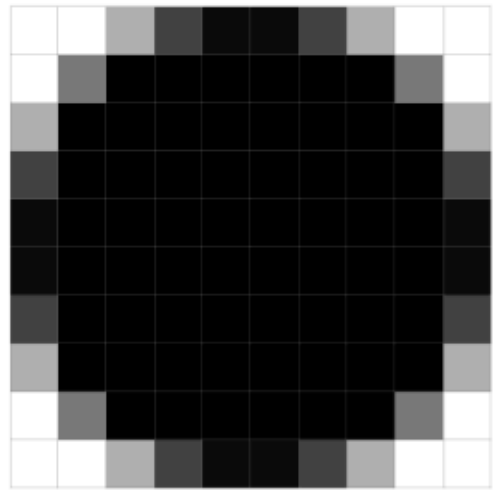


Model

- 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

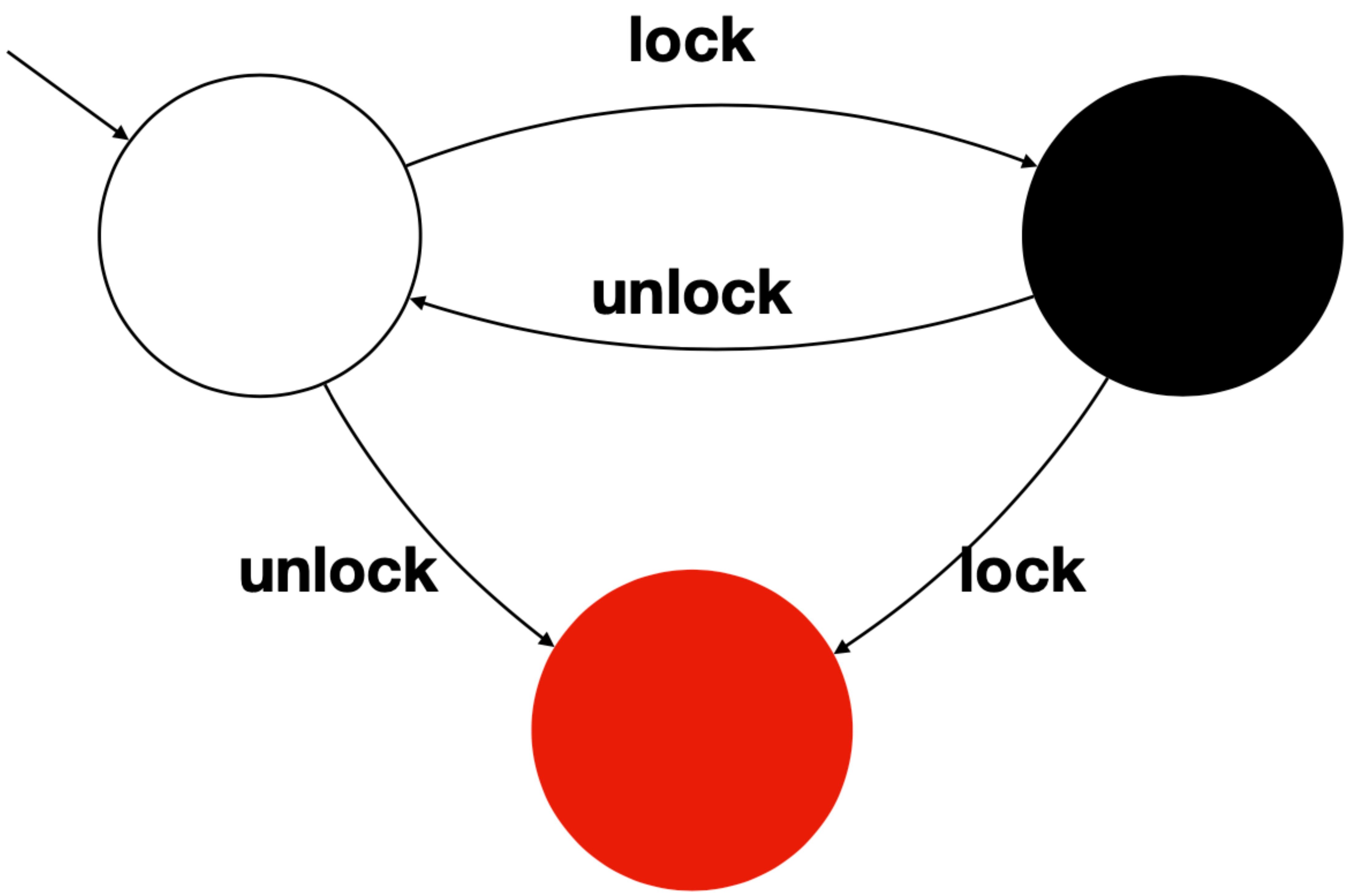


Program



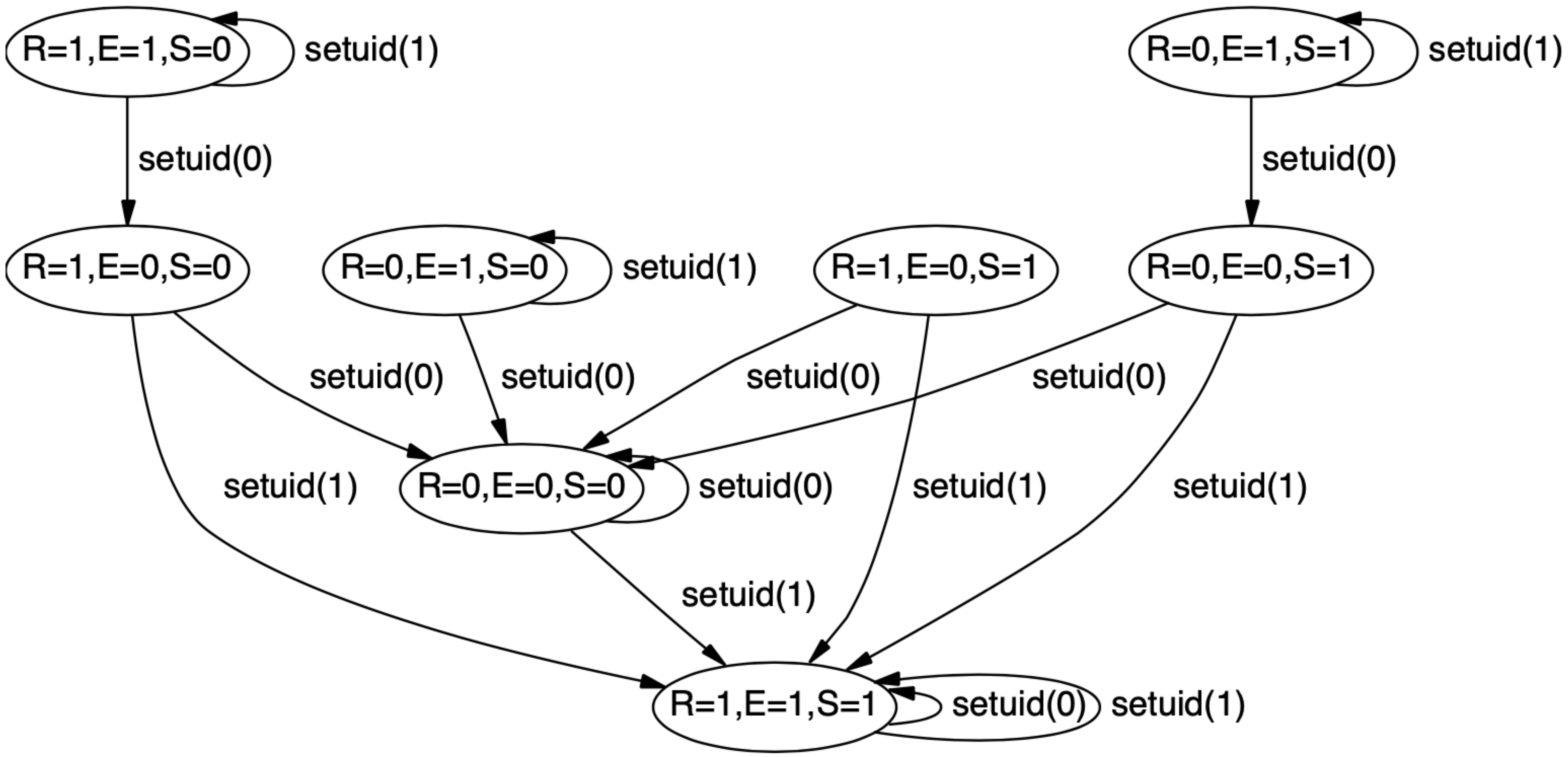
Model

Example: 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

Target Program

```
int main(){
  int x = input();
  x = 2 * x - 1;
  while (x > 0) {
    x = x - 2;
  }
  assert(x != 0);
  return 10 / x;
}
```

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;
  }
```

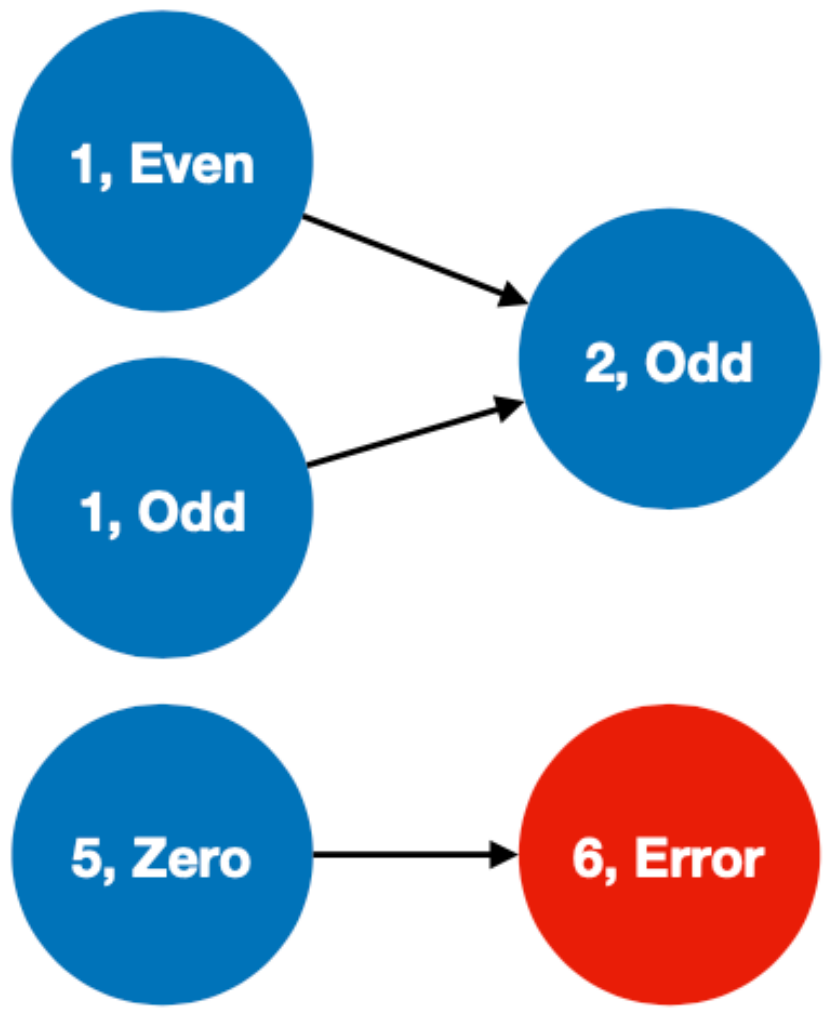
Example: Model & Specification

- 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

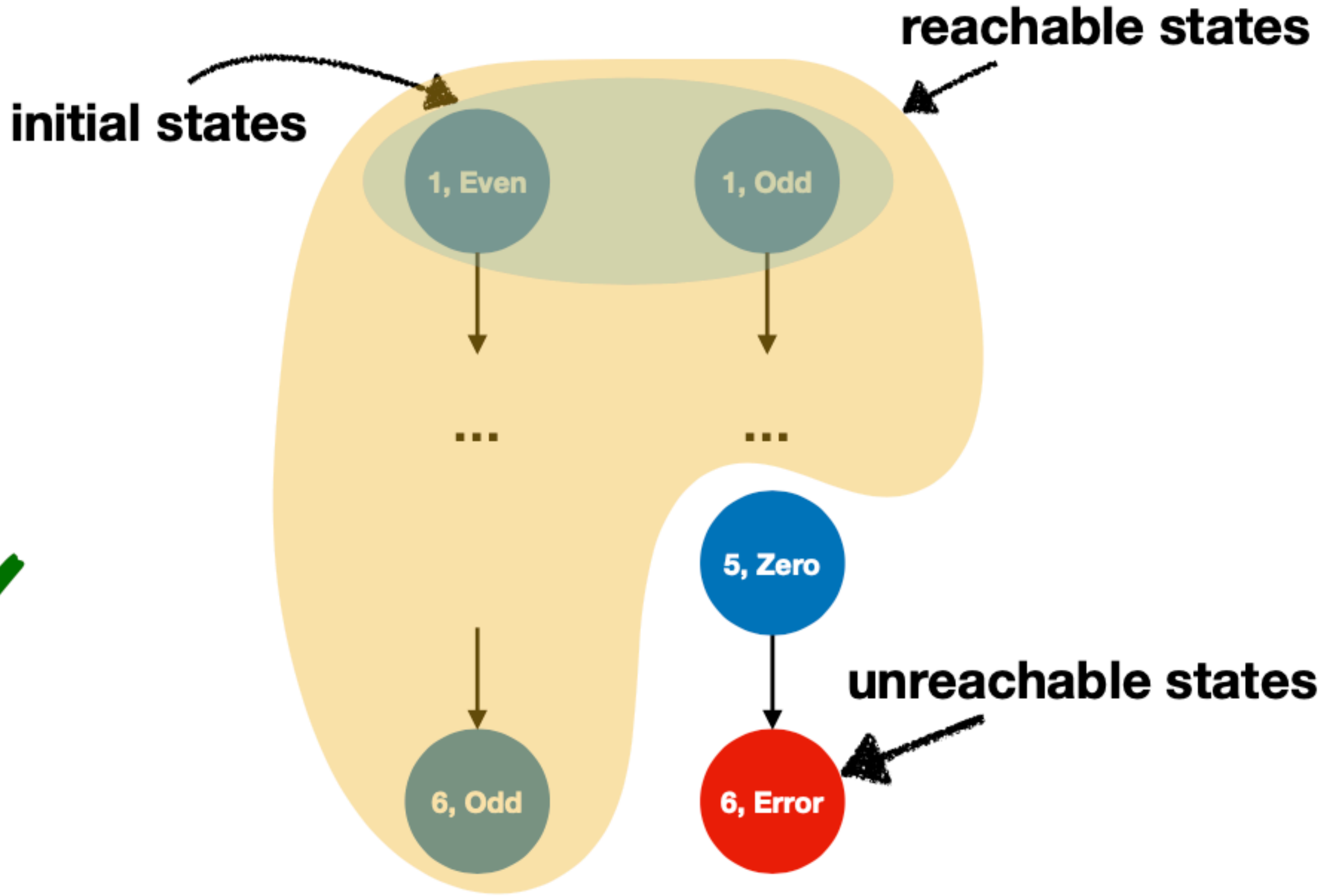


Example: Reachability Check

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

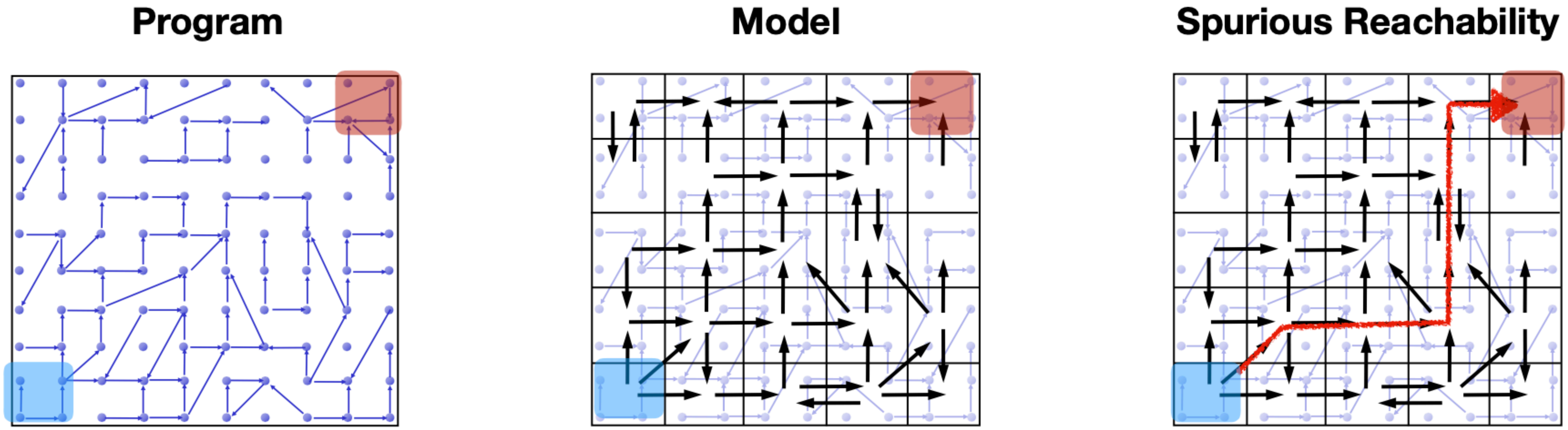
Target Program

```
  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;
  }
```



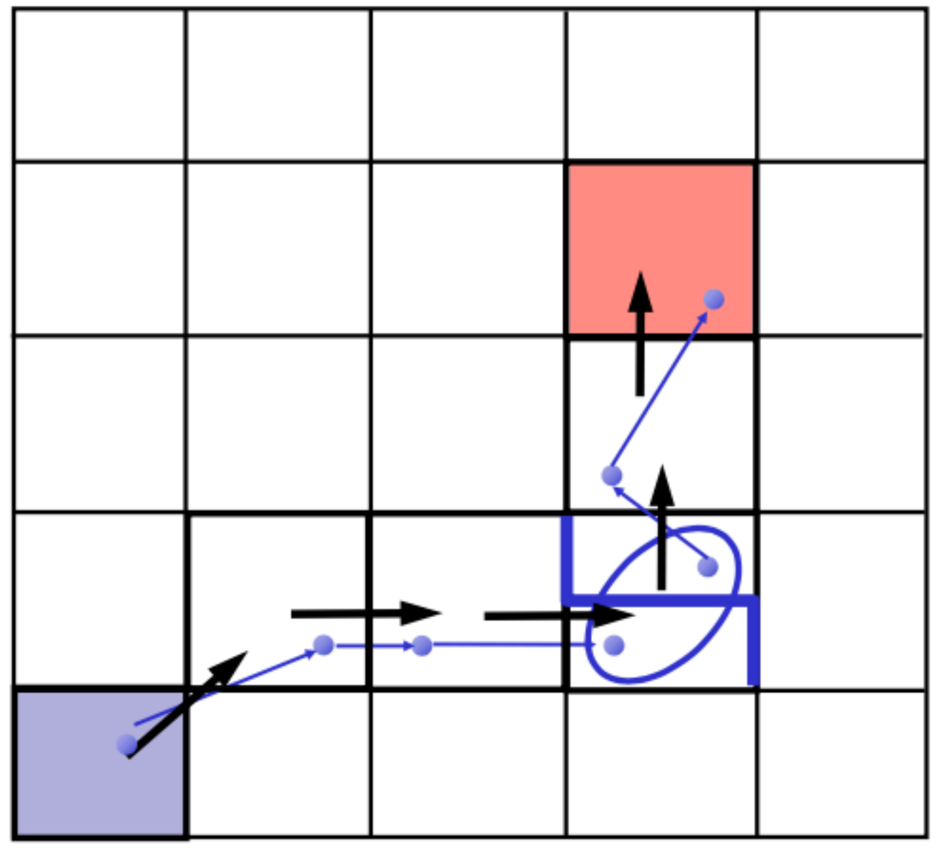
Spurious Reachability

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



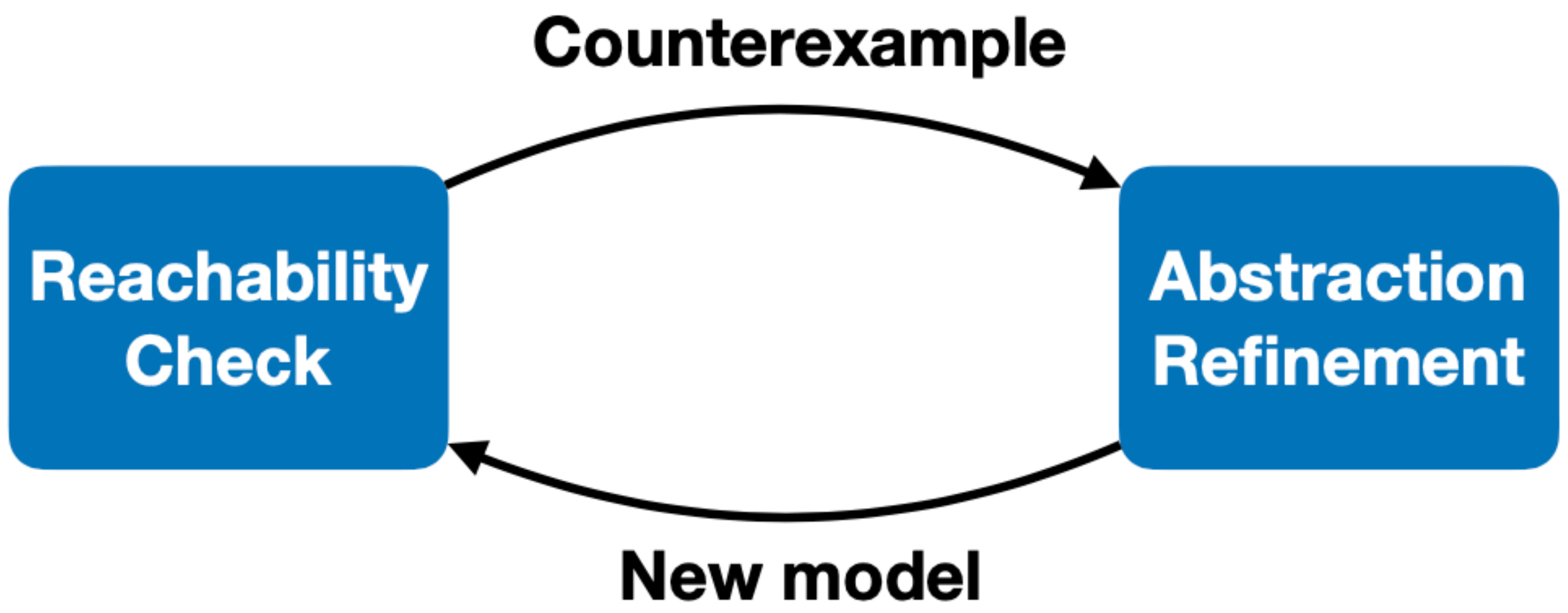
Abstraction Refinement

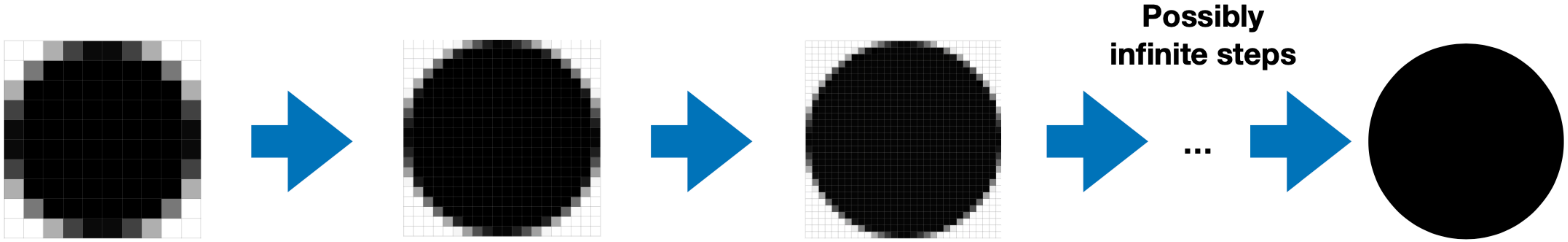
- Automatically refine the model when a spurious counterexample is found
  - New model: to conclude the spurious error is infeasible
  - Until a real counterexample is found or a proof is completed
- May not terminate



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

Industrial Applications of Astrée

The main applications of Astrée appeared two years after starting the project. Since then, Astrée has achieved the following unprecedented results on the static analysis of synchronous, time-triggered, real-time, safety critical, embedded software written or automatically generated in the C programming language:

- In Nov. 2003, Astrée was able to prove completely automatically the absence of any RTE in the primary flight control software of the Airbus A340 fly-by-wire system, a program of 132,000 lines of C analyzed in $1^{h}20$ on a 2.8 GHz 32-bit PC using 300 Mb of memory (and 50mn on a 64-bit AMD Athlon™ 64 using 580 Mb of memory).

- From Jan. 2004 on, Astrée was extended to analyze the electric flight control codes then in development and test for the A380 series. The operational application by Airbus France at the end of 2004 was just in time before the A380 maiden flight on Wednesday, 27 April, 2005.

- In April 2008, Astrée was able to prove completely automatically the absence of any RTE in a C version of the automatic docking software of the Jules Vernes Automated Transfer Vehicle (ATV) enabling ESA to transport payloads to the International Space Station [32].

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

OPLSS: Static Analysis

(1) **Concepts in Static Analysis**

(2) **Operational / Denotational Semantics**

(3) **Abstract Interpretation**

(4) **Automatic Derivation of Static Analysis**

[OPLSS: Static Analysis]

Static Analysis for Android Multilingual Applications

Sukyoung Ryu

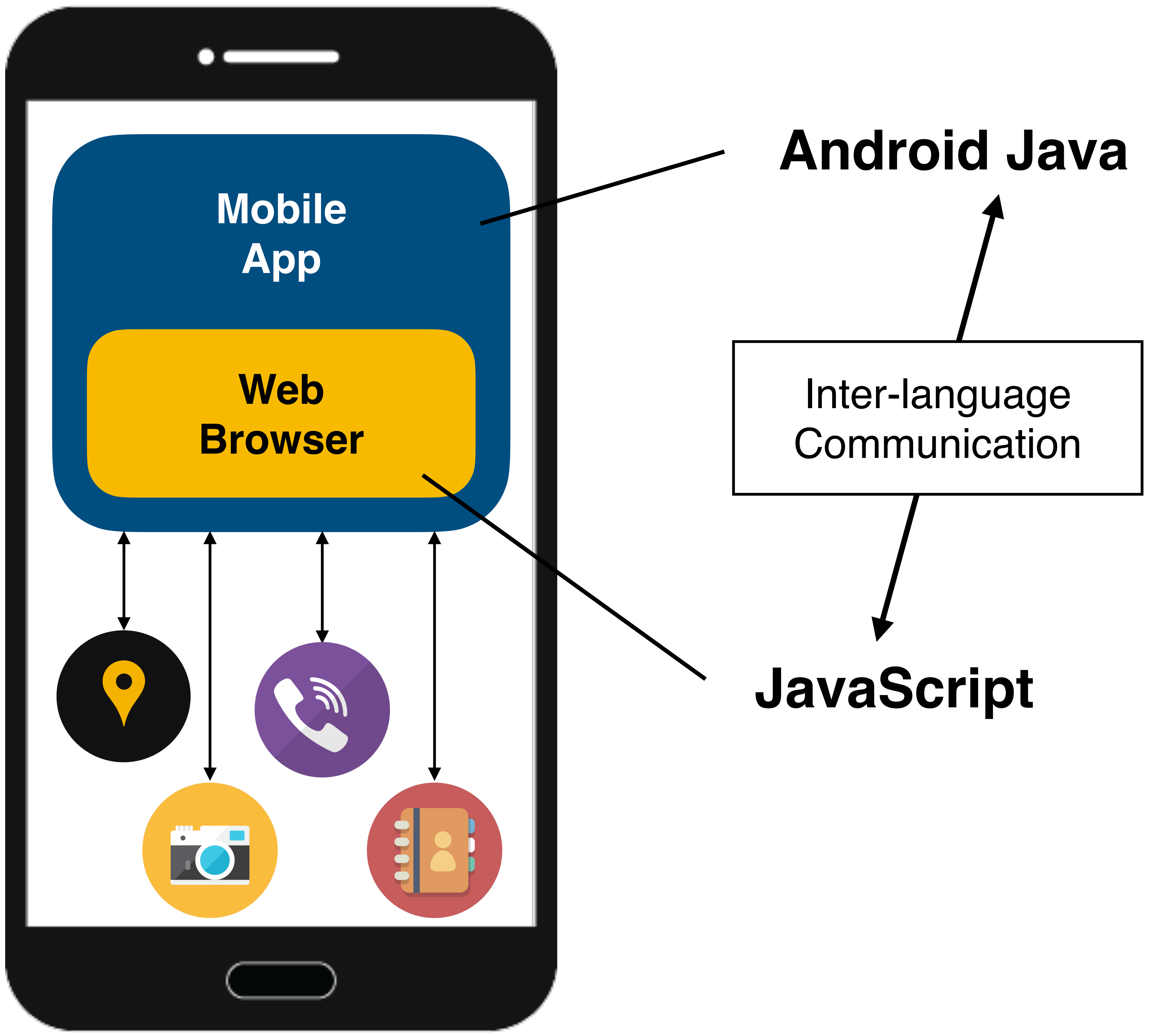
with PLRG@KAIST and friends

July 3, 2023

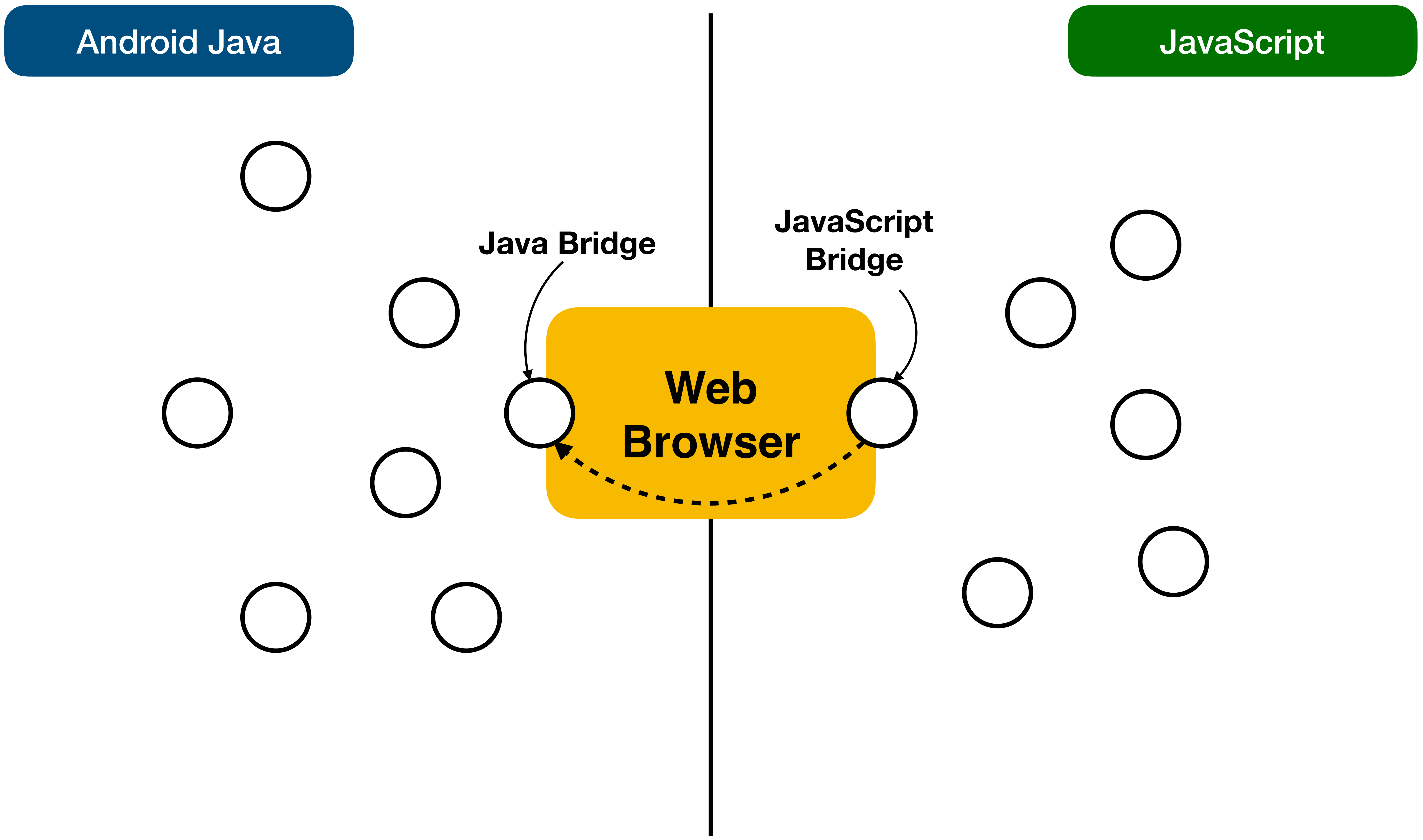
Static Analysis for Android Multilingual Applications

- **“Bittersweet ADB: Attacks and Defenses”**
  ACM Symposium on Information, Computer and Communications Security, 2015 with Sungjae Hwang, Sungho Lee, and Yongdae Kim
- **“All about Activity Injection: Threats, Semantics, and Detection”**
  IEEE/ACM International Conference on Automated Software Engineering, 2017 with Sungho Lee and Sungjae Hwang
- **“HybriDroid: Static Analysis Framework for Android Hybrid Applications”**
  IEEE/ACM International Conference on Automated Software Engineering, 2016 with Sungho Lee and Julian Dolby
- **“Towards Understanding and Reasoning about Android Interoperations”**
  ACM/IEEE International Conference on Software Engineering, 2019 with Sora Bae and *Sungho Lee*
- **“Adlib: Analyzer for Mobile Ad Platform Libraries”**
  ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019 with Sungho Lee

HybriDroid: Android Hybrid Apps



HybriDroid: Inter-language Communication



HybriDroid: Inter-language Communication

Android Java

```
Class JSApp {
  @JavascriptInterface
  public int alert(String m) {
      …
  }
  …
}

…

addJavascriptInterface(
           new JSApp(), “app”);
```

Java Bridge

JavaScript

```
app.alert(“hello hybrid”);
```

JavaScript Bridge

HybriDroid: Inter-language Communication

Android Java

```
Class JSApp {
  @JavascriptInterface
  public int alert(String m) {
      …
  }
  …
}

…

addJavascriptInterface(
           new JSApp(), “app”);
```

Java Bridge

JavaScript

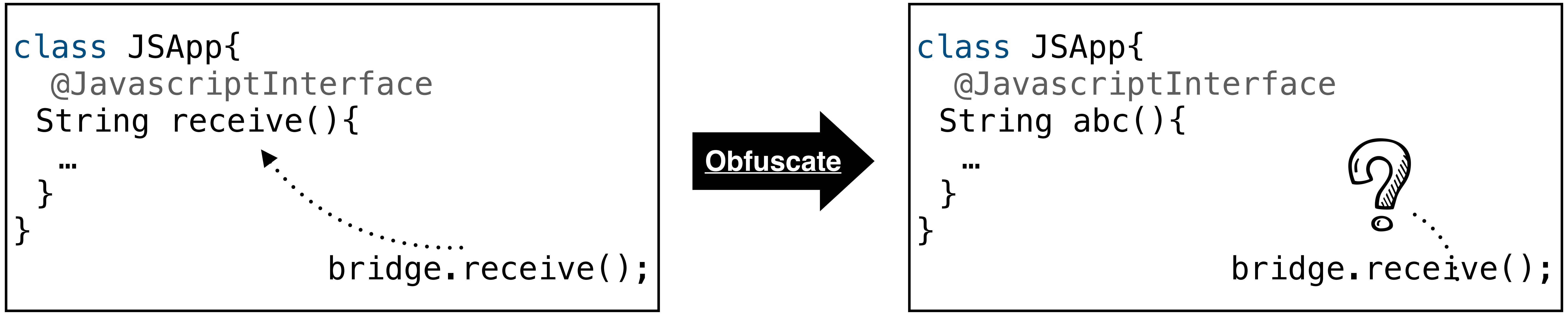
MethodNotFound exception

```
app.alert(“hello hybrid”, 3);
```

JavaScript Bridge

HybriDroid: Bug Detection

Rank	Hybrid App	Bug Type (#)	#FP	#TP	Bug Cause (#)	Time
1 – 100	com.gameloft.android.ANMP.GloftDMHM	MethodNotFound (1)	0	1	Obfuscation (1)	2404 sec.
	com.creativemobile.DragRacing	MethodNotFound (1)	1	0		3192 sec.
	com.gau.go.launcherex	MethodNotFound (2)	2	0		5432 sec.
	com.tripadvisor.tripadvisor	MethodNotFound (1)	0	1	Obfuscation (1)	4028 sec.
	com.dianxinos.dxbs	MethodNotFound (1)	0	1	Obfuscation (1)	1924 sec.
10,000 – 10,100	com.magmamobile.game.LostWords	MethodNotFound (1)	1	0		475 sec.
20,000 – 20,100	com.daishin	MethodNotFound (1)	0	1	Undeclared Method (1)	6572 sec.
100,000 – 100,100	com.carezone.caredroid.careapp	MethodNotFound (5)	0	5	Missing Annotation (5)	2357 sec.
	com.pateam.kanomthai	MethodNotFound (2)	0	2	Missing Annotation (2)	4209 sec.
	com.acc5.l6	MethodNotFound (6)	0	6	Missing Annotation (6)	367 sec.
	jp.cleanup.android	MethodNotFound (1)	1	0		253 sec.
	ligamexicana.futbol	MethodNotFound (2)	2	0		253 sec.
200,000 – 200,100	com.sysapk.weighter	MethodNotFound (1)	0	1	Missing Annotation (1)	106 sec.
	com.youmustescape3guide.free	MethodNotFound (6)	0	6	Missing Annotation (6)	445 sec.
Total		MethodNotFound (31)	7	24	Missing Annotation (20) Obfuscation (3) Undeclared Method (1)	2287 sec.



Static Analysis for Android Multilingual Applications

- **"Bittersweet ADB: Attacks and Defenses"**
  ACM Symposium on Information, Computer and Communications Security, 2015 with Sungjae Hwang, Sungho Lee, and Yongdae Kim
- **"All about Activity Injection: Threats, Semantics, and Detection"**
  IEEE/ACM International Conference on Automated Software Engineering, 2017 with Sungho Lee and Sungjae Hwang
- **"HybriDroid: Static Analysis Framework for Android Hybrid Applications"**
  IEEE/ACM International Conference on Automated Software Engineering, 2016 with Sungho Lee and Julian Dolby
- **"Towards Understanding and Reasoning about Android Interoperations"**
  ACM/IEEE International Conference on Software Engineering, 2019 with Sora Bae and Sungho Lee
- **"Adlib: Analyzer for Mobile Ad Platform Libraries"**
  ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019 with Sungho Lee

Inter-language Operation: Types and Values

Android Java

```
Class JSApp {
  @JavascriptInterface
  public int alert(String m) {
      …
  }
  …
}

…

addJavascriptInterface(
           new JSApp(), “app”);
```

Java Bridge

JavaScript

```
app.alert(“hello hybrid”);
```

JavaScript Bridge

Inter-language Operation: Types and Values

Android Java

Chapter 4. Types, Values, and Variables

The Java programming language is a *statically typed* language, which means that every variable and every expression has a type that is known at compile time.

The Java programming language is also a *strongly typed* language, because types limit the values that a variable (§4.12) can hold or that an expression can produce,

The types of the Java programming language are divided into two categories: primitive types and reference types. The primitive types (§4.2) are the `boolean` type an
special null type. An object (§4.3.1) is a dynamically created instance of a class type or a dynamically created array. The values of a reference type are references to c

≠

JavaScript

6 ECMAScript Data Types and Values

Algorithms within this specification manipulate values each of which has an associated type. The possible value types are exactly those defined in this clause. Types are further subclassified into ECM

Within this specification, the notation "Type(*x*)" is used as shorthand for "the *type* of *x*" where "type" refers to the ECMAScript language and specification types defined in this clause. When the term
equivalent to saying "no value of any type".

Inter-language Operation: Overloading

Android Java

8.4.9. Overloading

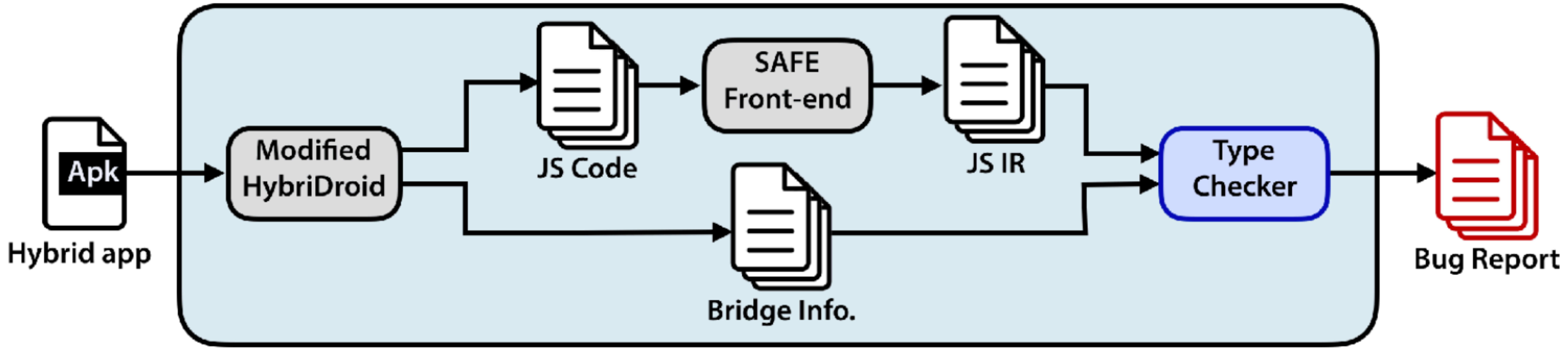
If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name

This fact causes no difficulty and never of itself results in a compile-time error. There is no required relationship between the return types or between the

When a method is invoked (§15.12), the number of actual arguments (and any explicit type arguments) and the compile-time types of the arguments are

JavaScript

Formalization of Android Interoperation



- Identify the under-documented Android interoperation behaviors
  - Discovered previously-unknown, unintuitive, and surprising behaviors
- Present the first formal semantics of Android interoperation
- Develop a light-weight type system detecting interoperation bugs
  - More true bugs more efficiently than HybriDroid

OPLSS: Static Analysis

(1) **Concepts in Static Analysis**

(2) **Operational / Denotational Semantics**

(3) **Abstract Interpretation**

(4) **Automatic Derivation of Static Analysis**