Announcement:
The lesson scheduled for Friday, the 24th, will not be held

Course Focus and Next Steps

In this course, we have explored how to model systems and then verify them using model checking techniques.

This represents the main line of the course.

Now, we will take a side path and examine some program analysis techniques.

We will focus on dynamic analysis techniques.

What is program analysis?

Testing and Analysis

What you know from previous courses

- Manual testing or semi-automated testing
- JUnit, Pytest, Selenium, etc.

Manual ”analysis” of programs:

- Code inspection, debugging, etc.

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

class GeneralTestSuite {

    private ExampleService service;

    @BeforeAll
    static void initAll() {
        System.out.println("Starting test suite...");
    }

    @BeforeEach
    void init() {
        service = new ExampleService();
    }

    @Test
    void testAddition() {
        assertEquals(4, service.add(2, 2), "Addition should return correct result");
    }
```

```
    @Test
    void testDivisionByZeroThrows() {
        assertThrows(ArithmeticException.class, () -> service.divide(10, 0));
    }

    @Test
    void testStringNotNull() {
        String result = service.getMessage();
        assertNotNull(result, "Message should not be null");
        assertTrue(result.startsWith("Hello"), "Message should start with 'Hello'");
    }

    @AfterEach
    void tearDown() {
        service = null;
    }

    @AfterAll
    static void tearDownAll() {
        System.out.println("All tests completed.");
    }
}
```

```
// Dummy implementation under test
class ExampleService {
    int add(int a, int b) { return a + b; }
    int divide(int a, int b) { return a / b; }
    String getMessage() { return "Hello, World!"; }
}
```

The Key Concepts

@BeforeAll / @AfterAll for global setup and cleanup

@BeforeEach / @AfterEach for per-test setup and teardown

Assertions: assertEquals, assertNotNull, assertThrows ...

Good naming and descriptive messages


Code inspection

Rust Code

```
use std::env;

fn read_port() -> u16 {
    let port_str = env::var("APP_PORT").unwrap();
    let port: i32 = port_str.parse().unwrap();
    port as u16
}
```

The function read_port() reads the environment variable APP_PORT, parses it as a number, and returns it as a port.

Rust Code

```
use std::env;

fn read_port() -> u16 {
    let port_str = env::var("APP_PORT").unwrap();
    let port: i32 = port_str.parse().unwrap();
    port as u16
}
```

The function read_port() reads the environment variable APP_PORT, parses it as a number, and returns it as a port.

unwrap() is a method used on **Option<T>** and **Result<T, E>** types.
It retrieves the inner value but will **panic** if the value is None or Err.

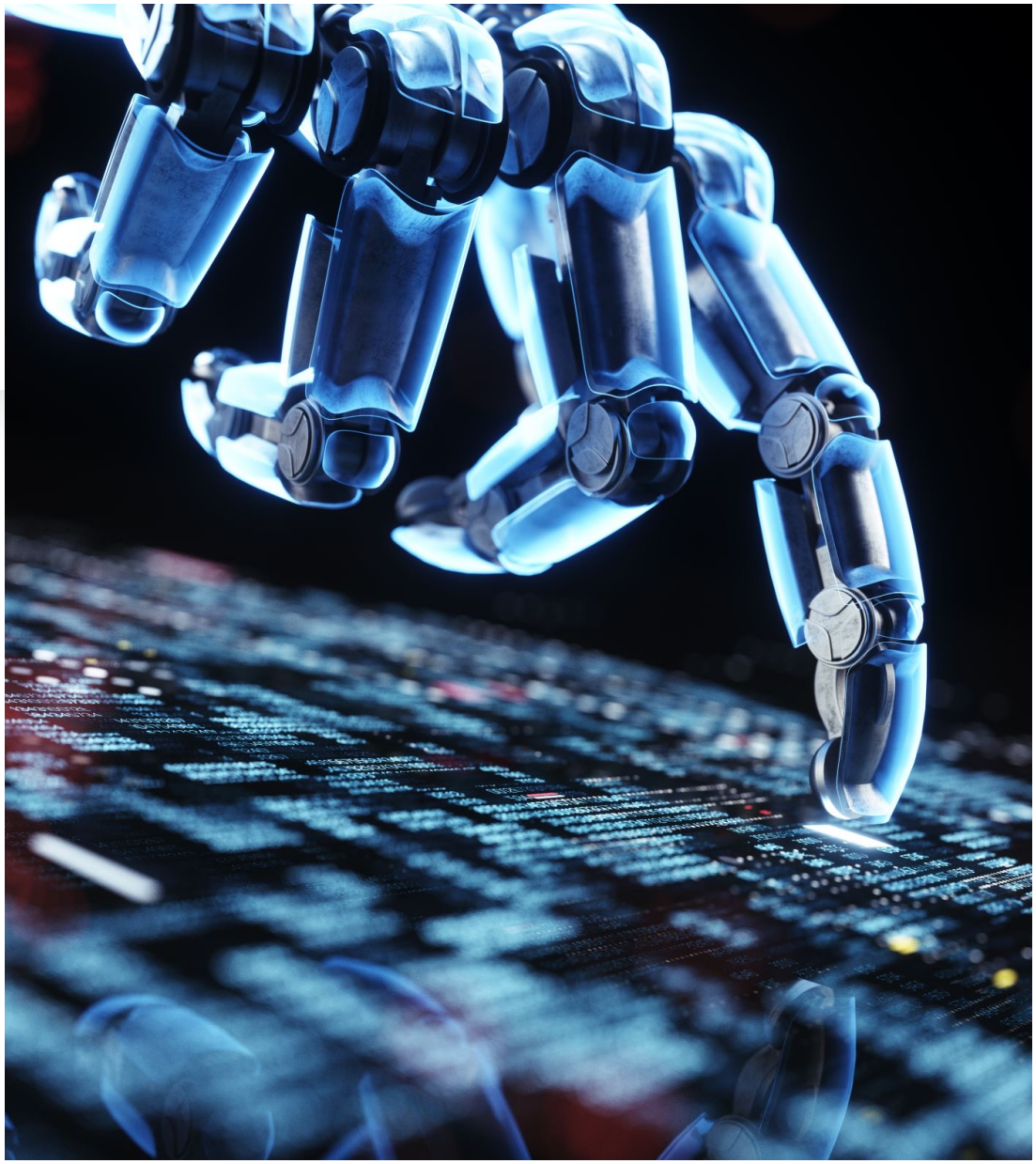
```
use std::env;

fn read_port() -> u16 {
    let port_str = env::var("APP_PORT").unwrap(); // panics if missing
    let port: i32 = port_str.parse().unwrap();    // panics if invalid
    port as u16
}
```

```
use std::{env, num::ParseIntError};

#[derive(Debug)]
enum PortError {
    MissingVar,
    Parse(ParseIntError),
}

fn read_port() -> Result<u16, PortError> {
    let raw = env::var("APP_PORT").map_err(|_| PortError::MissingVar)?;
    let port = raw.parse::<u16>().map_err(PortError::Parse)?;
    Ok(port)
}
```


Focus of this side path of the course:
Automated testing and
program analysis

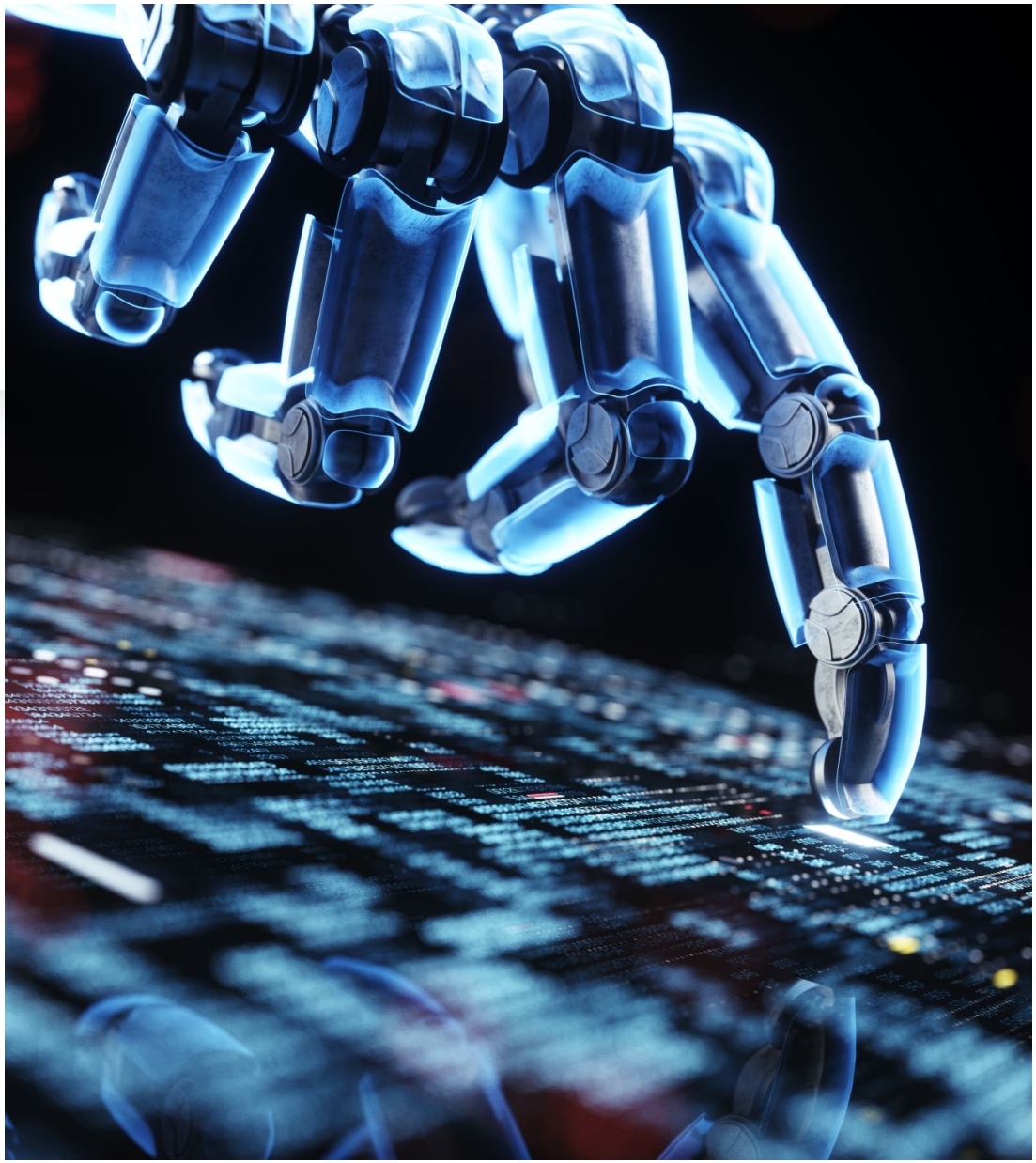
Why do we talk about program analysis and automated testing?

- All software has bugs
- Bugs are hard to find
- Bugs cause serious harm



Why do we talk about program analsysis and automated testing?

- All software has bugs
- Bugs are hard to find
- Bugs cause serious harm
- … Because no software is bug-free, an AI-based copilot can help us to spot and resolve bugs more efficiently.

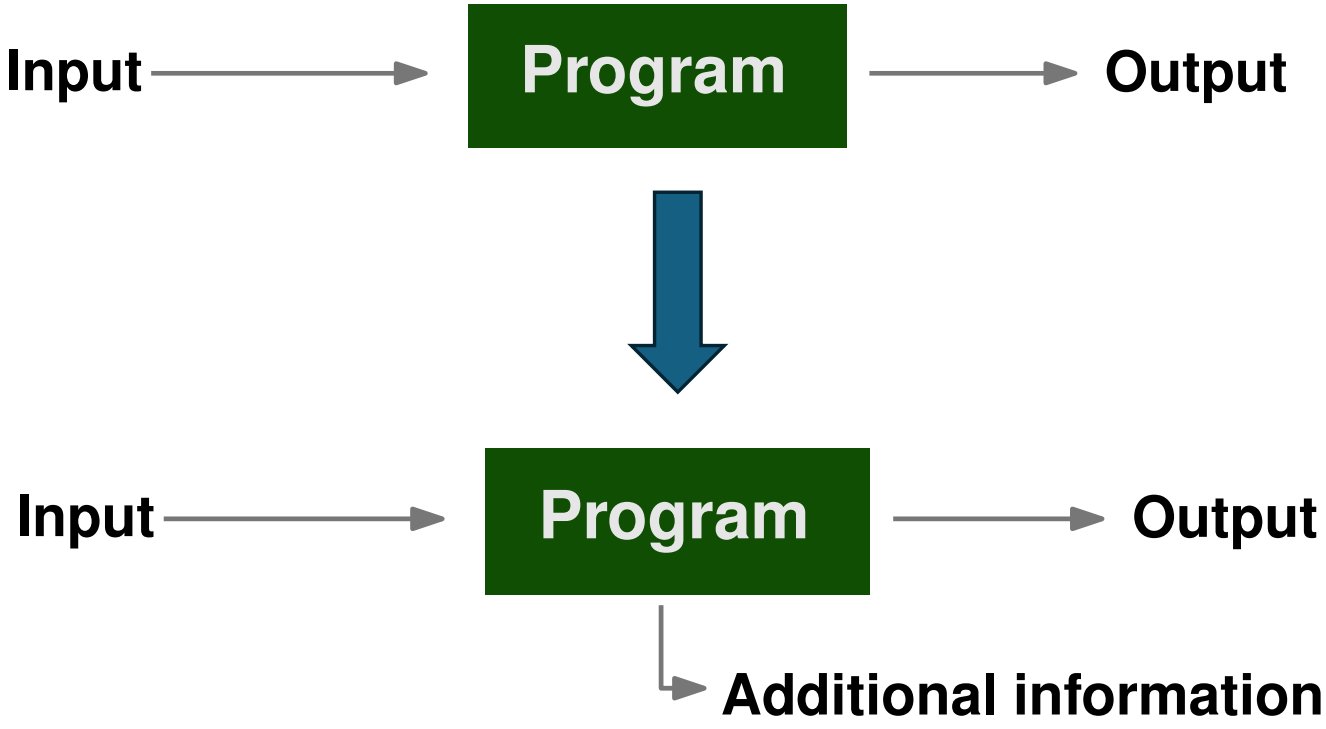


My personal statement

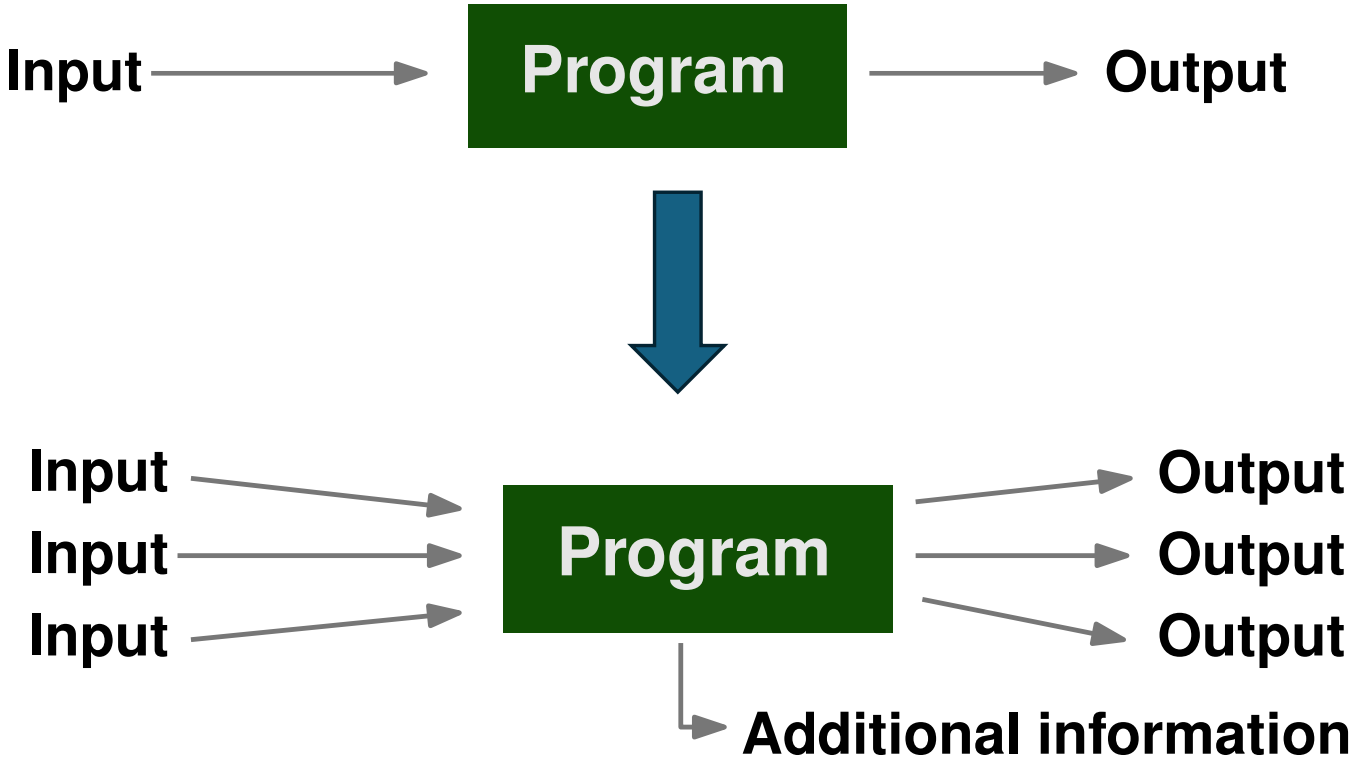
- While AI-based copilots offer valuable support in software development, human understanding of the generated code remains essential, as software (**even AI-generated software**) may still contain bugs.

What is program analysis

From Program Execution to Program Analysis



Program Analysis



Program analysis

Static

- Analyze source code, byte code, or binary
- Typically:
  - Consider all inputs
  - **Overapproximate possible behavior**

Dynamic

- Analyze program execution
- Typically:
  - Consider current input
  - **Underapproximate possible behavior**

Program analysis example

```
//Jvascript
var r = Math.random();
var out = "yes";

if (r < 0.5)
    out = "no";

if (r === 1)
    out = "maybe";

console.log(out);
```

Program analysis example

```
//Jvascript
var r = Math.random();
var out = "yes";

if (r < 0.5)
    out = "no";

if (r === 1)
    out = "maybe";

console.log(out);
```

Overapproximation:
”yes”, ”no”, ”maybe”

Consider all paths

Program analysis example

```
//Jvascript
var r = Math.random();
var out = "yes";

if (r < 0.5)
    out = "no";

if (r === 1)
    out = "maybe";

console.log(out);
```

Underapproximation: ”yes”

Execute the program once

Program analysis example

```
//Jvascript
var r = Math.random();
var out = "yes";

if (r < 0.5)
    out = "no";

if (r === 1)
    out = "maybe";

console.log(out);
```

Sound and complete analysis::
”yes”, ”no”

The analysis explores both feasible paths

Program analysis example

```
//Jvascript
var r = Math.random();
var out = "yes";

if (r < 0.5)
    out = "no";

if (r === 1)
    out = "maybe";

console.log(out);
```

Math.random()
Returns a floating-point number r such that 0 ≤ r < 1
It never returns exactly 1.

Initialization
out starts as "yes".

First if condition (r < 0.5)
If the random number is less than 0.5, out becomes "no".
This happens roughly 50% of the time.

Second if condition (r === 1)
This would only run if r is exactly 1.
But since Math.random() never produces 1, this branch is impossible

Output
So out is either "no" or "yes" depending on r.

Another example

```
//Jvascript
var r = Math.random();

var out = r*2

console.log(out);
```

Another example

```
//Jvascript
var r = Math.random();

var out = r*2

console.log(out);
```

Method	Output	Discussion
Over-approximation		
Under-approximation		
Sound and Complete		

Another example

```
//Jvascript
var r = Math.random();

var out = r*2

console.log(out);
```

Method	Output	Discussion
Over-approximation	Any value	All paths
Under-approximation		
Sound and Complete		

Another example

```
//Jvascript
var r = Math.random();

var out = r*2

console.log(out);
```

Method	Result	Discussion
Over-approximation	Any value	All paths
Under-approximation	Some number in [0,2), e.g., 1.234	One execution
Sound and Complete		

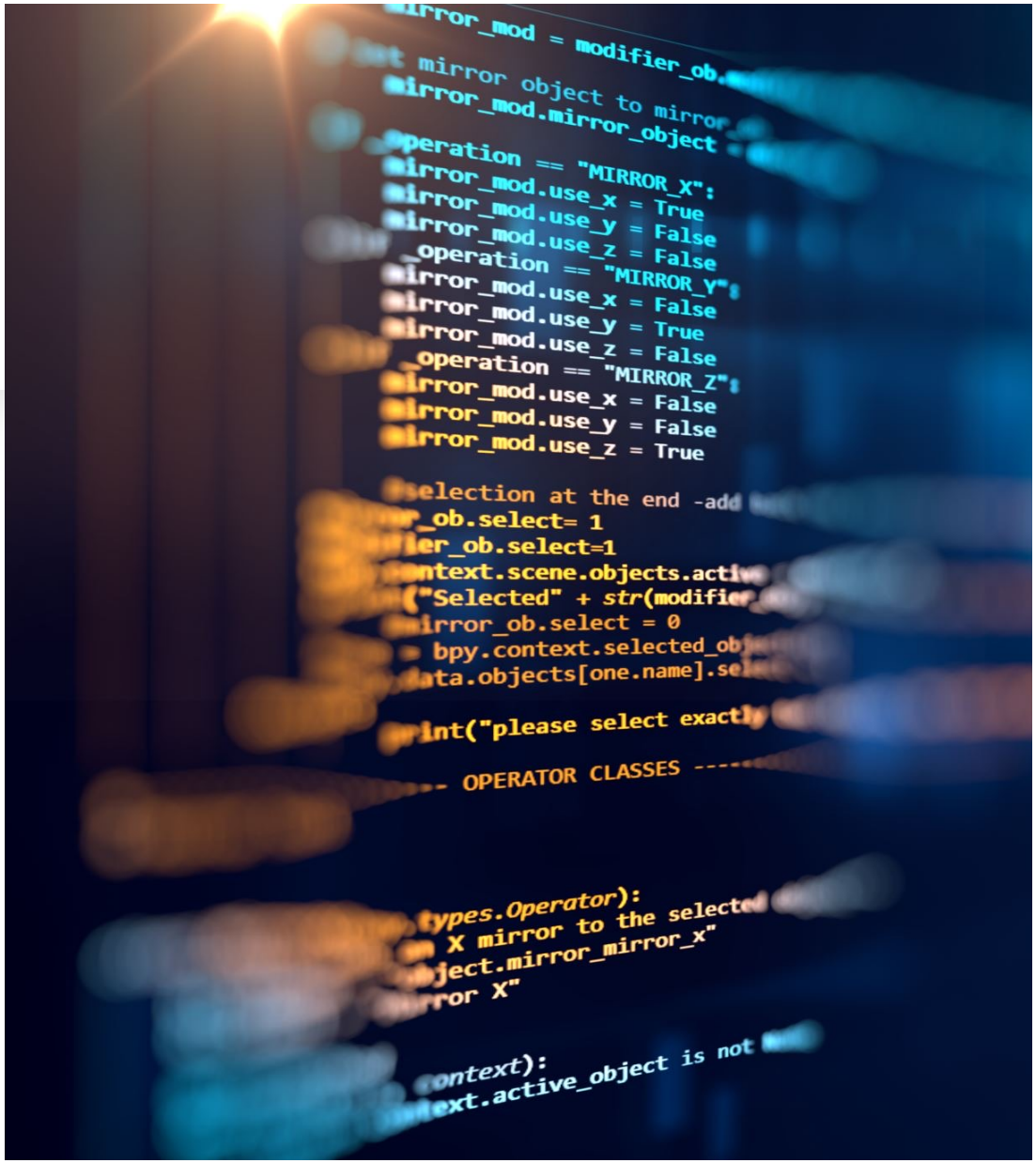
Another example

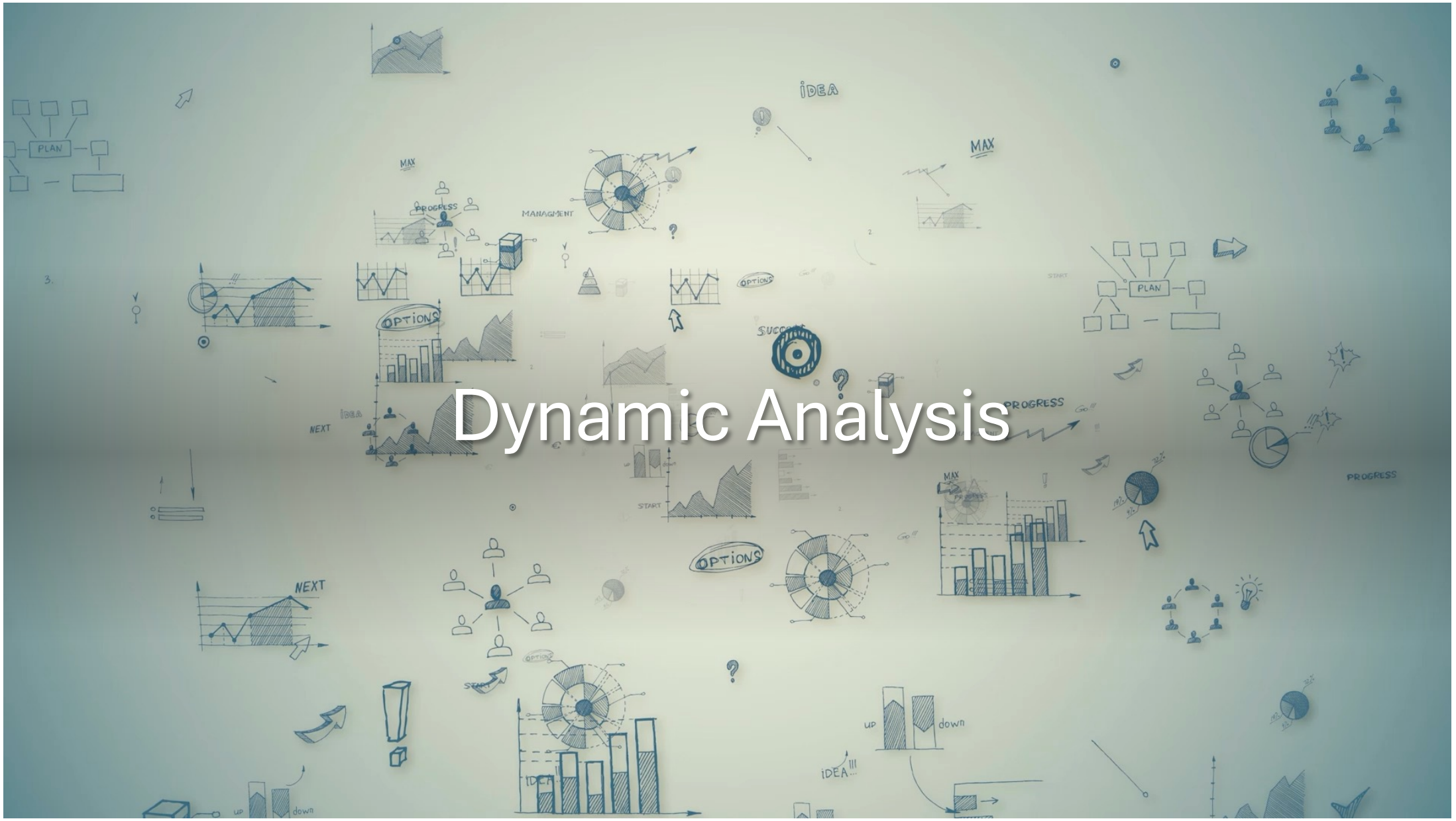
```
//Jvascript
var r = Math.random();

var out = r*2

console.log(out);
```

Method	Result	Discussion
Over-approximation	Any value	All paths
Under-approximation	Some number in [0,2), e.g., 1.234	One execution
Sound and Complete	??	Exploring all possible outputs: Practically impossible

How does program analysis help me?

- **Use program analysis tools**
  - Improve the quality of your code
- **Understand program analysis**
  - Better understanding of program code and program behaviours
- **Create your own tools**




Dynamic Analysis

Dynamic Analysis

Execute an **instrumented program** to gather information that can be analyzed to learn about a property of interest

Precise: All observed behavior actually happens

Incomplete: Very difficult to cover all possible behaviors

Dynamic Analysis Examples

Coverage: Track which lines or branches get executed

Call graph: Track which functions call which other functions

Slicing: Track dependencies to produce a reduced program

Some references

- Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation, Nethercote et al., PLDI 2007
- Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript, Sen et al., FSE 2013
- DynaPyt: A Dynamic Analysis Framework for Python, Eghbali et al., FSE 2022

Call graph analysis

```
function main() {
    let n = readInput();

    function a() {
        b();
    }

    function b() {
        if (n == 5) {
            c();
        }
    }

    function c() {
        if (n == 5) {
            c();
            n--;
        }
    }

    a();
}
```

Call graph analysis

Nodes
main, readInput, a, b, c

Edges
main → readInput
main → a
a → b
b → c *(if n == 5)*
c → c *(if n == 5)*. // **recursive call**

```
function main() {
    let n = readInput();

    function a() {
        b();
    }

    function b() {
        if (n == 5) {
            c();
        }
    }

    function c() {
        if (n == 5){
            n--;
           c();
        }
    }

    a();
}
```

The instrumented code

```
function main() {
    const calls = new Set();

    let n = readInput();
    calls.add("main->readInput");

    function a() {
        calls.add("a->b");
        b();
    }

    function b() {
        if (n == 5) {
            calls.add("b->c");
            c();
        }
    }

function c() {
        if (n == 5) {
            calls.add("c->c");
            n=--;
            c();
        }
    }

    calls.add("main->a");
    a();

    console.log(calls);
}
```

Different dynamic analyses, but with several commonalities

Specific runtime events to track

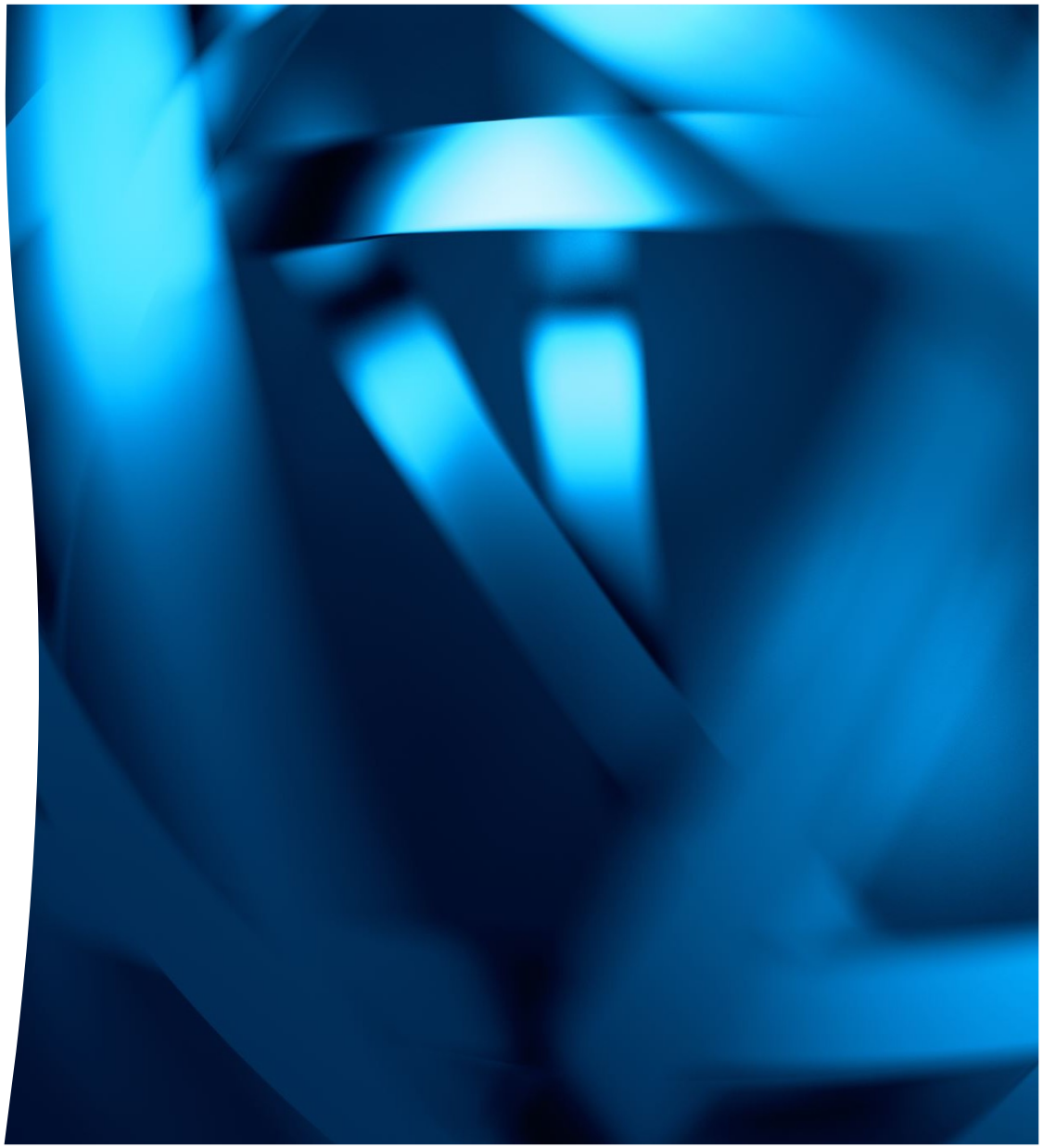
Analysis updates some state in response to events

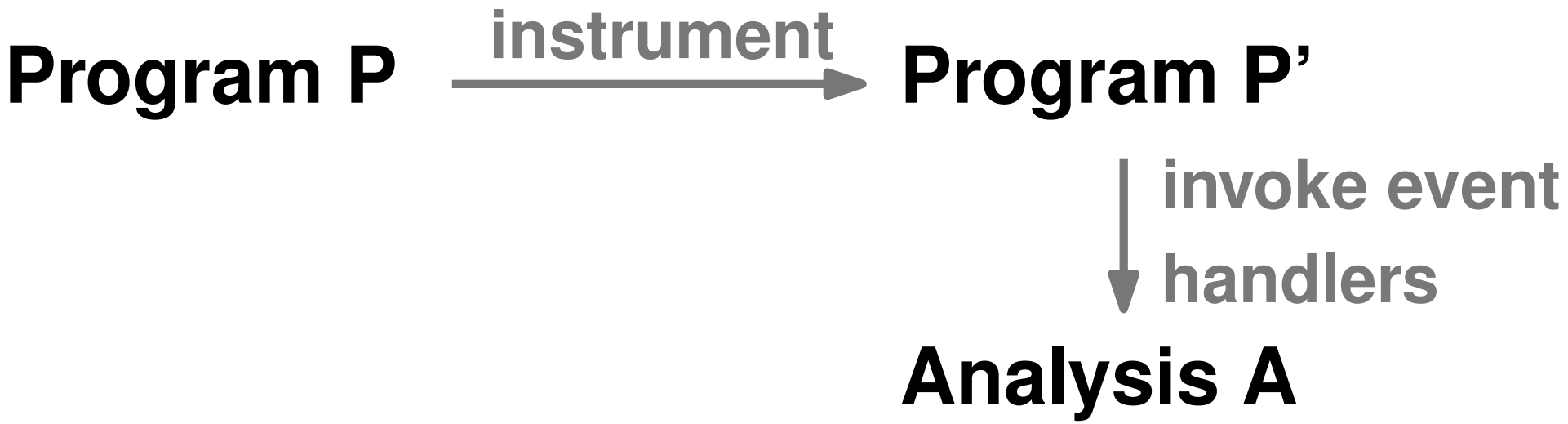
The ingredients

Identify the set (of kinds) of runtime events

The analysis can register for specific events

At runtime, instrumented program invokes event handlers





The idea (cont.)

Event	Example
Arithmetic operation	`2+3`
Boolean operation	`a > 0`
Branch	`if (c) ...`
Function call	`g()`
Return from function call	`x = g()`
Write into variable or field	`x.f = z`
Read of variable or field	`x.f = z`

Example

```
function main() {
    let a = readInput();
    let b = a + 3;

    if (b === -23) {
        foo();
    } else {
        b = 5;
    }
}
```

Example

```
function main() {
    let a = readInput();
    let b = a + 3;

    if (b === -23) {
        foo();
    } else {
        b = 5;
    }
}
```

Runtime events:

- Arithmetic operations
- Boolean operations
- Reads of variables
- Writes into variables
- Function calls

Input: -26

Example

```
function main() {
    let a = readInput();
    let b = a + 3;

    if (b === -23) {
        foo();
    } else {
        b = 5;
    }
}
```

Runtime events:

- Arithmetic operations
- Boolean operations
- Reads of variables
- Writes into variables
- Function calls

Input: -26

What sequence of events get triggered?

Example

```
function main() {
    let a = readInput();
    let b = a + 3;

    if (b === -23) {
        foo();
    } else {
        b = 5;
    }
}
```

Runtime events:

- Call of **readInput**
- Write of **-26** to **a**
- Read of **a** (-26)
- Arithmetic op (-26+3 = -23)
- Write of **-23** to **b**
- Read of **b** (-23)
- Boolean op (-23 == -23 aka true)
- Call foo

Input: -26

This is the sequence of events triggered!!

Remark and questions

Easy enough for small examples.

- But what happens with **large codebases**?
- How can we **formally guide** the **instrumentation of code and the analysis**?

Formally: Estended Operational Semantics

Tracking runtime events: Additional behavior performed during program execution

Formally describe by extending the operational semantics

Some Notation

Runtime events:

- Write of **-26** to **a ===>** $\boldsymbol{a \leftarrow -26}$
- The if branch is taken **===> if true**

Some Notation

CONFIGURATION

$$\langle P, s, e \rangle$$

P: program
s: store (mapping from program variables to values)
e: sequence of run-time events

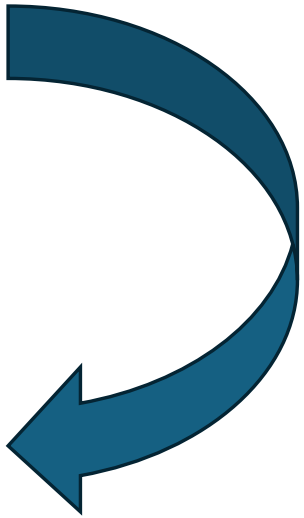
Example

$$\frac{s(x) = n}{\langle !\, x, s\rangle \rightarrow \langle n, s\,\rangle}\ \text{VAR}$$

Example

$$\frac{s(x) = n}{\langle !\, x, s\rangle \rightarrow \langle n, s\,\rangle} \text{ VAR}$$

$$\frac{s(x) = n}{\langle !\, x, s, e\rangle \rightarrow \langle n, s, e\,\rangle} \text{ VAR}$$



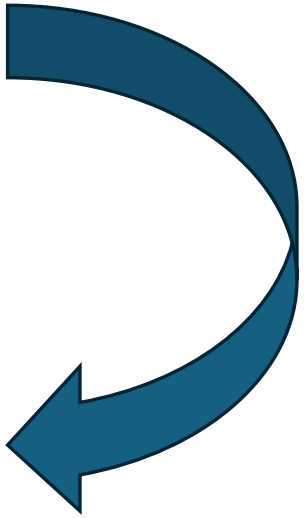
Run-time events are left unchanged

Example

$$\frac{}{\langle x ::= n, s\rangle \rightarrow \langle skip, s[x == n]\,\rangle}\ \text{ASS}$$

Example

$$\frac{}{\langle x ::= n, s\rangle \rightarrow \langle skip, s[x == n]\,\rangle}\ \text{ASS}$$



$$\frac{}{\langle x ::= n, s, e\rangle \rightarrow \langle skip, s[x == n], e\,;(x \leftarrow n)\,\rangle}\ \text{ASS}$$

$$\frac{}{\langle if\ true\ then\ C1\ else\ C2, s, e \rangle \rightarrow \langle C1, s, e; (if\ true) \rangle}$$

QUIZ: Extend the operational rules for conditional – all cases

The Valgrind Framework

Valgrind is a *dynamic binary instrumentation framework*.

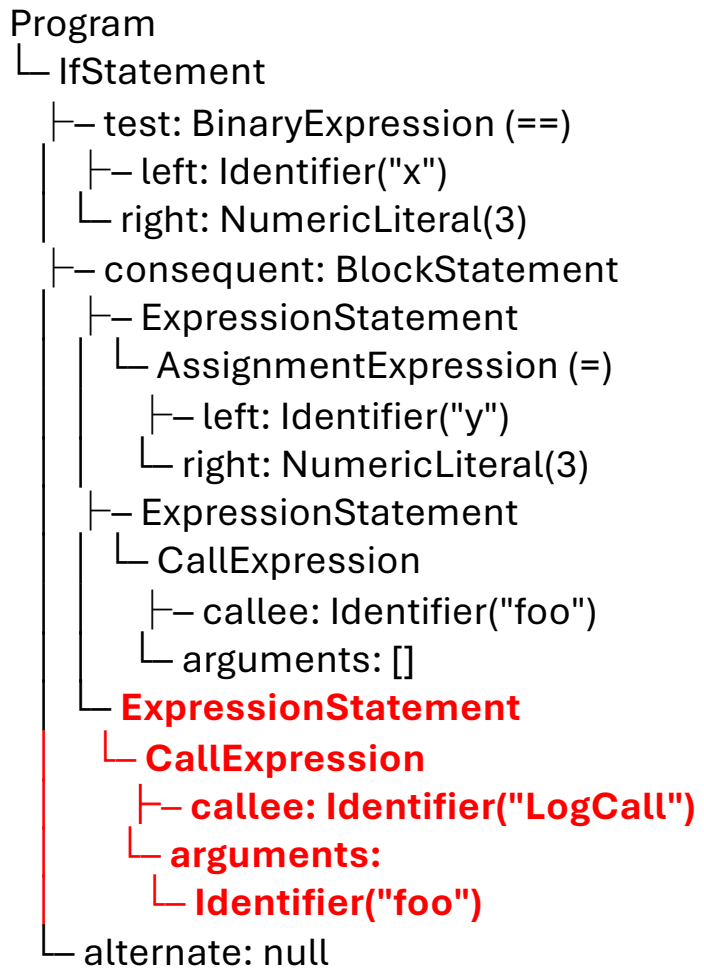
Valgrind runs programs inside a virtual machine and **monitors its behavior at runtime** instrumenting the source code.

The Instrumentation code

```
if (x === 3) {
    y = 3;
    foo();
    LogCall(foo);
}
```

The instrumented AST

```
if (x === 3) {
    y = 3;
    foo();
    LogCall(foo);
}
```



Implementation steps

1. The instrumented AST **is compiled into** an intermediate form (IR).
2. The Valgrind IR allows handling of run-time events around memory operations, system calls, and thread interactions.
3. This IR runs on on the Valgrind virtual machine
4. Valgrind intercepts and logs all the run-time events of interest.
5. When the program finishes (or fails), Valgrind produces a **detailed report** describing:
   - Where run-time events occurred (with call stacks)
   - What kind of run-time events they were
   - Which allocations caused memory leaks

Best Practice

A robust workflow for C/C++ (and analogous for other languages) often looks like this:	**Static phase** — run:	**Dynamic phase** — run:
	• clang-tidy: enforces code style, detects suspicious logic. • cppcheck or Coverity: deeper semantic and data-flow analysis.	• valgrind --leak-check=full ./program: check runtime memory safety. • valgrind --tool=helgrind ./program: check for race conditions.

What is the significance of all this within the framework of this course?

What have we learned?

Two key aspects:

Instrumenting the language and its operational semantics (understood as a guide to implementation) to capture and manage **run-time events**.

Analyzing run-time events to gain a better understanding of what **dynamic program analysis** means for the program under examination.

History dependent Access Control

A Running Example

History Dependent Access Control: Key Ideas

History-dependent access control: access permissions depend not only on *who* the principal is, but also on *what actions* have occurred previously in the program's execution.

Security policies as predicates over execution histories

access is granted only if a specific authentication step was executed earlier.

Provides a **formal model** connecting **program semantics** with **security enforcement**.

History dependent access control: discussion

History dependent access control allows one express access policies of the form:

Allow access only if this request has been authenticated by a trusted login program,

Allow a module to access resource only when invoked through certain routines

Ensures **fine-grained control: execution traces** areused to reason about **dynamic, context-sensitive access control**.

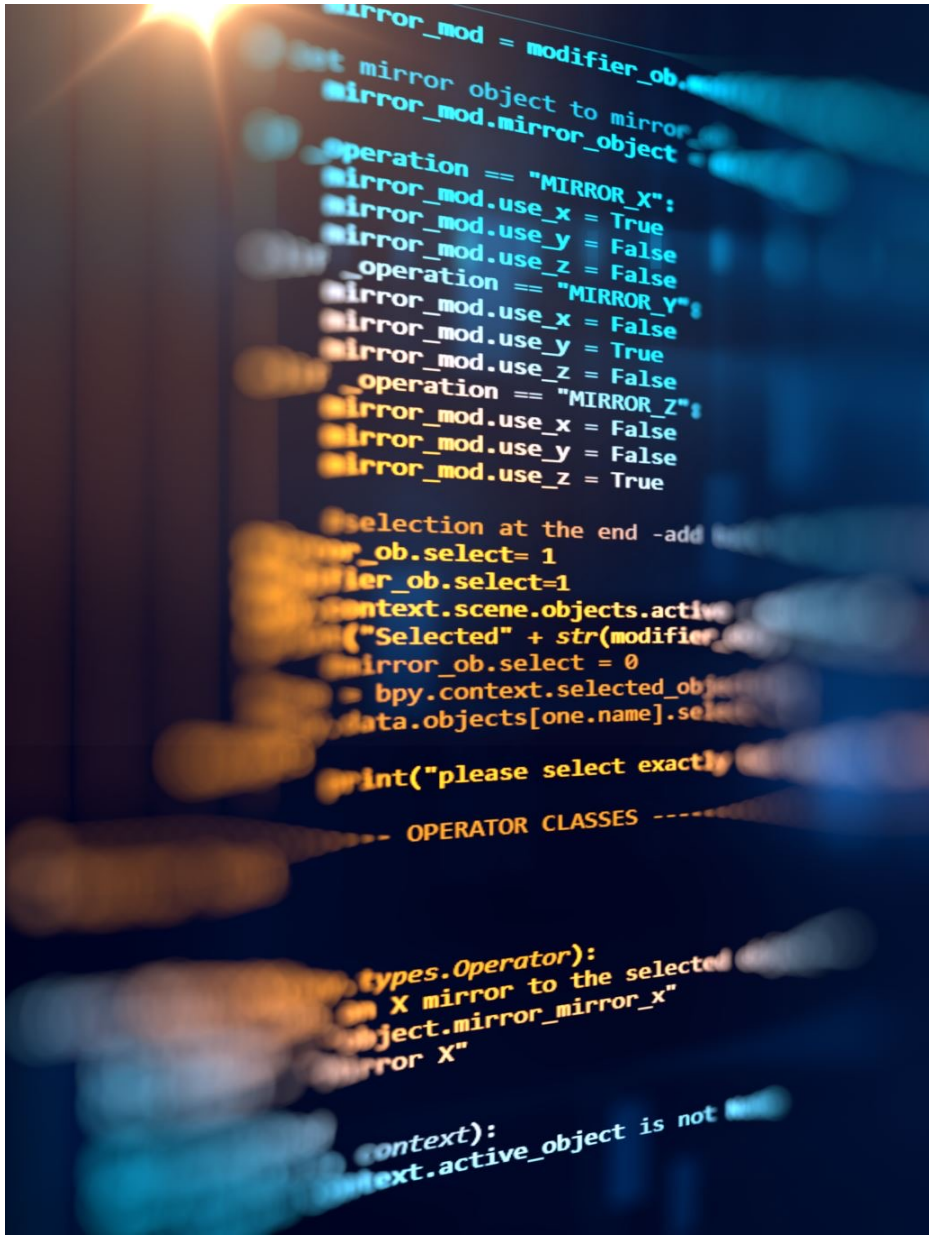
References

- **M. Abadi and C. Fournet.** *Access Control Based on Execution History.*
  In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS 2003)*, San Diego, CA, 2003.

References

M. Abadi and C. Fournet. *Access Control Based on Execution History.* In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS 2003)*, 2003.

Massimo Bartoletti, Pierpaolo Degano, Gian-Luigi Ferrari: History-Based Access Control with Local Policies. In Proc, FoSSaCS 2005.



Motivating example

- Consider a simple **web browser** that displays HTML pages and executes suitable code.
- Code may be **trusted** (for instance, because they are signed or downloaded from a trusted third party) or **untrusted**.
- The browser enforces a **security policy** which is **always applied to untrusted code:**
- The security policy states that *code cannot connect to the network after it has read from the local disk*.

We define the browser as a function that processes a URL *u*.
If the URL refers to an HTML page, the page is displayed.
If the URL refers to code, it is executed when trusted; otherwise, it is executed under the control of a security policy.

```
(* u : url  *)
(* execute : (unit -> unit)-> unit *)
(* enforce : policy -> (unit -> unit)-> unit *)

let browser (u : url) : unit =
  if html u then
    display u
  else if trusted u then
    excute  u
  else
    enforce p u  //p is the browser security policy
```

Trusted programna

We consider three trusted programs:

- **Read** to read files,
- **Write** to write files,
- **Connect** to open network connections.

These programs are overly simplified, because we are only interested in the events they execute.

Trusted Read

```
(* An action type *)
type action = Read | Write | Connect

(* Dispatcher: maps an action to a thunk (unit -> unit) *)
let action : action -> unit -> unit = function
  | Read  -> (fun () -> print_endline "Performing READ")
  | Write -> (fun () -> print_endline "Performing WRITE")
  | Connect -> (fun () -> print_endline "Performing CONNECT")

(* The function: ignores its argument and executes Read *)
let TrustedRead : 'a -> unit =
  fun _ -> action Read ()
```

Runtime enforcement: key idea

- **Execution monitors** enforce history-based security policies at run time
  - They **observe** computations and **abort** when a violation is imminent.
- **Events** = observations of **security-relevant actions**
  - Examples: opening a socket, reading/writing a file.
- **Histories** = (possibly infinite) **sequences of events**.
- **Policy as a global invariant**: must hold **at every point** during execution.
- **Expressive power**: execution monitors enforce **exactly the class of safety properties** (not liveness).

Trusted code

The behavior of trusted code, when executed within the browser, does not violate security policies;

for trusted code, the security monitor is vacuous (i.e., disabled).

Untrusted code

```
(* An action type *)
type action = Read | Write | Connect

(* Dispatcher: maps an action to a thunk (unit -> unit) *)
let action : action -> unit -> unit = function
  | Read  -> (fun () -> print_endline "Performing READ")
  | Write -> (fun () -> print_endline "Performing WRITE")
  | Connect -> (fun () -> print_endline "Performing CONNECT")

(* The function: idoes not gnore its argument *)
let Untrusted : url ->'a -> unit =
  fun (z:url) _ -> Browser z
```

Execution

What is the behaviour of an untrusted code (**Untrusted Write**) executed under the security policy **Pw** stating that untrusted code cannot write the local file system?

Execution

The behaviour of an untrusted code (**Untrusted Write**) executed under the security policy **Pw** stating that untrusted code cannot write the local file system, is illustrated by the following trace:

$$\epsilon . Browser(Untrusted\ Write)$$

$$\downarrow$$

$$\epsilon, enforce\ Pw\ \ Write$$

$$\downarrow$$

$$w, raise\ Security\ Failure$$

$$sequence\ of\ \ actions, code$$

$$\uparrow$$

CONFIGURATION

Execution

The behaviour of an untrusted code (**Untrusted Read&Connect**) executed under the security policy **Pnc** stating that untrusted code cannot connect to the network after reading the local file system, is illustrated by the following trace:

$$\epsilon . Browser(UntrusteRead\&Connect)$$

$$\epsilon, enforce\ Pnc\ \ Read\ ;\ Connect$$

$$r, enforce\ Pnc\ \ Connect$$

$$\text{r;c}, raise\ Security\ Failure$$

we have a security exception, because the r;c does not satisfy the security policy Pnc

The Formal Representation

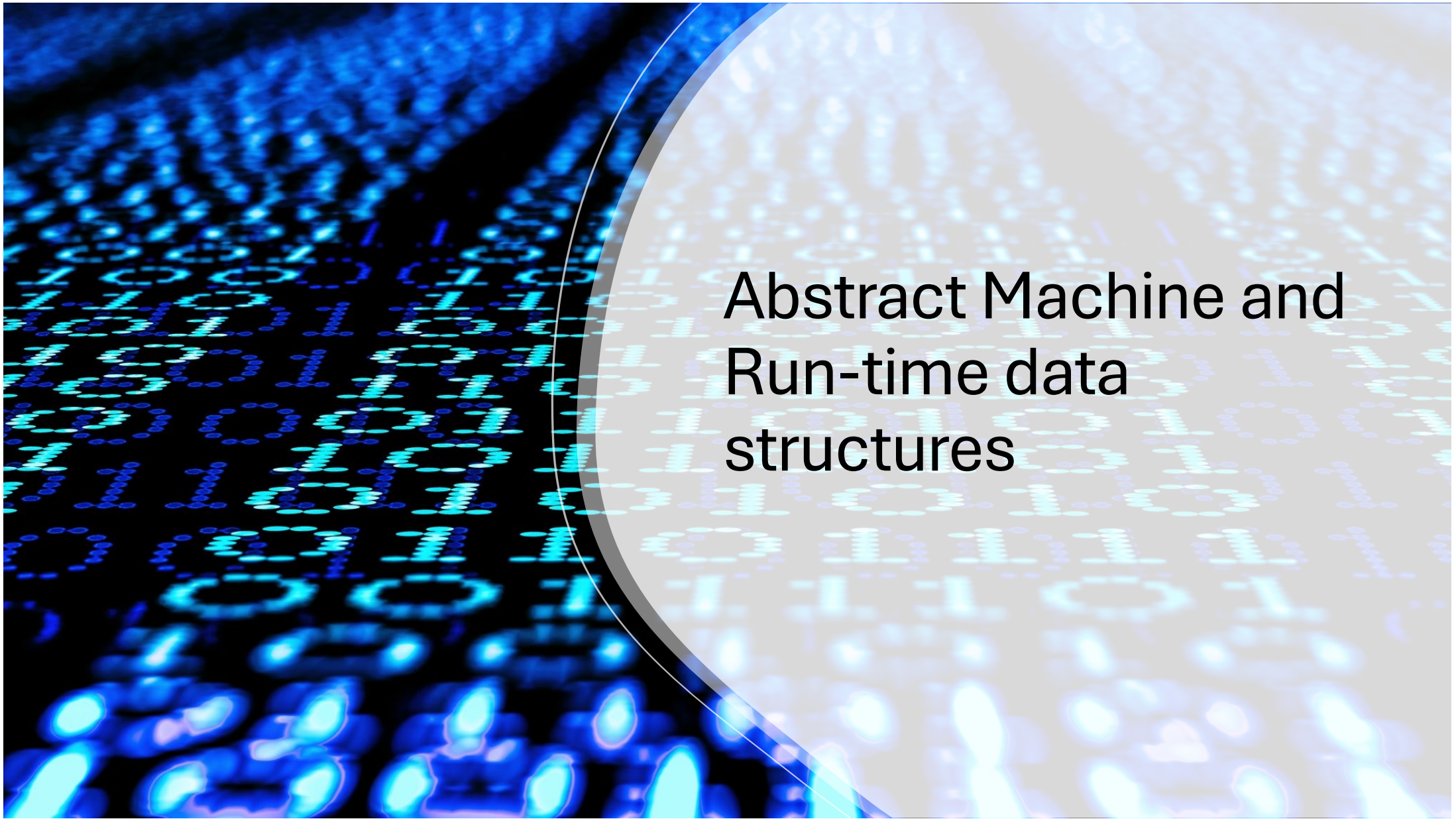
Syntax

EXPRESSIONS: Functional programming language

$$e, e' \quad ::= \quad x \mid \alpha \mid \texttt{if}\ b\ \texttt{then}\ e\ \texttt{else}\ e' \mid \lambda_z x.\, e \mid e\, e'$$

PROGRAMS: Sandbox monitoring execution under the policy

$$P ::= \varphi[e]$$


Abstract Machine and
Run-time data
structures

Execution Monitors

Runtime Monitoring: Checking policy adherence during execution (e.g., sandboxing)

Enforcement Mechanisms in VM: Virtual machines (e.g., WebAssembly, JVM) enforce security through restricted execution environments.

Execution Monitors: Intercepts security-sensitive operations and ensures they comply with policies.

Some challenging questions

Can we prove that mechanism **M** enforces the security policy **P**?	Are there limits to what is enforceable?	Policies?	OVERALL
• What is the mathematical definition of a policy? • What is the programming abstractions for declaring security policies? • What does it mean to enforce a policy within a programming language?	• Which enforcement approaches are best suited to which	• Are there some policies that are completely beyond any known enforcement strategy? • Are some enforcement approaches strictly more powerful than others?	• what is the landscape of policies, policy classes, and enforcement mechanisms?

Enforceable Security Policies

Enforceable Security Policies

F. Schneider, ACM Transactions on Information and System Security, February 2000

Abstract

A precise characterization is given for the class of security policies enforceable with mechanisms that work by monitoring system execution, and automata are introduced for specifying exactly that class of security policies. Techniques to enforce security policies specified by such automata are also discussed.

The Ideal Execution Monitor

1. **Sees *everything* a program is about to do before it does it**
2. **Can *instantly* and *completely* stop program execution (or prevent action)**
3. **Has *no other effect* on the program or system**

The Ideal Execution Monitor

1. **Sees *everything* a program is about to do before it does it**
2. **Can *instantly* and *completely* stop program execution (or prevent action)**
3. **Has *no other effect* on the program or system**

Can we implement this?

Probably not …….

Real ~~Ideal~~ Execution Monitor

1. **Sees** *~~everything~~* **most things a program is about to do before it does it**
2. **Can** *instantly* and *completely* stop program execution (or **prevent action**~~)~~
3. **Has** *~~no other~~* ***effect*** **on the program or system**

limited

Execution Monitor

Execution Monitors (EMs)

- EMs watch un-trusted programs at runtime
- Events (raised by the run time executions) are mediated by the EM
- Violations solicit EM interventions (e.g. termination)

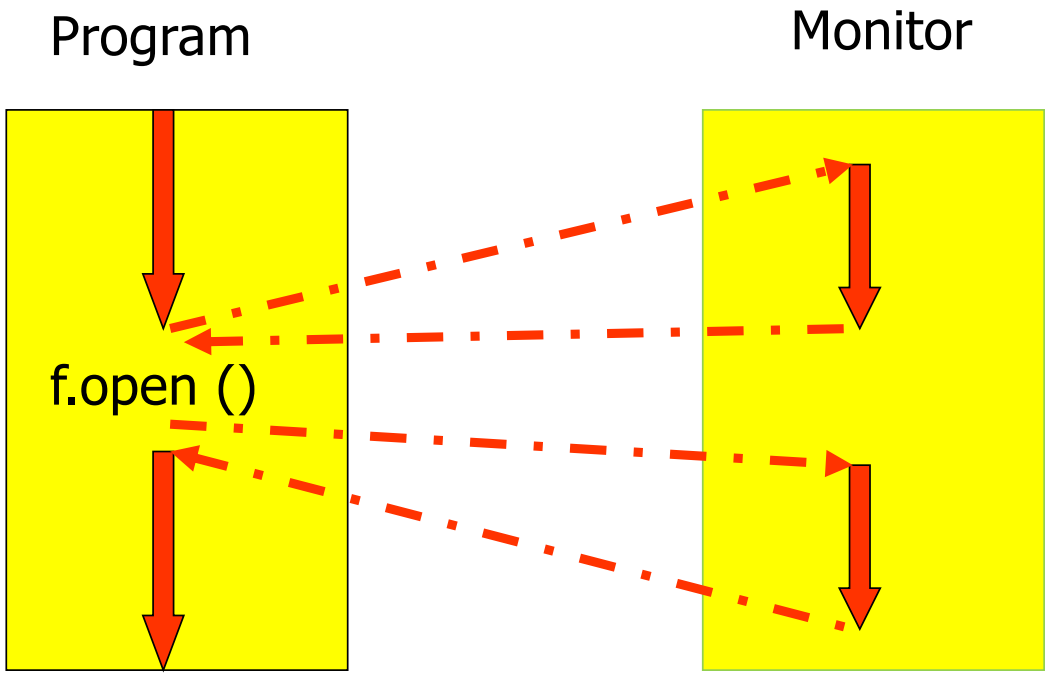
Example: File system access control

- EM is inside the OS
- decides policy violations using access control lists (ACLs)

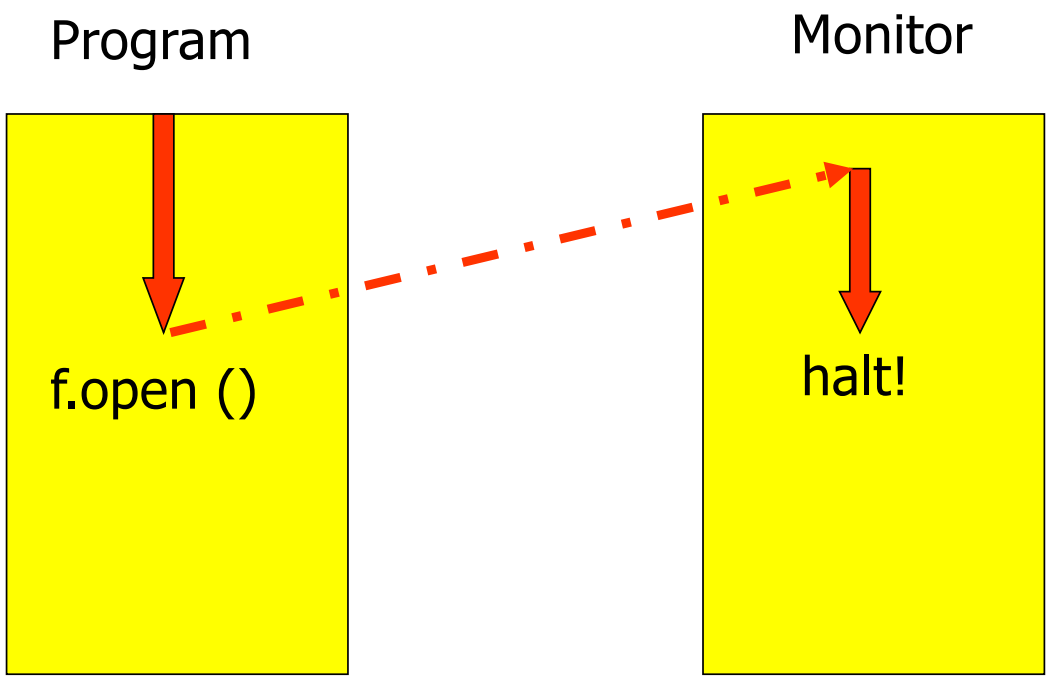
Execution Monitor in programming language design

- EMs are run-time structures that runs in parallel with the program
  - Tracks execution flow at runtime to **detect and prevent** security violations dynamically.
  - monitor decisions may be based on **execution history**

EM: Good Operations



EM: Bad Operation



Execution and Security Policies

What's a program (at run-time)?

- A set of possible executions

What's an execution?

- A sequence of states

What's a security policy?

- A predicate on a set of executions

Definitions and Notations

- An *execution (or trace)* **s** is a sequence of security-relevant program *events* **e** (also called actions)
- Sequence may be **finite** or **(countably) infinite**
  - **$s = e_1; e_2; ....; e_k ; e_{halt}$**
  - **$s = e_1; e_2; ....; e_k; .....$**
  - The empty sequence ε is an execution
  - If s is the execution **$e_1; e_2; ....; e_i; .... e_l; ....$**
    then **s[i]** is the execution **$e_1; e_2; ....; e_i;$**
- We simplify the formalism.
  - We model program termination as an infinite repetition of **e_{halt}** event.
  - Result: now all executions are infinite length sequences

Definitions and Notations (cont.)

- A program **S** is a **set** of sequences (possible executions)
  - A program is modelled as the set S = {s_1, s_2, ...}
- A security policy **P** is a **property** of programs
- A policy partitions the program space into two groups:
  - Permissible
  - Impermissible
- Impermissible programs are censored somehow (e.g.,terminated on violating runs)

Execution Monitors cannot enforce all Security Policies

- Some policies depend on:
  - Knowing about the future
    - If the program charges the credit card, it must eventually ship the goods
  - Knowing about all possible executions
    - Information flow – can't tell if a program reveals secret information without knowing about other possible executions
- Execution Monitors can only know about past of the particular execution at hand

EM-enforceable policies

1. EM policies are universally quantified predicates over executions
   - $\forall s. P(s)$
   - **Policy P is called the detector.**

EM-enforceable policies

1. EM policies are universally quantified predicates over executions
   - $\forall s. P(s)$
   - **Policy P is called the detector.**
2. The detector must must be prefix-closed
   - **P(ε) holds**
   - **P(s;e) holds then P(s) holds**

EM-enforceable policies

1. EM policies are universally quantified predicates over executions
   - $\forall s. P(s)$
   - **Policy P is called the detector.**
2. The detector must must be prefix-closed
   - **P(ε) holds**
   - **P(s;e) holds then P(s) holds**
3. If the detector is not satisfied by a sequence (the detector rejects the sequence) then it must do so in finite time
   - $\neg P(s) \implies \exists i \neg Ps[i]$

EM-enforceable policies

1. EM policies are universally quantified predicates over executions
2. The detector must must be prefix-closed
3. If the detector is not satisfied by a sequence (the detector recjects the sequence) then it must do so in finite time
4. **Fact**
   - **A policy satisfies (1), (2), and (3) if and only if it is a *safety policy***
   - **Lamport 1977: Safety policies say that some "bad thing" never happens**
   - **EMs enforce safety policies!**

Safety properties: Nothing bad ever happens

Safety property can be enforced **using only traces** of program

- If P(t) does not hold, then all extensions of t are also bad

Amenable to **run-time enforcement**: don't need to know future

Examples:

- **access control** (e.g. checking file permissions on file open)
- **memory safety** (process does not read/write outside its own memory space)
- **type safety** (data accessed in accordance with type)

Formally…

- Ψ : set of all possible executions (can be infinite)
- Σ_S: set of executions possible by target program S
- P: security policy

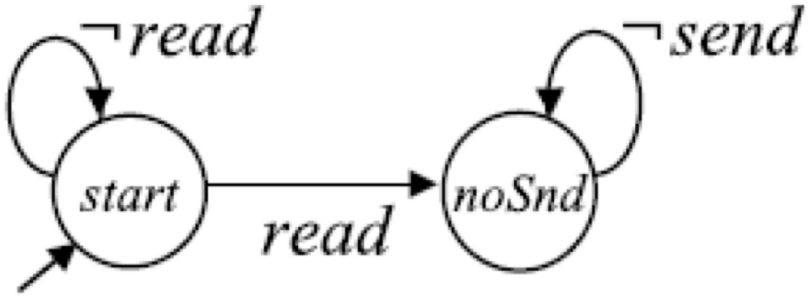
  set of executions $\rightarrow$ Boolean

Program S is secure wrt the security policy P iff $P\ (\Sigma_S)$ is true.

Security Automata

- Security Automata (Erlingsson & Schneider 1999)
- Formalization of security policies
  - finite state automaton
  - accepts language of permissible executions
  - alphabet = set of events
  - edge labels = event predicates
  - all states accepting (language is prefix-closed)

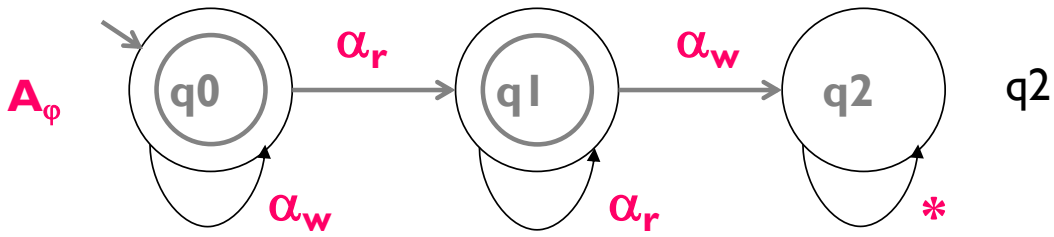
Security Automata (Example)



NO SENDS AFTER READS

CHINESE WALL POLICY

φ Chinese Wall: cannot write (α_w) after read (α_r)



q2 non-final sink

$\alpha_w \alpha_r \alpha_w \not\models \varphi$

Execution Monitor

```
type event = Read of string | Write of string | Connect of string
type history = event list
type policy = history -> event -> bool  (* true = allowed *)

type caps = {
  read    : string -> string;
  write   : string -> unit;
  connect : string -> unit;
}

let sandbox (phi : policy) (base : caps) (prog : caps -> 'a) : 'a =
  let hist = ref [] in
  let check e =
    if phi !hist e then hist := e :: !hist
    else failwith "Policy violation"
  in
  :
}
```

Syntax

EXPRESSIONS

$$e, e' \quad ::= \quad x \mid \alpha \mid \texttt{if}\ b\ \texttt{then}\ e\ \texttt{else}\ e' \mid \lambda_z x.\, e \mid e\, e'$$

PROGRAMS

$$P ::= \varphi[e]$$

NOTE: LAMBDA

ANONYMOUS FUNCTIONS λ **x.e**

λ **x. x+1**

OCAML NOTATION

fun x = e
fun x = x + 1

NOTE: LAMBDA

RECURSIVE FUNCTIONS λ_z x.e

λ_{fact} x. if x <= 1 then 1 else x * fact(x-1)

OCAML NOTATION

```
let rec fact x =
        if x <= 1 then 1 else x * fact (x - 1);;
```

NOTE: SEQUENTIAL COMPOSITION

$\lambda\_.\, e$ for $\lambda x.\, e$, for $x \notin fv(e)$

$e; e' = (\lambda\_.\, e')e.$

Operational semantics

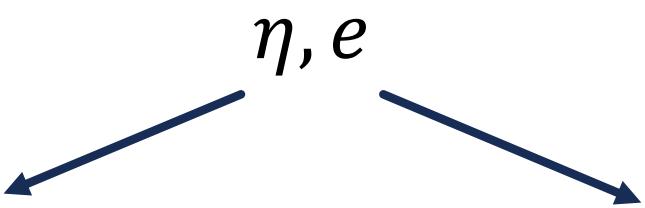
- The configurations are pairs

$$\eta, e$$

- A transition $\eta, e \to \eta', e'$ indicates that, starting from a history η, the expression e may evolve to e', possibly extending the history η.
- We write $\eta \vDash \varphi$ when the history η satisfies the policy φ.

ABSTRACT MACHINE: DEFINITION

Configuration of the abstract machine



Execution history

Program to be executed

record the events generated by program execution on its operational environment

ABSTRACT MACHINE: DEFINITION

Configuration of the abstract machine

$$\eta, e$$

Operational rules

$$\eta\ e \ \rightarrow \ \eta\ e'$$

NOTE: IN WHAT WE TRUST?

Configurations: η, e

The abstract mechine: **only trusts its own history**

Operational semantics

$$\frac{\eta, e_1 \to \eta', e_1'}{\eta, e_1 e_2 \to \eta', e_1' e_2} \qquad \frac{\eta, e_2 \to \eta', e_2'}{\eta, v e_2 \to \eta', v e_2'} \qquad \frac{}{\eta, (\lambda_z x.e) v \to \eta, e\{v/x, \lambda_z x.e/z\}}$$

Operational Semantics

$$\frac{\mathcal{B}(b) = true}{\eta, \texttt{if}\ b\ \texttt{then}\ e_0\ \texttt{else}\ e_1 \rightarrow \eta, e_0} \qquad \frac{\mathcal{B}(b) = false}{\eta, \texttt{if}\ b\ \texttt{then}\ e_0\ \texttt{else}\ e_1 \rightarrow \eta, e_1}$$

Operational Semantics

$$\frac{}{\eta, \alpha \longrightarrow \eta\alpha, *}$$

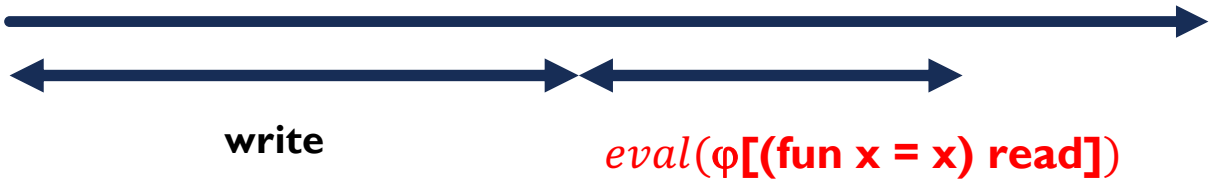
Operational Semantics

$$\frac{\eta, e \longrightarrow \eta', e' \quad \eta, \eta' \models \varphi}{\eta, \varphi[e] \longrightarrow \eta', \varphi[e']} \qquad \frac{\eta \models \varphi}{\eta, \varphi[v] \longrightarrow \eta, v}$$

EXAMPLE

φ[write; (fun x = x) read]

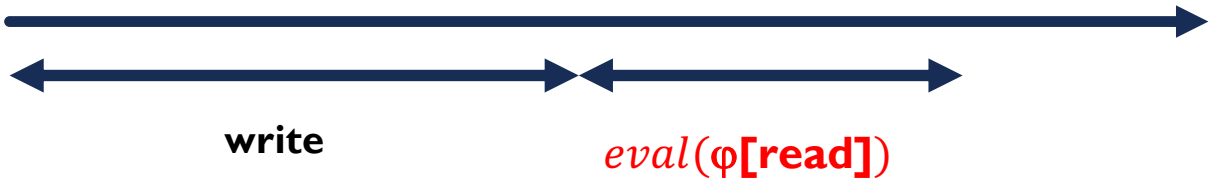
Policy φ = no read after write



EXAMPLE

φ[write; (fun x = x) read]

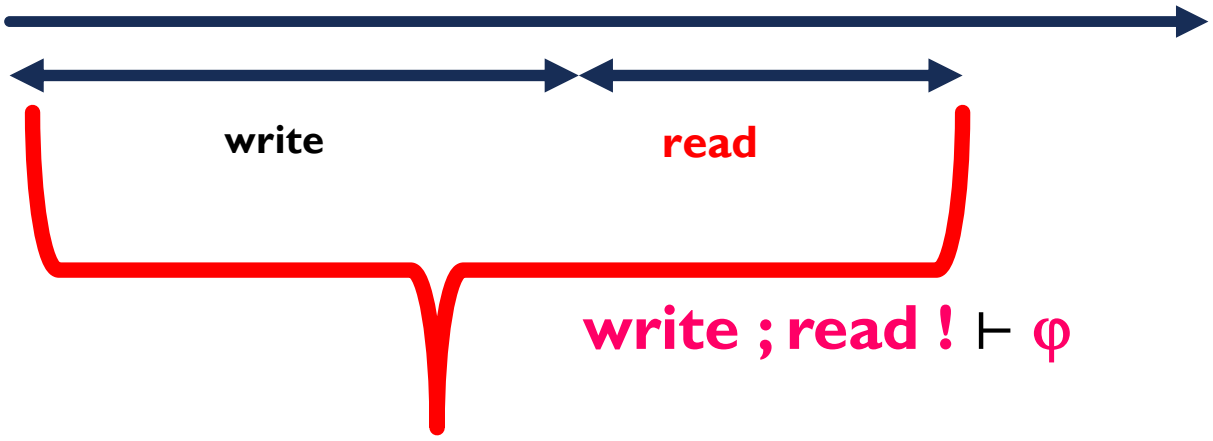
Policy φ = no read after write



EXAMPLE

φ[write; (fun x = x) read]

Policy φ = no read after write



Summary

- **Dynamic Analysis of History-Dependent Access Control**
- We have shown how to **characterize the dynamic analysis** of **history-dependent access control**, defining its **formal structure** and its **limitations** (safety security policies).

Next step

Can we do better?
Can we **integrate static and dynamic analysis**?

➡️ **Type & Effect system**

Amalgating static and dynamic enforcement of security properties

Static Observable Behaviours

- **access events** are the actions relevant for security (e.g. read/write local files, invoke/be invoked by a given service, etc)
  - mechanically inferred, or inserted by programmer.
  - their meaning is fixed globally.
  - access events cannot be hidden

CHECKING STATICALLY

1. We extract the **abstract behaviour** of the program via a suitable **type system** that ***over-approximates the possible run-time behavior of the program***
2. This abstract behaviour is **compiled** into a suitable form to be **model-checked** in order ***to verify whether or not the security constraints are satisfied***
3. As a result of the model checking we ensure that the program ***satisfies the security property***.

THE TYPE SYSTEM

Type & effect system

- **types** carry **effects** annotations **H** about abstract behaviour
- **effects H**, are called ***history expressions***, and over-approximate the actual execution histories

TYPES

(basic types are pretty standard)

$$\tau ::= \text{int} \mid \text{bool} \mid \text{unit} \mid \cdots \mid$$

$$\tau \xrightarrow{H} \tau'$$

Functional types

FUNCTIONAL TYPES

Functional types

The history expression H describes the latent effect associated with the functional abstraction (the set of possible execution histories)

When the abstraction is applied to a value, its execution will generate a run time behaviours belonging to the execution histories represented by H

INTERMEZZO

- Plain (traditiona) type systems have judgments of the form

$$\Gamma \vdash e : \tau$$

- to mean:
  - ***e* won't get stuck**
  - **if *e* produces a value, that value has type τ**

INTERMEZZO

- Adding effects to typing rules mean to compute statically something about "***how program executes***" .
- There are many things we may want to conservatively approximate

EFFECTS (HISTORY EXPRESSIONS)

$H ::=$

ε	**empty**
h	**variable**
α	**access event**
$H \cdot H'$	**sequence**
$H + H'$	**choice**
$\mu h.H$	**recursion**

WHAT IS THE MEANING OF HISTORY EXPRESSIONS?

When the program is sent into execution, it will generate one of the histories abstract represented by H.

History expressions are a sort of context-free grammars

The language generated by this grammars are abstract program behaviours

WHAT IS THE MEANING OF HISTORY EXPRESSIONS?

$$H = \mu h.\, \alpha + \beta.\, H$$

What is the set of run-time executions abstractly represented by H?

WHAT IS THE MEANING OF HISTORY EXPRESSIONS?

$$H = \mu h.\, \alpha + \beta . H$$

What is the set of run-time executions abstractly represented by H?

$$F(h) = \alpha + \beta \cdot h$$

We need to solve this fix point equation

$$H = F(H) = \{\alpha\} \cup \{\beta\} \cdot H.$$

The least fixpoint

WHAT IS THE MEANING OF HISTORY EXPRESSIONS?

$$H = \mu h.\, \alpha + \beta.H$$

What is the set of run-time executions abstractly represented by H?

$$F(h) = \alpha + \beta \cdot h$$

We need to solve this fix point equation

$$H = F(H) = \{\alpha\} \cup \{\beta\} \cdot H.$$

The least fixpoint

$$\begin{aligned} H &= \{\alpha\} \cup \{\beta\} \cdot H \\ &= \{\alpha\} \cup \{\beta\} \cdot (\{\alpha\} \cup \{\beta\} \cdot H) \\ &= \{\alpha,\, \beta\alpha\} \cup \{\beta^2\} \cdot H \\ &= \{\alpha,\, \beta\alpha,\, \beta^2\alpha,\, \beta^3\alpha, \ldots\} \end{aligned}$$

Unfolding the equation

WHAT IS THE MEANING OF HISTORY EXPRESSIONS?

$$H = \mu h.\, \alpha + \beta.H$$

What is the set of run-time executions abstractly represented by H?

$$\begin{aligned} H &= \{\alpha\} \cup \{\beta\} \cdot H \\ &= \{\alpha\} \cup \{\beta\} \cdot (\{\alpha\} \cup \{\beta\} \cdot H) \\ &= \{\alpha,\ \beta\alpha\} \cup \{\beta^2\} \cdot H \\ &= \{\alpha,\ \beta\alpha,\ \beta^2\alpha,\ \beta^3\alpha, \ldots\} \end{aligned}$$

The denotation is the set of all finite traces consisting of zero or more β events followed by one α event:

$$\boxed{H = \{\, \beta^n \alpha \mid n \geq 0 \,\}.}$$

WHAT IS THE MEANING OF HISTORY EXPRESSIONS?

$$H = \mu h.\, \alpha + \beta.H$$

$$\begin{aligned} H &= \{\alpha\} \cup \{\beta\} \cdot H \\ &= \{\alpha\} \cup \{\beta\} \cdot (\{\alpha\} \cup \{\beta\} \cdot H) \\ &= \{\alpha,\ \beta\alpha\} \cup \{\beta^2\} \cdot H \\ &= \{\alpha,\ \beta\alpha,\ \beta^2\alpha,\ \beta^3\alpha, \ldots\} \end{aligned}$$

What is the set of run-time executions abstractly represented by H?

the run-time executions perform any number of β events (possibly none) and then one α event— exactly capturing the behavior of the recursive equation

$$F(h) = \alpha + \beta \cdot h$$

THE DENOTATIONAL SEMANTICS OF HISTORY EXPRESSIONS

The denotational semantics is defined over the complete lattice

$(\wp(\Sigma) \ \subseteq)$ $\Sigma \ is \ the \ set \ of \ access \ events$

The environment ρ is used to map variables to sets of (finite) histories

THE DENOTATIONAL SEMANTICS OF HISTORY EXPRESSIONS

$$[\![\varepsilon]\!]_\rho = \varepsilon \qquad [\![\alpha]\!]_\rho = \alpha \qquad [\![h]\!]_\rho = \rho(h)$$

$$[\![H \cdot H']\!]_\rho = [\![H]\!]_\rho [\![H']\!]_\rho \qquad [\![H + H']\!]_\rho = [\![H]\!]_\rho \cup [\![H']\!]_\rho$$

$$[\![\mu h.H]\!]_\rho = \bigcup_{n \in \omega} f^n(\emptyset) \quad \text{where } f(X) = [\![H]\!]_{\rho\{X/h\}}$$

THE DENOTATIONAL SEMANTICS OF HISTORY EXPRESSIONS

$$[\![\varepsilon]\!]_\rho = \varepsilon \qquad [\![\alpha]\!]_\rho = \alpha \qquad [\![h]\!]_\rho = \rho(h)$$

$$[\![H \cdot H']\!]_\rho = [\![H]\!]_\rho [\![H']\!]_\rho \qquad [\![H + H']\!]_\rho = [\![H]\!]_\rho \cup [\![H']\!]_\rho$$

$$[\![\mu h.H]\!]_\rho = \bigcup_{n \in \omega} f^n(\emptyset) \quad \text{where } f(X) = [\![H]\!]_{\rho\{X/h\}}$$

The least fixpoint over the CPO $(\wp(\Sigma)\ \subseteq)$

THE TYPE AND EFFECT SYSTEM

Types and Type Environments

$$\tau \quad ::= \quad unit \;\mid\; \tau \xrightarrow{H} \tau \qquad\qquad \Gamma \quad ::= \quad \emptyset \;\mid\; \Gamma; x : \tau \quad (x \notin dom(\Gamma))$$

THE TYPE AND EFFECT SYSTEM

$\Gamma, H \vdash e : \tau$ **Typing Judgment**

Intuition: the expression e evaluates to a value of type τ, and produces a history belonging to the effect H .

$$\frac{\Gamma, H \vdash e : \tau}{\Gamma, H + H' \vdash e : \tau}$$

STRUCTURAL RULE: WEAKENING

$$\frac{}{\Gamma, \varepsilon \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma, \alpha \vdash \alpha : unit} \qquad \frac{}{\Gamma, \varepsilon \vdash * : unit}$$

THE TYPING RULES

$$\frac{\Gamma; x : \tau; z : \tau \xrightarrow{H} \tau', H \vdash e : \tau'}{\Gamma, \varepsilon \vdash \lambda_z x.e : \tau \xrightarrow{H} \tau'} \qquad \frac{\Gamma, H \vdash e : \tau \xrightarrow{H''} \tau' \quad \Gamma, H' \vdash e' : \tau}{\Gamma, H \cdot H' \cdot H'' \vdash ee' : \tau'}$$

THE TYPING RULES

$$\frac{\Gamma; x : \tau; z : \tau \xrightarrow{H} \tau', H \vdash e : \tau'}{\Gamma, \varepsilon \vdash \lambda_z x.e : \tau \xrightarrow{H} \tau'} \qquad \frac{\Gamma, H \vdash e : \tau \xrightarrow{H''} \tau' \quad \Gamma, H' \vdash e' : \tau}{\Gamma, H \cdot H' \cdot H'' \vdash ee' : \tau'}$$

THE TYPING RULES

$$\frac{\Gamma, H \vdash e : \tau \quad \Gamma, H \vdash e' : \tau}{\Gamma, H \vdash \texttt{if}\ b\ \texttt{then}\ e\ \texttt{else}\ e' : \tau}$$

THE TYPING RULES

$$\frac{\Gamma, H \vdash e : \tau}{\Gamma, \varphi[H] \vdash \varphi[e] : \tau}$$

TYPING PROGRAMS

EXAMPLE

We want to derive the type judgment

$$\emptyset, \varepsilon \vdash \lambda_z x.\ \text{if } b \text{ then } \alpha \text{ else } \beta;\ z\, x :\ \text{unit} \xrightarrow{H^\star} \text{unit},$$

with

$$H^\star = \mu h.(\alpha + \beta \cdot h).$$

STEP 1: TYPING THE BODY

Let $\Gamma = \{z : \text{unit} \xrightarrow{H} \text{unit},\ x : \text{unit}\}$.
We derive a type for the body if b then α else β; $z\,x$.

(a) By the EVENT rule: $\Gamma, \alpha \vdash \alpha : \text{unit} \quad \Gamma, \beta \vdash \beta : \text{unit}$.

(b) For the application $z\,x$:

$\Gamma, \varepsilon \vdash z : \text{unit} \xrightarrow{H} \text{unit} \quad \Gamma, \varepsilon \vdash x : \text{unit}$

then by the APP rule:

$\Gamma, H \vdash z\,x : \text{unit}$.

(c) Sequencing $\beta; z\,x \equiv (\lambda.\, z\,x)\,\beta$:

By ABS and APP rules: $\Gamma, \beta \cdot H \vdash \beta;\ z\,x : \text{unit}$.

(d) For the conditional:

$\Gamma, \alpha \vdash \alpha : \text{unit} \quad \Gamma, \beta \cdot H \vdash \beta;\ z\,x : \text{unit}$.

Using WK and IF rules, we obtain

$\Gamma, \alpha + \beta \cdot H \vdash \text{if } b \text{ then } \alpha \text{ else } \beta;\ z\,x : \text{unit}$.

STEP 2: NAME ABSTRACTION AND FIXPOINT

By the ABS rule for named abstractions:

$$\emptyset, \varepsilon \vdash \lambda_z x.\, e : \text{unit} \xrightarrow{H} \text{unit} \text{ provided that } H = \alpha + \beta \cdot H.$$

STEP 3: SOLVING THE CONSTRAINTS

The least fixpoint solution is $H^\star = \mu h.(\alpha + \beta \cdot h)$.

Hence the final typing judgment is:

$$\boxed{\emptyset, \varepsilon \vdash \lambda_z x.\ \text{if } b \text{ then } \alpha \text{ else } \beta;\ z\, x :\ \text{unit} \xrightarrow{\mu h.(\alpha+\beta\cdot h)} \text{unit}.}$$

STEP4: PUTTING EVEYTHING TOGETHER

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \alpha \vdash \alpha : \text{unit}}\ \text{ev}}{\Gamma, \alpha + \beta \cdot H \vdash \alpha : \text{unit}}\ \text{wk} \qquad \dfrac{\dfrac{\Gamma, \varepsilon \vdash \lambda.\, zx : \text{unit} \xrightarrow{H} \text{unit} \quad \Gamma, \beta \vdash \beta : \text{unit}}{\Gamma, \beta \cdot H \vdash \beta;\ zx : \text{unit}}\ \text{app}}{\Gamma, \alpha + \beta \cdot H \vdash \beta;\ zx : \text{unit}}\ \text{wk}}{\Gamma, \alpha + \beta \cdot H \vdash \text{if } b \text{ then } \alpha \text{ else } \beta;\ zx : \text{unit}}\ \text{if}}{\emptyset, \varepsilon \vdash \lambda_z x.\, \text{if } b \text{ then } \alpha \text{ else } \beta;\ zx : \text{unit} \xrightarrow{H} \text{unit}}\ \lambda$$

SOUNDNESS

$$If\ \Gamma, H \vdash e : \tau\ \ and\ \varepsilon, e \rightarrow^{*} \eta, e',\ then\ \eta \in [\![H]\!]$$

SAFETY

Assume e closed and $\Gamma, H \vdash e : \tau$.
Let $\varphi[e]$ be a program.
If H is valid for φ then e does not wrong

A computation goes wrong when attempting to execute an event forbidden by active policy

VERIFICATION: VALIDITY OF HISTORY EXPRESSIONS

- Idea: model checking Basic Process Algebras (BPAs) with Buchi automata.
  - BPA (the model) -- Buchi automata (the policy)
- The decision procedure for verifying that a BPA process p satisfies a ω-regular property φ amounts to constructing the pushdown automaton for p and the Buchi automaton for the negation of φ.
- The property holds if the (context-free) language accepted by the conjunction of the pushdown automaton and the Buchi automaton (which is still a pushdown automaton), is empty.
- This problem is decidable, and several algorithms and tools show this approach is feasible.

BASIC PROCESS ALGEBRA

$$p \quad ::= \quad \varepsilon \mid \alpha \mid p \cdot p' \mid p + p' \mid X$$

Operational Semantics of BPA processes

$$\frac{}{\alpha \xrightarrow{\alpha} \varepsilon} \qquad \frac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'} \qquad \frac{q \xrightarrow{\alpha} q'}{p + q \xrightarrow{\alpha} q'} \qquad \frac{p \xrightarrow{\alpha} p'}{p \cdot q \xrightarrow{\alpha} p' \cdot q} \qquad \frac{p \xrightarrow{\alpha} p' \quad X \stackrel{def}{=} p \in \Delta}{X \xrightarrow{\alpha} p'}$$

$$BPA(\varepsilon, \Gamma) = \langle \varepsilon, \emptyset \rangle$$
$$BPA(\alpha, \Gamma) = \langle \alpha, \emptyset \rangle$$
$$BPA(h, \Gamma) = \langle \Gamma(h), \emptyset \rangle$$
$$BPA(H_0 \cdot H_1, \Gamma) = \langle p_0 \cdot p_1, \Delta_0 \cup \Delta_1 \rangle, \text{ where } BPA(H_i, \Gamma) = \langle p_i, \Delta_i \rangle$$
$$BPA(H_0 + H_1, \Gamma) = \langle p_0 + p_1, \Delta_0 \cup \Delta_1 \rangle, \text{ where } BPA(H_i, \Gamma) = \langle p_i, \Delta_i \rangle$$
$$BPA(\mu h.H, \Gamma) = \langle X, \Delta \cup \{X \stackrel{def}{=} p\} \rangle, \text{ where } BPA(H, \Gamma\{X/h\}) = \langle p, \Delta \rangle$$

MAPPING HISTORY EXPRESSIONS TO BPA

A MAIN PROPERTY

The prefixes of the histories generated by a history expression H (i.e. $[\![H]\!]^{\pi}$) are all and only the finite prefixes of the strings that label the computations of $BPA(H)$. Recall that this is enough, because validity is a safety property.

THE FORMAL RESULT

- Validity of a history expression H can be decided by showing that the BPA generated by H satisfies a ω-regular formula.
- A closed expression e never violates the security property of the program $\varphi[e]$ if its effect is (model) checked valid.

OUR RUNNING EXAMPLE

The history expression

$$H = \alpha + \beta \cdot H$$

The Basic process Algebra

$$X = \alpha \cdot \epsilon + \beta \cdot X$$

OUR RUNNING EXAMPLE

BPA PROCESS

$$X = \alpha \cdot \epsilon + \beta \cdot X$$

UNFOLDING THE RECURSION

$$X = \alpha \cdot \epsilon + \beta(\alpha \cdot \epsilon + \beta \cdot X) = \alpha \cdot \epsilon + \beta \cdot \alpha \cdot \epsilon + \beta \cdot \beta \cdot X = \ \ldots$$

THE FINITE TRACES

$$TRACES(X) = \{ \beta^n \alpha \mid n \geq 0 \}$$

THE LEAST FIX POINT

APPROXIMATION

$X_0 = 0.$

$X_{n+1} = \alpha \cdot 0 + \beta \cdot X_n$

FIX POINT

$FIX(X) = \cup_n X_n$

THE LEAST FIX POINT

APPROXIMATION

$$X_0 = 0.$$

$$X_{n+1} = \alpha \cdot 0 + \beta \cdot X_n$$

FIX POINT

$$FIX(X) = \cup_n X_n$$

Kleene-style shorthand (not primitive in BPA, but intuitive):

$$\mathrm{FIX}(X) \equiv \beta^*.\alpha.\mathbf{0}$$

any number of β's, then α, then terminate.

THE PROPERTY

The (Büchi) property is "**no α after β**"

The finite-word language enforced is

$\mathcal{L}_{safety} = \alpha^* \beta^*$ *(no α occurs after the first β).*

THE MODEL

$L(H) = \{\beta^n \alpha \mid n \geq 0\} \cup \{\beta^\omega\}$ **we admit infinite runs.**

THE MODEL

$L(H) = \{\beta^n \alpha \mid n \geq 0\} \cup \{\beta^\omega\}$ **we admit infinite runs.**

Finite behaviour (for any k): $\beta^k \alpha$

Any finite behavior violates the property,
since an α appears **after** a β.

THE MODEL

$L(H) = \{\beta^n \alpha \mid n \geq 0\} \cup \{\beta^\omega\}$ **we admit infinite runs.**

Finite behaviour (for any k): $\beta^k \alpha$

Any finite behavior violates the property,
since an α appears **after** a β.

$L(H) \cap \neg L_{safety} \neq \emptyset$

H is **not valid** w.r.t. the
"no α after β" property.

A concrete counterexample is the trace $\beta\alpha$.

SUMMARY: INTEGRATE MODEL CHECKING AND STATIC ANALYSIS

- Combines **model checking** with **static analysis** to **prove properties** of *history-dependent access control* systems.
- Our initial approach uses a **single sandbox** to reason about local histories and access traces.
- In the paper, this approach is **extended to hierarchical sandboxes**, enabling verification across **nested or composed security contexts**.