# L2: Dynamic, Static and Hybrid Analysis

**Software Analysis**

IIT Guwahati · 28 July 2026

# Outline

- What is program analysis?
  - Three kinds: dynamic, static, hybrid
- Dynamic program analysis
- Static program analysis
- A worked example: discovering invariants

# What Is Program Analysis?

- A body of work to **automatically discover useful facts** about programs [1].
- An example of a useful fact: **a programming error**.
- Broadly classified into three kinds:
  - **Dynamic** — at execution time
  - **Static** — at compile time
  - **Hybrid** — combines static and dynamic

[1] Mayur Naik, *CIS 547: Software Analysis*, University of Pennsylvania, Lecture 1

# Three Kinds of Program Analysis

| **Dynamic** | **Static** | **Hybrid** |
| --- | --- | --- |
| Observe program execution | Analyze program code | Combine both approaches |

- Dynamic: requires program inputs.
- Static: no execution required.
- Hybrid: combines static and dynamic information.

# Dynamic Program Analysis

Infers facts about a program by **monitoring its runs**.

**Array bound checking** — Purify[1]

**Memory leak detection** — Valgrind[2]

**Data race detection** — Eraser[3]

**Finding likely invariants** — Daikon[4]

[1] Hastings & Joyce, *Purify: Fast Detection of Memory Leaks and Access Errors*, USENIX Winter 1992 [2] valgrind.org; Nethercote & Seward, PLDI 2007 [3] Savage et al., ACM TOCS 15(4), 1997 [4] Ernst et al., IEEE TSE 27(2), 2001

# Static Program Analysis

Infers facts about a program by **inspecting its source (or binary) code**.

**Suspicious error patterns** — Lint[1], FindBugs[2], Coverity[3]

**Checking API usage rules** — Microsoft SLAM[4]

**Memory leak detection** — Facebook Infer[5]

**Verifying invariants** — ESC/Java[6]

The program is never run — so the facts must hold for *every* run.

[1] Johnson, *Lint, a C Program Checker*, Bell Labs CSTR 65, 1978 [2] Hovemeyer & Pugh, SIGPLAN Not. 39(12), 2004 [3] Bessey et al., CACM 53(2), 2010 [4] Ball & Rajamani, POPL 2002 [5] fbinfer.com; Calcagno et al., NFM 2015 [6] Flanagan et al., PLDI 2002

# What Counts as a “Useful Fact”?

- We said program analysis **automatically discovers useful facts**. Useful for *whom*, and useful for *what*?

- A fact is useful if it lets us **rule out a bad behaviour** or **justify a transformation**:
  - `p` is never `NULL` at line 42 $\rightarrow$ no null dereference here.
  - `i` stays within `0..n-1` $\rightarrow$ this array access is in bounds.
  - `f` is never called $\rightarrow$ delete it.
  - `x` does not depend on the input $\rightarrow$ compute it at compile time.

- Every one of these is a claim about **all runs**, not about the run you happened to try.

# Program Invariants

An **invariant** is a program fact that is true in **every** run of the program.

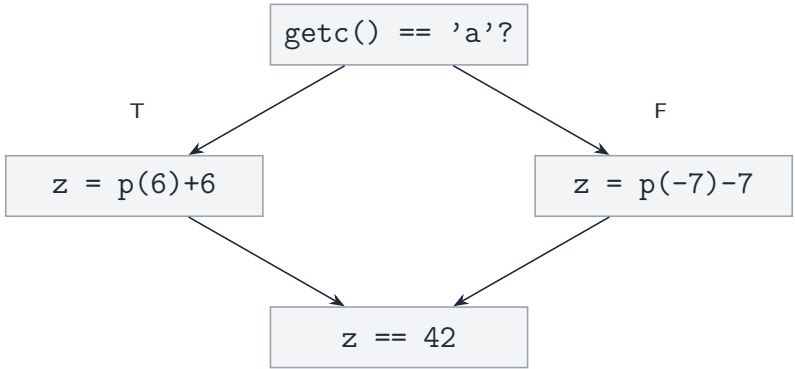
```
int p(int x) { return x*x; }

void main() {
    int z;
    if (getc() == 'a')
        z = p(6) + 6;
    else
        z = p(-7) - 7;
}
```

An invariant at the end of this program is **(z == c)** for some constant **c**.

**What is c?**

# Computing c




$$c = 42$$

# Why This Fact Is Useful

```
int p(int x) { return x*x; }

void main() {
    int z;
    if (getc() == 'a')
        z = p(6) + 6;
    else
        z = p(-7) - 7;
    if (z != 42)
        disaster();
}
```

Since **(z == 42)** is an invariant,

this program can **never** call `disaster()`.

# Discovering Invariants by Dynamic Analysis

- Daikon runs the program a **finite** number of times and reports what held.
- From any run of our program, it concludes:
  - **(z == 42)** is a *likely* invariant
- It **cannot** prove z is *always* 42, so it cannot rule out `disaster()`.
- But it **can** be conclusive in the negative direction:
  - **(z == 30)** is *definitely not* an invariant — one run refutes it.

Daikon; Ernst et al., *Dynamically Discovering Likely Program Invariants to Support Program Evolution*, IEEE TSE 27(2), 2001

# Discovering Invariants by Static Analysis

**(z == 42)** is **definitely** an invariant

**(z == 30)** is **definitely not** an invariant

Static analysis can show at **compile time** that the program will never call `disaster()` at **runtime**.

No inputs. No test cases. No execution.

# Where We Are Going

- We showed static analysis *can* prove `(z == 42)` on this program.
- Natural question: **why not do this for every program?**
- Because there are **fundamental limits** — not engineering limits.
  - Some questions about programs are **undecidable**.
  - Every real analysis must therefore **approximate**.

**Static program analysis aims to automatically answer questions about the possible behaviour of programs.**

Møller & Schwartzbach, *Static Program Analysis*, Aarhus University, 2024

# Why Do We Analyse Programs?

**Optimization**

*make it faster*

**Correctness**

*make it right*

**Security**

*make it safe*

**Concurrency**

*make it well-behaved*

**Development**

*make it understandable*

Taxonomy after SPA, §1.1

# Analysis for Program Optimization

Optimizing compilers — including JIT compilers and interpreters — need program properties to generate efficient code.

- Is function `f` **unreachable** from `main`? → dead code: shrink the binary
- Is an expression inside a loop the **same in every iteration**? → hoist it out of the loop
- Does `x` **depend on the program input**? → if not, precompute at compile time
- What are the **lower and upper bounds** of integer `x`? → pick a smaller runtime representation
- Do `p` and `q` point to **disjoint data structures**? → safe to parallelize

SPA, §1.1

# Analysis for Program Correctness

- Does there exist an input leading to a **null pointer dereference**, **division by zero**, or **arithmetic overflow**?
- Are all variables **initialized** before they are read?
- Are arrays always accessed **within their bounds**?
- Can there be **dangling references** — use of pointers to memory that has been freed?
- Does the program **terminate** on every input?
  - Even in reactive systems such as an OS, individual components (e.g. device drivers) are expected to always terminate.

SPA, §1.1

# Safe vs. Unsafe Languages

The *same* bug has very different consequences depending on the language.

**C / C++ — unsafe**

Out-of-bounds write *silently* corrupts whatever is next in memory.

**Undefined behaviour.**

→ attacker-controlled code execution

**Java / C# — safe**

VM checks every access.

Throws `ArrayIndexOutOfBoundsException`.

**Defined behaviour.**

→ crash, not takeover

**Rust — safe**

Ownership checked by the **compiler**.

Most errors rejected before the program runs.

→ Many memory-corruption exploits become impossible in safe Rust.

# JVM Safety

- **No raw pointers.** References cannot be forged from integers or moved with arithmetic.
- **Bounds checks on every array access** — inserted by the compiler, enforced by the VM.
- **Garbage collection.** Memory is freed only when unreachable → **no dangling pointers, no double-free** *by construction*.
- **Bytecode verifier** rejects malformed code at class-load time — itself a static analysis.
- The price: a **runtime cost** for checks and GC, and errors surface **only when that line runs**.

# Rust Safety — Without a Runtime

```
let q = String::from("hello");
let p = q;                  // ownership MOVES from q to p
println!("{}", q);          // compile ERROR: q was moved
```

- **Ownership:** every value has exactly **one** owner; freeing happens when the owner goes out of scope.
- **Borrowing:** many readers **or** one writer — never both at once.
- Enforced by the **borrow checker** — a static analysis in the compiler.

**No GC. No runtime checks.**

# Correctness Properties from Specifications

Some properties are not universal — they depend on a **specification the programmer supplies**.

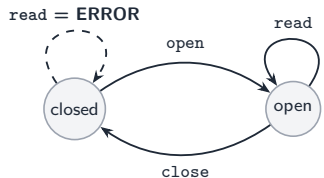
```
int balance = withdraw(account, amount);
assert(balance >= 0);    // program-specific: never overdraw
```

- **Are all assertions guaranteed to succeed?** An assertion expresses a correctness property that must hold in **all** executions.
- No compiler knows "never overdraw" — only you do. The analyser checks *your* claim.

# Typestate Properties

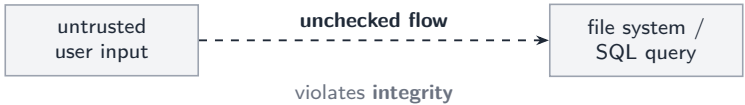
Many APIs require operations in a **particular order** — the legal operations depend on the object's current *state*.

```
FILE *f = fopen("data.txt", "r");
fclose(f);
fread(buf, 1, 10, f);          // read AFTER close
Iterator it = c.iterator();
Object x = it.next();          // next() without hasNext()
```




SPA, §1.1

# Security: Integrity and Confidentiality




SPA, §1.1

# Concurrency: Data Races

Both threads run `balance = balance + 100;` — **not** an atomic operation.

```
Thread 1                        Thread 2
--------                        --------
read   balance    (100)
                                read   balance    (100)
add    100        (200)
                                add    100        (200)
write  balance    (200)
                                write  balance    (200)
```

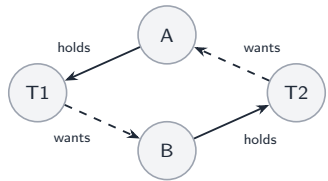
Deposited **200**, balance rose by **100**. One update is lost.

Two accesses, same location, at least one a write, no synchronization → **data race**.

# Concurrency: Deadlock

The fix for races — locks — introduces a new failure mode.

```
Thread 1                        Thread 2
--------                        --------
lock(A);                        lock(B);
lock(B);   waits for T2         lock(A);   waits for T1
```




A **cycle** in the wait-for graph → both threads wait forever.

Static fix: impose a global **lock order**. Static check: prove no cycle exists.

# Analysis for Program Development

Modern IDEs run program analyses constantly to support **debugging**, **refactoring**, and **program understanding**:

*Which function may possibly be called on line 117?*

- "Go to definition" on a virtual call, a function pointer, or a callback is a **static analysis**.
- "Rename this method safely" must know **every** call site — and no others.

You have been using program analysis every day without noticing.

SPA, §1.1

*"Program testing can be used to show the presence of bugs, but never to show their absence!"*

— Edsger W. Dijkstra, 1970

*Notes on Structured Programming*, EWD249, §3; an earlier form appears in the 1969 NATO Software Engineering conference report

# Reasoning About Programs Is Hard

Does this terminate for every integer `n` (arbitrary-precision)?

```
while (n > 1) {
    if (n % 2 == 0)      // if n is even, divide it by two
        n = n / 2;
    else                 // if n is odd, multiply by three and
        add one
        n = 3 * n + 1;
}
```

This is the **Collatz conjecture** (1937). Checked for all inputs up to $2^{68}$[1] — **nobody has proved it**.

A *conjecture* is a statement believed true but not yet proved. Testing $2^{68}$ inputs is still not a proof.

[1] Barina, *Convergence verification of the Collatz problem*, J. Supercomputing 77:2681–2688, 2021

# Reasoning About Programs Is Hard

Does this output `true` for some integer inputs?

```
x = input; y = input; z = input;
output: x*x*x + y*y*y + z*z*z == 42
```

Open from **1954 until 2019** — answered only after roughly a **million hours** of computing.[1]

$$(-80538738812075974)^3 + 80435758145817515^3 + 12602123297335631^3 = 42$$

**Yes** — but no test suite was ever going to stumble onto *that* input.

[1] Booker & Sutherland, *On a question of Mordell*, PNAS 118(11), 2021; solution for $k = 42$ found 2019 on Charity Engine

# Rice's Theorem

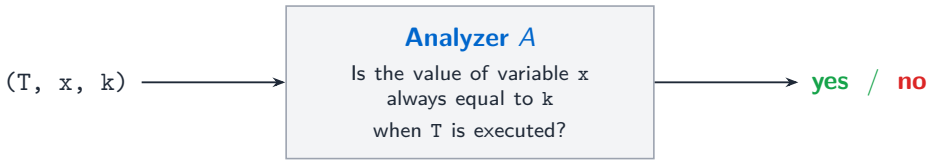
All **interesting** questions about the I/O behaviour of programs
(written in a **Turing-complete** language)
are **undecidable**.

- **Undecidable** = no algorithm can answer it correctly for *all* programs.
- **Turing-complete** = expressive enough to simulate any Turing machine. (C, Java, Python — essentially every language you use.)
- **Interesting** = non-trivial; true for some programs, false for others.

Rice, *Classes of Recursively Enumerable Sets and Their Decision Problems*, Trans. AMS 74:358–366, 1953

# A Constant-Value Analyzer

Suppose an analyzer $A$ exists that decides: *is variable* `x` *always equal to* `k` *whenever* `T` *is executed?*




Input: a program `T`, one of its variables `x`, a value `k`.
Output: a definite **yes** or **no** — always correct, always terminating.

**Claim: no such $A$ can exist.**

# The Reduction: $A$ Would Solve the Halting Problem

Feed $A$ this program, where `TM(j)` simulates the $j$-th Turing machine on empty input:

```
x = 17; if (TM(j)) x = 18;
```

- If machine $j$ **never halts**: `TM(j)` never returns, `x = 18` never runs $\rightarrow$ `x` is always **17**.
- If machine $j$ **halts**: `x` becomes **18** $\rightarrow$ `x` is not always 17.

`x` is constantly 17 $\iff$ machine $j$ does **not** halt

So $A$ decides the **halting problem** — proved **impossible** in 1937.
Contradiction. Therefore $A$ does not exist. ■

Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem*, Proc. LMS 42:230–265, 1937

# Approximate Answers

- Rice's theorem kills the **perfect** analyser — one that always answers **yes** or **no**, always correctly.
- It says **nothing** about analysers allowed to answer **maybe**.
- Impossible to decide a property for *any* program.
  Entirely possible to give **useful answers for most realistic programs**.
- An *ideal* analyser does not exist — so there is **always room** for a more precise approximation. That is why this field is still alive, and why you can do research in it.

SPA, §1.2

# Approximation Is Enough to Find Real Bugs

```c
int main(int argc, char *argv[]) {
  if (argc == 42) {
    char *p, *q;
    p = NULL;
    printf("%s", p);
    q = (char *)malloc(100);
    p = q;
    free(q);
    *p = 'x';
    p = (char *)malloc(100);
    p = (char *)malloc(100);
    q = p;
    strcat(p, q);
    assert(argc > 87);
  }
}
```

- `gcc -Wall` reports **no errors** at all.
- Every bug sits behind `if (argc == 42)` — testing reaches them only with **exactly 42 arguments**.
- Random testing has essentially **no chance** of generating that input.
- Yet *approximate* answers about null values, pointer targets and branch conditions catch **all six** — without running the program once.

After SPA, §1.3

# The Six Errors

1. `printf("%s", p)` with `p = NULL` — **null pointer dereference**.
2. `*p = 'x'` after `free(q)`, where `p == q` — **use-after-free** (dangling pointer write).
3. `p = malloc(100)` twice in a row — the first block becomes unreachable: **memory leak**.
4. `strcat(p, q)` reads **uninitialized memory** — `malloc` does not zero, so there is no terminating `\0`.
5. `strcat(p, q)` with `p == q` — overlapping source and destination is **undefined behaviour**, and it **overflows** the 100-byte buffer.
6. `assert(argc > 87)` inside `if (argc == 42)` — **always fails** when reached.

Each needs a different *approximate* fact: null values, pointer targets, allocation reachability, initialization, aliasing, branch conditions.

# Conservative Approximation

- An approximation is **conservative** (or **safe**) if all errors lean to the **same side**.
- Which side is "safe" is determined by the **intended application**.
- A "maybe" must never be silently reported as a definite answer.
- An analyser that errs *unpredictably* — sometimes missing bugs, sometimes inventing them — gives you **no guarantee at all**. One-sidedness is what makes an approximation worth having.

SPA, §1.3

# Soundness Depends on Your Goal

A program analyser is **sound** if it never gives incorrect results — though it may answer **maybe**.

### Verification tool

Sound = **never misses an error** of the kind it targets.

May produce **spurious warnings** (false positives).

*"If I say safe, it is safe."*

### Automated testing tool

Sound = **all reported errors are genuine**.

May **miss errors** (false negatives).

*"If I report it, it is real."*

Same word. **Opposite** meanings.

Always ask: sound *with respect to what*?

SPA, §1.3