Dynamic Analysis & Fuzzing

Sergey Mechtaev

mechtaev@pku.edu.cn

Peking University

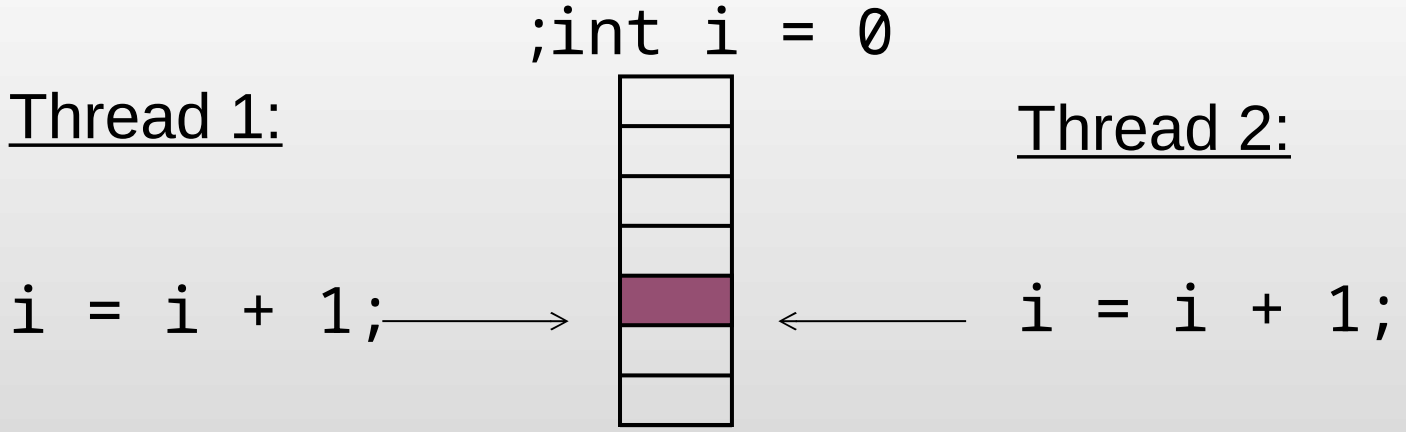
Dynamic Analysis

Definition. Dynamic program analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor.

- Detecting race conditions
- Detecting buffer overflows
- Detecting memory leaks
- Dynamic taint analysis

Race Conditions

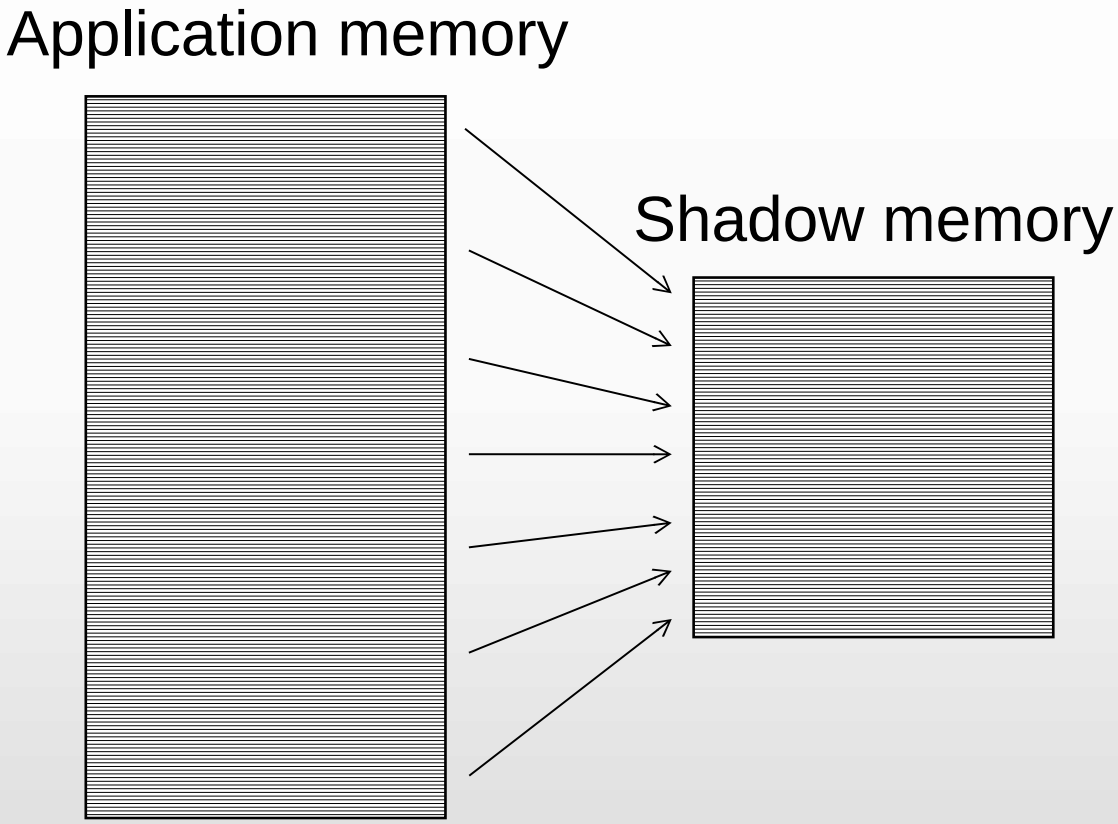
Definition. A race condition is the situation in which multiple threads or processes read and write a shared data item and the final result depends on the order of execution.



Race Condition Impact

- Endanger system robustness
  - Corrupted memory and state, violated data structure invariants
  - Inconsistent and/or wrong behavior, crashes
- Endanger system security
  - Time-of-check to time-of-use problem
- Unpredictable and non deterministic
- Real example
  - Therac-25 radiation therapy machine led to the death of three patients and injuries to several more in late ‘80s

Shadow Memory



Shadow memory is used to track memory accesses by threads during program execution

Happens-Before Relation

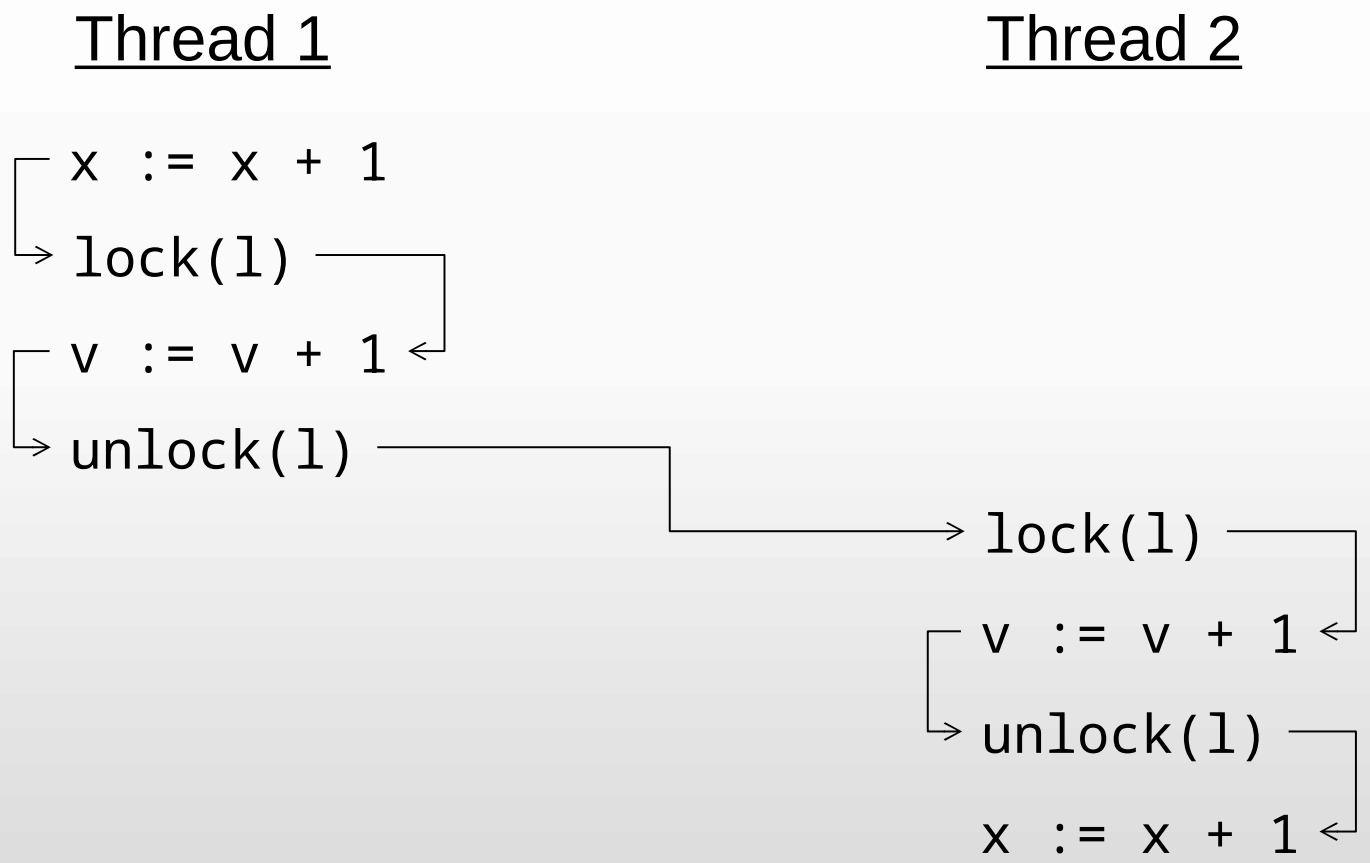
Given events A and B, A happens-before B iff

- A and B are executed by the same thread t and t executes A before B
- A and B are executed by different threads and A sends a message to B
  - A send may have a single receiver, multiple receivers or none
  - Send events: lock release, notify(), notifyAll(), start()
  - Receive evens: lock acquire, join(), wait();
- There exists event C such that A happens-before C and C happens-before B

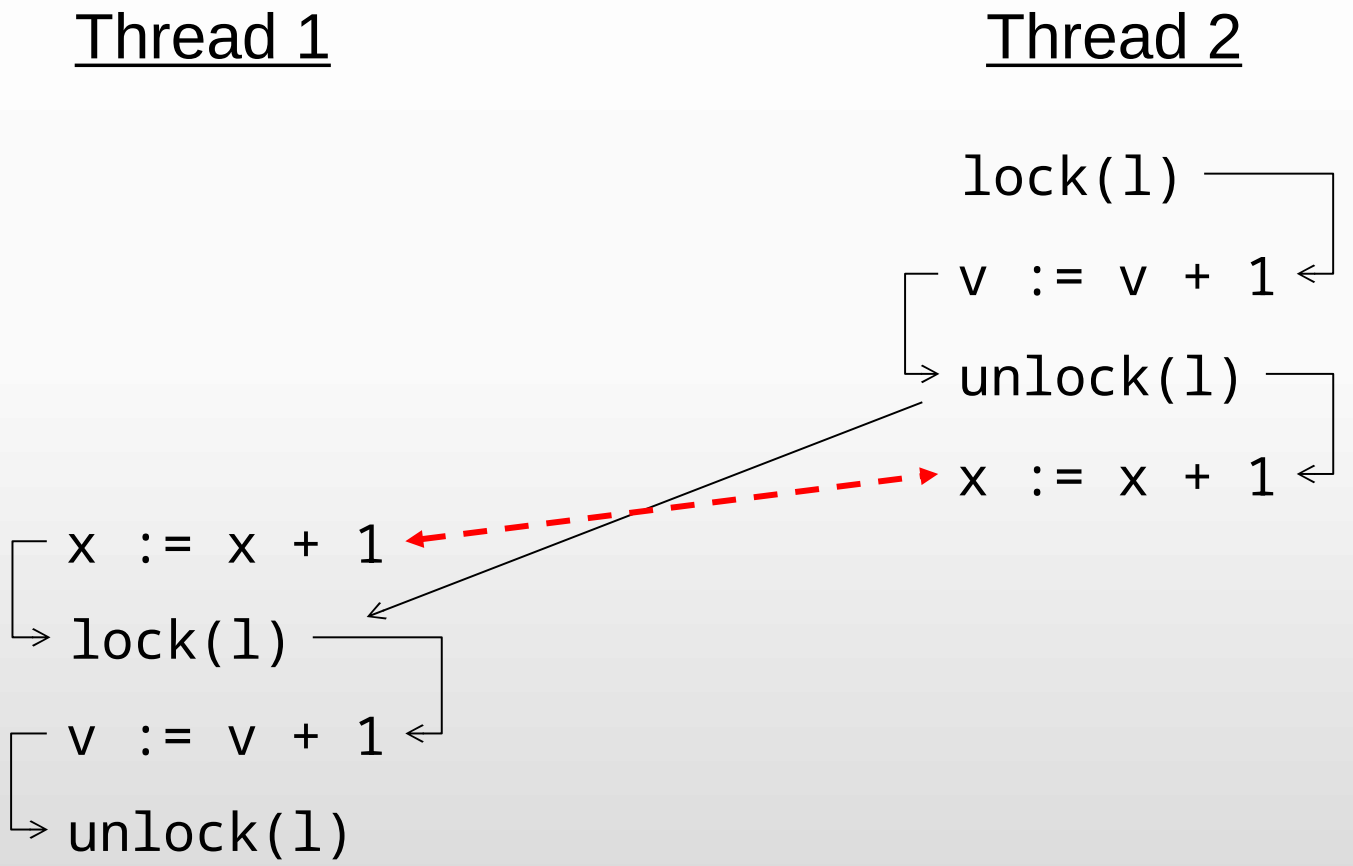
Race Condition Detection

- There are two concurrent events A and B (neither A happens-before B nor B happens-before A)
- A and B access the same memory word
- At least one of the two accesses is a write access

Race Condition Example



Race Condition Example

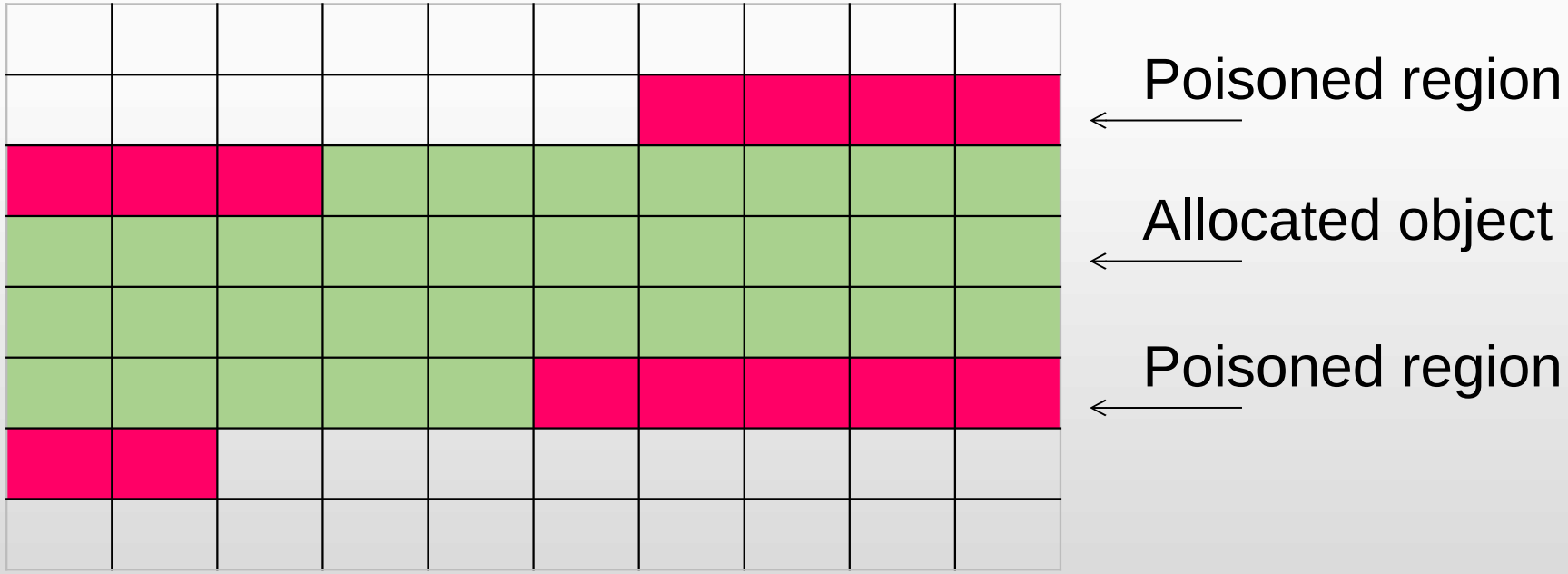


Memory Errors

- Use after free (dangling pointer dereference)
- Heap buffer overflow
- Stack buffer overflow
- Global buffer overflow
- Use after return
- Use after scope
- Initialization order bugs

Memory Errors Detection

- Poisoned memory around allocated blocks, and inside deallocated blocks



Memory Errors Detection

Use code instrumentation to check reads and writes of poisoned memory

- Before:

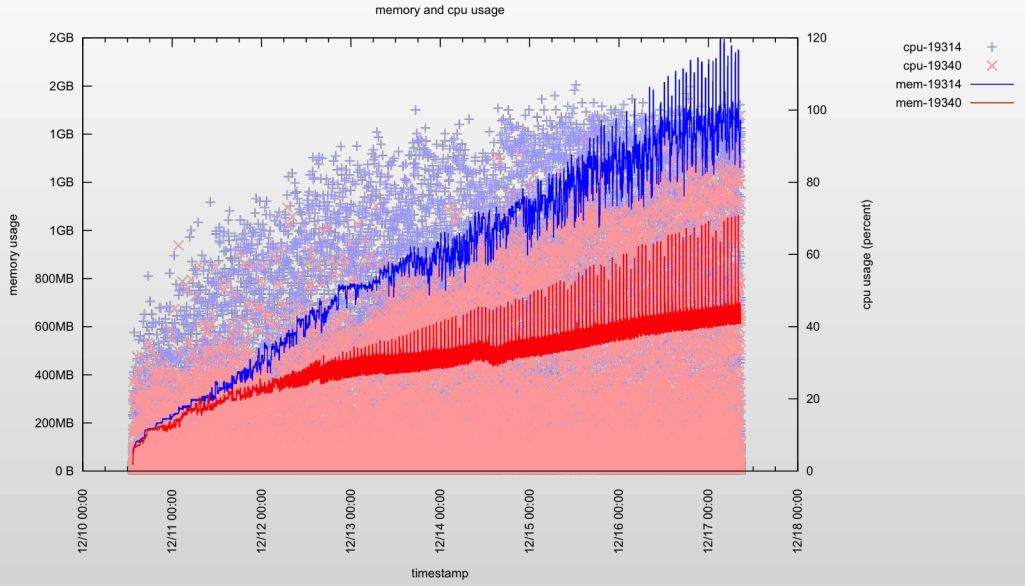
```
*address = ...;  // or: ... = *address;
```

- After:

```
if (IsPoisoned(address)) {
   ReportError(address, kAccessSize, kIsWrite);
 }
 *address = ...;  // or: ... = *address;
```

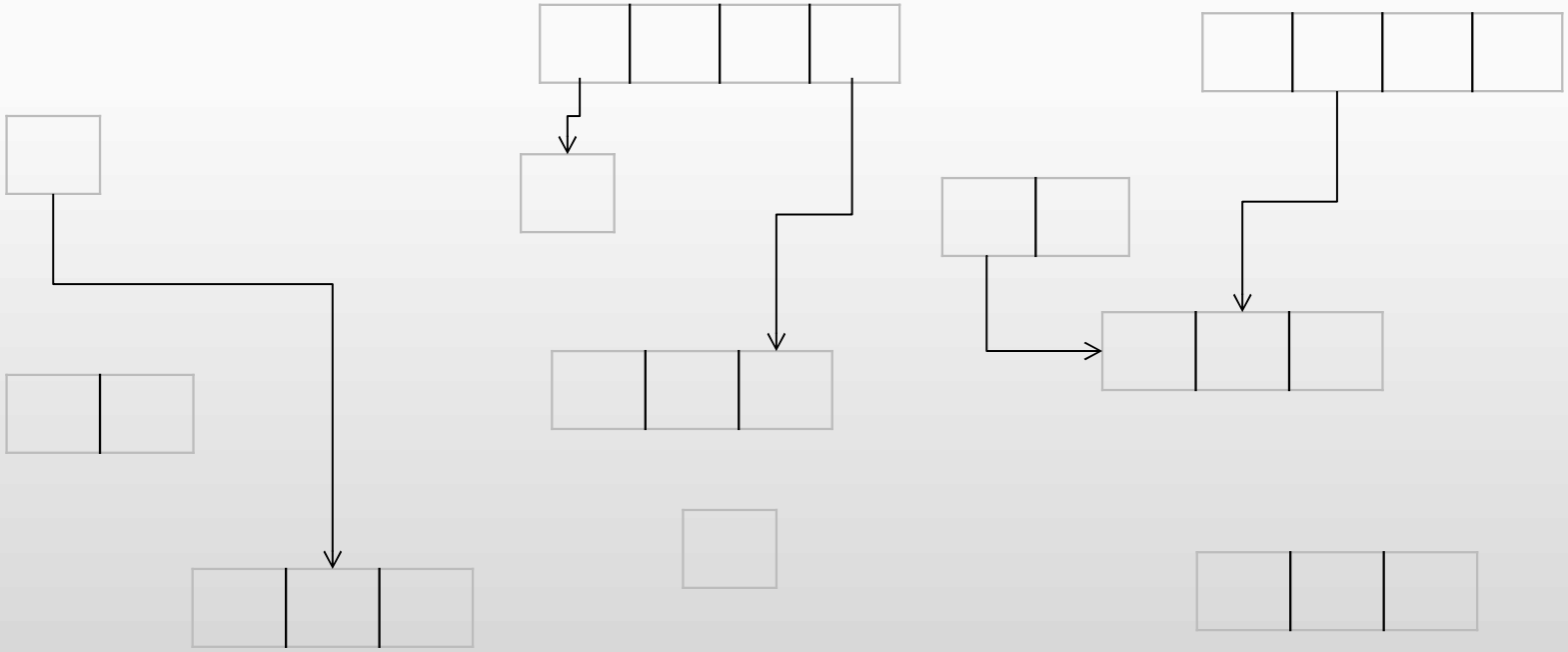
Memory Leaks

Definition. A type of resource leak that occurs when a computer program incorrectly manages memory allocations in such a way that memory which is no longer needed is not released.



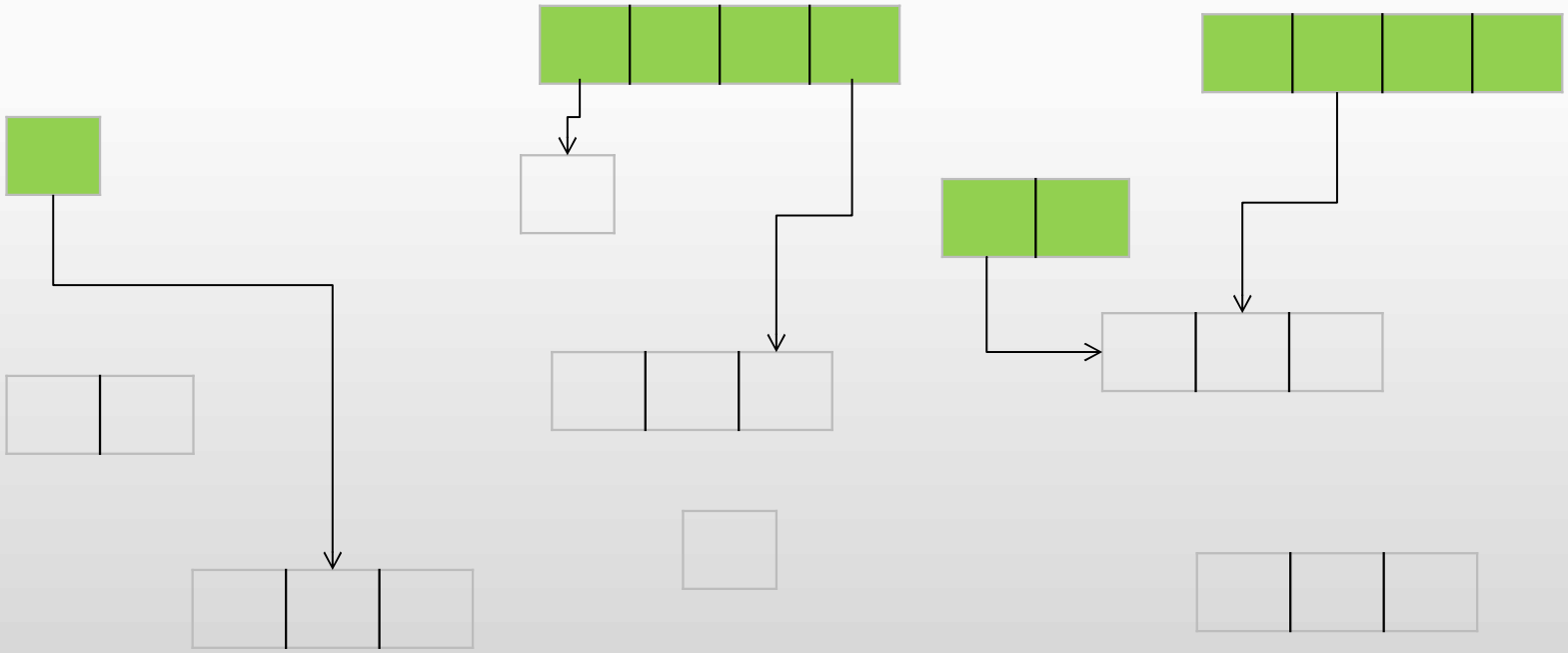
Memory Leak Detection

1. Execute the program and check memory at the end of the process' lifetime



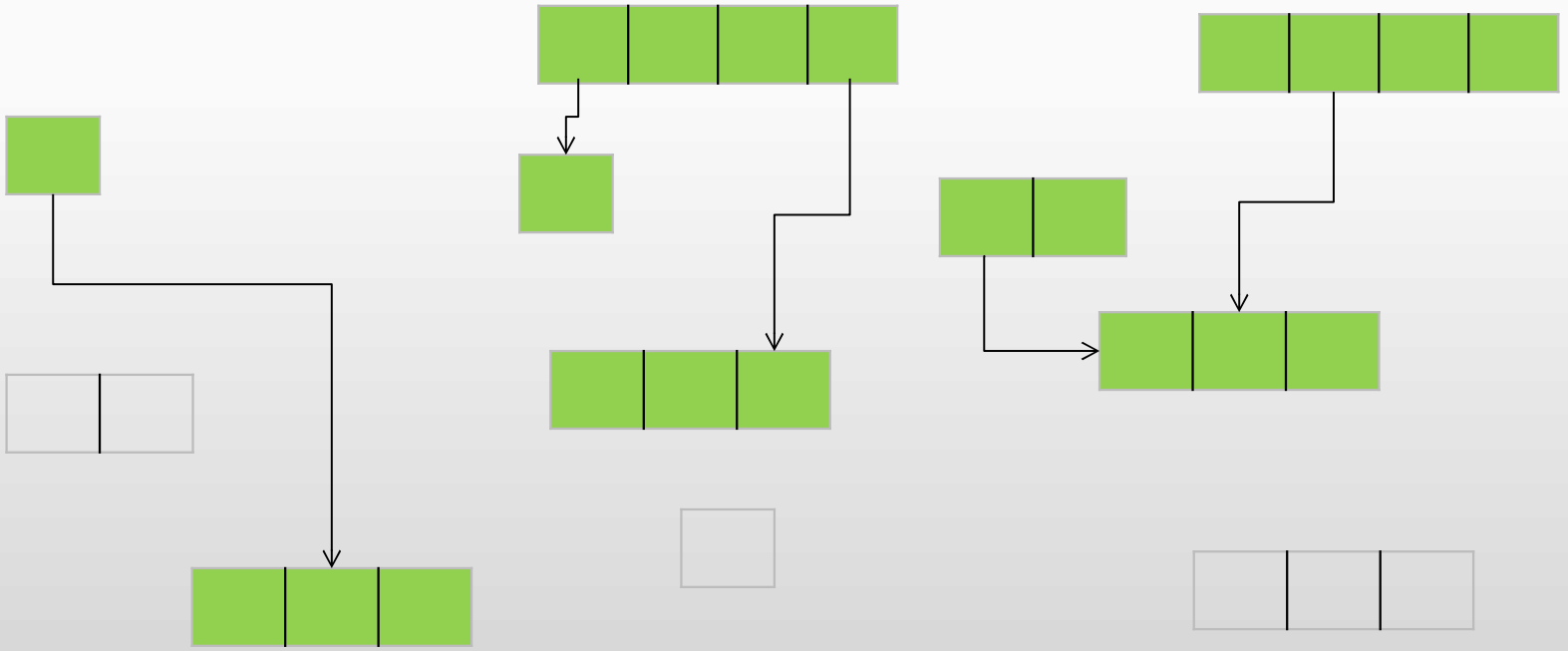
Memory Leak Detection

2. Obtain the root set of live memory (global variables, registers, etc)



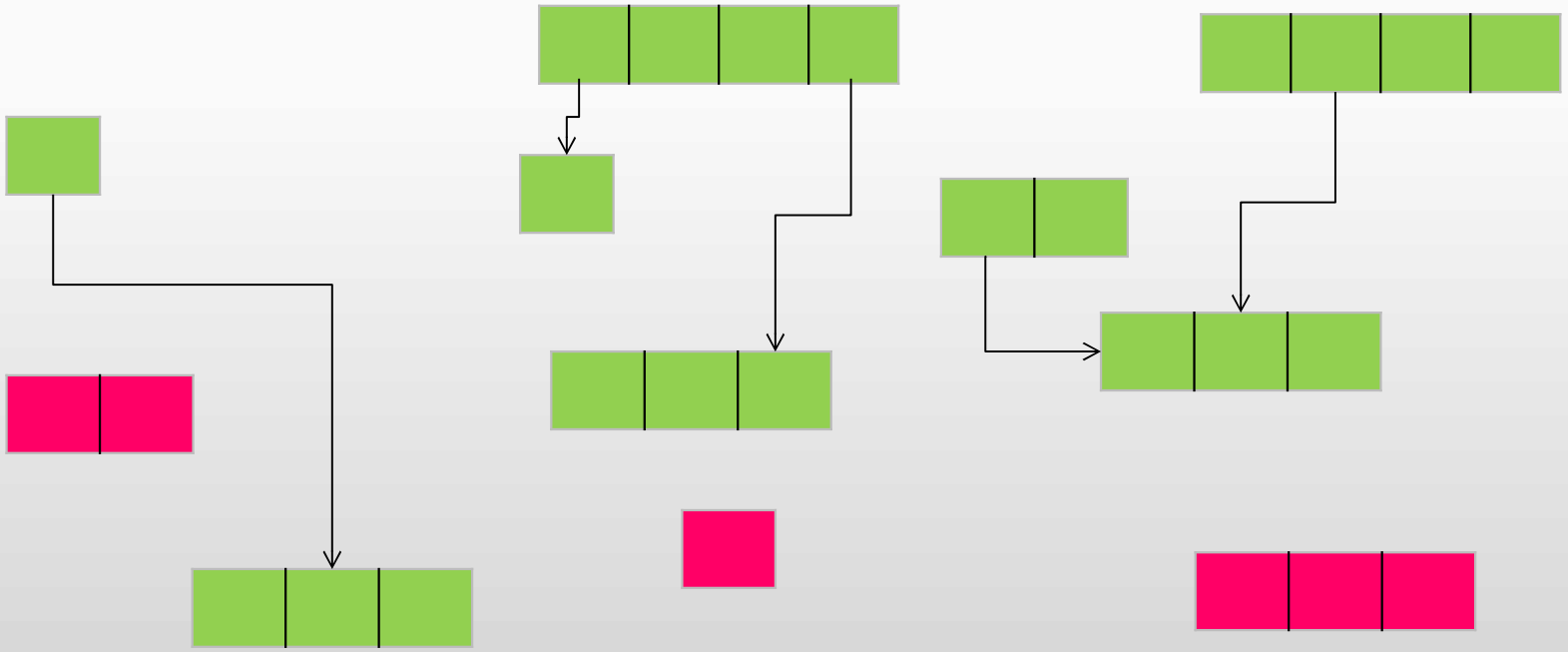
Memory Leak Detection

3. Scan memory to find all memory blocks reachable from the roots



Memory Leak Detection

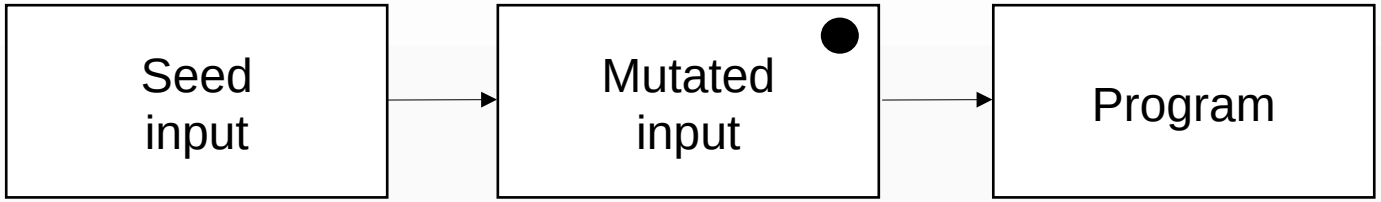
4. Report unreachable blocks as leaks



Blackbox Fuzzing

- Given a program simply feed random inputs and see whether it exhibits incorrect behavior (e.g., using dynamic analysis)
- + easy, low programmer cost
- - inefficient
  - Inputs often require structures, random inputs are mostly malformed
  - Inputs that trigger an incorrect behavior is a very small fraction, probably of getting lucky is very low

Mutation-based Fuzzing



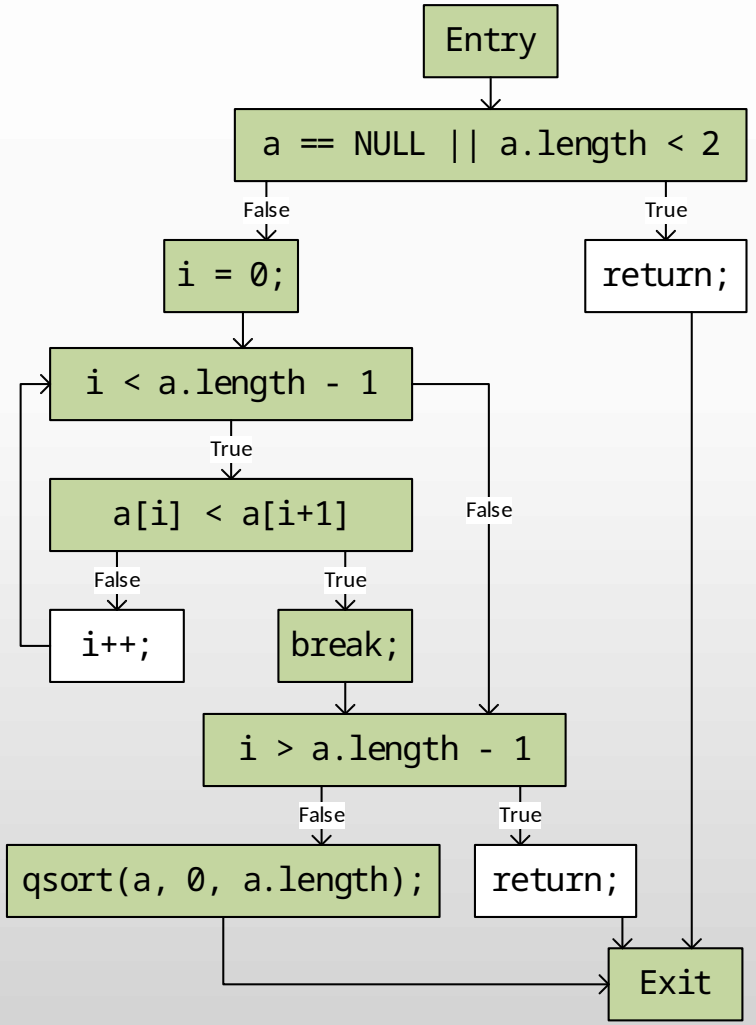
- Take a well-formed input, randomly perturb (flipping bit, etc.)
- Anomalies are added to existing valid inputs (either random or follow heuristics e.g., removing NULL, shift characters)
- + easy, improved efficiency
- - limited by initial corpus
- - mutated inputs are still often invalid

Code Coverage

Coverage for the input

a = [3, 7, 5]

Executes 7 out of 10 blocks, so statement coverage is 70%



Path ID (AFL)

For a given program P, assume there is an order of arcs in CFG: A1, A2, A3, …

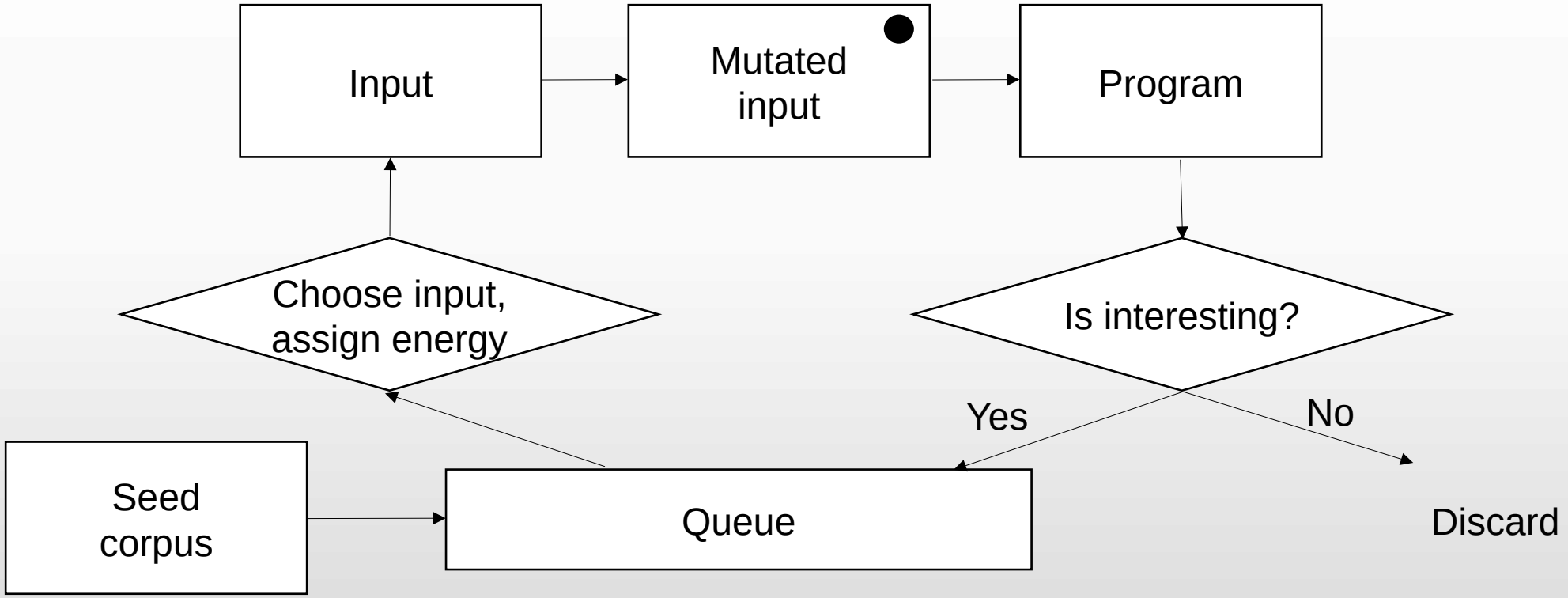
For a given input I, PathID(I) is a vector [N1, N2, N3, ...], where the Ni is the number of times Ai has executed by I.

If two inputs have different path IDs, then they follow different paths.

Coverage-guided Graybox Fuzzing

- Special type of mutation-based fuzzing
  - Run mutated inputs on instrumented program and compute Path IDs
  - Search for mutants that discover new Path IDs
  - Use metaheuristics such as genetic algorithms
- Examples: AFL, libfuzzer

Coverage-guided Graybox Fuzzing



Is Interesting? And Power Schedules

- Is Interesting?
  - Generally, checks if new paths are uncovered
  - AFL defines "is interesting" for path ID [N1, N2, N3, ...] if any of round(log(Ni)) is different from other paths.
- Power schedule
  - Energy determines how many time an input is fuzzed
  - Optimisation: assigns low energy to seeds exercising high-frequency paths and high energy to seeds exercising low-frequency paths [AFLFast, CSS 2016]