CS 569
Selected Topics in Software Engineering: Program Analysis & Evaluation

Random Testing

Oregon State University, Winter 2024

Random Testing (Fuzzing)

- Idea: feed random inputs to a program
- Observe whether it behaves “correctly”
  - Output is correct as per specification
  - Does not crash (universal test oracle)
- Special case of mutation analysis

The Infinite Monkey Theorem

Given enough time, a hypothetical monkey typing at random would, as part of its output, almost surely produce one of Shakespeare's plays (or any other text).

Source: https://en.wikipedia.org/wiki/Infinite_monkey_theorem

The First Fuzzing Study

- Conducted by Barton Miller at University of Wisconsin
- 1990: Command-line fuzzer, testing reliability of UNIX programs
  - Bombards utilities with random data
- 1995: Expanded to GUI-based applications (X Windows), network protocols, and system library APIs
- Later: Command-line and GUI-based Windows and OS X apps

Fuzzing UNIX Utilities: Aftermath

- 1990: Command-line fuzzer, testing reliability of 88 UNIX programs -> caused 25-33% of UNIX utility programs to crash or hang
- 1995: Expanded to GUI-based applications (X Windows), network protocols, and system library APIs -> systems got better but not much
- “Even worse is that many of the bugs reported in 1990 are still present in code releases of 1995”

A Silver Lining: Security Bugs

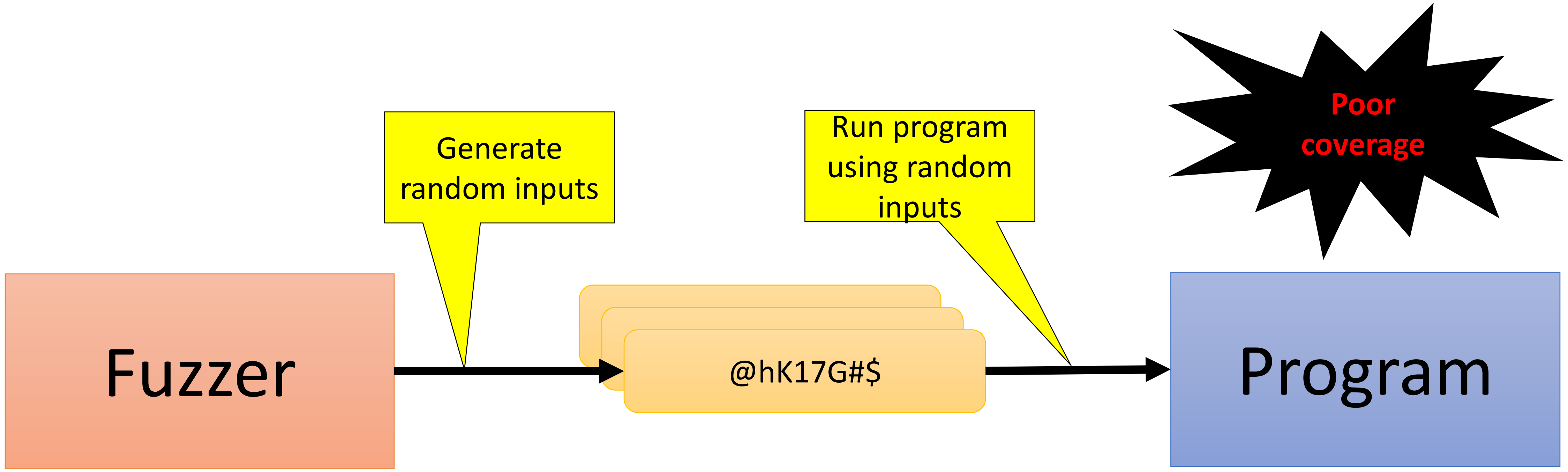
- gets() function in C has no parameter limiting input length

=> programmer must make assumptions about the structure of input

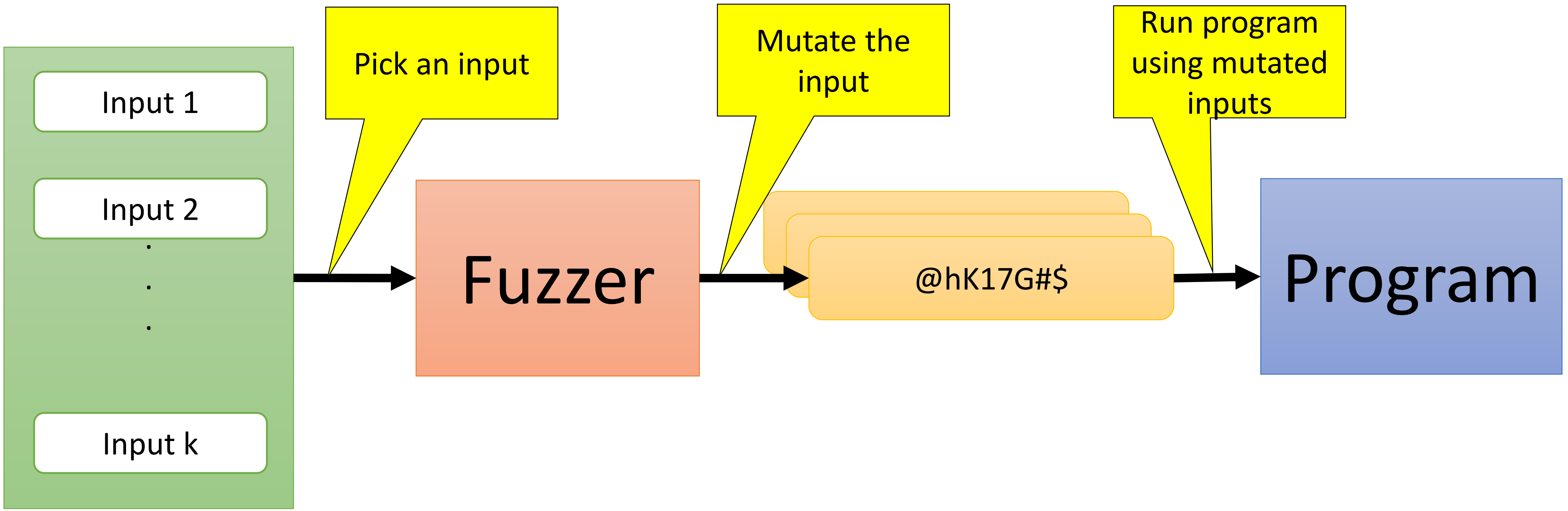
- Causes reliability issues and security breaches
  - Second most common cause of errors in 1995 study

- Solution: use fgets(), which includes argument to limit the maximum length of input data

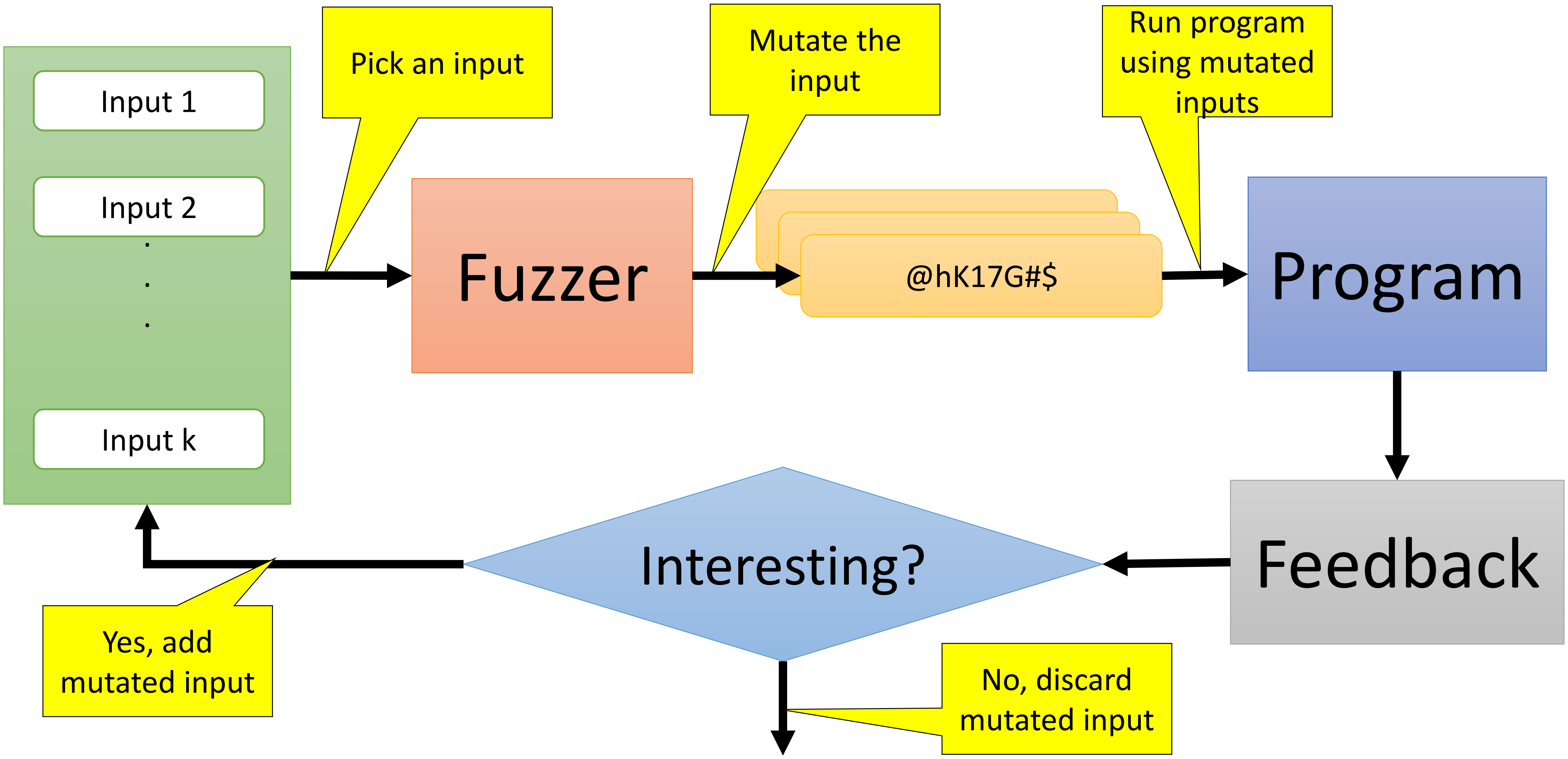
First Generation Fuzzer



Second Generation Fuzzer



Third Generation Fuzzer



What Kinds of Bugs can Fuzzing Find?

- Memory errors
  - Spatial (out-of-bound access) and Temporal (use-after-free)
- Other undefined behaviors
  - Integer overflow, null dereference, divide by zero, uninitialized read, …
- Assertion violations
- Infinite loops (using timeout)
- Concurrency bugs
  - Data races, deadlocks, …

Random Testing: Pros and Cons

- Pros
  - Easy to implement
  - Provably good coverage given enough tests
  - Can work with programs of any format
  - Appealing to find security vulnerabilities
- Cons
  - Inefficient test suite
  - Might find bugs that are unimportant
  - Low code coverage in practice

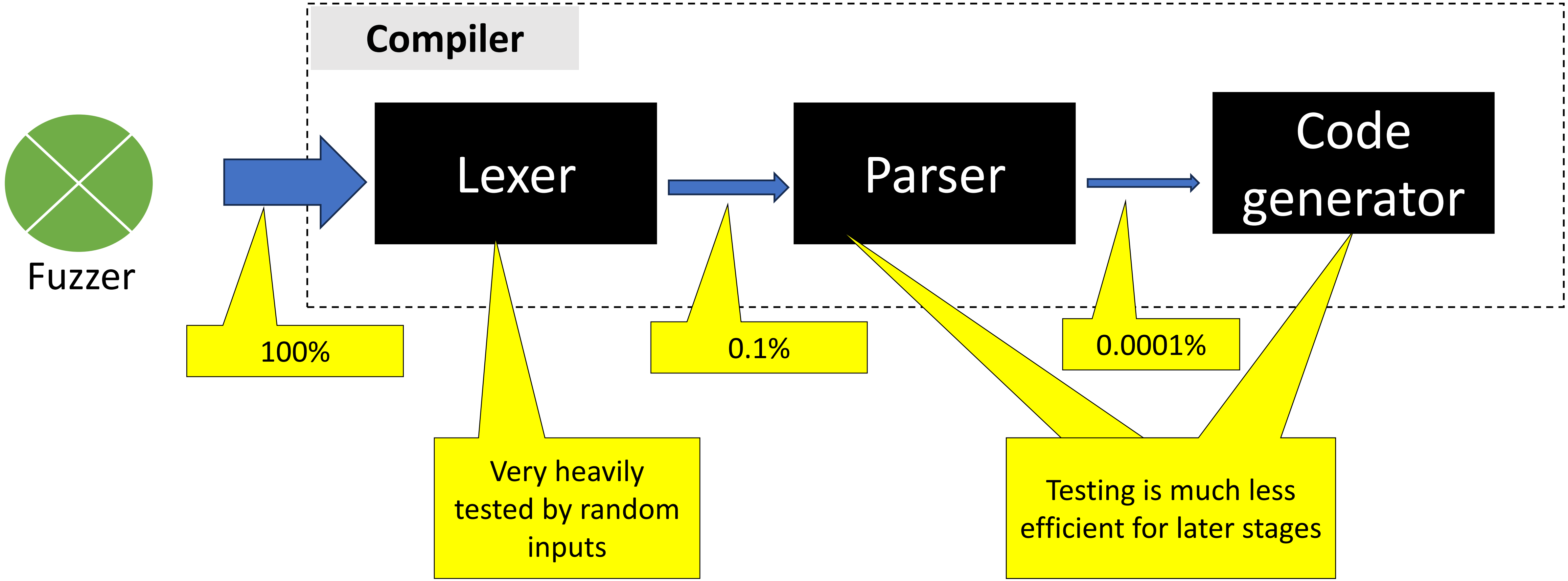
Uneven Code Coverage: Example 1

Hit by 5M+ inputs

Hit by 0 inputs!

```
void test_me(int x)
{
    int y = x + 3;
    if (y == 13){
        ERROR;
    }
}
```

Uneven Code Coverage: Example 2



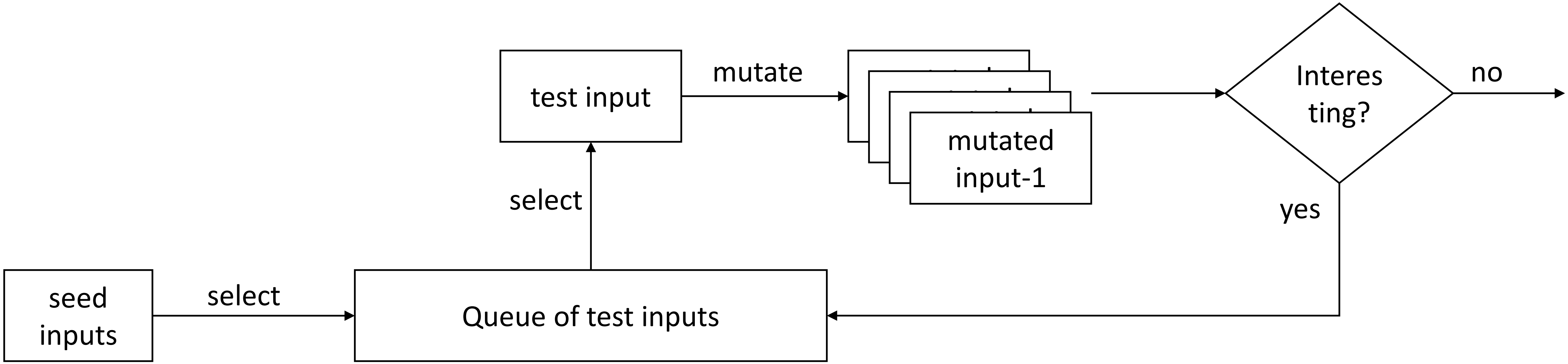
American Fuzzy Lop: AFL

- Gray-box Fuzzing
- Guide input generation toward a goal
  - Guidance based on lightweight program analysis
- Three main steps
  - Randomly generate inputs
  - Get feedback from test executions
    - E.g., what code is covered?
  - Mutate inputs that have covered new code

American Fuzzy Lop: AFL

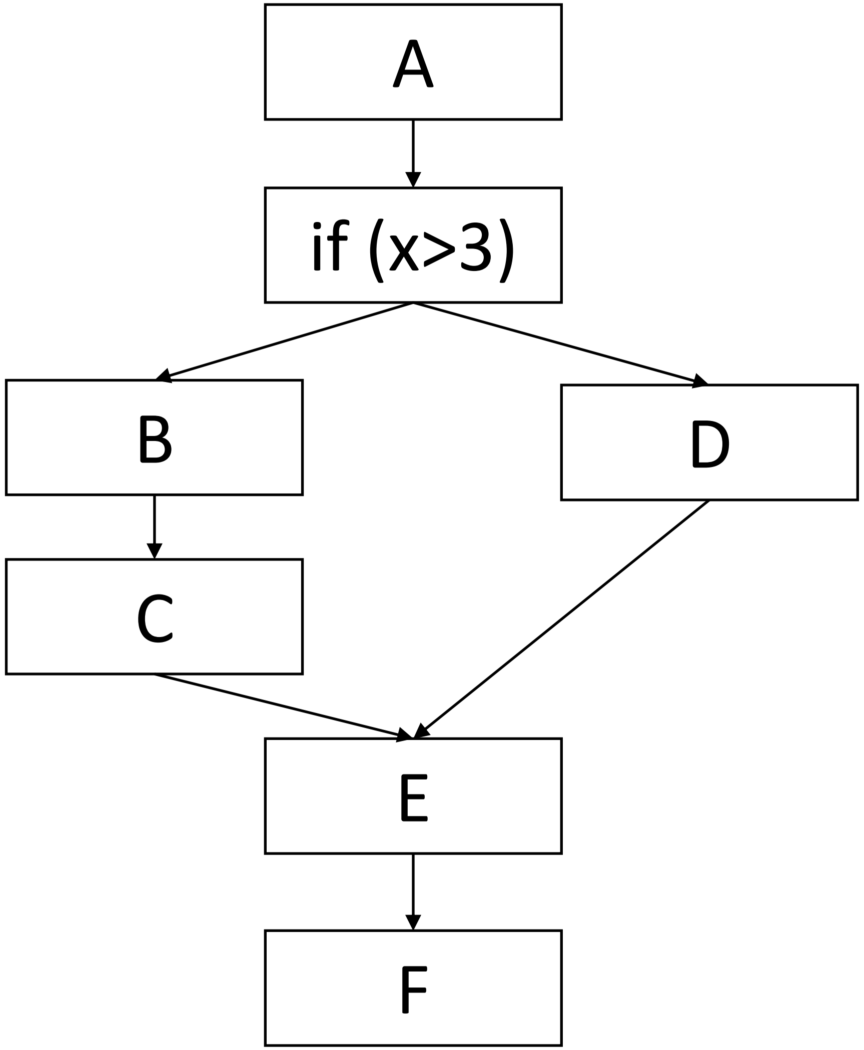
- Simple yet effective fuzzing tool
  - Targets C/C++ programs
  - Inputs are e.g., files read by the program
- Widely used in industry
  - Maintained by Google
  - Find security-related bugs
  - Has found many bugs in OpenSSL, PhP, Firefox, etc.

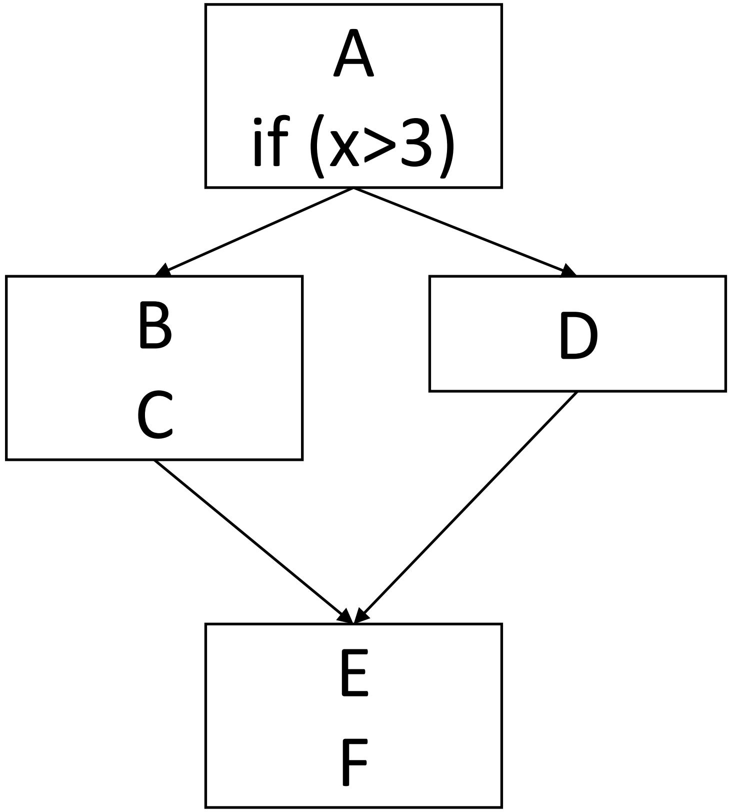
American Fuzzy Lop: AFL Overview



Control Flow Graphs – Basic Blocks

```
A
if (x>3){
      B
      C
}else{
      D
}
      E
      F
```





AFL – Measuring Coverage

- AFL uses Branch coverage
  - Branches between basic blocks
- Rationale: reaching a code location may not be enough to trigger a bug, but state also matters
- Tradeoff between
  - Effort spent on measuring coverage
  - Guidance it provides to the fuzzer

AFL – Example

Sequence of basic blocks that are executed

Branches covered (i.e. edges in CFG)

AFL – Example

Sequence of basic blocks that are executed

A -> B -> C -> D -> E

Branches covered (i.e. edges in CFG)

AB, BC, CD, DE

AFL – Example

Sequence of basic blocks that are executed	Branches covered (i.e. edges in CFG)
A -> B -> C -> D -> E	AB, BC, CD, DE
A -> B -> D -> C -> E	AB, BD, DC, CE

Using branch coverage (instead of line coverage) benefits AFL

AFL – Efficient Implementation

- AFL instruments all branching points:

```
cur_location = /*RANDOMLY GENERATED AT COMPILE TIME*/;

shared_mem[cur_location ^ prev_location]++;

prev_location = cur_location >> 1;
```

Works well with separately compiled components

Combine current and prev block into a fixed size hash and increment the number of times it is seen

To distinguish between A->B and B->A

AFL – Detecting New Behaviors

- Inputs that trigger a new edge in the CFG: considered as new behavior
- Alternative: Consider new paths
  - More expensive to track
  - Path explosion problem

AFL – Example

Sequence of basic blocks that are executed	**Branches covered (i.e. edges in CFG)**
Exec1: A -> B -> C -> D -> E	AB, BC, CD, DE (new => keep input)

AFL – Example

Sequence of basic blocks that are executed	**Branches covered (i.e. edges in CFG)**
Exec1: A -> B -> C -> D -> E	AB, BC, CD, DE (new => keep input)
Exec2: A -> B -> C -> A -> E	AB, BC, CA, AE (new => keep input)

AFL – Example

Sequence of basic blocks that are executed	**Branches covered (i.e. edges in CFG)**
Exec1: A -> B -> C -> D -> E	AB, BC, CD, DE (new => keep input)
Exec2: A -> B -> C -> A -> E	AB, BC, CA, AE (new => keep input)
Exec3: A -> B -> C -> A -> B -> C -> D -> E	AB, BC, CA, AB, BC, CD, DE (not new => discard input)

AFL – Example

Sequence of basic blocks that are executed	**Branches covered (i.e. edges in CFG)**
Exec1: A -> B -> C -> D -> E	AB, BC, CD, DE (new => keep input)
Exec2: A -> B -> C -> A -> E	AB, BC, CA, AE (new => keep input)
Exec3: A -> B -> C -> A -> B -> C -> D -> E	AB, BC, CA, AB, BC, CD, DE (not new => discard input)

Indicates a loop. What if new behavior occurs after k iterations of the loop?

AFL Refinement – Edge Hit Counts

- Refinement of the previous definition of “new behaviors”
- For each edge, count how often it is taken:
  - Approximate counts based on buckets of increasing size
    - 1, 2, 3, 4-7, 8-15, 16-31, etc.
  - Rationale: focus on relevant differences in hit counts

AFL – Evolving the Input Queue

- Maintain queue of inputs
  - Initially: Seed inputs provided by the user
  - Once used, keep input if it covers a new edge
  - Add new inputs by mutating existing input
- In practice: Queue sizes of 1k to 10k

AFL – Mutation Operators

- Goal: Create new inputs from existing inputs
- Random transformations of bytes in an existing input
  - Bit flips with varying lengths and stepovers
  - Addition and Subtraction of small integers
  - Insertion of known interesting integers
    - E.g., 0, 1, INT_MAX, INT_MIN
  - Splicing of different inputs

AFL – More Tricks for Fast Fuzzing

- Time and memory limits
  - Discard input when execution is too expensive
- Pruning the queue
  - Periodically select subset of inputs that still cover every edge seen so far
- Prioritize how many mutants to generate from an input in the queue
  - E.g., focus on unusual paths or try to reach specific locations

AFL – Real World Impact

- Open-source tool maintained mostly by Google
  - Initially created by a single developers
  - Various improvements proposed in academia and industry
- Fuzzers regularly check various security-critical components
  - Many thousands of compute hours
  - Hundreds of detected vulnerabilities

LibFuzzer

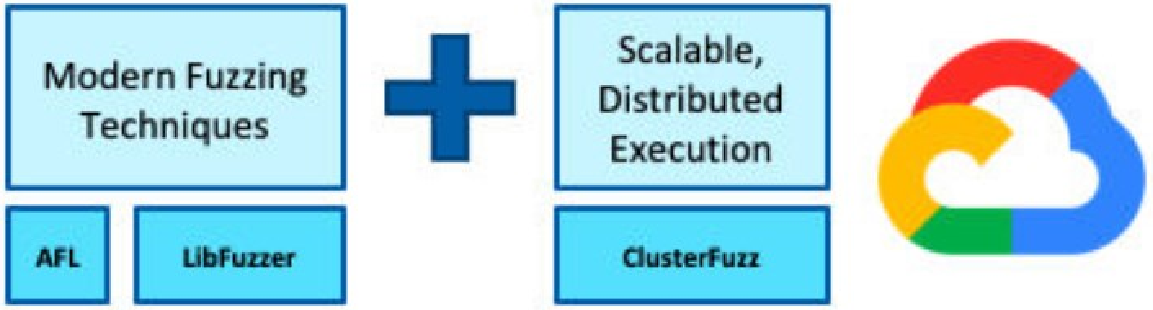
- Motivation: enable fuzzing libraries (software components) instead of whole application
- User provides fuzzing entry points called fuzz targets
- Intuition: if program has X lines of code and Y fuzz targets, then fuzzer has to cover only X/Y lines of code, on average per target.

LibFuzzer

- Motivation: enable fuzzing libraries (software components) instead of whole application
- User provides fuzzing entry points called fuzz targets
- Intuition: if program has X lines of code and Y fuzz targets, then fuzzer has to cover only X/Y lines of code, on average per target.

OSS-Fuzz

- Continuous fuzzing infrastructure hosted on Google cloud platform



- OSS-Fuzz has discovered over 17,400 bugs from 2016 – 2019 in many large projects (openssl, llvm, postgresql, git, firefox, …)

ClusterFuzz

- Google's scalable fuzzing infrastructure used to fuzz Chrome browser
- As of Jan 2019, it has found ~16,000 bugs in Chrome and ~11,000 bugs in over 160 projects integrated with OSS-Fuzz
- Features:
  - Highly scalable (runs over 1000s of machines)
  - Automatically tracks bugs found by underlying Fuzzers
  - Test case minimization
  - Statistics to analyze fuzzers performance
  - Easy to use web interface for reports

Domain-Specific Fuzzing

- Random testing is a paradigm as opposed to a technique
- Two Case Studies:
  - Mobile Apps: Google's Monkey Tool for Android Apps
  - Concurrent programs: Microsoft's Cuzz Tool

Testing Mobile Apps

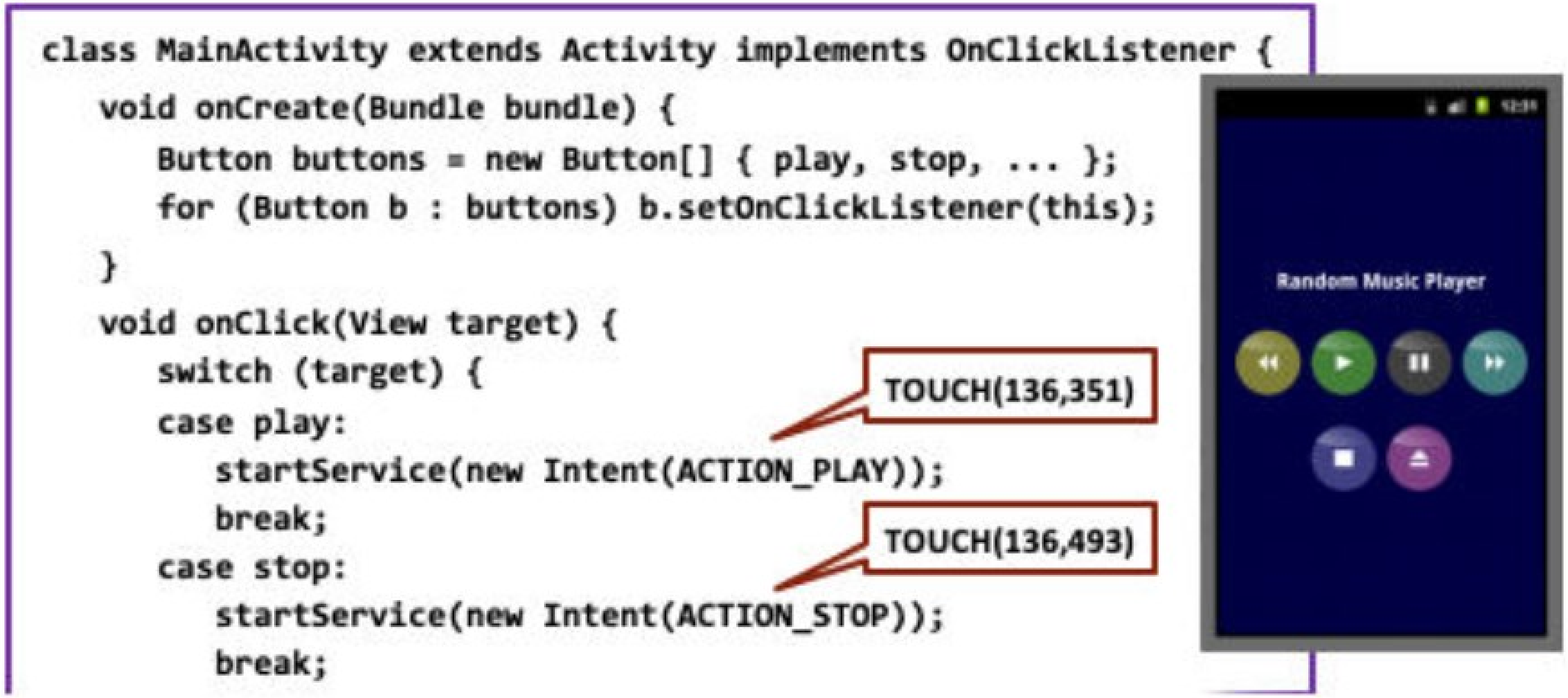
Android Framework Interface

```
class MainActivity extends Activity implements OnClickListener {
    void onCreate(Bundle bundle) {
        Button buttons = new Button[] { play, stop, ... };
        for (Button b : buttons) b.setOnClickListener(this);
    }
    void onClick(View target) {
        switch (target) {
        case play:
            startService(new Intent(ACTION_PLAY));
            break;
        case stop:
            startService(new Intent(ACTION_STOP));
            break;
        ...
    }
```

Function called by Android framework whenever user taps on any button

Music player App

Testing Mobile Apps



- TOUCH(x,y) where x and y are randomly generated: x in [0,480] and y in [0,800]

Testing Mobile Apps

```
class MainActivity extends Activity implements OnClickListener {
  void onCreate(Bundle bundle) {
    Button buttons = new Button[] { play, stop, ... };
    for (Button b : buttons) b.setOnClickListener(this);
  }
  void onClick(View target) {
    switch (target) {
    case play:
      startService(new Intent(ACTION_PLAY));
      break;
    case stop:
      startService(new Intent(ACTION_STOP));
      break;
```

TOUCH(136,351)

TOUCH(136,493)

- TOUCH(x,y) where x and y are randomly generated: x in [0,480] and y in [0,800]
- Monkey tool can generate many other kinds of input events (E.g., key press on keyboard, input from devices’ trackball, incoming phone call, change GPS location, …

Testing Mobile Apps – Sequence of Events

Click on Eject button to select the audio file or see the default one

TOUCH(x1, y1) TOUCH(x2, y2) TOUCH(x3, y3)

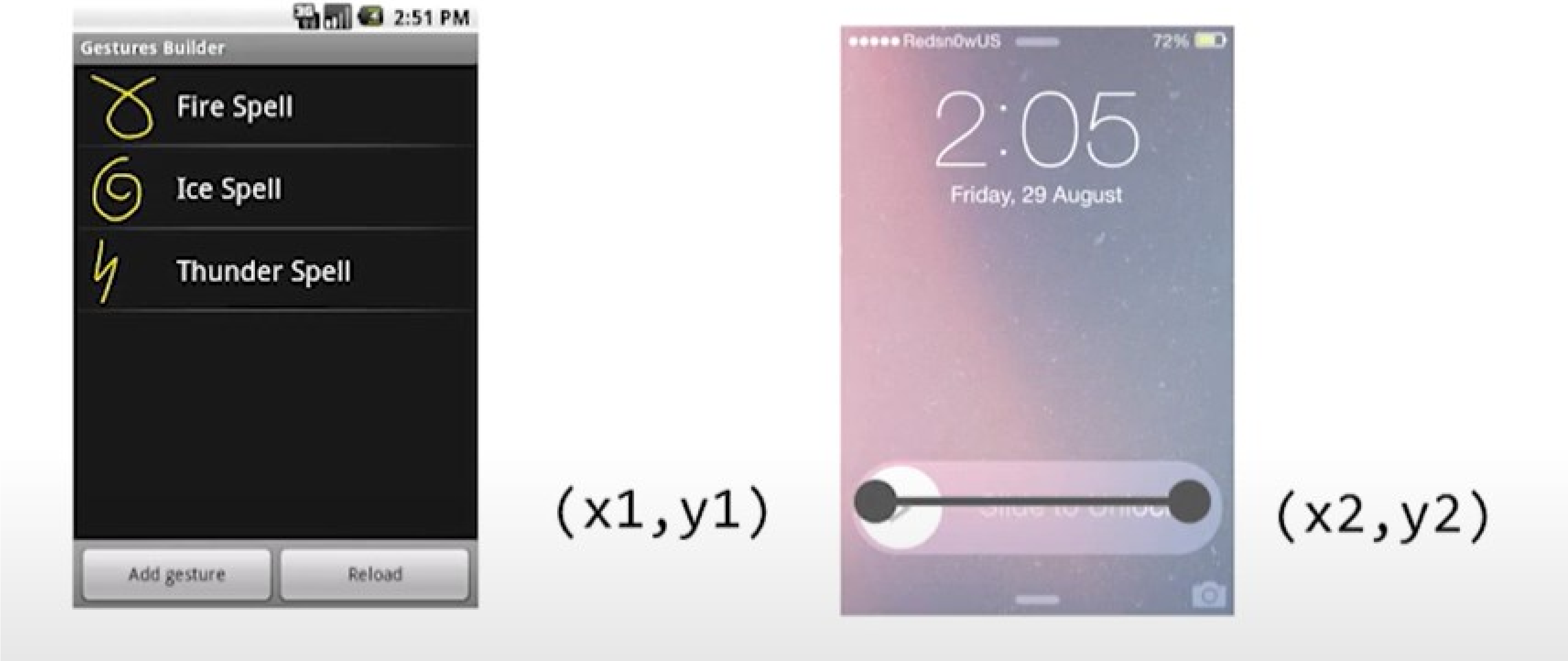
Stop playing the audio file

Dialog box asking user to select/view audio file and play it

Testing Mobile Apps – Generating Gestures

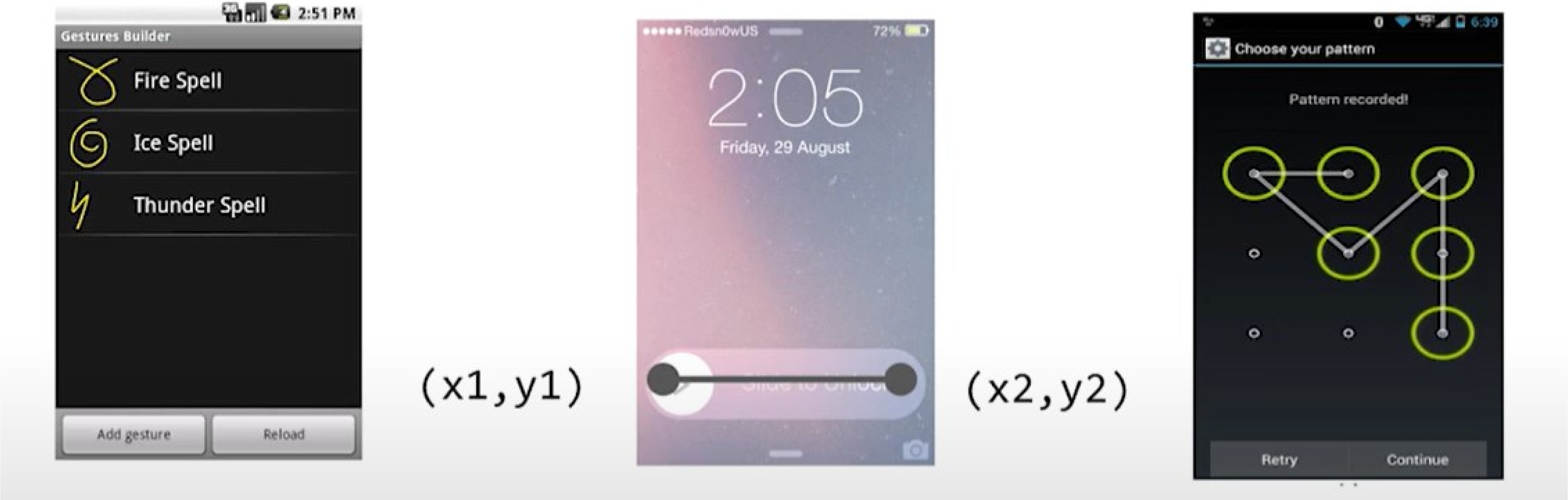
Testing Mobile Apps – Generating Gestures

DOWN(x1,y1) MOVE(x2,y2) UP(x2,y2)



Testing Mobile Apps – Generating Gestures

DOWN(x1,y1)   MOVE(x2,y2)   UP(x2,y2)



Grammar-Based Fuzzing

- Recall: Context Free Grammar
  - Terminals
  - Non-terminals
  - Production rules
  - Start symbol
- Advantages of specifying program inputs using context-free grammars
  - Systematic and Efficient Test Generation
  - Expressive enough to handle complex input formats (e.g., XML, JSON)
  - Provides a basis to fuzz wide range of software components:
    - configurations, APIs, GUIs, protocols, simulations, etc.

Grammar of Monkey Events

$$
\begin{aligned}
\mathit{test\_case} &:= \mathit{event}\,^{*} \\
\mathit{event} &:= \mathit{action}\,(\,x\,,\,y\,)\;|\;\ldots \\
\mathit{action} &:= \mathbf{DOWN}\;|\;\mathbf{MOVE}\;|\;\mathbf{UP} \\
x &:= 0\;|\;1\;|\;\ldots\;|\;\mathtt{x\_limit} \\
y &:= 0\;|\;1\;|\;\ldots\;|\;\mathtt{y\_limit}
\end{aligned}
$$

Quiz(1/2): Monkey Events

Use Monkey Grammer to write the event sequences for the specifications

$$
\begin{aligned}
test\_case &:= event\ * \\
event &:= action\ (\ x\ ,\ y\ )\ \mid\ \ldots \\
action &:= \mathbf{DOWN}\ \mid\ \mathbf{MOVE}\ \mid\ \mathbf{UP} \\
x &:= \mathtt{0}\ \mid\ \mathtt{1}\ \mid\ \ldots\ \mid\ \mathtt{x\_limit} \\
y &:= \mathtt{0}\ \mid\ \mathtt{1}\ \mid\ \ldots\ \mid\ \mathtt{y\_limit}
\end{aligned}
$$

Give the specification of a TOUCH event at pixel (80,215).

TOUCH events are a pair of DOWN and UP events at a single place on the screen.

Quiz(2/2): Monkey Events

Use Monkey Grammer to write the event sequences for the specifications

```
test_case := event *
    event := action ( x , y ) | ...
   action := DOWN | MOVE | UP
        x := 0 | 1 | ... | x_limit
        y := 0 | 1 | ... | y_limit
```

Give the specification of a TOUCH event at pixel (80,215).

```
DOWN(80,215) UP(80,215)
```

TOUCH events are a pair of DOWN and UP events at a single place on the screen.

Give the specification of a MOTION event from pixel (80,215) to pixel (80,100) to pixel (370,100).

MOTION events consist of a DOWN event somewhere on the screen, a sequence of MOVE events, and an UP event.

Quiz(2/2): Monkey Events

Use Monkey Grammer to write the event sequences for the specifications

```
test_case  :=  event *
    event  :=  action ( x , y ) | ...
   action  :=  DOWN | MOVE | UP
        x  :=  0 | 1 | ... | x_limit
        y  :=  0 | 1 | ... | y_limit
```

Give the specification of a TOUCH event at pixel (80,215).

```
DOWN(80,215) UP(80,215)
```

TOUCH events are a pair of DOWN and UP events at a single place on the screen.

Give the specification of a MOTION event from pixel (80,215) to pixel (80,100) to pixel (370,100).

```
DOWN(80,215) MOVE(80,100)
MOVE(370,100) UP(370,100)
```

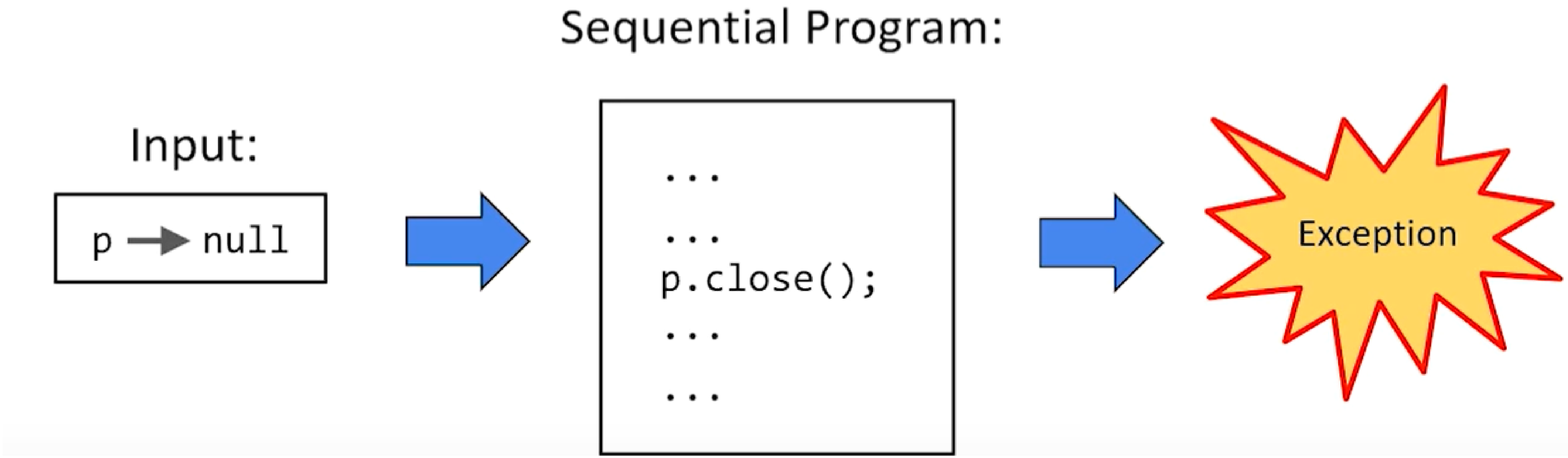
MOTION events consist of a DOWN event somewhere on the screen, a sequence of MOVE events, and an UP event.

Testing Concurrent Programs

Sequential Program:

```
...
...
p.close();
...
...
```

Testing Concurrent Programs



Testing Concurrent Programs

Input:

```
p → new File()
```

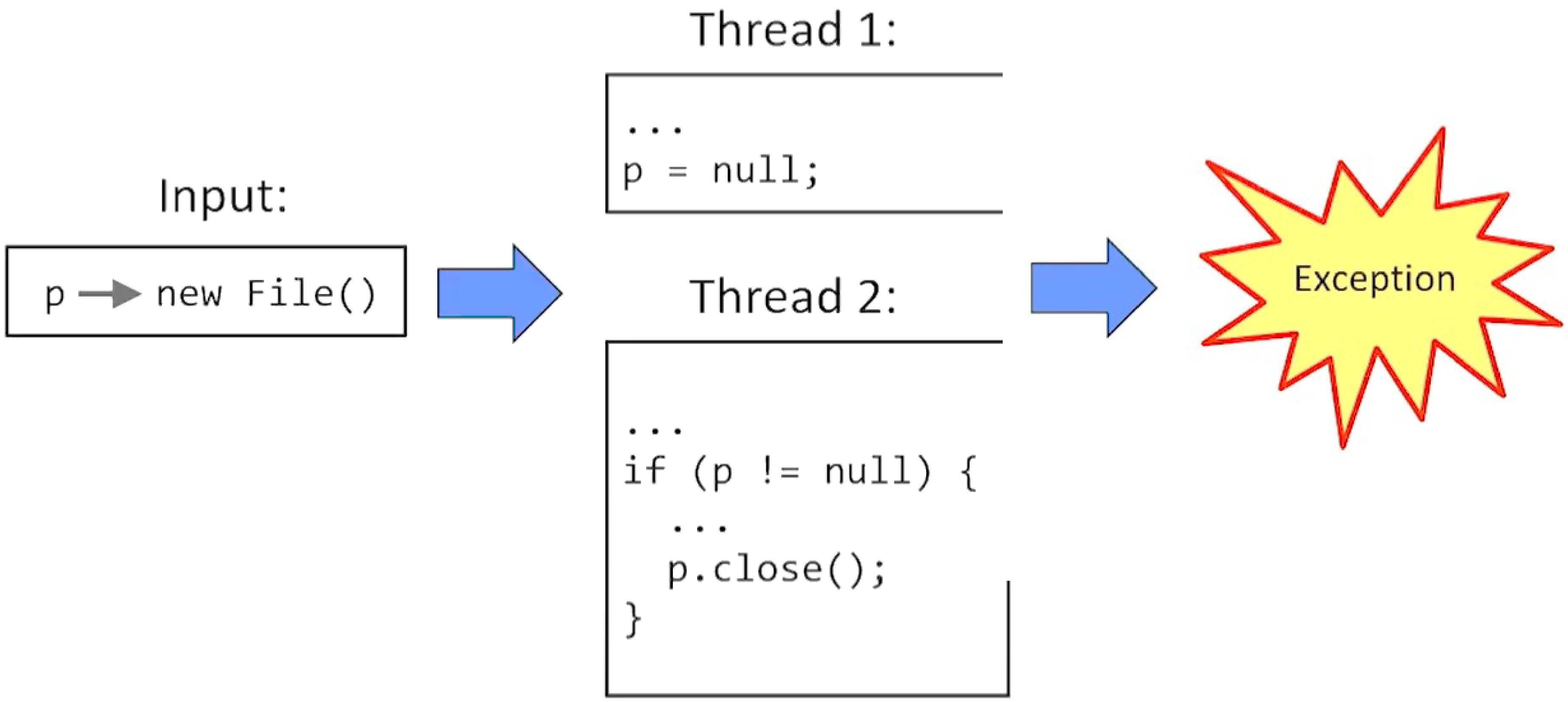
Thread 1:

```
...
p = null;
```

Thread 2:

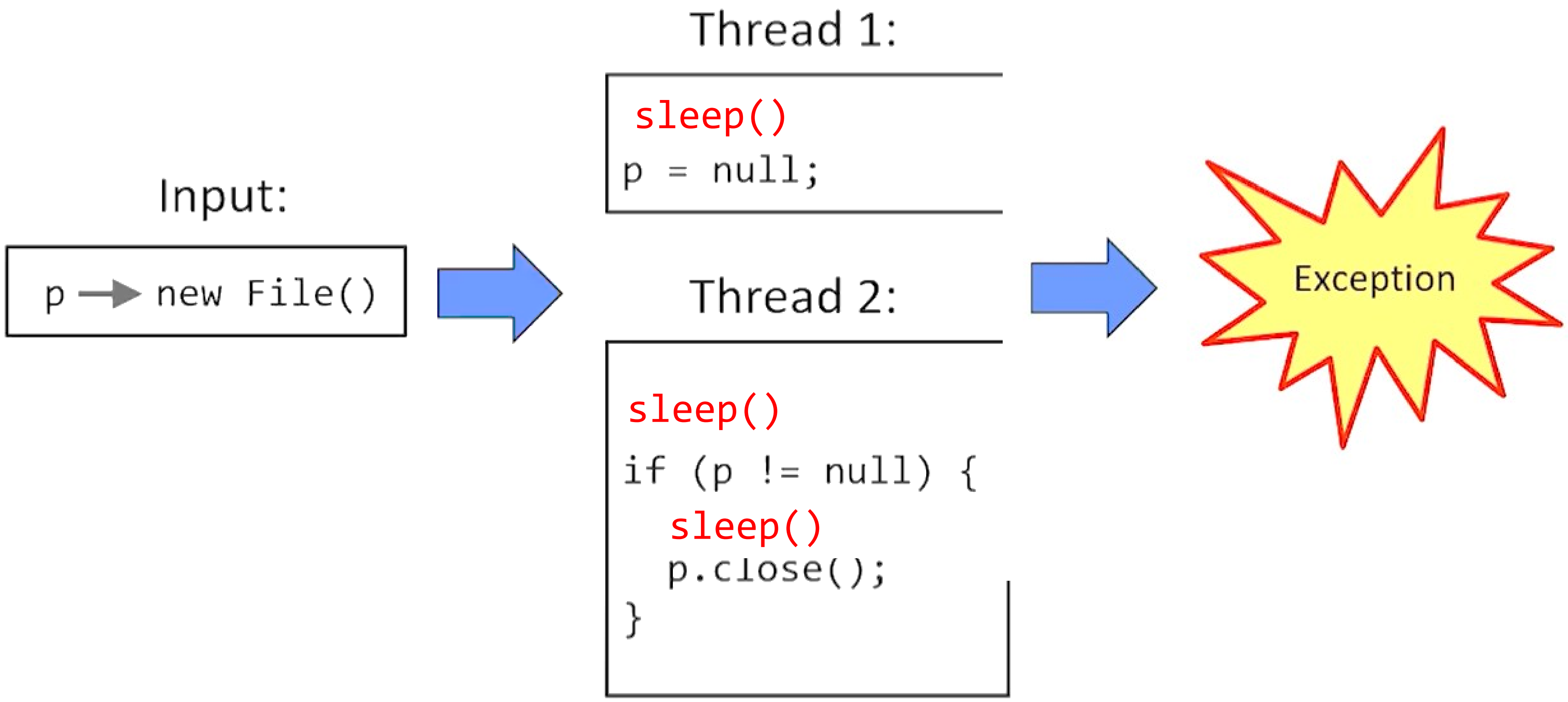
```
...
if (p != null) {
  ...
  p.close();
}
```

Testing Concurrent Programs



T2:S1 -> T1:S1 -> T2:S2

Testing Concurrent Programs



Cuzz: Fuzzing Thread Schedules

- Introduces sleep() calls
  - Automatically (instead of manually)
  - Systematically before each statement (as opposed to those chosen by tester)
    - Less tedious and less error-prone
- Gives worst-case probabilistic guarantees on bug finding

Depth of Concurrency Bug

- Bug depth = number of scheduling constraints that must be satisfied to trigger the bug
- Scheduling constraint = requirement on the ordering between two statements in different threads.

Depth of Concurrency Bug: Example 1

- Bug depth = number of scheduling constraints that must be satisfied to trigger the bug
- Scheduling constraint = requirement on the ordering between two statements in different threads.

Thread 1:

```
...
T t = new T();
```

Thread 2:

```
...
if(t.state ==1)
  ...
```

Depth of Concurrency Bug: Example 1

- Bug depth = number of scheduling constraints that must be satisfied to trigger the bug
- Scheduling constraint = requirement on the ordering between two statements in different threads.

Thread 1:

```
...
T t = new T();
```

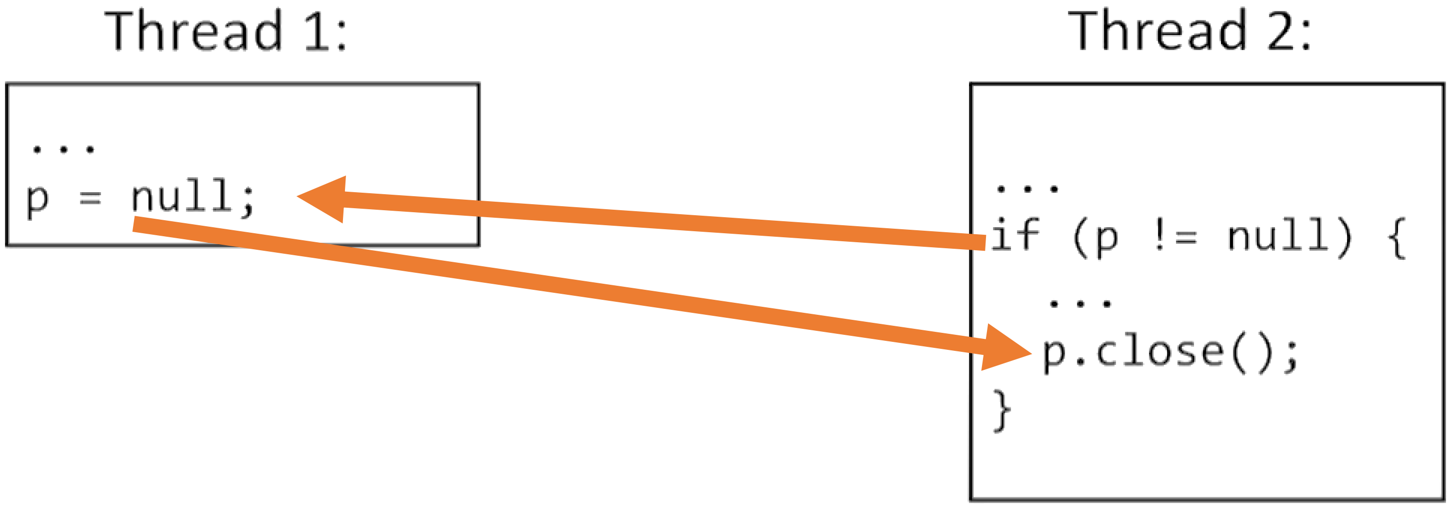
Thread 2:

```
...
if(t.state ==1)
  ...
```

Depth of this concurrency bug is 1

Depth of Concurrency Bug: Example 2

- Bug depth = number of scheduling constraints that must be satisfied to trigger the bug
- Scheduling constraint = requirement on the ordering between two statements in different threads.



Depth of Concurrency Bug: Example 2

- Bug depth = number of scheduling constraints that must be satisfied to trigger the bug
- Scheduling constraint = requirement on the ordering between two statements in different threads.

Thread 1:

```
...
p = null;
```

Thread 2:

```
...
if (p != null) {
  ...
  p.close();
}
```

Depth of this concurrency bug is 2

Depth of Concurrency Bug

- Bug depth = number of scheduling constraints that must be satisfied to trigger the bug
- Scheduling constraint = requirement on the ordering between two statements in different threads.
- Observation exploited by Cuzz: most concurrency bugs typically have small depth
  - “small test case” hypothesis: if there is a bug, there will be some small input that will trigger the bug

Quiz: Concurrency Bug

Specify the depth of the concurrency bug in the following example:

Then specify all ordering constraints needed to trigger the bug. Use notation (x,y) to mean statement x comes before statement y, and separate multiple constraints by a space.

Thread 1

```
1:  lock(a);
2:  lock(b);
3:  g = g + 1;
4:  unlock(b);
5:  unlock(a);
```

Thread 2

```
6:  lock(b);
7:  lock(a);
8:  g = 0;
9:  unlock(a);
10: unlock(b);
```

Quiz: Concurrency Bug

Specify the depth of the concurrency bug in the following example: 2

Then specify all ordering constraints needed to trigger the bug. Use notation (x,y) to mean statement x comes before statement y, and separate multiple constraints by a space.

(1,7) (6,2)

Thread 1

```
1:  lock(a);
2:  lock(b);
3:  g = g + 1;
4:  unlock(b);
5:  unlock(a);
```

Thread 2

```
6:  lock(b);
7:  lock(a);
8:  g = 0;
9:  unlock(a);
10: unlock(b);
```

Deadlock preventing both threads to proceed and makes program hang

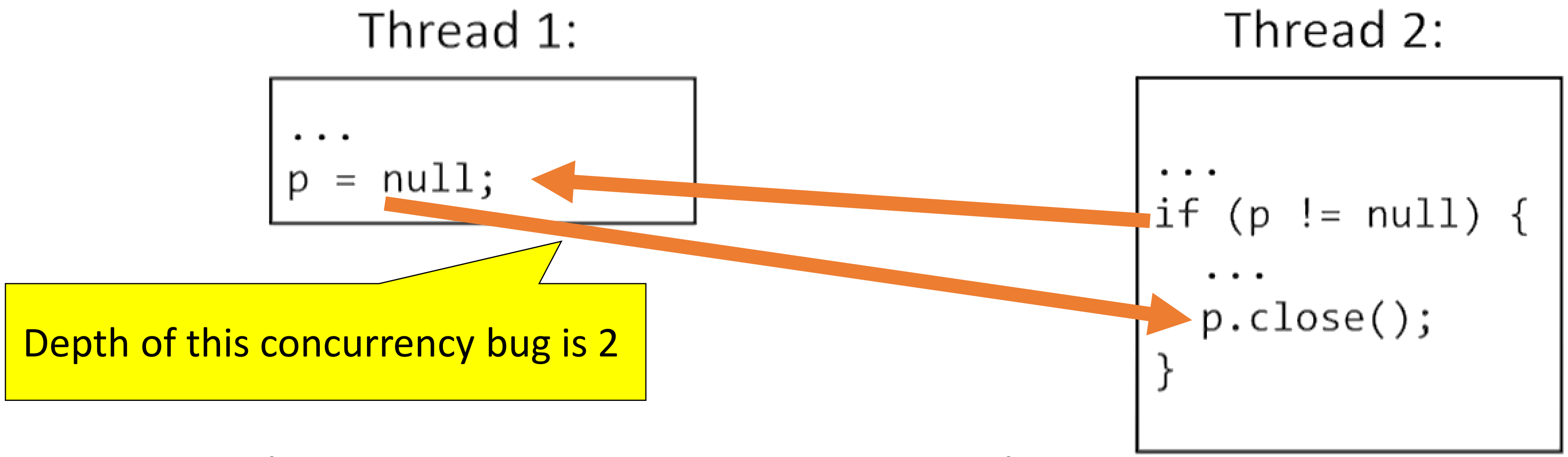
Cuzz: Probabilistic Guarantee

- Given a program with:
  - n threads (~tens)
  - k steps (~millions)
  - Bug depth d (1 or 2)

- Cuzz will find the bug with probability >= $1/ nk^{(d-1)}$ in each run

Worst case guarantee

Cuzz: Probabilistic Guarantee



- Probability (choose correct thread ordering) >= 1/2 (generally 1/ n for n threads)
- Probability (switch thread ordering at correct statement) >= 1/k, where k = total statements in program
- Probability (triggering bug) >= 1/nk
- For a bug of depth d, thread priorities are changed d-1 times, i.e., Cuzz needs to pick d-1 statements in program to switch thread execution ordering.
  - Probability (picking right set of d-1 statement >= $1/k^{(d-1)}$
- Probability (triggering bug) >= $1/nk^{(d-1)}$

What Have We Learnt

- Random Testing is effective is effective for testing mobile apps, security, and concurrency
- It is not a technique but a paradigm that you can apply to solve many different kinds of problems (e.g., Domain-Specific fuzzers)
- Complements and does not replace formal testing
- Must generate inputs from a reasonable distribution to be effective
- May be less effective for systems with multiple layers (e.g. compilers)

Next Class

Delta Debugging

Reminder

- **Paper Presentation** assignment will be released today.
- It will be done in groups of **up to three students**.
- **Project Idea presentation** in next class.
- Each group will give 10 min presentation. The submission for this assignment is due **before** class.

    1. **The Problem**. Tell us what you are going to build. If you are doing a research-focused project, tell us the research question(s) you will answer. If you are building a system, describe the prototype and tthe basic functionality and where your work will focus.
    2. **The Design.** Tell us how you will build what you are building. If you are doing a research-focused project, tell us the design of your experiment(s). If you are building a system, show up some early design.
    3. **The Evaluation.** Tell us how you will know you succeeded. If you are doing a research-focused project, tell us what data you will use, how you will know that your results make sense, what statistical tests you'll apply, etc. If you are building a system, tell us your testing plan and how you will execute it.
    4. **The Plan.** Tell us your planned timeline and each group member's responsibilities.