Week 4: Lecture B

Runtime Feedback

Wednesday, January 31, 2024

Recap: Lab 1

- **Lab 1: Beginner Fuzzing (due 2/07 by 11:59PM)**
  - Familiarize yourself with AFL++ and its features
  - Check out its documentation in **docs/**
- **Pick three features, evaluate them, and discuss your findings**
  - E.g., impacts on code coverage, speed, crash discovery
  - What insights do you have?
  - Why did one feature work better than another?
- **Deliverable:** a **1–3 page report** detailing your findings
  - Feel free to make it your own (e.g., pictures, text, etc.)
- Need a Linux environment
  - Use the **CS 4440 VM** if you don't have one!

Recap: Lab 1

- Pick any **target program** you like, e.g.:
  - FuzzGoat fuzzing benchmark
  - FoRTE-FuzzBench
  - HexHive's Magma

- Skills you'll learn along the way:
  - **Compiling** a C/C++ program
  - Inserting AFL++'s **instrumentation**
  - Initiating **fuzzing** with AFL++
  - Interpreting AFL++'s **results**

Recap: Key Dates

- **Jan. 24** — **Lab 1 released**
- **Feb. 07** — **Lab 1 due**
- **Feb. 14** — Lab 2 due
- **Feb. 19** — No class (President's Day)
- **Feb. 28** — Lab 3 due
- **Feb. 28** — **5-minute project proposals**
- **Mar. 04 & 06** — No class (Spring Break)
- **Apr. 17 & 22** — **Final project presentations**

cs.utah.edu/~snagy/courses/cs5963/schedule

Part 1: Course Intro and Research 101

Monday Meeting	Wednesday Meeting
Jan. 08 **Course Introduction**	Jan. 10 **Research 101: Ideas**
Jan. 15 **No Class (Martin Luther King Jr. Day)**	Jan. 17 **Research 101: Writing**
Jan. 22 **Research 101: Reviewing and Presenting** **Sign up for paper presentations by 11:59pm**	Jan. 24 **Introduction to Fuzzing** ▶ Readings: **Beginner Fuzzing Lab released**

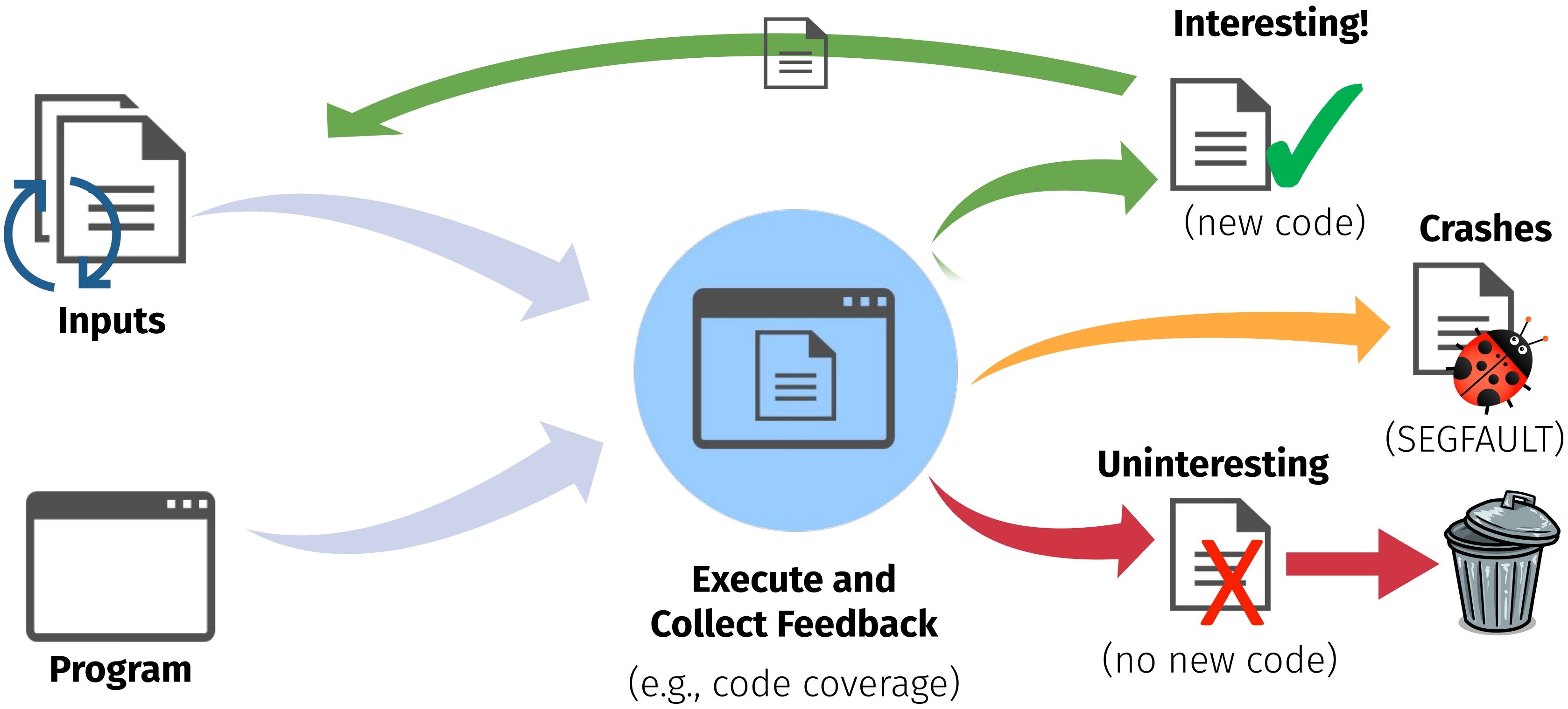
Part 2: Fuzzing Fundamentals

Monday Meeting	Wednesday Meeting
Jan. 29 **Input Generation** ▶ Readings:	Jan. 31 **Runtime Feedback** ▶ Readings:
Feb. 05 **Bugs & Triage I** ▶ Readings: **Triage Lab released**	Feb. 07 **Bugs & Triage II** ▶ Readings: **Beginner Fuzzing Lab due by 11:59pm**
Feb. 12 **Harnessing I** ▶ Readings: **Harnessing Lab released**	Feb. 14 **Harnessing II** ▶ Readings: **Triage Lab due by 11:59pm**

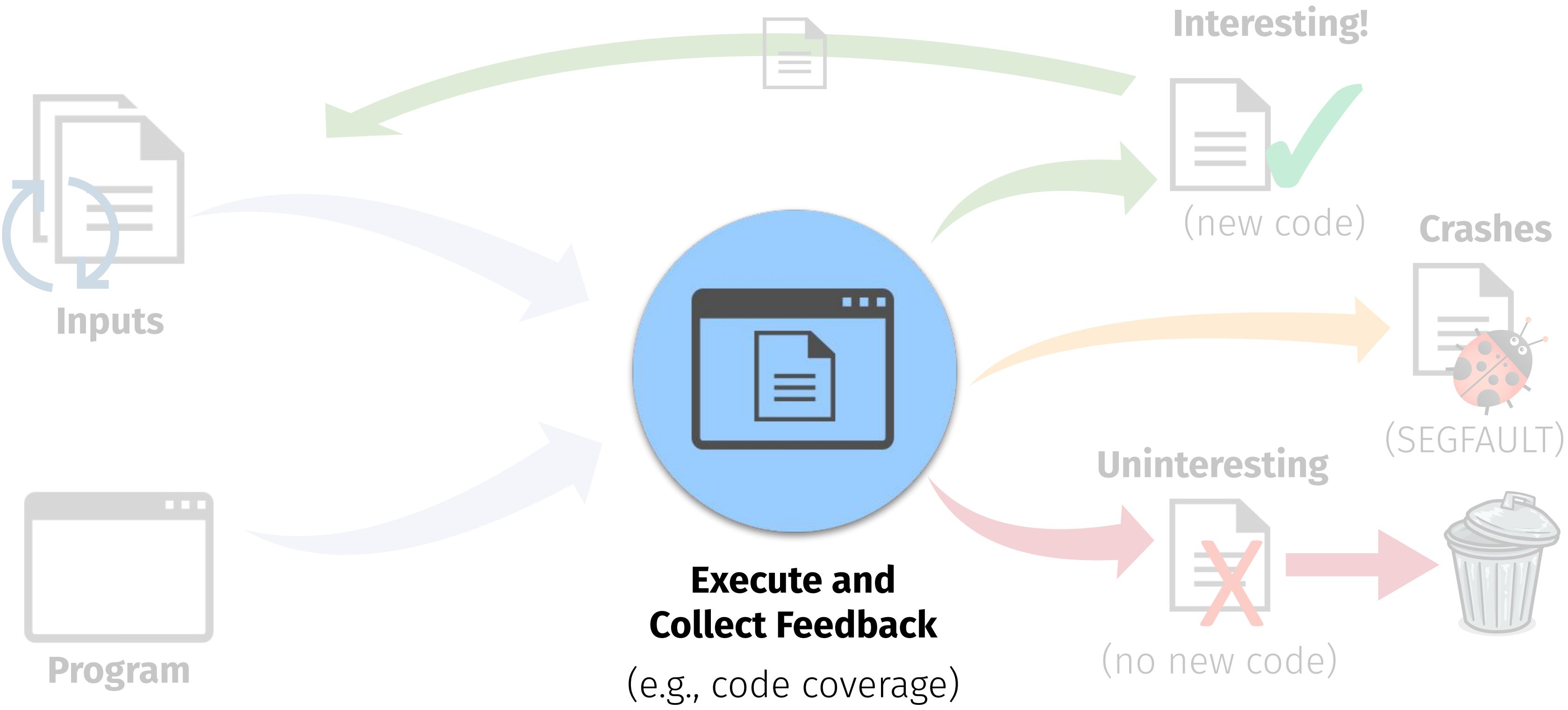
Questions?

Runtime Feedback

Recap: Coverage-guided Fuzzing

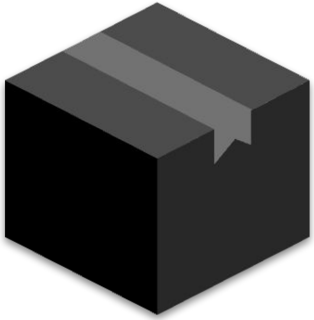


Recap: Coverage-guided Fuzzing



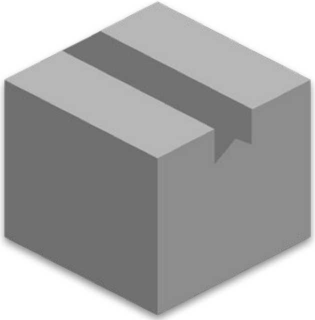
Types of Feedback-driven Fuzzers

Black-box



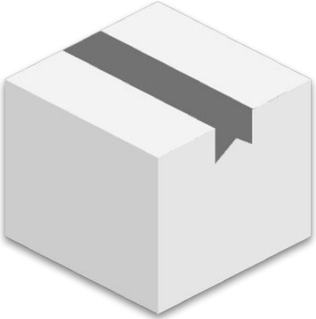
Zero Introspection

Grey-box



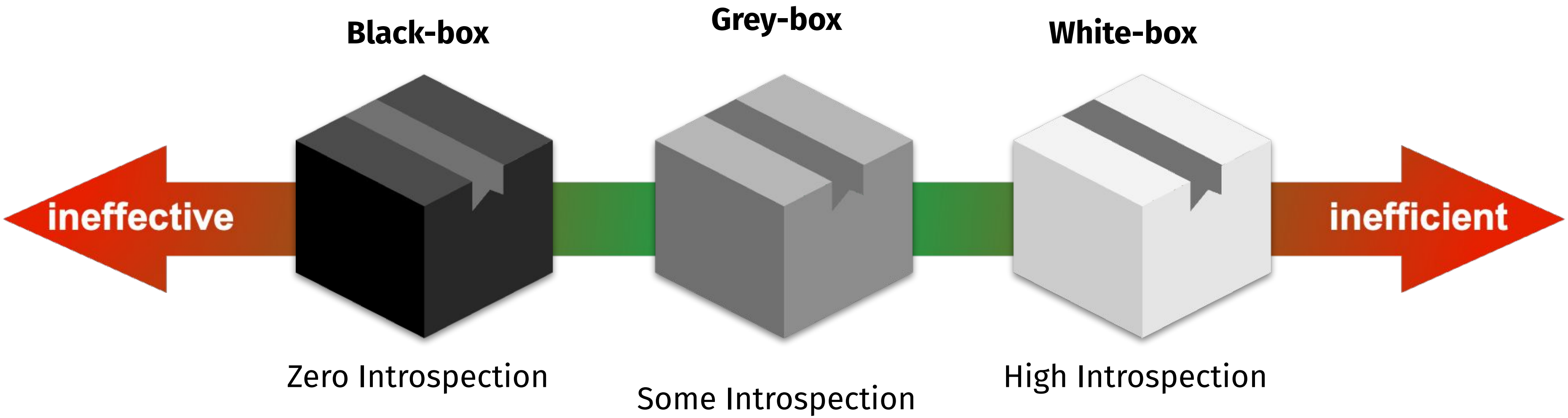
Some Introspection

White-box

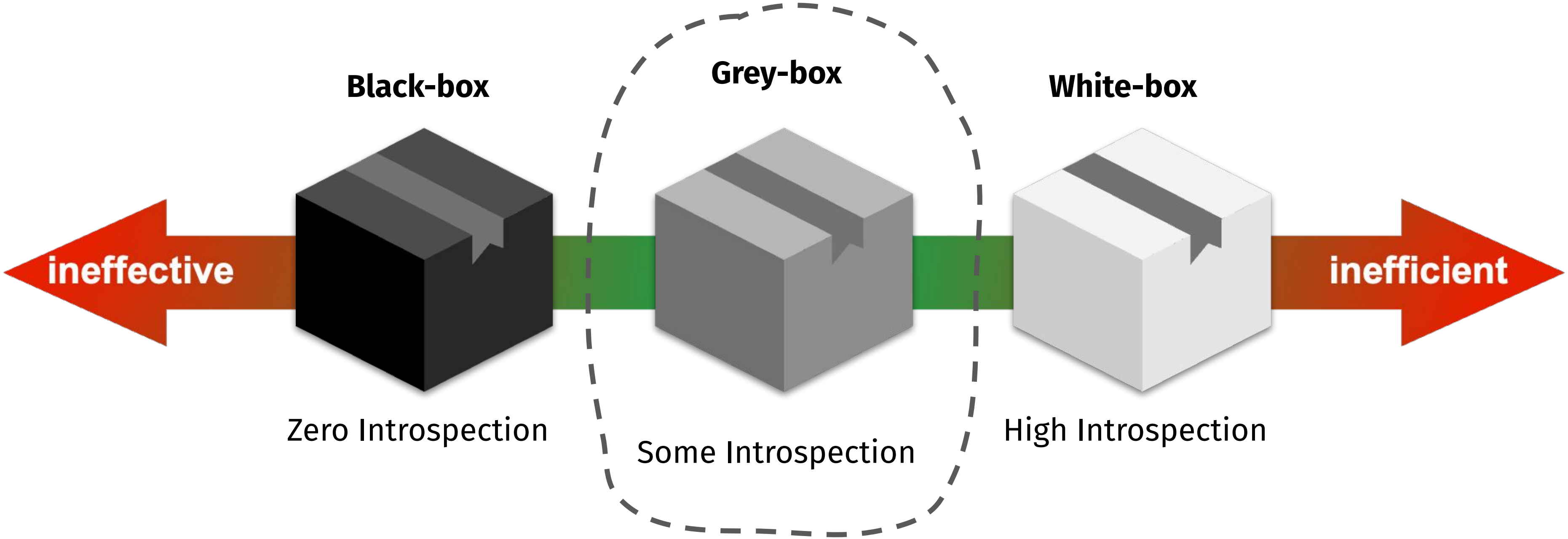


High Introspection

Types of Feedback-driven Fuzzers



Types of Feedback-driven Fuzzers



Feedback Considerations

- What makes a test case **interesting** for your target?
- How to **collect this** information from your target?
- How to **store and post-process** this information?

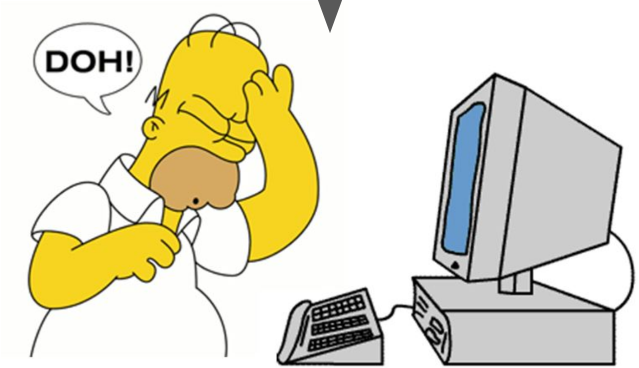
Feedback depends on your goals...

- Fuzzing something **for the first time**
  - Limited or no feedback
- Targeting a **certain code region**
  - Distances to that location, constrained coverage
- Hunting **use-after-free vulnerabilities**
  - Temporal memory accesses (`malloc()` → `free()` → use)
- Finding **resource exhaustion bugs**
  - Execution path length, execution duration

Trade-offs

- How **costly** is it to collect?
  - Runtime overhead
- Special **data structures** to store it?
  - Post-processing overhead
  - Implementation cost
- How **selective** will it be?
  - Not everything should look “interesting”
- **Does it even help?**

```
total execs  : 3202
exec speed   : 10.7/sec (slow!)
```



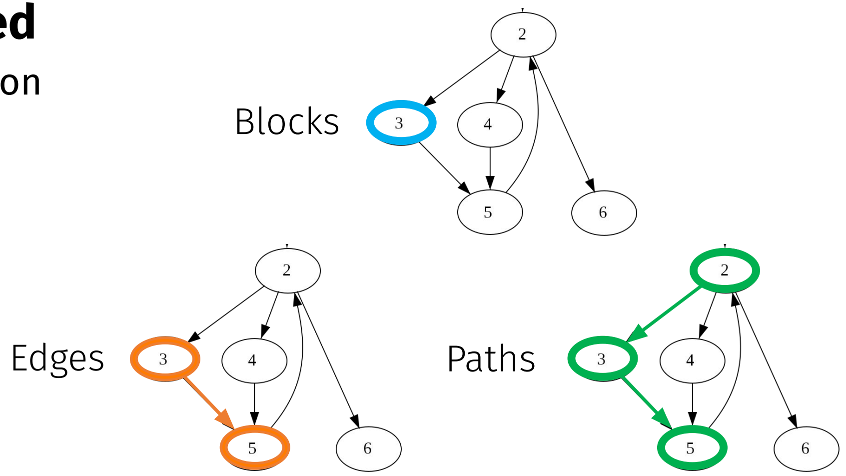
Code Coverage

Coverage-guided Fuzzing

- **Code coverage:** parts of the target code exercised by a test case

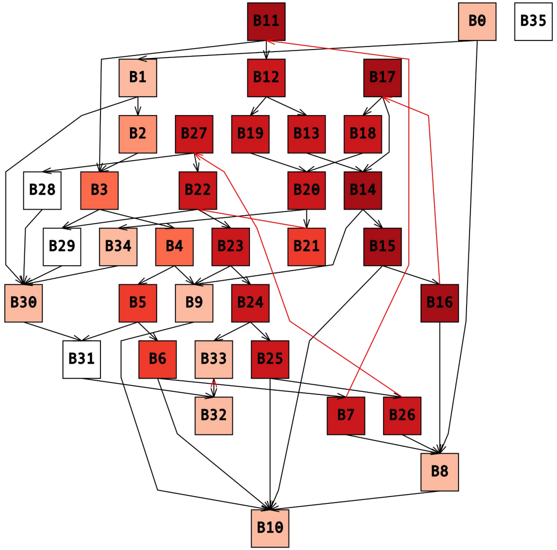
- Most fuzzing today is **coverage-guided**
  - Good balance of performance and precision

- Various metrics in use today:
  - Basic blocks
  - Edges
  - Hit counts
  - Instructions
  - Path approximations



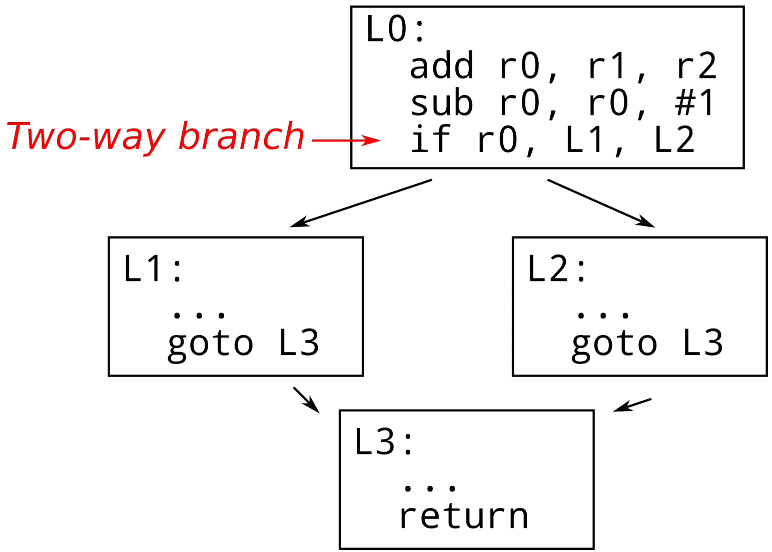
Program Control-flow Graphs (CFGs)

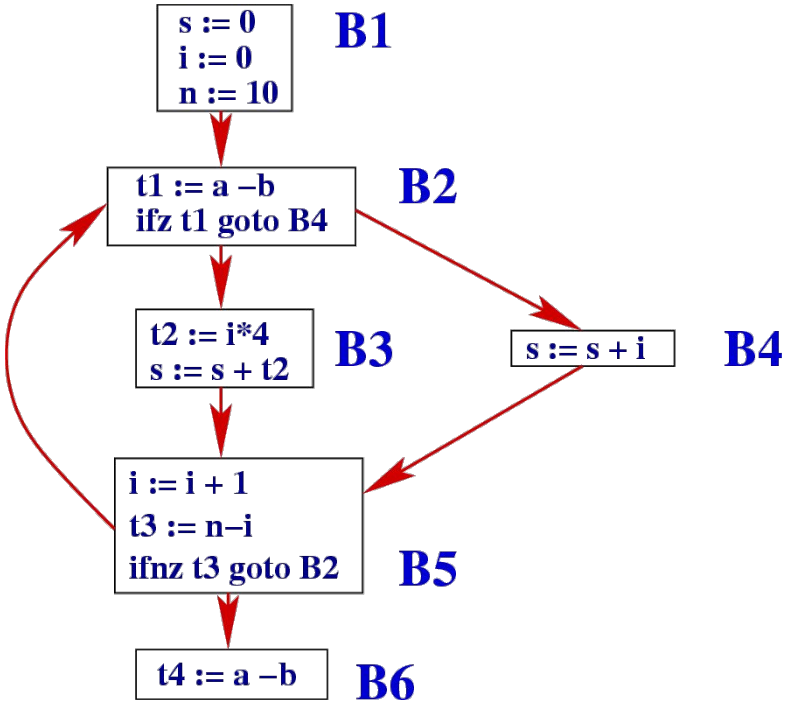
- Graph representation of **every possible program path**
  - Directed graph
  - **Nodes:** basic blocks
  - **Edges:** control-flow transitions between blocks



- Essential to software analysis
  - Compiler optimization
  - Static vulnerability discovery
  - **Code coverage measurement**

CFG Examples

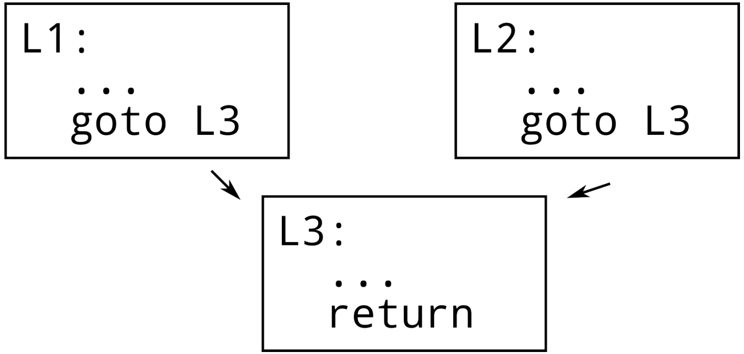




Basic Block Coverage

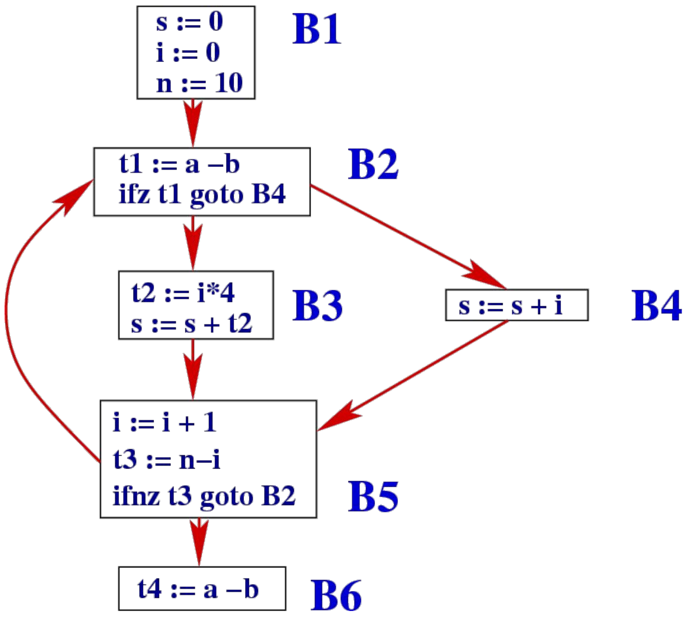
- **Basic blocks:** straight-lined code sequences entered-by / ending-in transfer
  - The nodes of the a program's control-flow graph
  - No control-flow transfer within a basic block

- Control-flow transfer instructions:
  - Jumps
  - Calls
  - Returns
  - Fall-through to next sequential block



Edge Coverage

- **Edges:** transitions between basic blocks
  - Jumps:
    - To basic blocks
  - Calls:
    - To function entries
  - Returns:
    - To post-call caller basic block
  - Fall-throughs:
    - To next sequential basic block



Instruction Coverage

- **Instructions:** the program's individual operations
  - What the processor executes
  - More common to measure in post-fuzzing coverage analysis

Line	Branch	Exec	Source
1			`// example.cpp`
2			
3		1	`int foo(int param)`
4			`{`
5	✗✓	1	`    if (param)`
6			`    {`
7			`        return 1;`
8			`    }`
9			`    else`
10			`    {`
11		1	`        return 0;`
12			`    }`
13			`}`
14			
15		1	`int main(int argc, char* argv[])`
16			`{`
17		1	`    foo(0);`
18			
19		1	`    return 0;`
20			`}`
21			

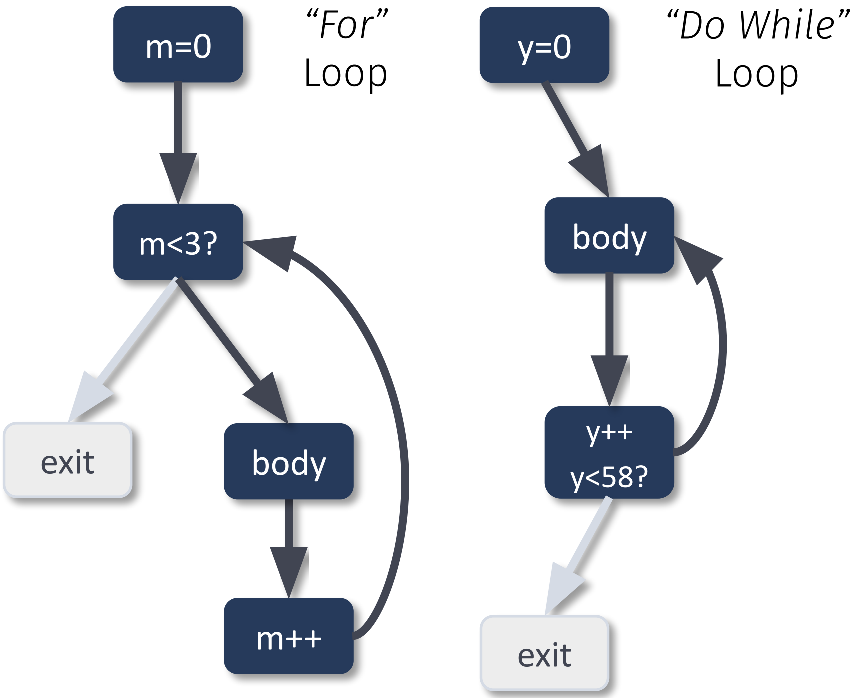
Hit Counts

- Execution is not just forward
  - Most program execution is spent in **loops**

```
for(m = 0; m < 3; m++){
      /* loop body */
}
```

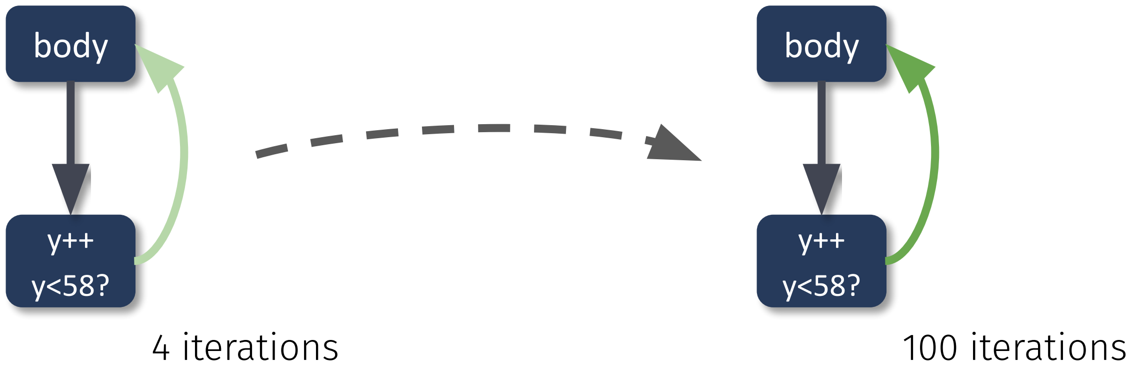
```
while(d < 14){
      /* loop body */
      d = d + 1;
}
```

```
do {
      /* loop body */
      y = y + 1;
} while (y < 58);
```



Hit Count Coverage

- **Hit counts:** execution frequencies of blocks, edges, etc.
  - Used to discern “interesting” changes in covering already-seen code
    - Looping for a higher number of consecutive iterations
    - Greater recursion depth



Common Coverage Metrics in Fuzzing

Fuzzer	Coverage	Fuzzer	Coverage	Fuzzer	Coverage
AFL	Edges + Counts	EnFuzz	Edges + Counts	ProFuzzer	Edges + Counts
AFL++	Edges + Counts	FairFuzz	Edges + Counts	QSYM	Edges + Counts
AFLFast	Edges + Counts	honggFuzz	Edges	REDQUEEN	Edges + Counts
AFLSmart	Edges + Counts	GRIMOIRE	Edges + Counts	SAVIOR	Edges + Counts
Angora	Edges + Counts	laf-Intel	Edges + Counts	SLF	Edges + Counts
CollAFL	Edges + Counts	libFuzzer	Edges + Counts	Steelix	Edges + Counts
DigFuzz	Edges + Counts	Matryoshka	Edges + Counts	Superion	Edges + Counts
Driller	Edges + Counts	MOpt	Edges + Counts	TIFF	Blocks + Counts
Eclipser	Edges + Counts	NEUZZ	Edges + Counts	VUzzer	Blocks + Counts

Questions?

Feedback Collection

Program Instrumentation

- Transforming a program to add extra behavior
  - Add new functionality that was not originally there
  - E.g., tracing of test cases' code coverage
- **Source-available** programs
  - Bake-in instrumentation at compile-time
  - Or at assembly-time
- **Binary-only** programs
  - Statically reverse-engineer its semantics
  - Dynamically on-the-fly as it is executing
  - **Way more complicated and difficult**

Instrumentation Platforms

Compiler — LLVM Compiler Infrastructure, GCC

Binary (static) — Dyninst

Binary (dynamic) — Pin, DynamoRIO, QEMU

speed

AFL's Edge Coverage

- Edge coverage via hashed basic block tuples

```
cur_location = <COMPILE_TIME_RANDOM>;
Shared_mem [cur_location ⊕ prev_location]++;
prev_location = cur_location >> 1;
```

AFL’s Edge Coverage

- Edge coverage via hashed basic block tuples
  - Each basic block assigned a random ID at compile-time

```
cur_location = <COMPILE_TIME_RANDOM>;
Shared_mem [cur_location ⊕ prev_location]++;
prev_location = cur_location >> 1;
```

AFL’s Edge Coverage

- Edge coverage via hashed basic block tuples
  - Each basic block assigned a random ID at compile-time

```
cur_location = <COMPILE_TIME_RANDOM>;
Shared_mem [cur_location ⊕ prev_location]++;
prev_location = cur_location >> 1;
```

  - **Edge hash:** current basic block ID is **XOR’d** to previous basic block’s

AFL’s Edge Coverage

- Edge coverage via hashed basic block tuples
  - Each basic block assigned a random ID at compile-time

```
cur_location = <COMPILE_TIME_RANDOM>;
Shared_mem [cur_location ⊕ prev_location]++;
prev_location = cur_location >> 1;
```

  - **Edge hash:** current basic block ID is **XOR’d** to previous basic block’s
    - Edge-specific hit counter incremented by one for each exercising

AFL's Edge Coverage

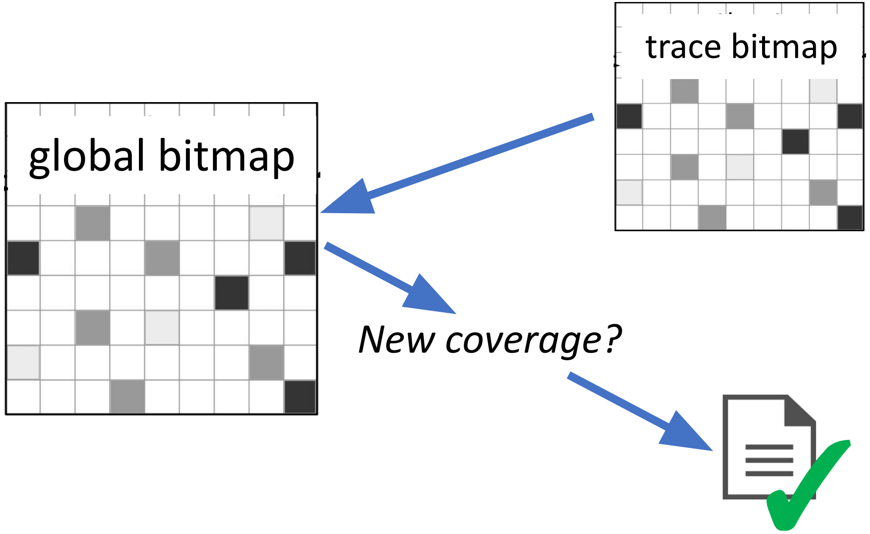
- Edge coverage via hashed basic block tuples
  - Each basic block assigned a random ID at compile-time

```
cur_location = <COMPILE_TIME_RANDOM>;
Shared_mem [cur_location ⊕ prev_location]++;
prev_location = cur_location >> 1;
```

  - **Edge hash:** current basic block ID is **XOR'd** to previous basic block's
    - Edge-specific hit counter incremented by one for each exercising
  - **Right shift** current block to preserve edge **directionality** (because XOR is commutative)
    - Enables A→B to be seen as distinct from B→A; also A→A from B→B

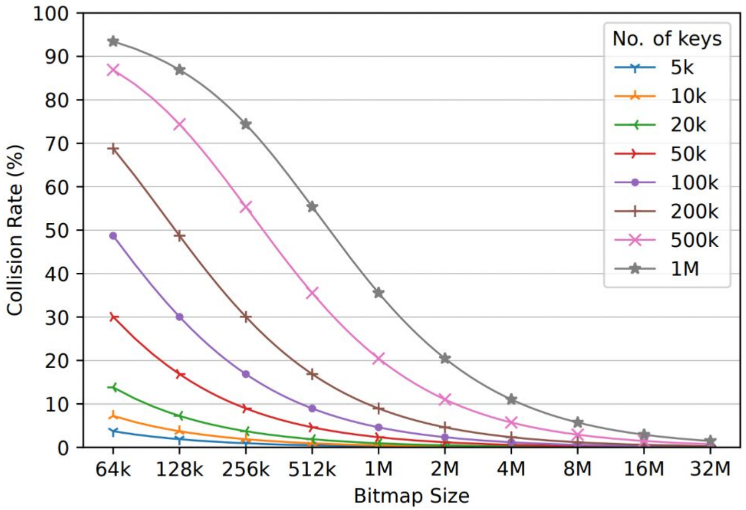
AFL’s Coverage Storage

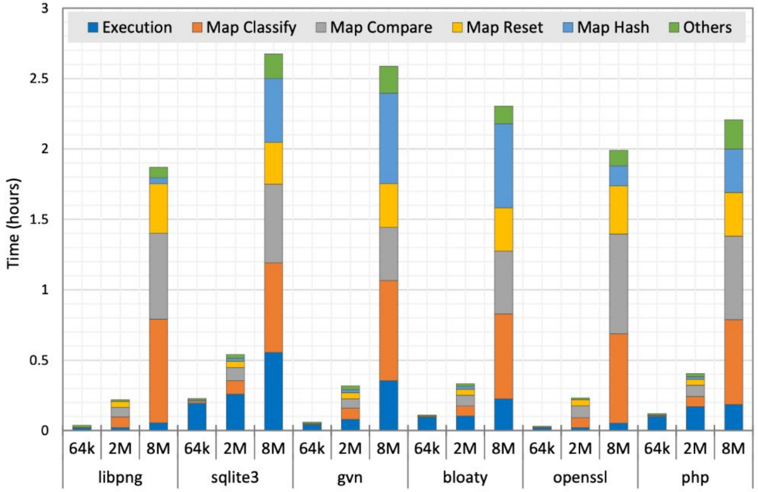
- Data structure: the **edge bitmap**
  - EdgeIDs = trace bitmap indexes
    - trace_bitmap[edge_id]++
  - Global bitmap updated only if trace contains **previously-unseen index(es)**
    - The union of all covered edges
  - **Default size: 64kB** (65536 entries)
    - Why?



Trade-offs

- **Performance:** 64kB small enough to fit in most systems' L-2 cache
- **Hash collisions:** with more edges = more collisions = lost edges
  - Increasing bitmap size to compensate leads to big slowdowns





Source: BigMap: Future-proofing Fuzzers with Efficient Large Maps

AFL's Hit Count Coverage

- Edge execution frequencies **discretized** to 8 "buckets"
  - Artifact of bitmap implementation (edge ID's map to 8-bit counters)

[1]	[2]	[3]	[4,7]
[8,15]	[16,31]	[32,127]	[128+]

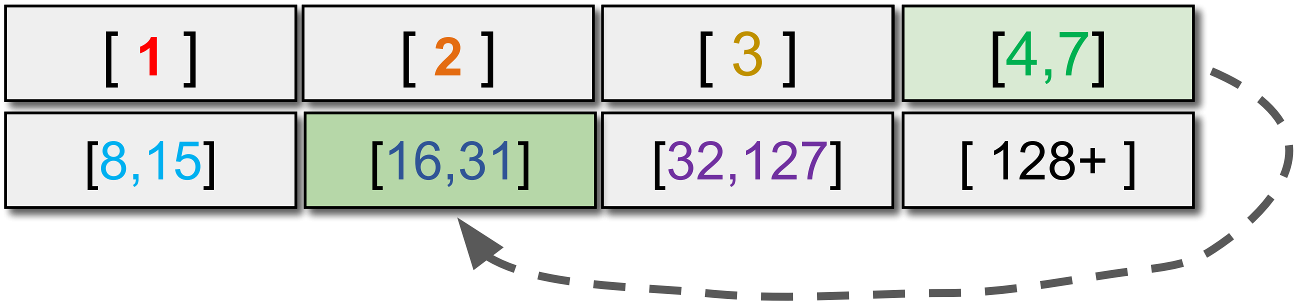
AFL's Hit Count Coverage

- Edge execution frequencies **discretized** to 8 "buckets"
  - Artifact of bitmap implementation (edge ID's map to 8-bit counters)
- Flag changes to higher buckets as interesting

[1]	[2]	[3]	[4,7]
[8,15]	[16,31]	[32,127]	[128+]

AFL’s Hit Count Coverage

- Edge execution frequencies **discretized** to 8 “buckets”
  - Artifact of bitmap implementation (edge ID’s map to 8-bit counters)
- Flag changes to higher buckets as interesting
  - E.g., an already seen edge’s count “jumps” from [4-7] to [16-31]



Trade-offs

- **Captures many interesting program state changes**
  - Deeper loop coverage
  - Deeper recursion depth

- **Not all loops are the same**
  - Miss subtle hit count changes
  - Biased to spending time in loops
    - Some loops you should avoid
  - **Still an open research problem**

Trade-offs are target-dependent!

Building a good fuzzer is all about finding the right balance of **performance & precision**.

Trade-offs are target-dependent!

Building a good fuzzer is all about finding the right balance of **performance & precision**.

Simple is (usually) better.

Questions?