Week 11: Lecture A

Directed Fuzzing

Monday, March 25, 2024

How are semester projects going?

Smoothly?

Obstacles?

Recap: Project Schedule

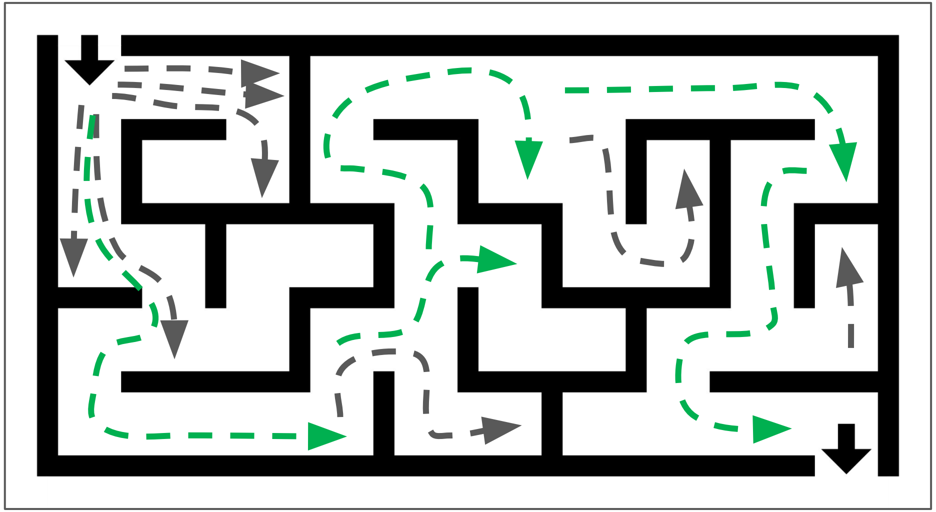
- **Mar. 27th:** in-class project workday
- **Apr. 17th & 22nd:** final presentations
  - 15–20 minute slide deck and discussion
  - What you did, and why, and what results

Questions?

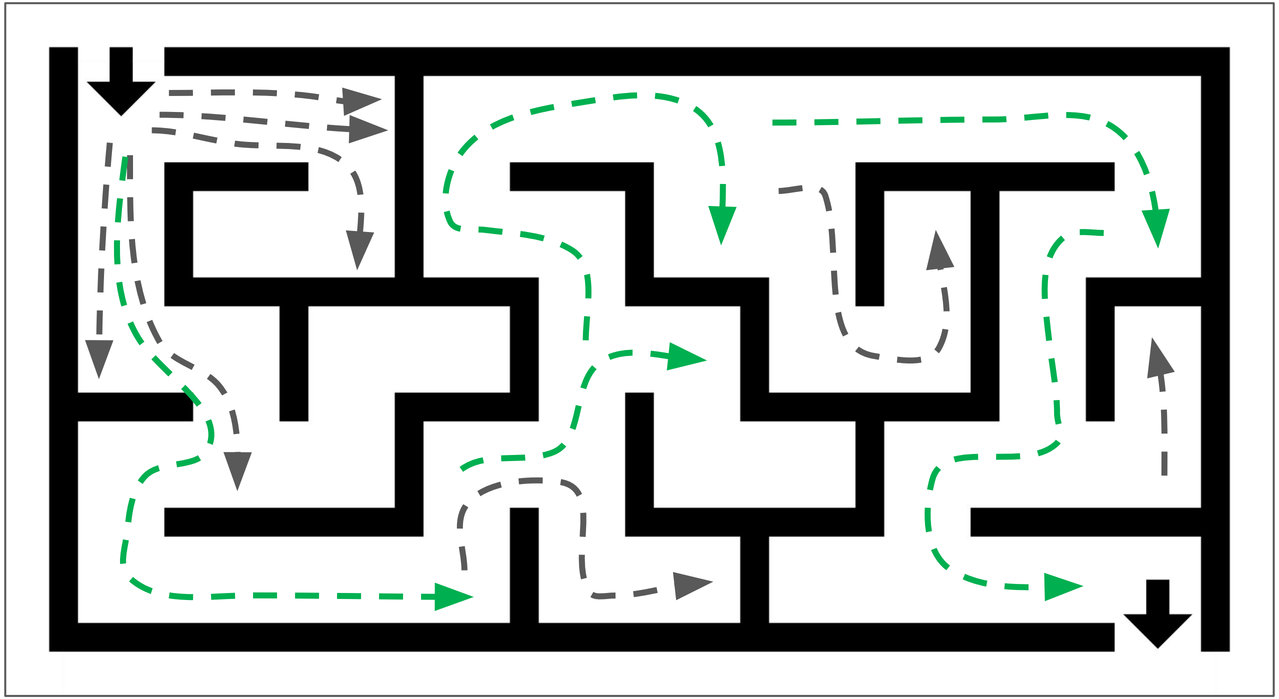
Directed Fuzzing

Recap: Coverage-guided Fuzzing

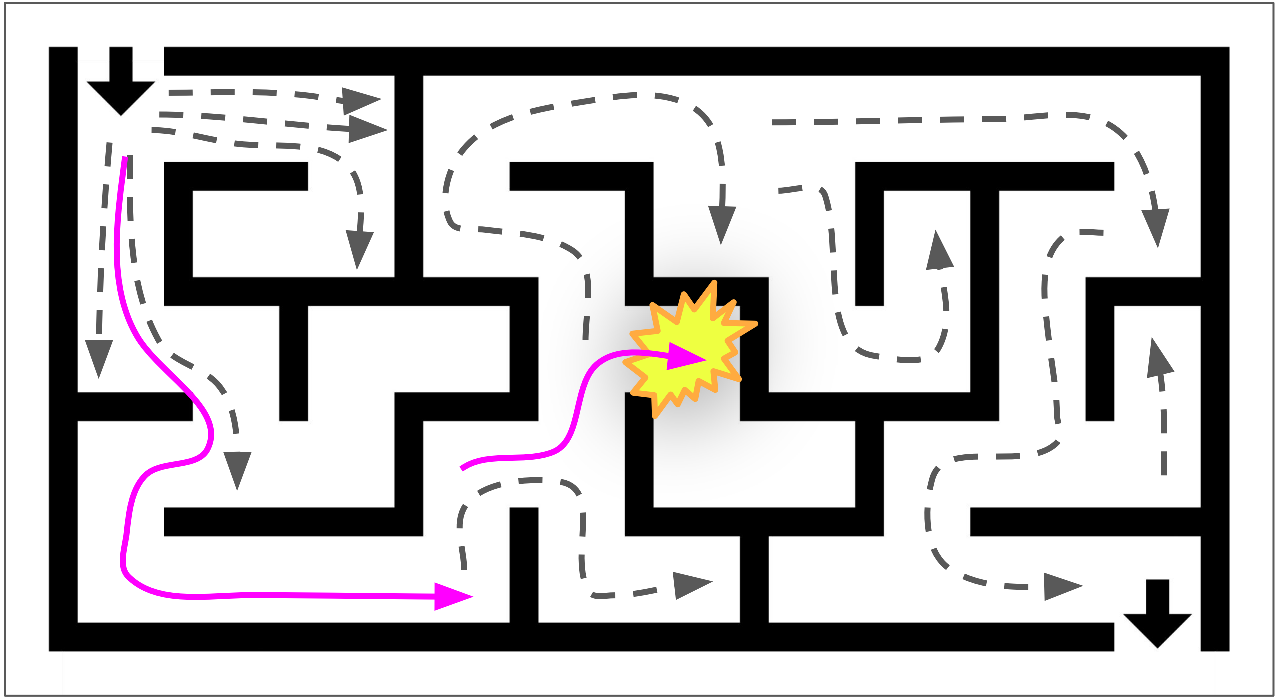
- Idea: track some measure of exploration “progress”
  - Coverage of program code
  - Stack traces
  - Memory accesses
- Pinpoint inputs that further progress over the others
- **Mutate only those inputs**



What if I only want to fuzz *one* location?



What if I only want to fuzz *one* location?



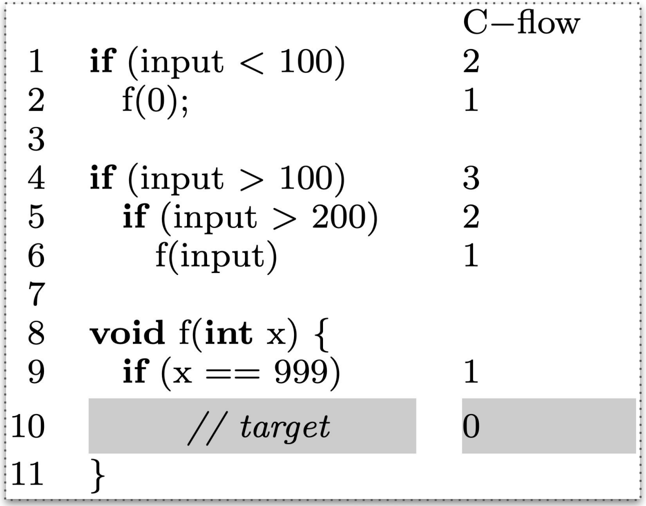
What if I only want to fuzz *one* location?

- **Regression testing**
  - Did my PR break the software?
- **Patch testing**
  - Have I actually fixed this vulnerability?
- **Crash reproduction**
  - Is this random person’s bug report valid?

“Directed” Fuzzing

- Guided fuzzing steered to **specific locations**
  - E.g., Patch-changed code lines
  - E.g., An ASAN-reported crash line

- **Key differences versus guided fuzzing:**
  - **Instrumentation:**
    - Track **distance** relative to targeted site(s)
    - Compute this for **every** generated test case
  - **Seed selection:**
    - Pick inputs that get you **closer** to target(s)
    - Progress stalls? Pick a new input and restart



```
                                  C−flow
if (input < 100)                  2
   f(0);                          1

if (input > 100)                  3
   if (input > 200)               2
      f(input)                    1

void f(int x) {
   if (x == 999)                  1
         // target                0
}
```

Source: KATCH: High-Coverage Testing of Software Patches

Directed Fuzzing

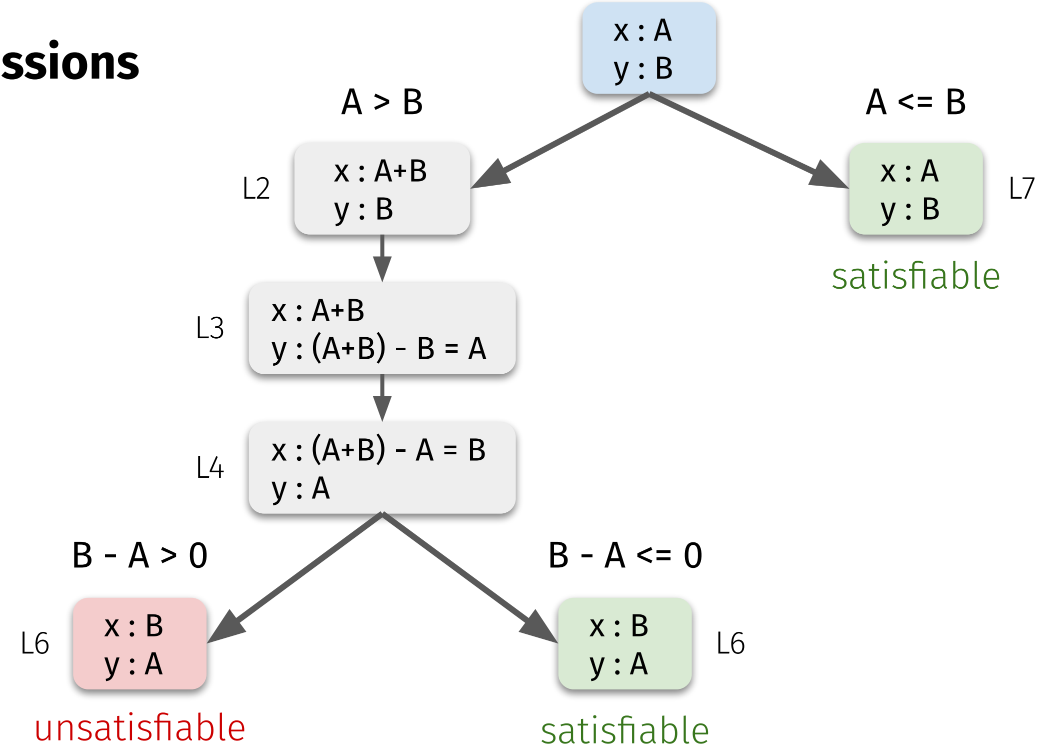
Recap: Symbolic Execution

- Solve paths as **symbolic expressions**

```
0. def f (x, y):
1.    if (x > y):
2.       x = x + y
3.       y = x - y
4.       x = x - y
5.       if (x - y > 0):
6.          assert false
7.    return (x, y)
```

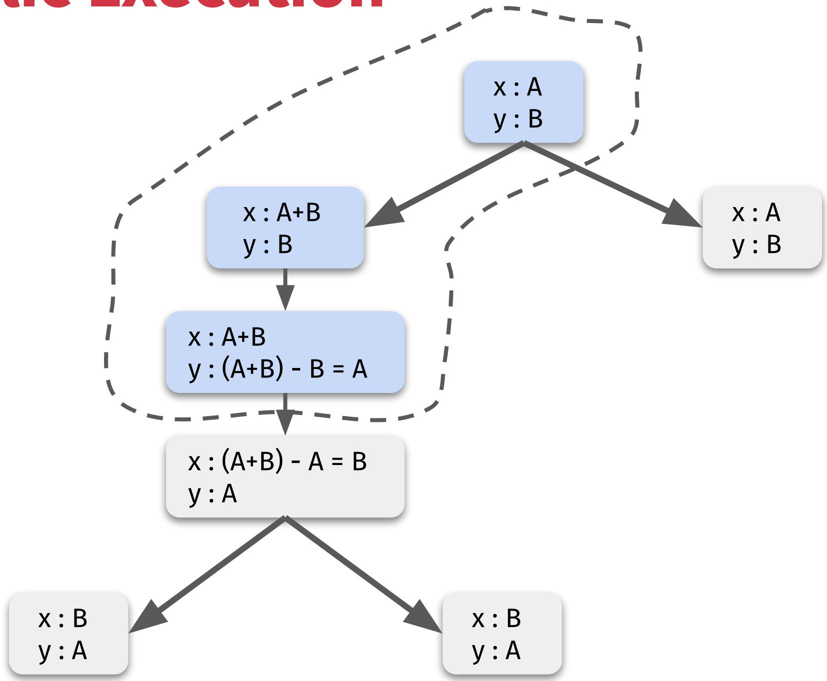
Possible path constraints:

- (A > B) and (B-A > 0) = unsatisfiable
- (A > B) and (B-A <= 0) = satisfiable
- (A <= B) = satisfiable



Directed Symbolic Execution

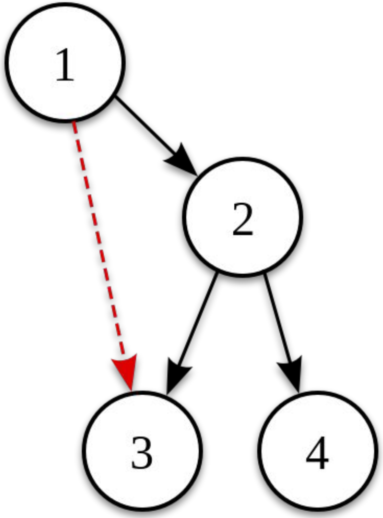
- **Early directed testing relied on SE**
  - E.g., KATCH (built atop of KLEE)
  - Primarily used for patch testing
- **Idea:** perform SE on specific paths
  - **Recap:** SE models paths symbolically
    - Find all satisfiable assignments
    - Generates branch-solving inputs
- **Trade-offs:**
  - Far too **heavyweight** to be practical
    - Not great on complex programs



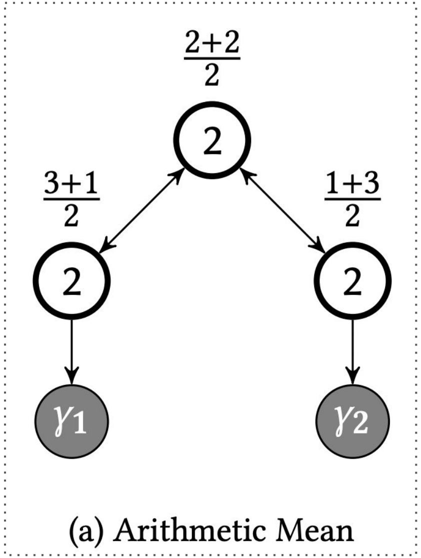
Directed Fuzzing

- **Direct successor to DSE**
  - Originator: AFL-Go

- **Idea:** minimize **seed–target distance**
  - Obtain each basic block's distance to target(s)
    - Computed during instrumentation time
  - Aggregate seed distance over block distances
    - Ideally minimize this over time



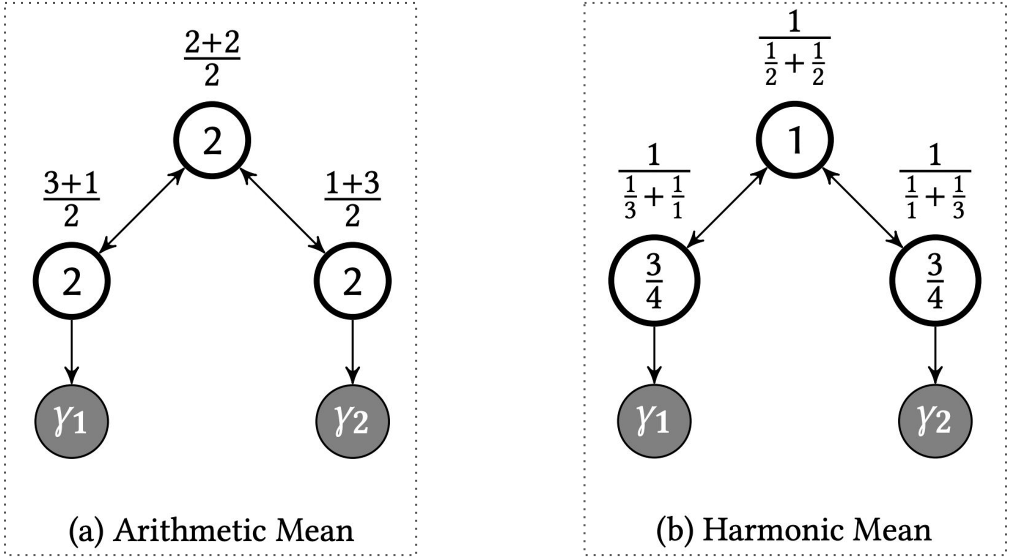
Distance Measurements



(a) Arithmetic Mean

Source: Directed Greybox Fuzzing

Distance Measurements

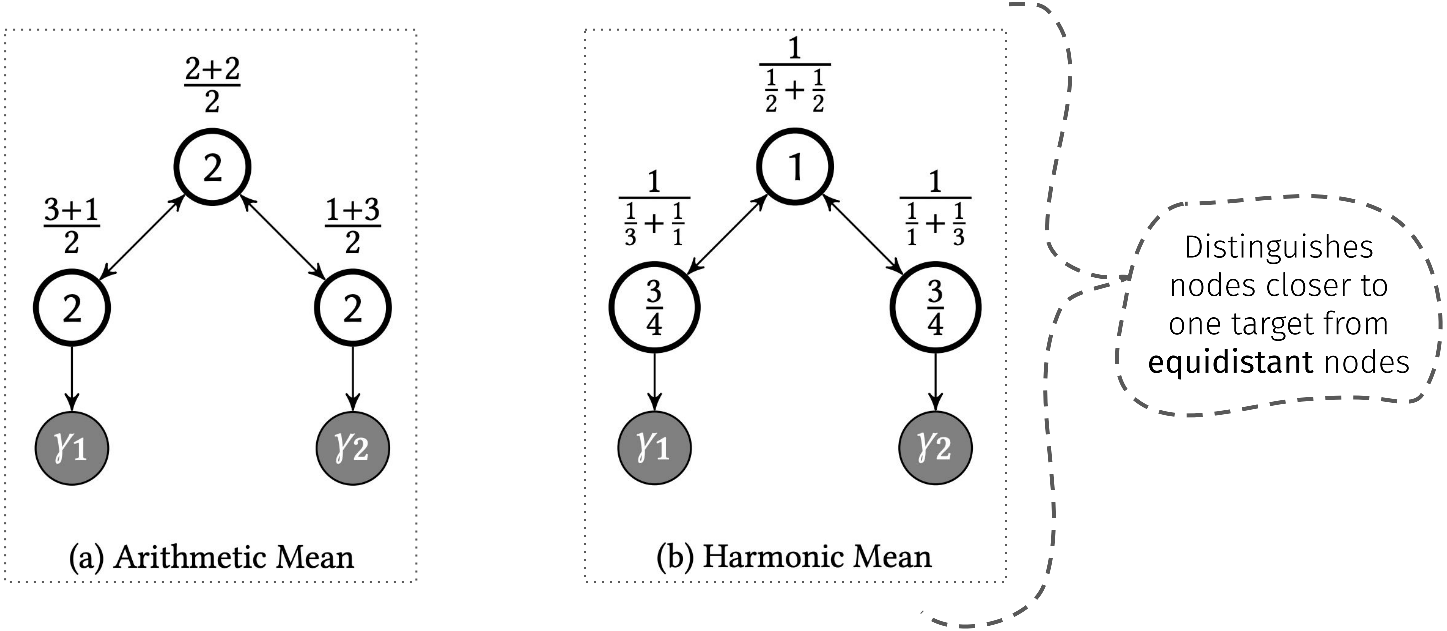


(a) Arithmetic Mean

(b) Harmonic Mean

Source: Directed Greybox Fuzzing

Distance Measurements



$\frac{2+2}{2}$
2
$\frac{3+1}{2}$
$\frac{1+3}{2}$
2
2
y_1
y_2
$\frac{1}{\frac{1}{2}+\frac{1}{2}}$
1
$\frac{1}{\frac{1}{3}+\frac{1}{1}}$
$\frac{1}{\frac{1}{1}+\frac{1}{3}}$
$\frac{3}{4}$
$\frac{3}{4}$
y_1
y_2

(a) Arithmetic Mean

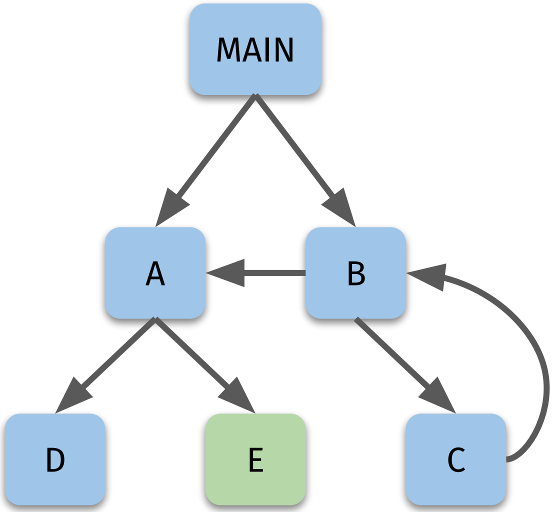
(b) Harmonic Mean

Distinguishes nodes closer to one target from **equidistant** nodes

Source: Directed Greybox Fuzzing

Function-level Distances

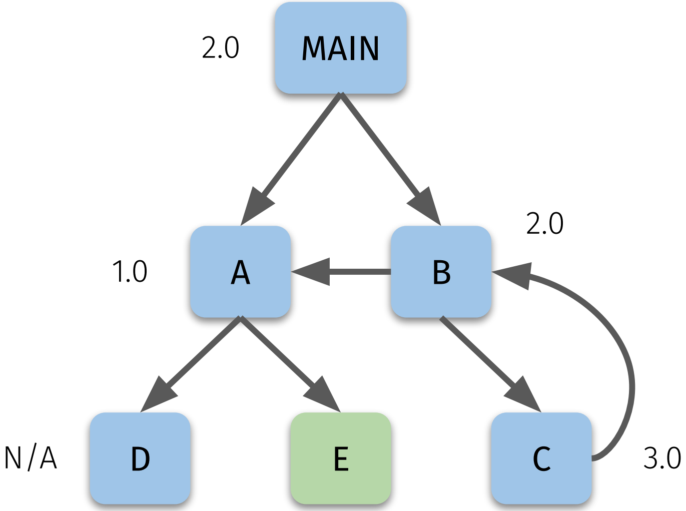
- Obtain the program's **call graph**
  - Relationships among all subroutines
  - Here, our target function is **E**



Function-level Distances

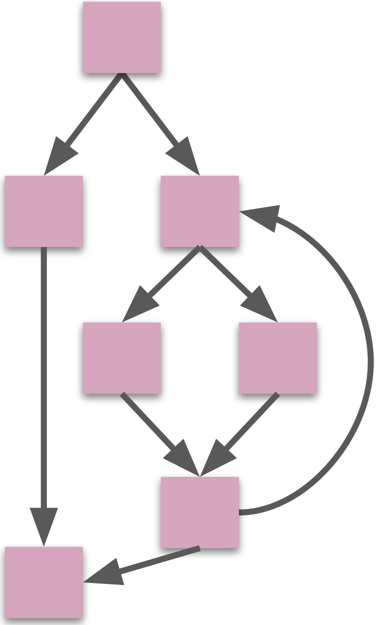
- Obtain the program's **call graph**
  - Relationships among all subroutines
  - Here, our target function is **E**

- Assign each *f* a harmonic distance
  - Relative to the **target function(s)**
  - No path to target? No score (e.g., **D**)



Block-level Distances

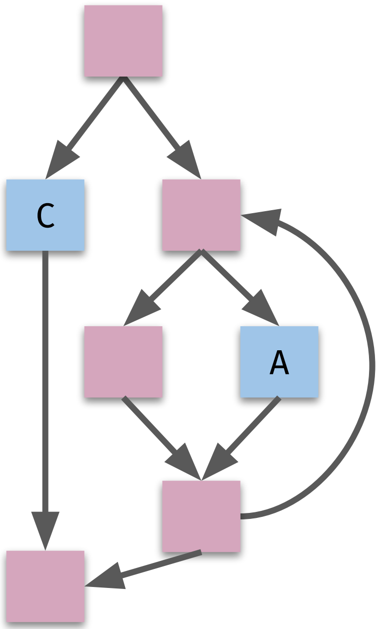
- Obtain **control-flow graph** for each *f*
  - Transitions between basic blocks in *f*
  - Here, we have a CFG for function **B**



***Block*-level Distances**

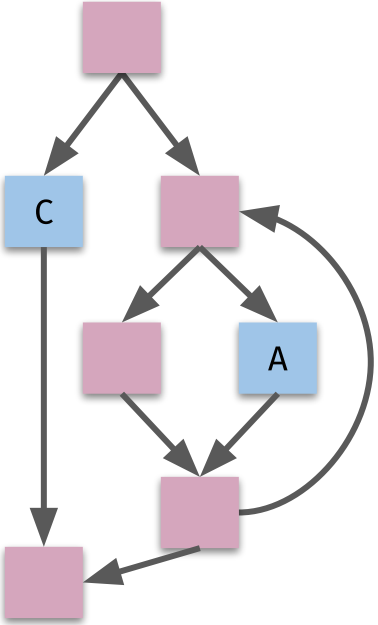
- Obtain **control-flow graph** for each *f*
  - Transitions between basic blocks in *f*
  - Here, we have a CFG for function **B**

- Identify basic blocks that call **functions**
  - Here, calls to functions **A** and **C**



Block-level Distances

- Obtain **control-flow graph** for each *f*
  - Transitions between basic blocks in *f*
  - Here, we have a CFG for function **B**
- Identify basic blocks that call **functions**
  - Here, calls to functions **A** and **C**
- Assign distances to each *b* in *f*

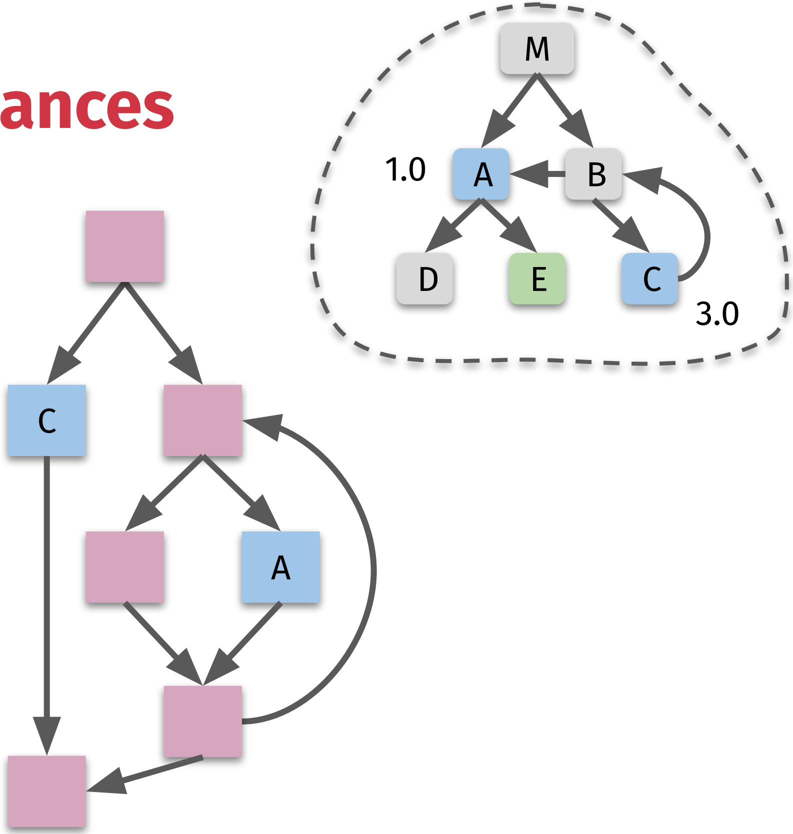


Block-level Distances

- Obtain **control-flow graph** for each *f*
  - Transitions between basic blocks in *f*
  - Here, we have a CFG for function **B**

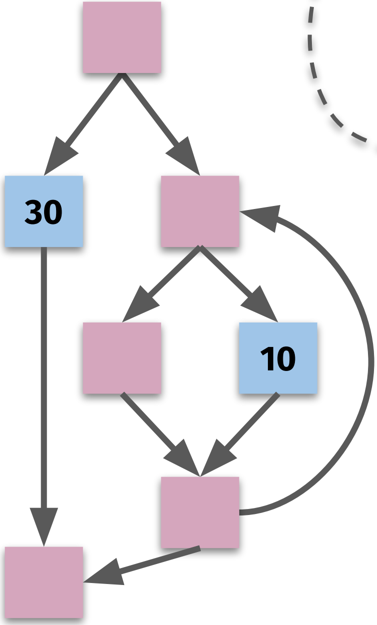
- Identify basic blocks that call **functions**
  - Here, calls to functions **A** and **C**

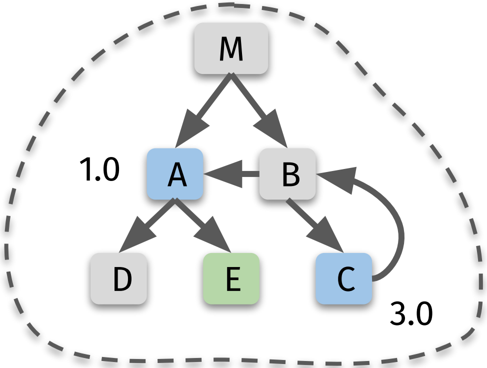
- Assign distances to each *b* in *f*
  - **Callers:** 10 * (callee’s function-level distance)



Block-level Distances

- Obtain **control-flow graph** for each *f*
  - Transitions between basic blocks in *f*
  - Here, we have a CFG for function **B**
- Identify basic blocks that call **functions**
  - Here, calls to functions **A** and **C**
- Assign distances to each *b* in *f*
  - **Callers:** 10 * (callee's function-level distance)
    - Choice of 10 seems arbitrary



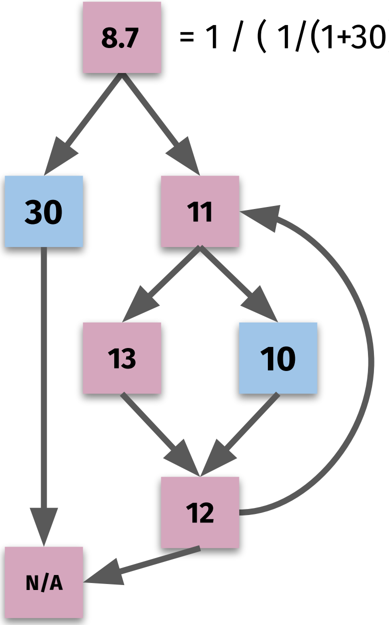


Block-level Distances

- Obtain **control-flow graph** for each *f*
  - Transitions between basic blocks in *f*
  - Here, we have a CFG for function **B**

- Identify basic blocks that call **functions**
  - Here, calls to functions **A** and **C**

- Assign distances to each *b* in *f*
  - **Callers:** 10 * (callee's function-level distance)
    - Choice of 10 seems arbitrary
  - **Rest:** harmonic distances to caller blocks
    - No path to a caller? No score



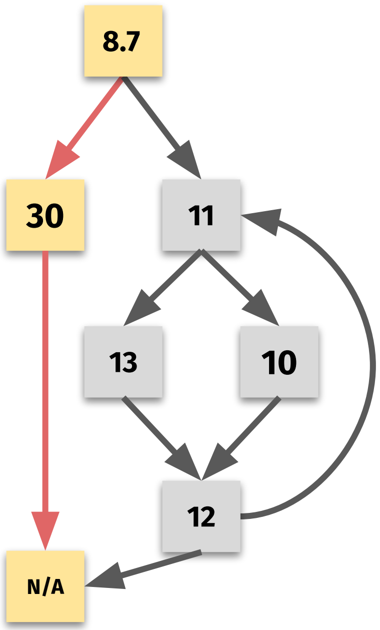
= 1 / (1/(1+30) + 1/(2+10))

Aggregating Distance

- **Normalize cumulative block distances over edges taken**

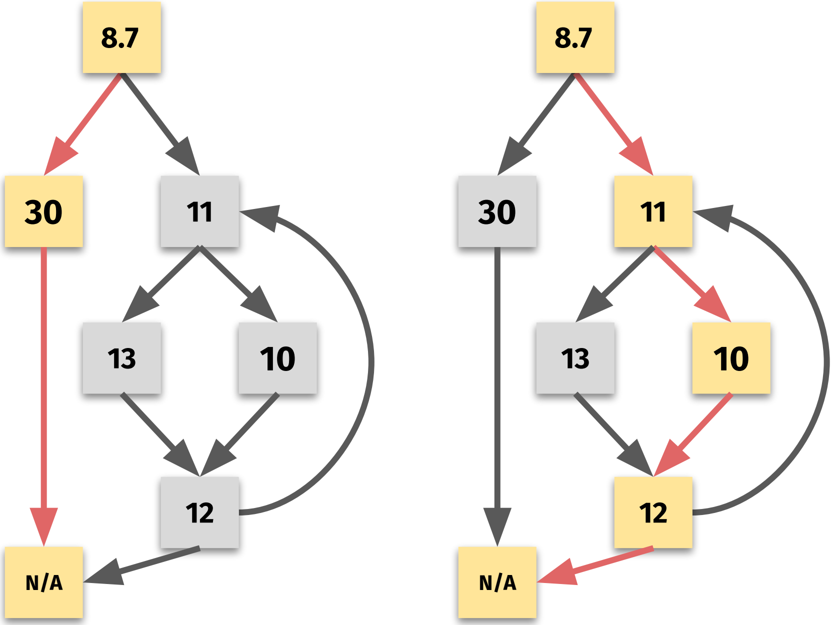
Aggregating Distance

- **Normalize cumulative block distances over edges taken**
  - E.g., seed one = (8.7 + 30) / 2
    - Seed Distance = **19.35**



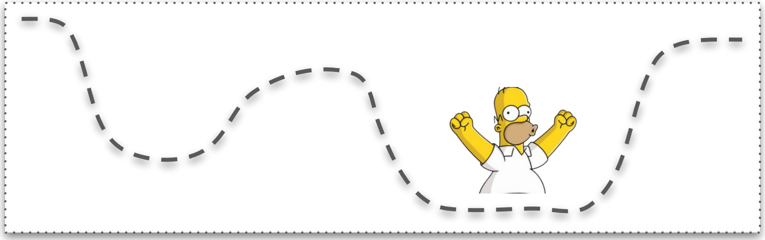
Aggregating Distance

- **Normalize cumulative block distances over edges taken**
  - E.g., seed one = (8.7 + 30) / 2
    - Seed Distance = **19.35**
  - E.g., seed two = (8.7 + 11 + 10 + 12) / 4
    - Seed Distance = **10.42**



Closing the Distance

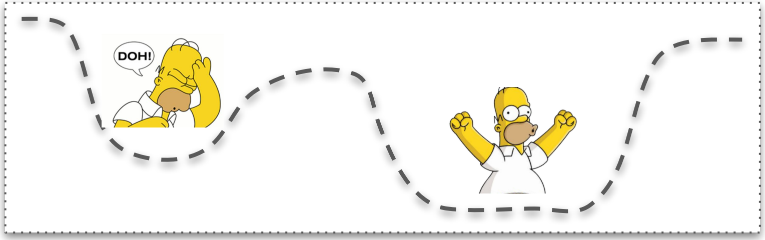
- By minimizing distance, we are treating programs as **gradients**
  - Want to converge on this gradient's **global minima**



Closing the Distance

- By minimizing distance, we are treating programs as **gradients**
  - Want to converge on this gradient's **global minima**

- **Problem:** programs are spaghetti code
  - More likely to reach a **local minima** at first
  - **Can get stuck really easily on bad paths**

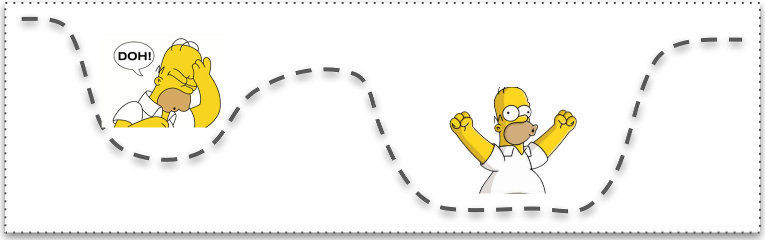


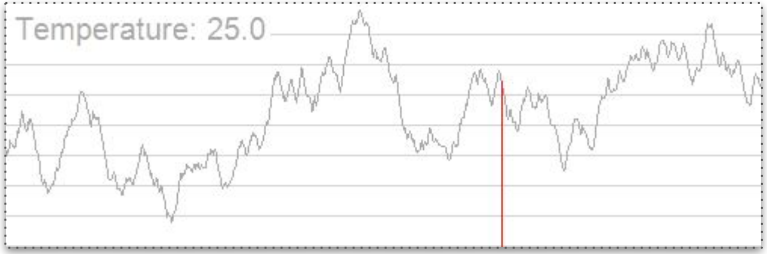
Closing the Distance

- By minimizing distance, we are treating programs as **gradients**
  - Want to converge on this gradient’s **global minima**

- **Problem:** programs are spaghetti code
  - More likely to reach a **local minima** at first
  - **Can get stuck really easily on bad paths**

- Solution: **simulated annealing**
  - Mutate candidate inputs at random
  - Eventually converge on global minima

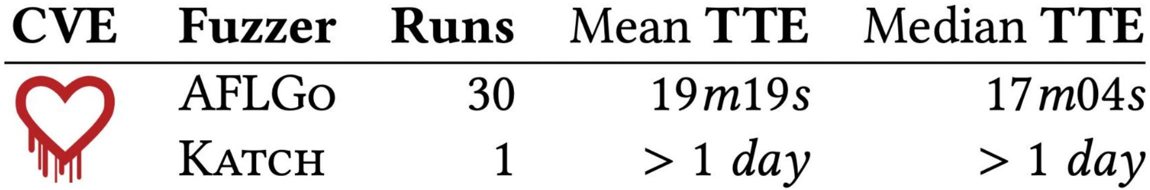




Simulated annealing for a global **maxima**

Results

- Unsurprisingly, **significantly faster** than Directed Symbolic Execution
  - **Cool finding:** able to reproduce the HeartBleed bug in 20 minutes!



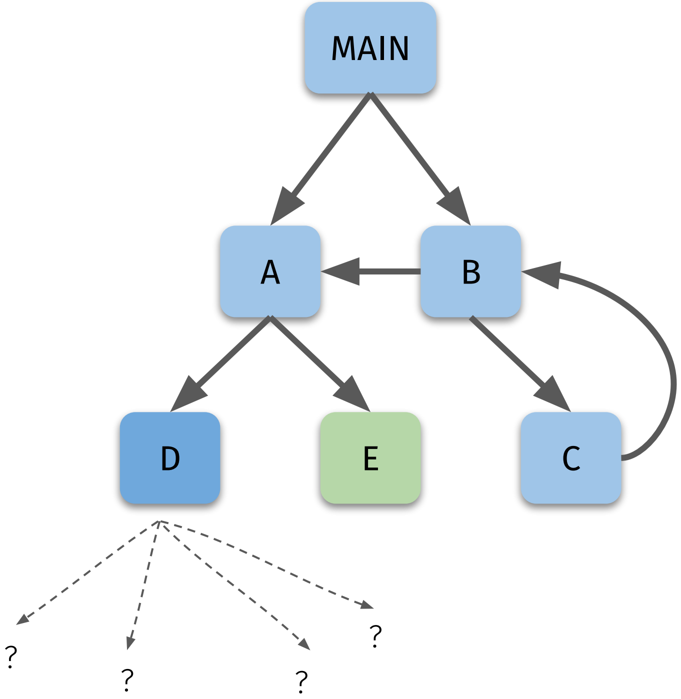
CVE	Fuzzer	Runs	Mean TTE	Median TTE
♥	AFLGo	30	19*m*19*s*	17*m*04*s*
	Katch	1	> 1 *day*	> 1 *day*

Figure 3: Time-to-Exposure (TTE), AFLGo versus Katch.

Source: Directed Greybox Fuzzing

Problem: Indirect Control Flow

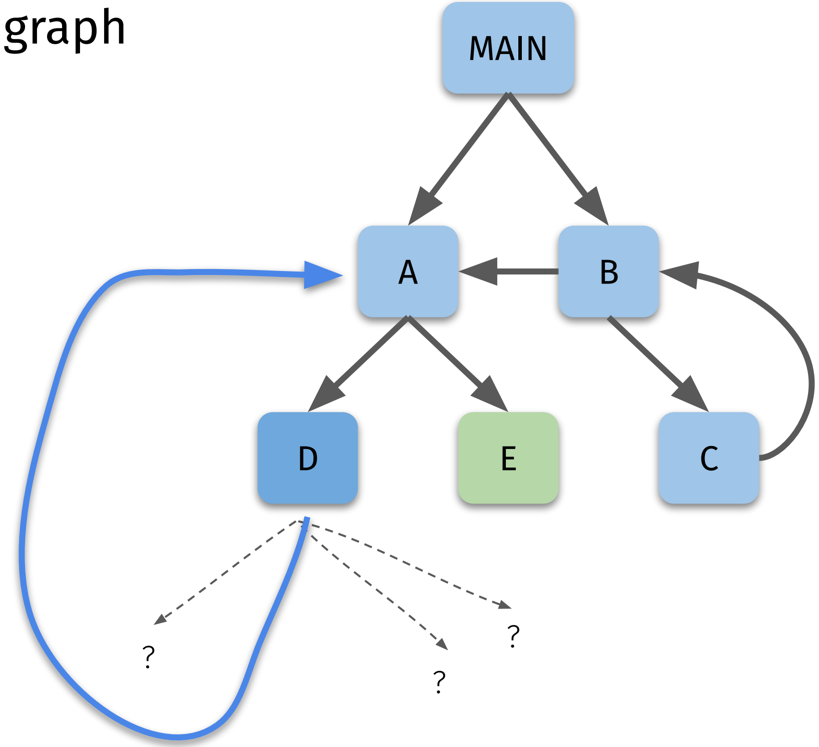
- **Indirect control-flow edges:**
  - E.g., `CALL $R1`, `JMP $R1`
- **Cannot be recovered statically**
  - Destinations resolved only at runtime
  - General case is undecidable
  - **Potentially miss shorter paths**



Problem: Indirect Control Flow

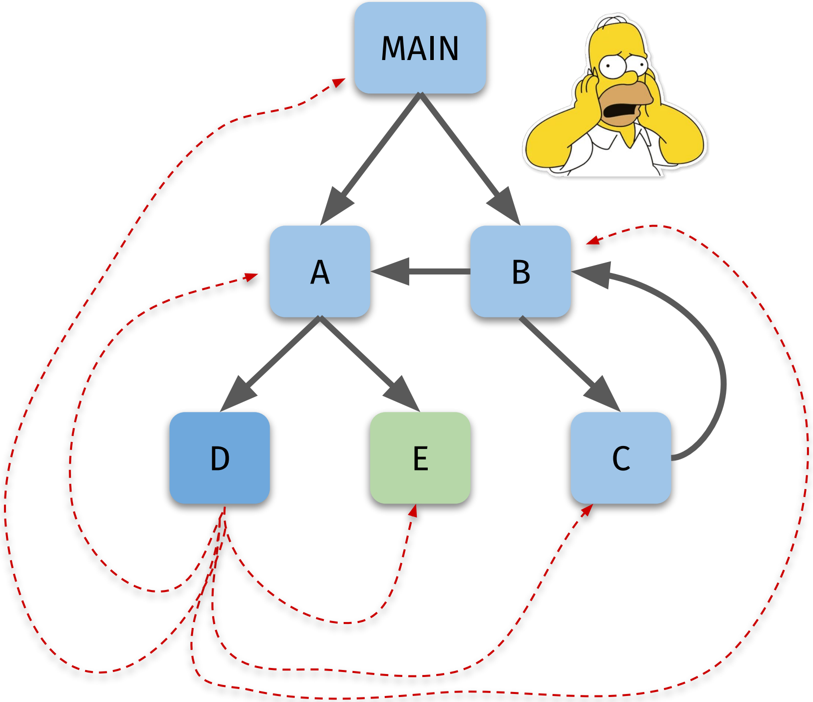
- **Solution 1:** dynamic control-flow graph
  - Initialize CFG with whatever edges are obtainable statically
  - As fuzzing continues, incorporate indirect edges as they are covered

- **Trade-offs:**
  - Higher runtime overhead
    - Tracking, bookkeeping
  - Only considers seen paths
    - CFG still incomplete



Problem: Indirect Control Flow

- **Solution 2:** value set analysis
  - Statically determine possible values that flow into all indirect calls, jumps
- **Trade-offs:**
  - Very high analysis cost
    - Enumerate all instructions
    - Track all memory accesses
  - Most severely over-approximate
    - E.g., *D*'s set may be *all* functions



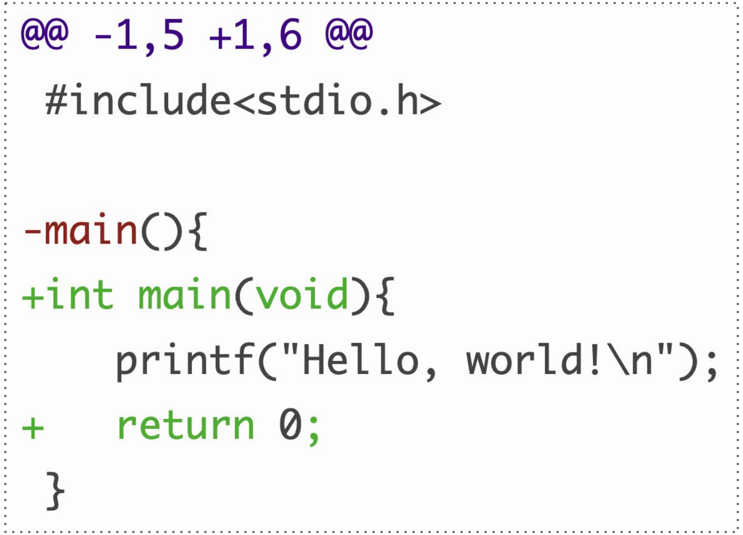
Questions?

Bug-tailored Directed Fuzzing

Motivation

- Sometimes must fuzz **multiple targets**
  - E.g., patch-changed source lines
  - E.g., reproducing specific bugs

- General-purpose directed fuzzing
  - Distances relative to these sites
  - **No ranking or sequential order**
    - Tries to reach all sites at once

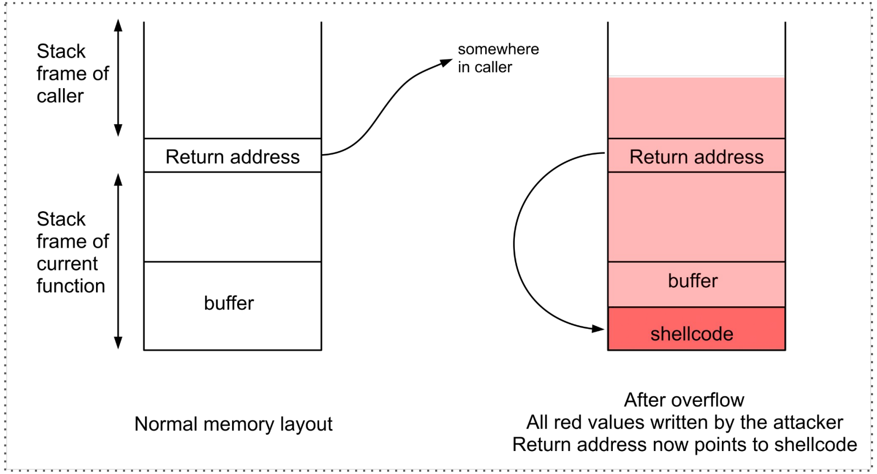


```
@@ -1,5 +1,6 @@
 #include<stdio.h>

-main(){
+int main(void){
     printf("Hello, world!\n");
+    return 0;
 }
```

Recap: “Spatial” Memory Safety

- **Spatial** = relating to **occupying space**
- **Spatial memory safety** violations
  - Buffer overflows
  - Heap overflows
  - Underflows
  - Invalid reads/writes
  - Uninitialized data
  - …
- Directed fuzzing on **limited target set**



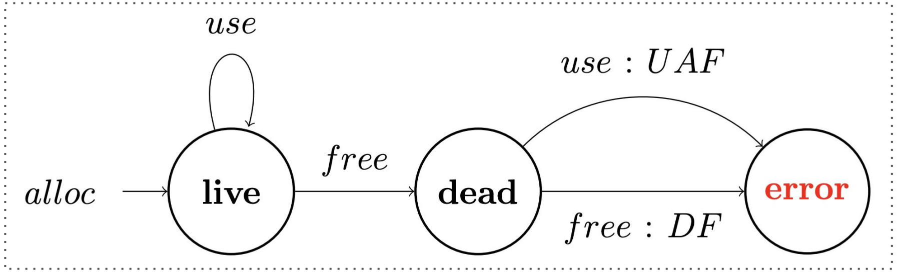
Normal memory layout

After overflow
All red values written by the attacker
Return address now points to shellcode

Recap: "Temporal" Memory Safety

- **Temporal** = relates to **time**

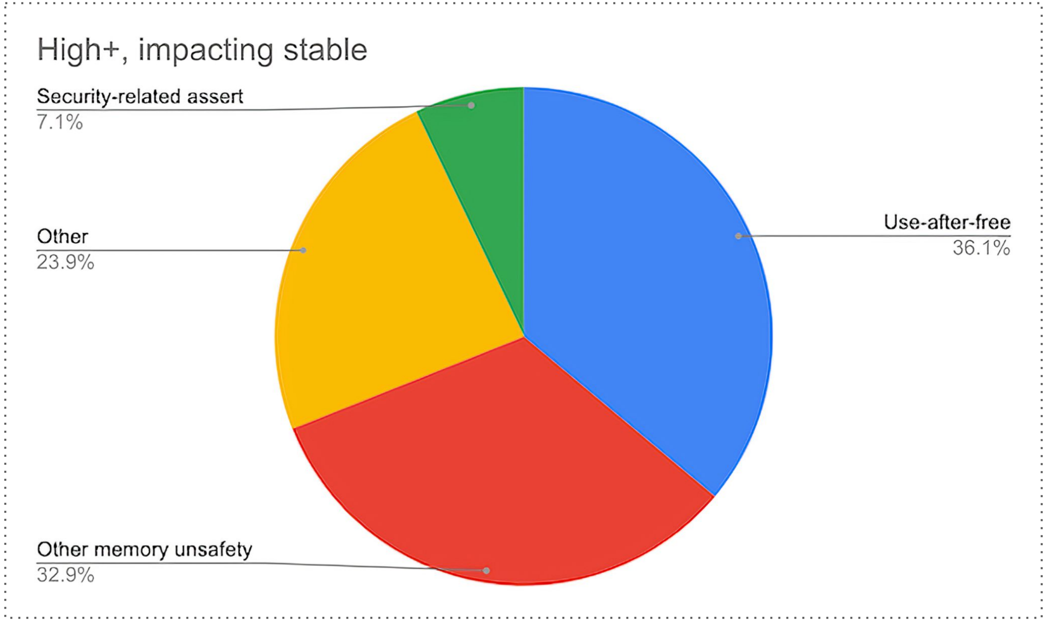
- **Temporal memory safety** violations
  - Dangling pointers
    - Heap use-after-free (UAF)
    - Double free (DF)



- Requires a **sequence of events**
  - Thus, must fuzz **multiple targets in order**

Recap: Use-After-Frees (UAFs)

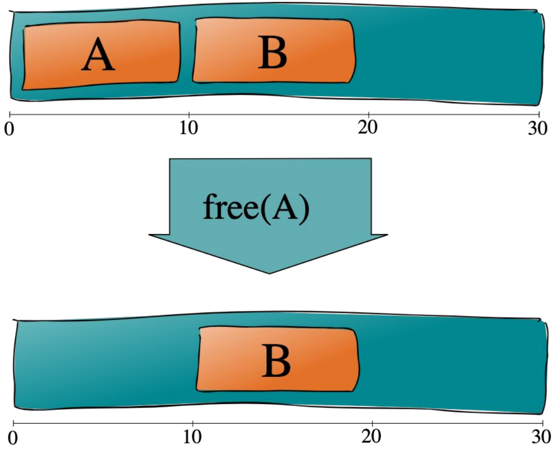
- **Over one third** of Chromium vulnerabilities



Source: https://www.chromium.org/Home/chromium-security/memory-safety/

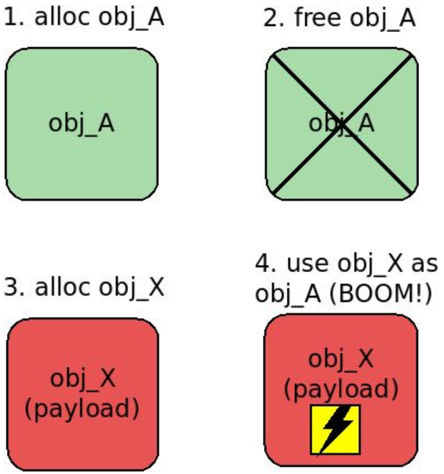
A (crash) course on UAFs

- The Heap = **dynamically**-allocated memory
  - Allocated via **malloc()**, and freed via **free()**
  - Chunks may get allocated, freed, split, coalesced
  - Regions accessed via **pointers**
- Management is **programmer's job**
  - Pointers must point to **live objects**
  - Must point to objects of the **right type**
  - Only pointers to **functions** can be executed
  - …



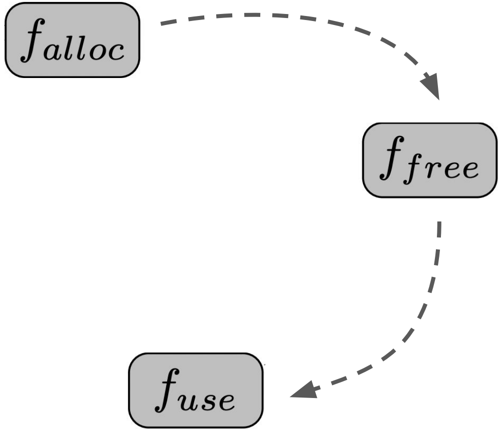
A (crash) course on UAFs

- Are use-after-frees exploitable?
  - Overwrite a free'd chunk
    - Leak information
    - Redirect execution
    - Type confusion
    - **Other evil things**
  - Short answer: **very much so!**



Fuzzing for UAFs

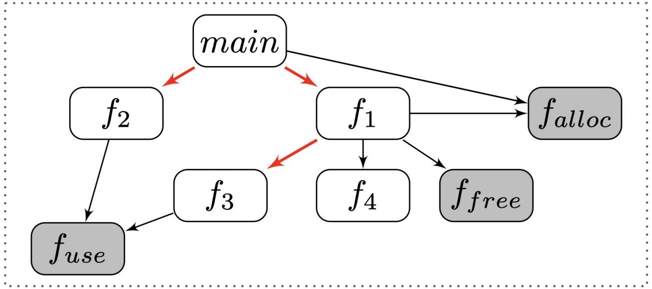
- **What call sequence is required for a UAF?**
  - An object **allocation** (e.g., `malloc()`)
  - A **`free()`** of that same object
  - A **use** (dereference) of that same object
    - E.g., calling a function pointer

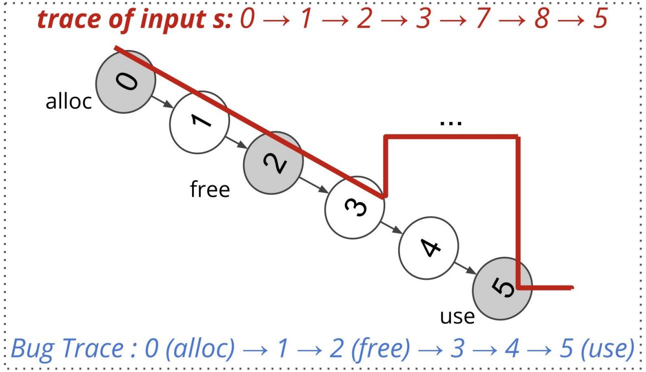


Directed Fuzzing for UAFs

- **What call sequence is required for a UAF?**
  - An object **allocation** (e.g., `malloc()`)
  - A **`free()`** of that same object
  - A **use** (dereference) of that same object
    - E.g., calling a function pointer

- **Pick inputs that *match* this call sequence**
  - Mine their locations statically
  - **Pick inputs that hit them in order**





trace of input s: *0 → 1 → 2 → 3 → 7 → 8 → 5*

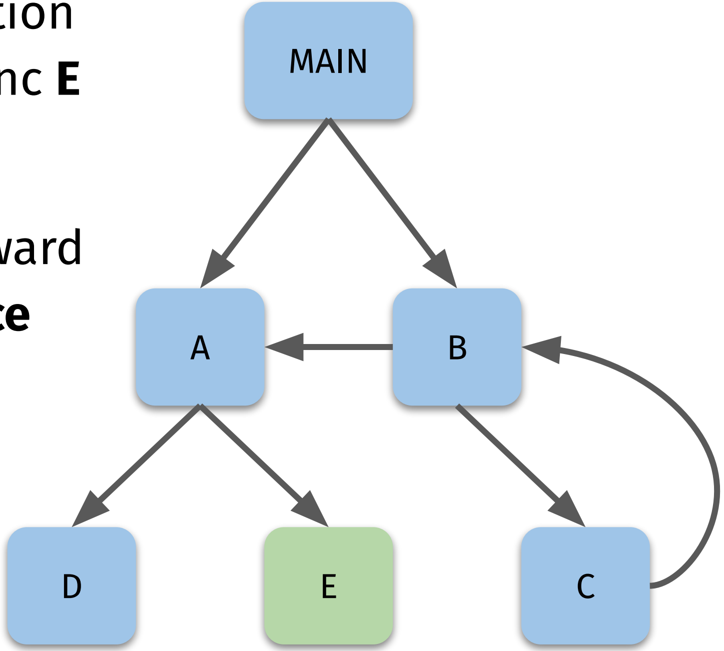
Bug Trace : 0 (alloc) → 1 → 2 (free) → 3 → 4 → 5 (use)

Source: https://i.blackhat.com/USA-20/Thursday/us-20-Bardin-About-Directed-Fuzzing-And-Use-After-Free-How-To-Find-Complex-And-Silent-Bugs.pdf

Sequence Awareness

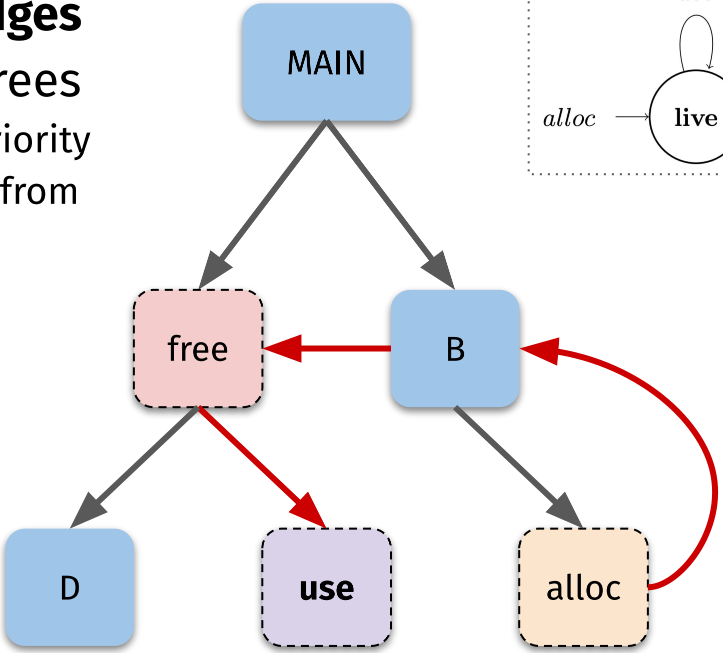
- **AFL-Go:** biases exploration toward single target func **E**
  - **No sequential ordering**

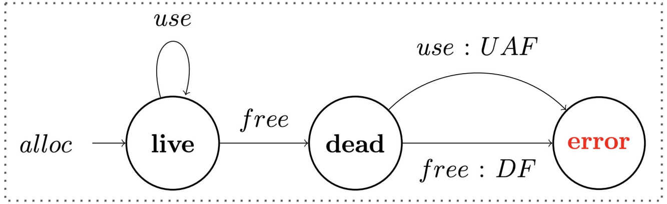
- For UAFs, must bias toward hitting correct **sequence**



Sequence Awareness

- **Solution: weight the edges** between allocs, uses, frees
  - Small weights = more priority
  - Bias the fuzzer to move from one state to the other

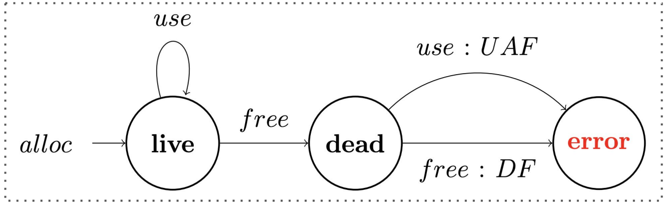


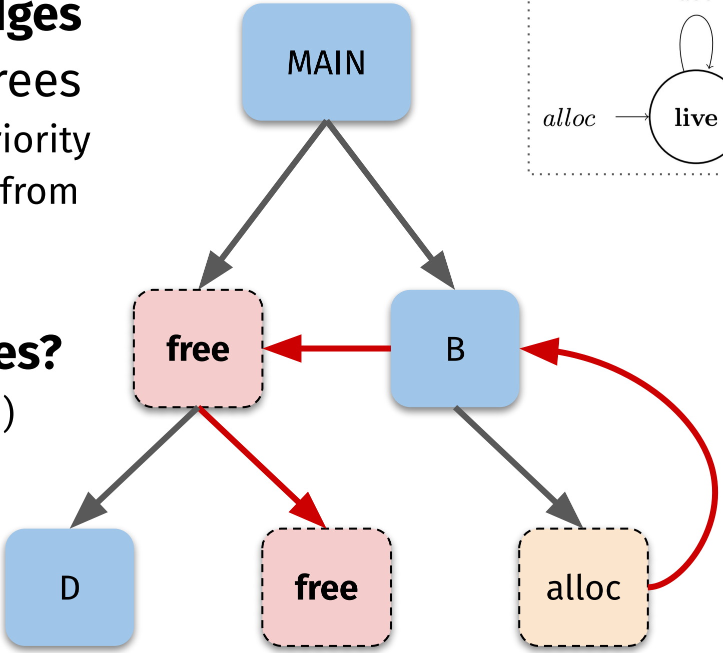


Sequence Awareness

- **Solution: weight the edges** between allocs, uses, frees
  - Small weights = more priority
  - Bias the fuzzer to move from one state to the other

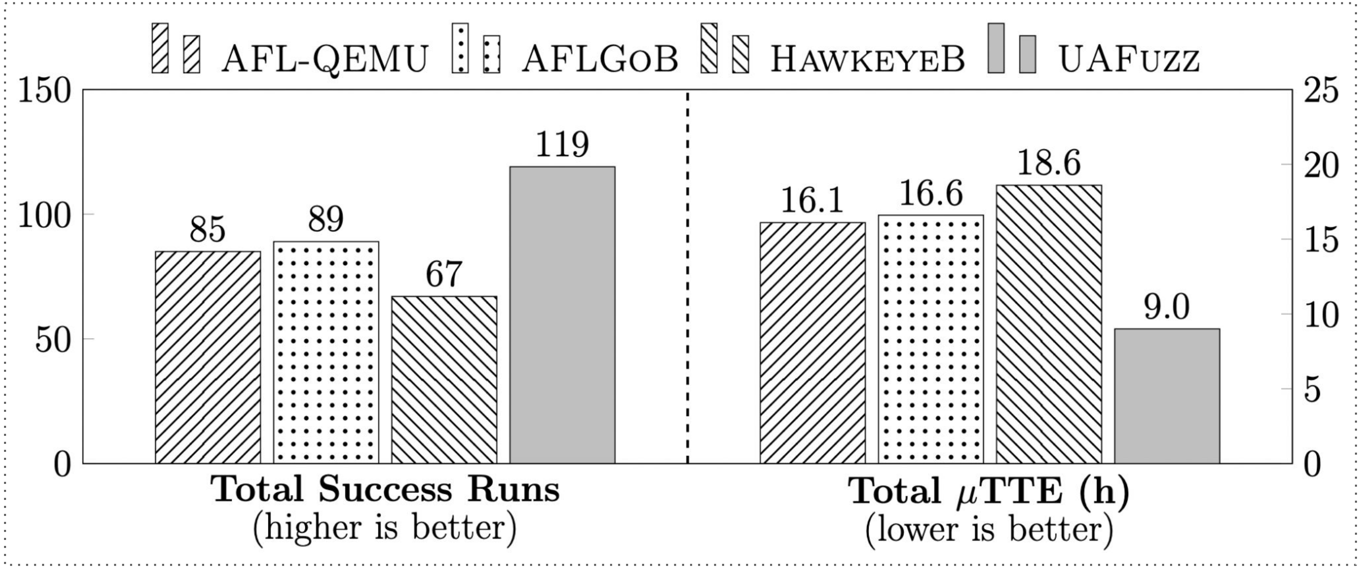
- **What about double frees?**
  - Just hit a second `free()`





Results

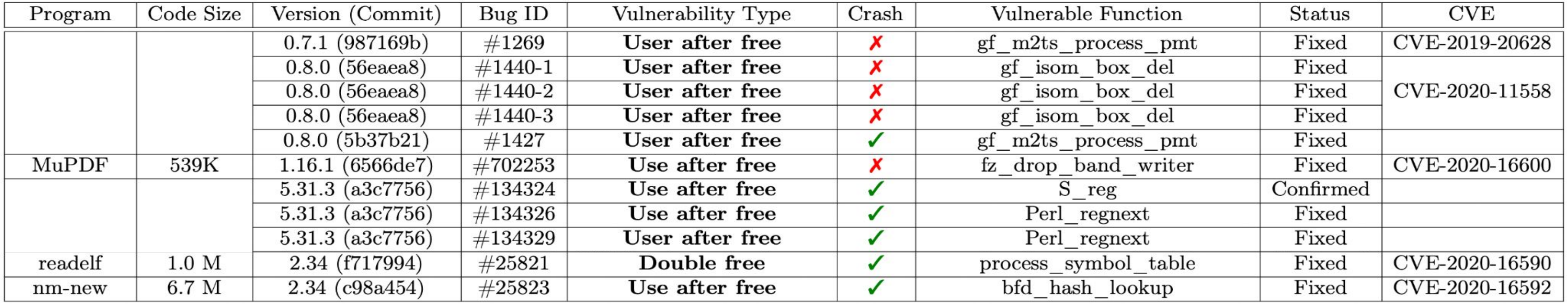
- **UAFuzz:** binary-level fuzzer for use-after-frees



Source: https://i.blackhat.com/USA-20/Thursday/us-20-Bardin-About-Directed-Fuzzing-And-Use-After-Free-How-To-Find-Complex-And-Silent-Bugs.pdf

Results

- **UAFuzz:** binary-level fuzzer for use-after-frees



Program	Code Size	Version (Commit)	Bug ID	Vulnerability Type	Crash	Vulnerable Function	Status	CVE
		0.7.1 (987169b)	#1269	**User after free**	✗	gf_m2ts_process_pmt	Fixed	CVE-2019-20628
		0.8.0 (56eaea8)	#1440-1	**User after free**	✗	gf_isom_box_del	Fixed	CVE-2020-11558
		0.8.0 (56eaea8)	#1440-2	**User after free**	✗	gf_isom_box_del	Fixed	
		0.8.0 (56eaea8)	#1440-3	**User after free**	✗	gf_isom_box_del	Fixed	
		0.8.0 (5b37b21)	#1427	**User after free**	✓	gf_m2ts_process_pmt	Fixed	
MuPDF	539K	1.16.1 (6566de7)	#702253	**Use after free**	✗	fz_drop_band_writer	Fixed	CVE-2020-16600
		5.31.3 (a3c7756)	#134324	**Use after free**	✓	S_reg	Confirmed	
		5.31.3 (a3c7756)	#134326	**Use after free**	✓	Perl_regnext	Fixed	
		5.31.3 (a3c7756)	#134329	**User after free**	✓	Perl_regnext	Fixed	
readelf	1.0 M	2.34 (f717994)	#25821	**Double free**	✓	process_symbol_table	Fixed	CVE-2020-16590
nm-new	6.7 M	2.34 (c98a454)	#25823	**Use after free**	✓	bfd_hash_lookup	Fixed	CVE-2020-16592

- Discovered **many new dangling pointer** vulnerabilities

Source: https://i.blackhat.com/USA-20/Thursday/us-20-Bardin-About-Directed-Fuzzing-And-Use-After-Free-How-To-Find-Complex-And-Silent-Bugs.pdf

Trade-offs

- **The more program introspection, the better**
  - Open-source is **always easier** than closed-source
    - Likely won't scale to many closed-source targets
    - E.g., Microsoft Word
  - **Static analysis becomes very costly**
    - Target identification
    - Distance computation

- **Can this be extended to other bug types?**
  - Yes… if it can be **expressed** as a temporal ordering
    - E.g., heap overflows (allocation + access)
    - Others? **(open research problem)**

Questions?