Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities

Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, Yulei Sui

2019.07.07

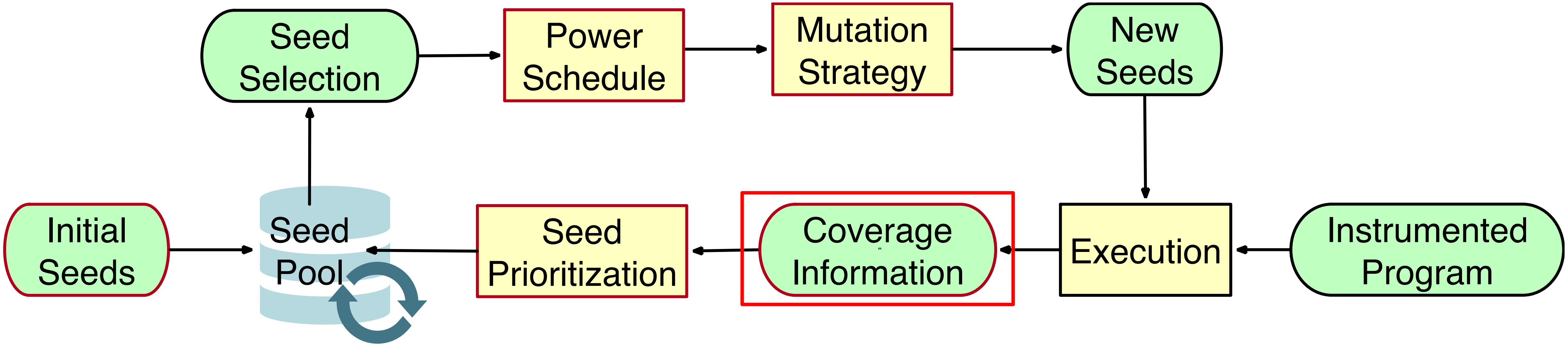
Fuzzing Techniques

- Fuzzing Technique
  - ✓ Automated software testing technique
  - ✓ Provide invalid, unexpected, or random data to a program
  - ✓ The program is instrumented, e.g., branch coverage
  - ✓ The most popular technique to find vulnerabilities
    e.g., AFL, libFuzzer, ClusterFuzz, OSS-Fuzz

- Fuzzing Technique Types
  - ✓ Generation-based fuzzing
    generate inputs from templates, e.g., grammar, specification
  - ✓ Mutation-based fuzzing
    randomly mutate inputs from seed test cases

Challenges in fuzzing techniques

Solutions in Fuzzing



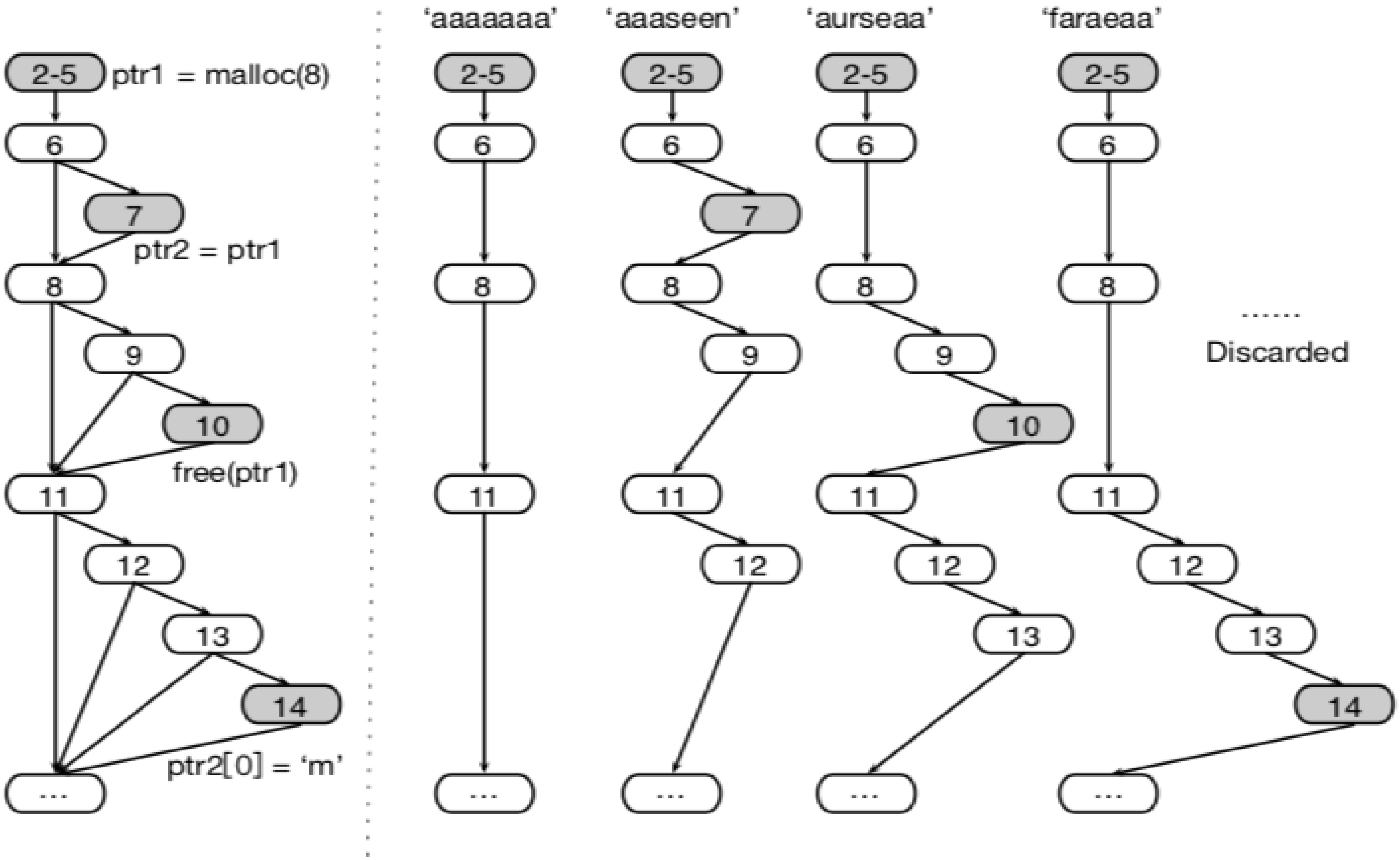
Whether the existing branch coverage is enough?
e.g., use-after-free vulnerability

Challenges in Fuzzing

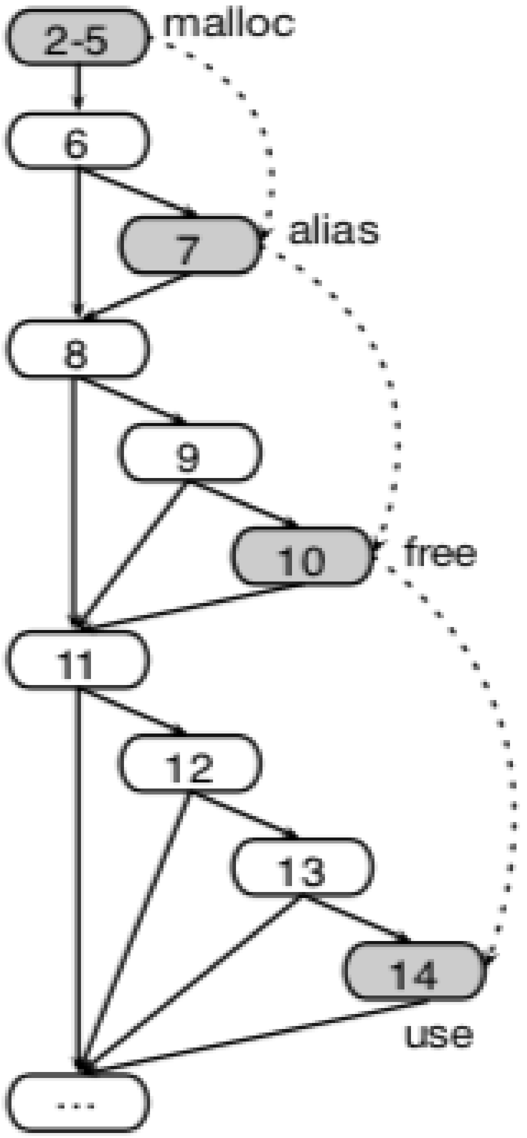
```
1  void main() {
2    char buf[7];
3    read(0, buf, 7)
4    char* ptr1 = malloc(8);
5    char* ptr2 = malloc(8);
6    if(buf[5] == 'e')
7      ptr2 = ptr1;
8    if(buf[3] == 's')
9      if(buf[1] == 'u')
10       free(ptr1);
11   if(buf[4] == 'e')
12     if(buf[2] == 'r')
13       if(buf[0] == 'f')
14         ptr2[0] = 'm';
15   ...
16 }
```

- The program has a use-after-free vulnerability
  - ✓ Line4: allocate the memory
  - ✓ Line7: *ptr1* and *ptr2* become alias
  - ✓ Line10: the memory is freed
  - ✓ Line14: use the freed memory

Challenges in Fuzzing

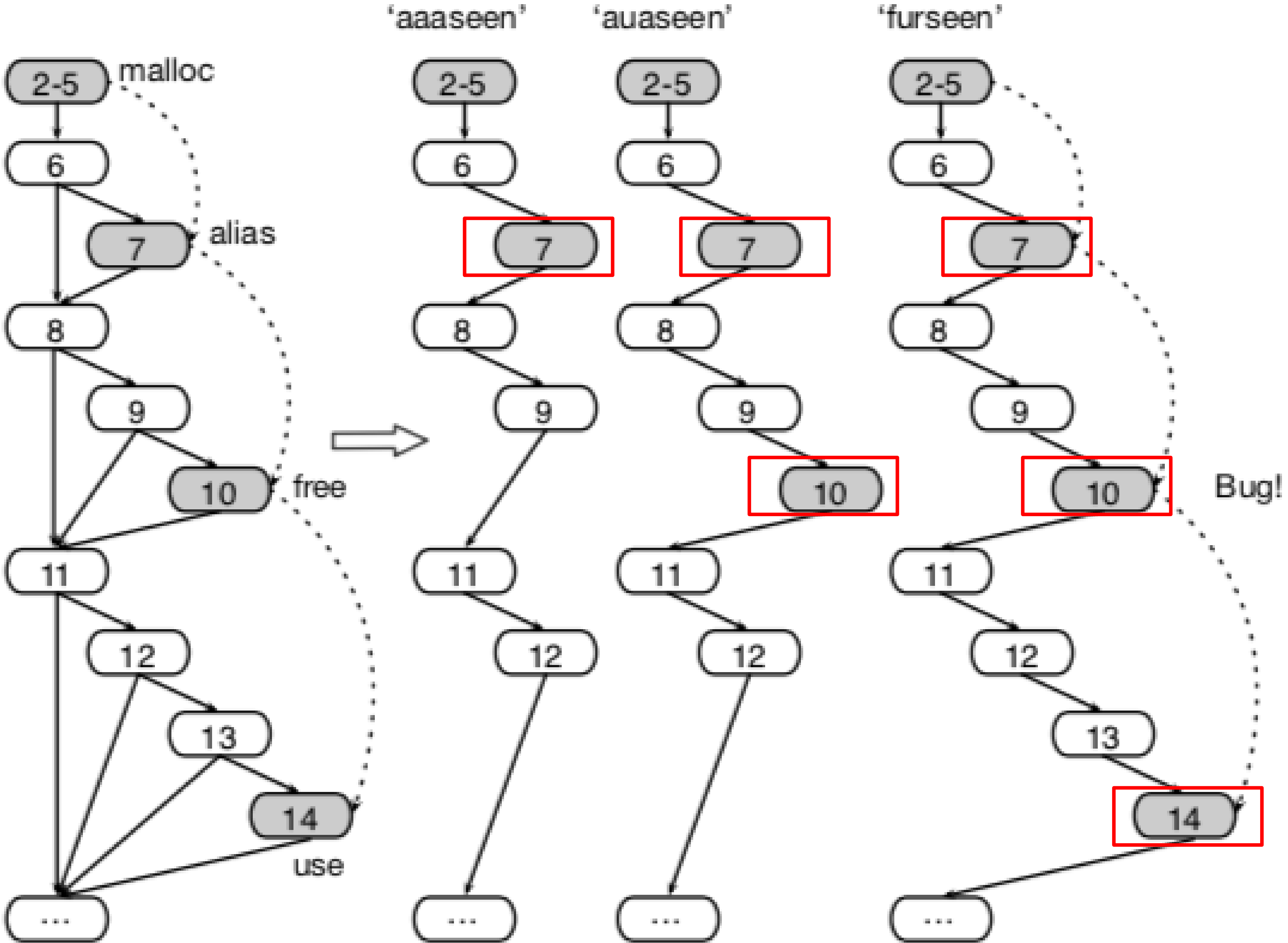


Challenges in Fuzzing



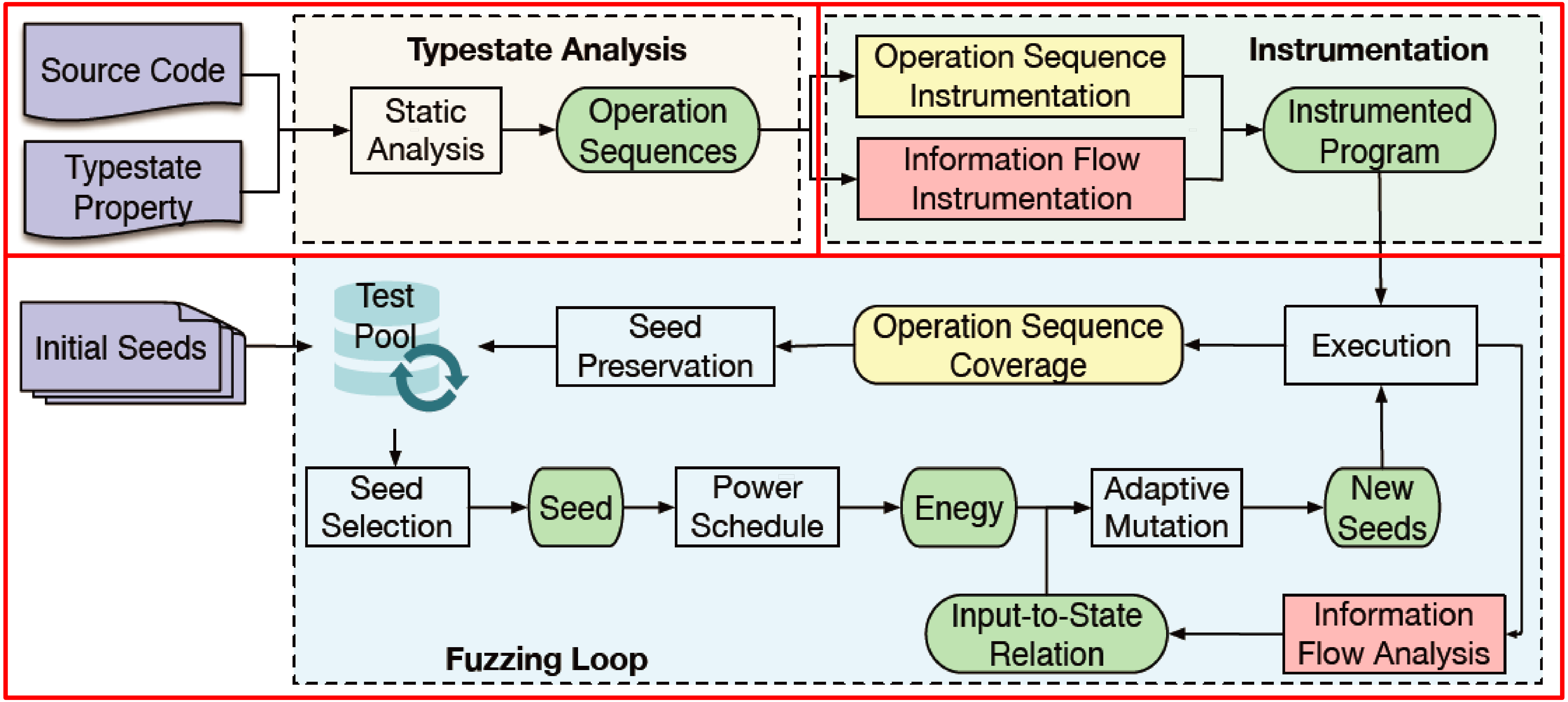
- To expose the use-after-free vulnerability
  - ✓ Line: 4 → 7 → 10 → 14
- Challenges in detecting UaF
  - ✓ How to identify the above operation sequence?
  - ✓ How to cover this operation sequence?

Challenges in Fuzzing



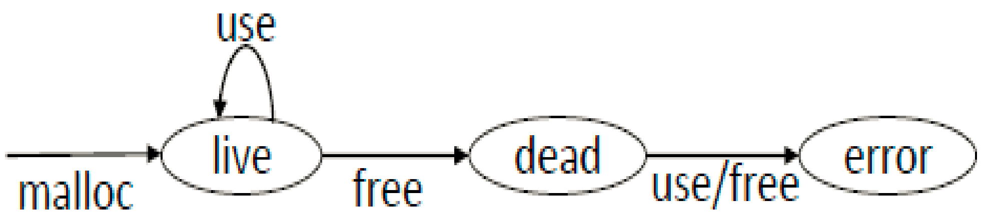
UAFL: Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities

Overview of UAFL

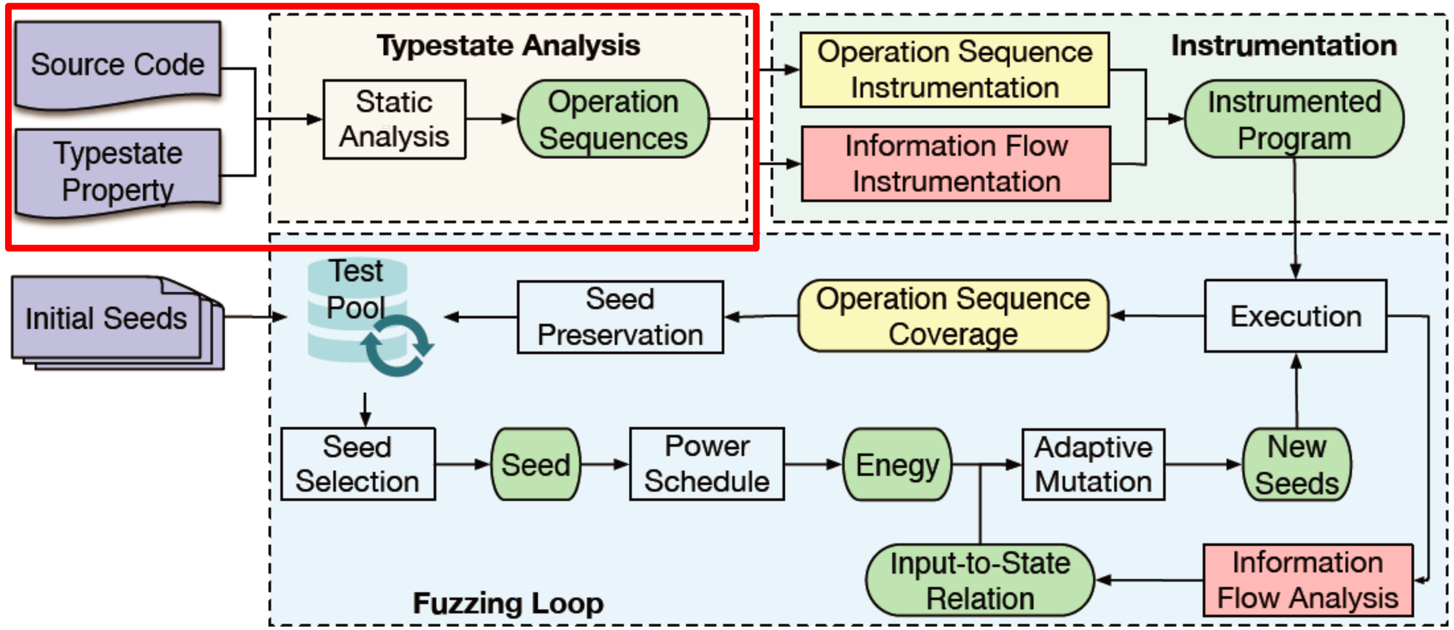


Static Typestate Analysis

- Typtestate property

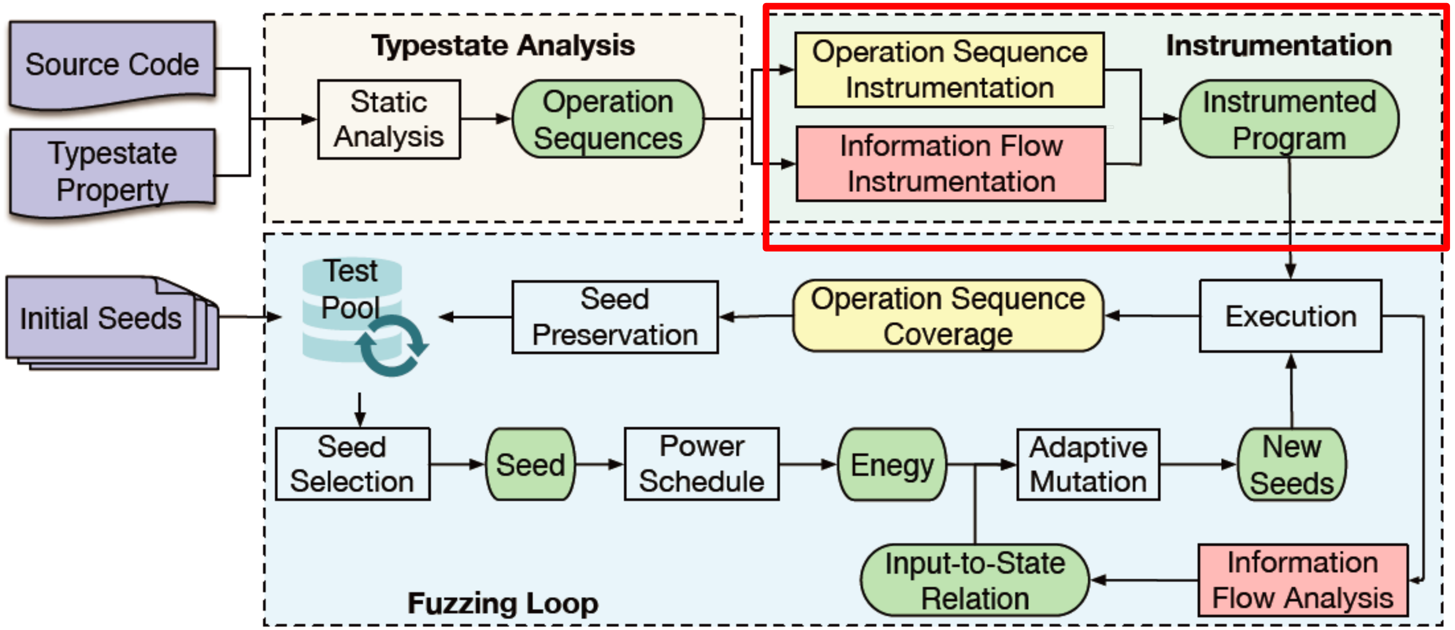


- Static Analysis
  - ✓ Lightweight path-insensitive static analysis
  - ✓ SVF: inter-procedural static value-flow analysis
- Operation Sequence
  - ✓ Line: 4 ⟶ 7 ⟶ 10 ⟶ 14



Program Instrumentation

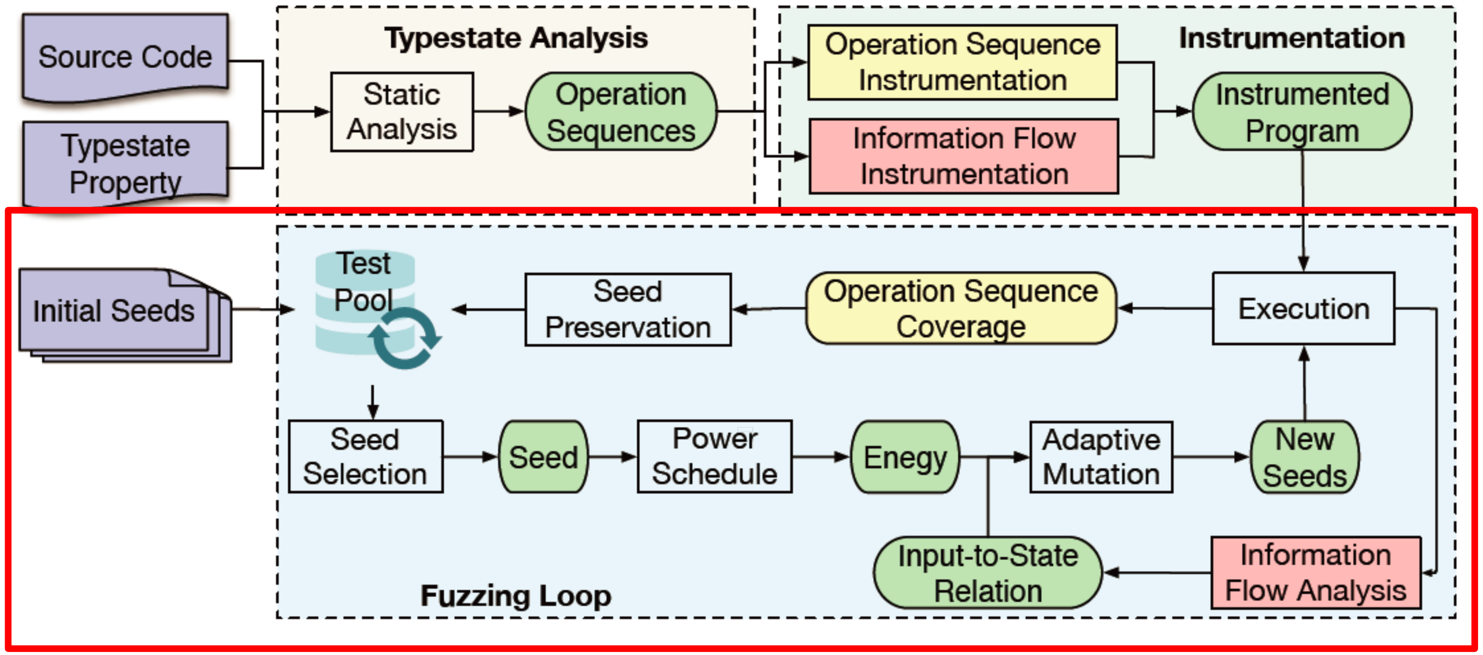
- Operation Sequences
  - ✓ $\forall s_i \in (s_1, \cdots, s_n)$, instrument whether s_i is executed
  - ✓ $\forall s_i \in (s_1, \cdots, s_n)$, retrieve the execution information of $\forall s_j \in (s_1, \cdots, s_i)$



- Information Flow
  - ✓ $IFStrength(x, y, V_x, V_y) = H(x, V_x) - H(x|y, V_x, V_y)$

Fuzzing loop

- Seed Selection
  - ✓ Seeds that cover more operation sequences
- Mutation Strategies
  - ✓ Input fields related to operation sequences are more important
  - ✓ Information flow analysis based mutation
- Feedback
  - ✓ Gradually cover the operation sequences



Evaluations for UAFL

Overhead of Static Analysis

Program	Version	LoC	*T_BB*	BB_{UAF}	BB_{IF}	BB_{Free}	#OS	T(s)
readelf	2.28	1,844k	16,967	2,681 (15.8%)	1,103 (6.5%)	91	41,605	262
readelf	2.31	3,277k	19,973	3,647 (18.2%)	1,555 (7.8%)	98	130,102	508
jpegoptim	1.45	2k	634	36 (5.7%)	28 (4.4%)	5	44	1
liblouis	3.2.0	53k	2,957	486 (16.4%)	190 (6.4%)	8	422	18
lrzip	0.631	19k	9,356	1,051 (11.2%)	467 (5.0%)	6	313	150
Mini XML	2.12	15k	4,237	890 (21.0%)	788 (18.6%)	10	486	44
boringssl	—	162k	22,547	3,701 (16.4%)	3,265 (14.4%)	32	84,069	2,005
GNU cflow	1.6	50k	5,095	1,402 (27.5%)	751 (14.7%)	33	4330	30
Boolector	3.0.0	141k	26,866	11,511 (42.8%)	9,031 (33.6%)	4	28,586	2,387
openh264	1.8.0	143k	12,735	2,090 (16.4%)	927 (7.3%)	1	1,219	1,127
libpff	—	125k	18,569	6,371 (34.3%)	6,041 (32.5%)	60	20,865	122
mjs	1.20.1	40k	4,937	546 (11.0%)	343 (6.9%)	16	1,143	24
ImageMagick	7.0.8	485k	31,190	1,573 (5.0%)	1,336 (4.3%)	3	55,877	2,185
nasm	2.14	101k	13,965	3,812 (27.2%)	3,390 (24.2%)	2	3,357	2,210
Avg.	-	462k	13,573	2,842 (19.2%)	2,087 (13.3%)	26	26,601	1,148

Time to Expose UaF

Program	Vulnerabilities	Time Usage to Expose the Vulnerabilities (hours)							
		UAFL	UAFL_{NoIF}	AFL	AFLFast	FairFuzz	MOpt	Angora	QSYM
readelf-2.28	CVE-2017-6966	**0.59**	1.32	6.09	1.43	0.68	3.61	T/O	6.20
readelf-2.31	CVE-2018-20623	0.10	0.10	0.10	0.10	T/O	0.10	**0.02**	0.10
jpegoptim	CVE-2018-11416	**0.09**	0.10	0.59	0.88	1.08	1.49	T/O	1.95
liblouis	CVE-2017-13741	**1.11**	1.81	15.81	T/O	6.96	17.38	T/O	13.42
lrzip	CVE-2018-11496	**0.01**	0.01	0.01	0.01	0.01	0.01	0.01	0.01
Mini XML	CVE-2018-20592	**0.38**	0.93	1.28	2.59	0.54	16.7	T/O	18.99
boringssl	Google Test-suit	**0.33**	1.06	T/O	T/O	4.67	7.62	–	T/O
GNU cflow	uaf-issue-1	**1.80**	12.21	23.29	T/O	20.02	T/O	T/O	T/O
Boolector	uaf-issue-2	0.83	0.97	1.09	0.82	**0.39**	1.66	–	1.16
openh264	uaf-issue-3	**8.17**	13.00	15.80	11.15	8.17	15.39	T/O	18.45
libpff	uaf-issue-4	**1.39**	1.39	4.21	4.11	3.98	4.35	T/O	4.98
mjs	uaf-issue-5	**1.21**	1.23	3.10	3.02	1.45	4.6	T/O	6.71
ImageMagick	uaf-issue-6	**6.29**	13.92	T/O	T/O	T/O	T/O	T/O	T/O
nasm	CVE-2018-19216	**2.59**	4.69	8.32	3.45	2.86	11.46	2.75	9.64
	CVE-2018-20535	**17.03**	T/O	T/O	T/O	T/O	T/O	T/O	T/O
Missed Vulnerabilities		0	1	3	5	3	3	10	4
Avg. Time Usage		2.79 ± 0.32	5.12 ± 0.32	10.11	9.84	8.18	10.42	18.67	11.84
UAFL's Speedup		–	1.75×	3.25×	3.16×	2.63×	3.35×	6.00×	3.80×
UAFL_{NoIF}'s Speedup		–	–	1.86×	1.81×	1.50×	1.92×	3.43×	2.18×

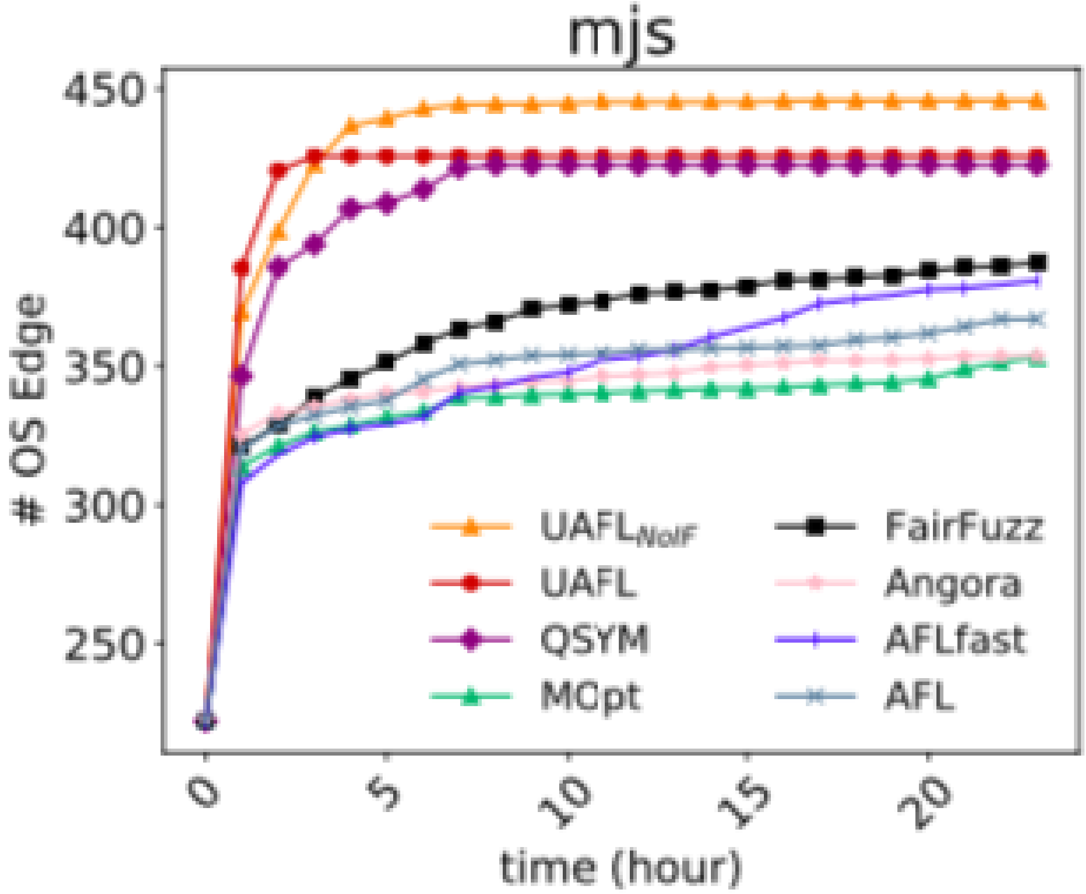
* T/O means the fuzzer cannot discover vulnerabilities within 24 hours across 8 runs. When we calculate the average time usage, we replace T/O with 24 hours.
* Angora does not work on the programs *boringssl* and *Boolector*, denoted by '-', because it throws the exceptions during the instrumentation.

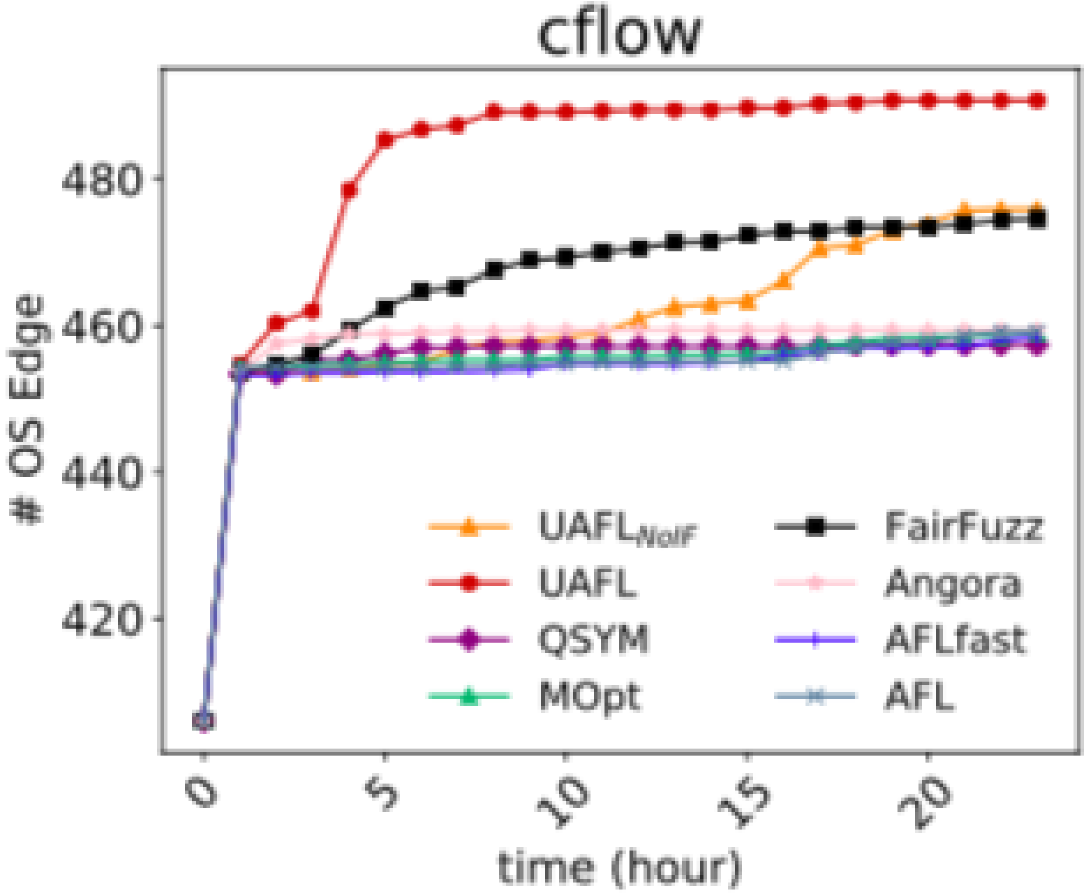
Statistical Tests

Program	Vulnerability	$\hat{A}_{12}$ (UAFL)						$\hat{A}_{12}$ (UAFL$_{NoIF}$)					
		AFL	AFLFast	FairFuzz	MOpt	Angora	QSYM	AFL	AFLFast	FairFuzz	MOpt	Angora	QSYM
readelf	CVE-2017-6966	**0.906**	**0.898**	**1.000**	0.609	**1.000**	**0.968**	**0.796**	0.546	**1.000**	0.453	**1.000**	**0.828**
readelf	CVE-2018-20623	0.500	0.500	0.500	0.500	**0.000**	0.500	0.500	0.500	0.500	0.500	**0.000**	0.500
jpegoptim	CVE-2018-11416	**0.995**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**0.995**	**1.0**	**1.000**	**1.000**	**1.000**	**1.000**
liblouis	CVE-2017-13741	**0.828**	**0.937**	**0.851**	**1.000**	**1.000**	**0.984**	**0.875**	**0.937**	**0.867**	**1.000**	**1.000**	**0.968**
lrzip	CVE-2018-11496	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500
Mini XML	CVE-2018-20592	**0.968**	**1.000**	**0.812**	**1.000**	**1.000**	**1.000**	0.617	**0.929**	**0.750**	**0.781**	**1.000**	**0.781**
boringssl	Google Test-suit	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**0.828**	**1.000**	**1.000**	**1.000**
GNU cflow	uaf-issue-1	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**0.937**	**0.968**	0.609	**0.968**	**1.000**	**1.000**
Boolector	uaf-issue-2	0.720	**1.000**	**0.030**	**0.880**	**1.000**	**0.780**	0.620	**1.000**	**0.020**	**0.820**	**1.000**	0.720
openh264	uaf-issue-3	**0.937**	**0.781**	**0.150**	**1.000**	**1.000**	**1.000**	0.687	0.359	**0.031**	0.640	**1.000**	**0.875**
libpff	uaf-issue-4	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
mjs	uaf-issue-5	**0.980**	**0.980**	0.590	**1.000**	**1.000**	**1.000**	**0.880**	**0.890**	0.604	0.987	**1.000**	**0.987**
ImageMagick	uaf-issue-6	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
nasm	CVE-2018-19216	0.960	0.600	0.560	**0.920**	0.600	0.800	**0.800**	0.319	0.280	**0.840**	0.280	0.800
	CVE-2018-20535	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	0.500	0.500	0.500	0.500	0.500	0.500
Significant better ($\hat{A}_{12} > 0.71$, bold)		11/15	12/15	9/15	12/15	12/15	12/15	9/15	9/15	7/15	9/15	11/15	10/15

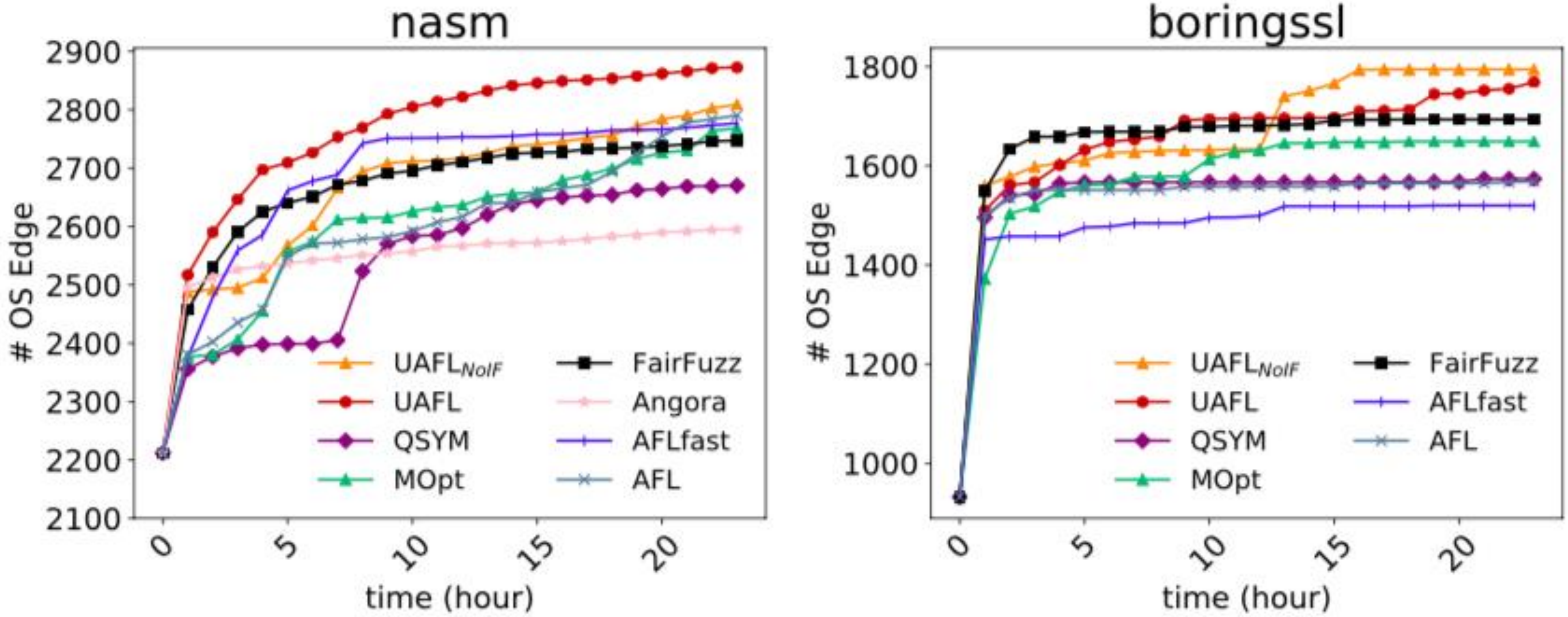
* We highlight the $\hat{A}_{12}$ in the bold if its corresponding *Mann-Whitney U* test is significant.

Operation Sequence Coverage

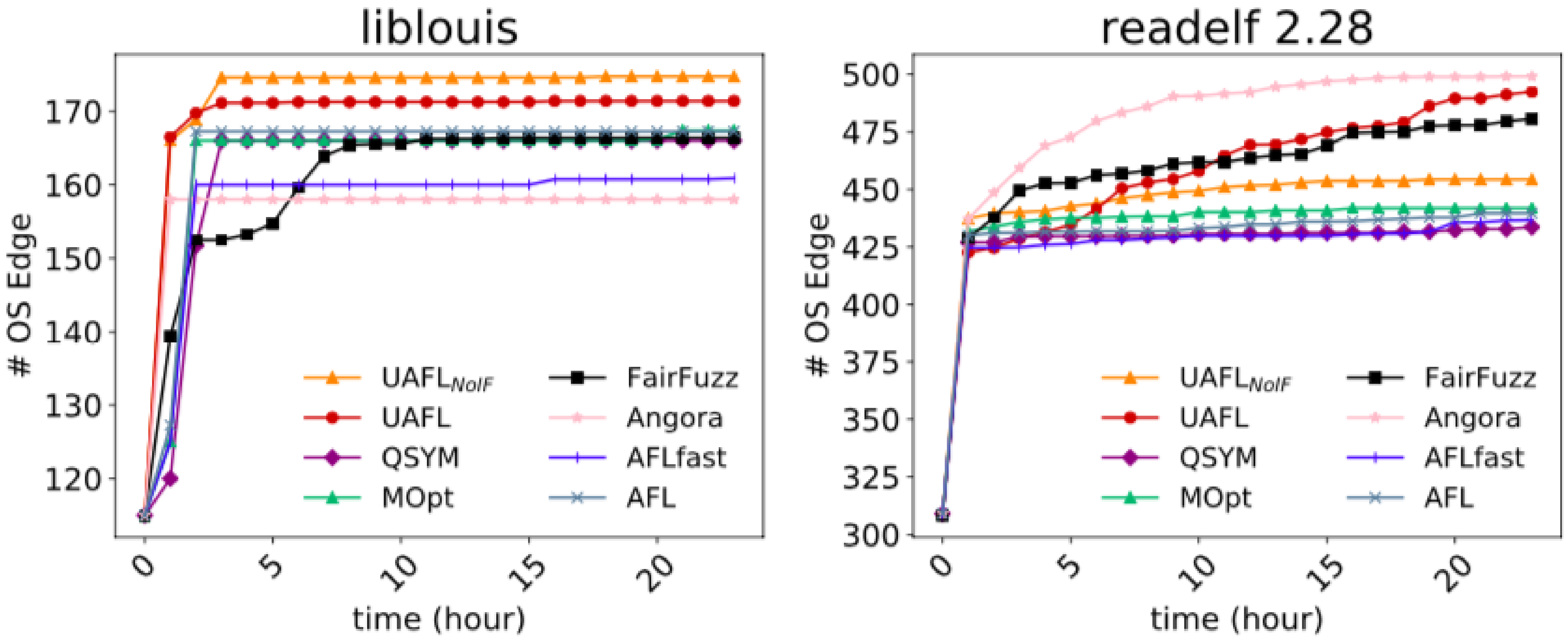




Operation Sequence Coverage



Operation Sequence Coverage



Thanks!

R&Q?