Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling

Chijin Zhou[1], Mingzhe Wang[1], Jie Liang [1],
Zhe Liu[2], Yu Jiang [1]

[1]School of Software, Tsinghua University, Beijing, China
[2]Computer Science and Technology, NUAA, Nanjing, China

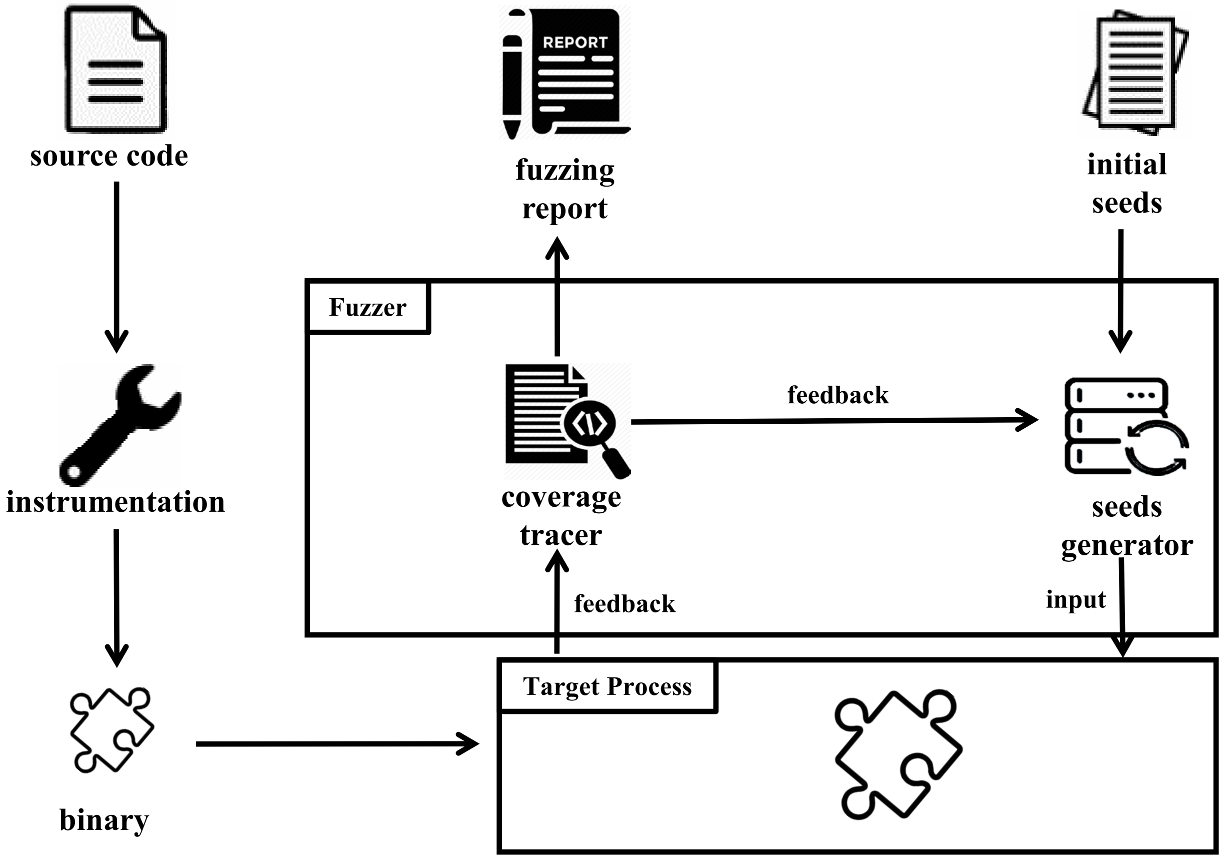
Fuzzing is popular

- Fuzzing is widely used for software vulnerability
  - Project Springfield — Microsoft
  - OSS-Fuzz — Google
    - Has found more than 16,000 bugs

- Some fuzzed projects

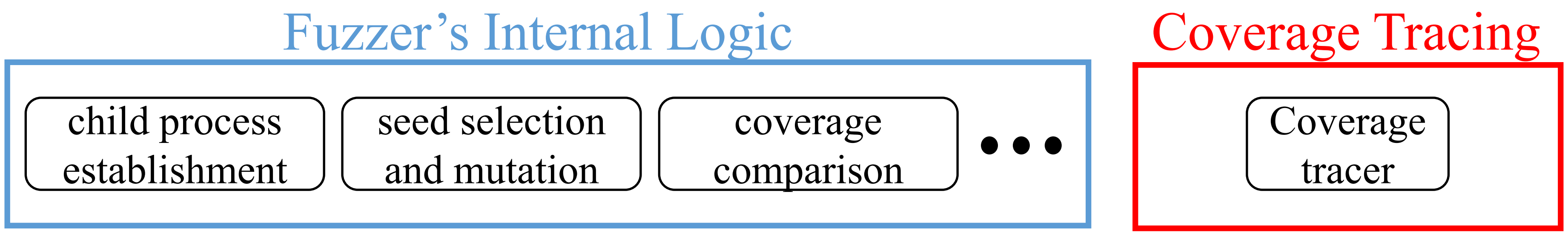
SQLite, chrome, FFmpeg, android, OpenSSL, TensorFlow™, node js, mozilla Firefox

General Workflow of Coverage-guided Fuzzing



Runtime of Coverage-guided Fuzzing

Take American Fuzzy Lop (AFL) as example, the fuzzer's runtime consists of two parts:



Limitation of Coverage-guided Fuzzing

Observation: tracing coverage is costly

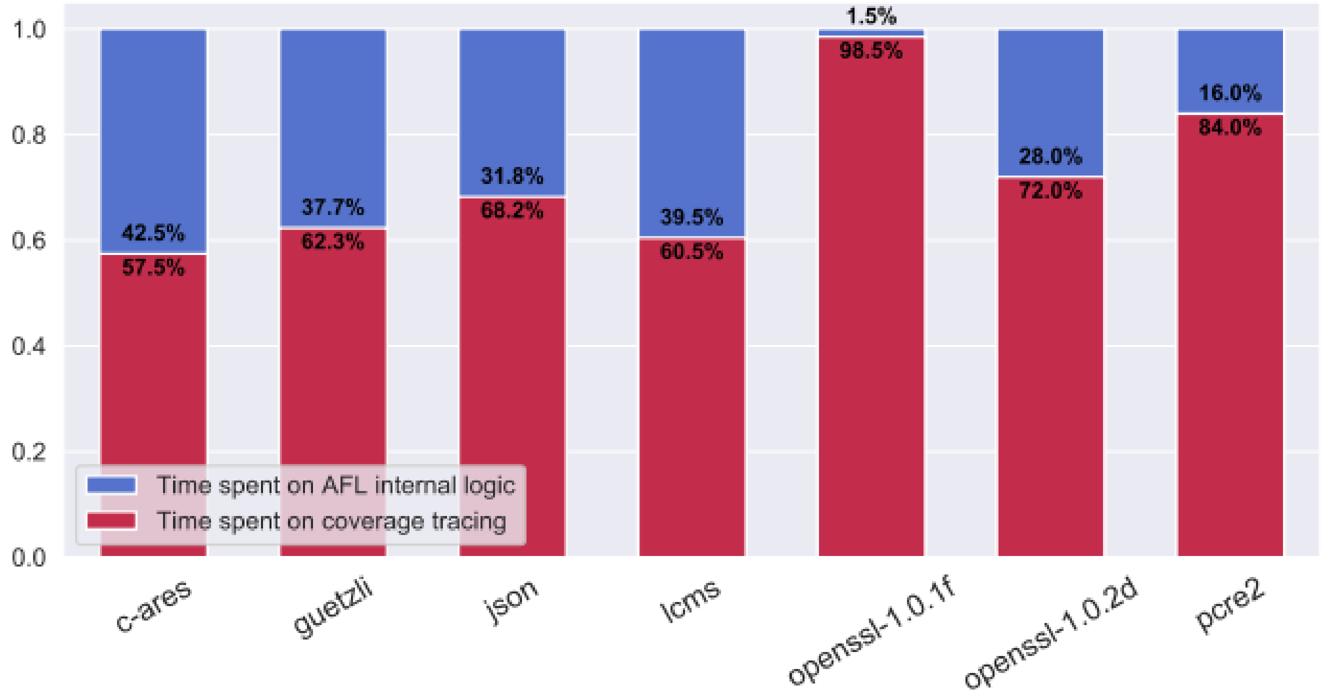
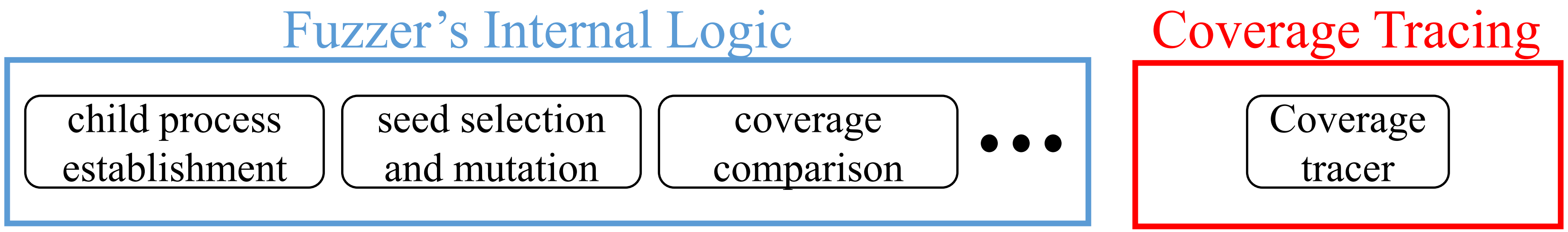


Figure 3: Percentage of internal logic execution time and edge level coverage tracing time in AFL.

AFL spends an average of 71.85% and up to 98.5% of its runtime to trace coverage

Limitation of Coverage-guided Fuzzing

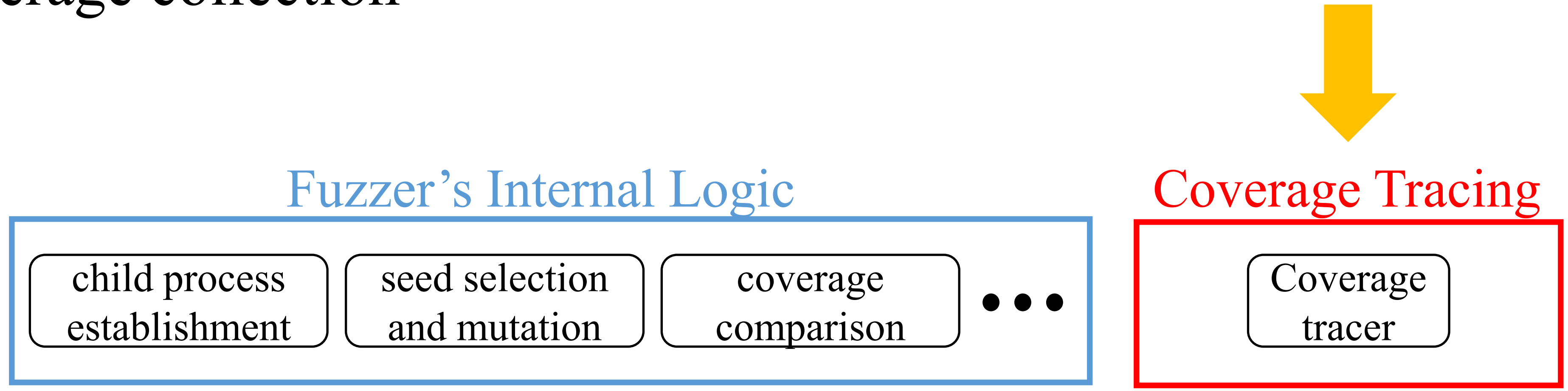
Observation: tracing coverage is costly



Average Cost: 28.15% 71.85%

Focus of This Paper

Target: boost fuzzing speed while preserve fine-grained coverage collection

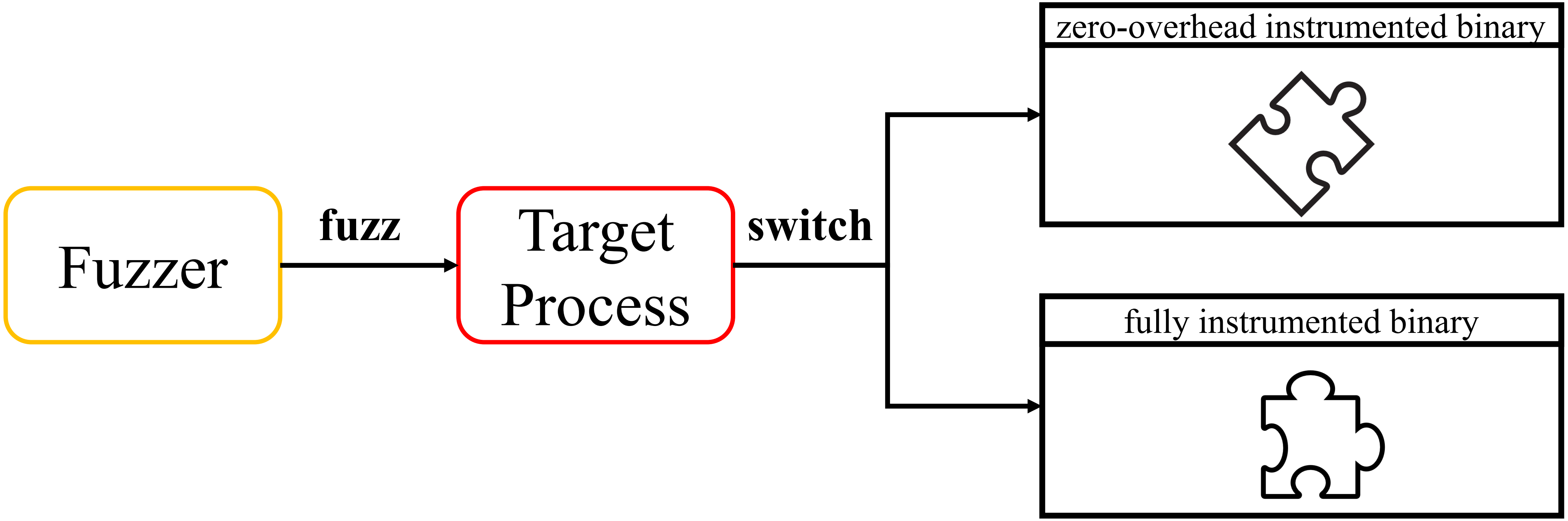


Average Cost: 28.15% 71.85%

Our Solution: Zeror

Main idea:

Switching between diversely-instrumented binaries



Our Solution: Zeror

Main idea:
Switching between diversely-instrumented binaries.

Approaches:

- A self-modifying tracing mechanism to provide a zero-overhead instrumentation for coverage collection
- A real-time scheduling mechanism to support adaptive switch between the zero-overhead instrumented binary and the fully instrumented binary for better vulnerability detection

Self-modifying Tracing

Key insight:

Instrument program in edge level, and dynamically remove visited instrumentation points during fuzzing process

Problems:

(1) How to instrument program

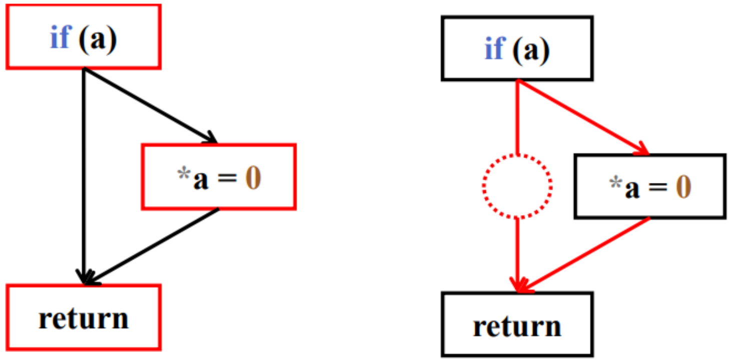
(2) How to remove visited instrumentation points

Self-modifying Tracing

How to instrument program

Solution: add dummy block to critical edge

```
void foo(int *a)
{
  if (a)
    *a = 0;
}
```

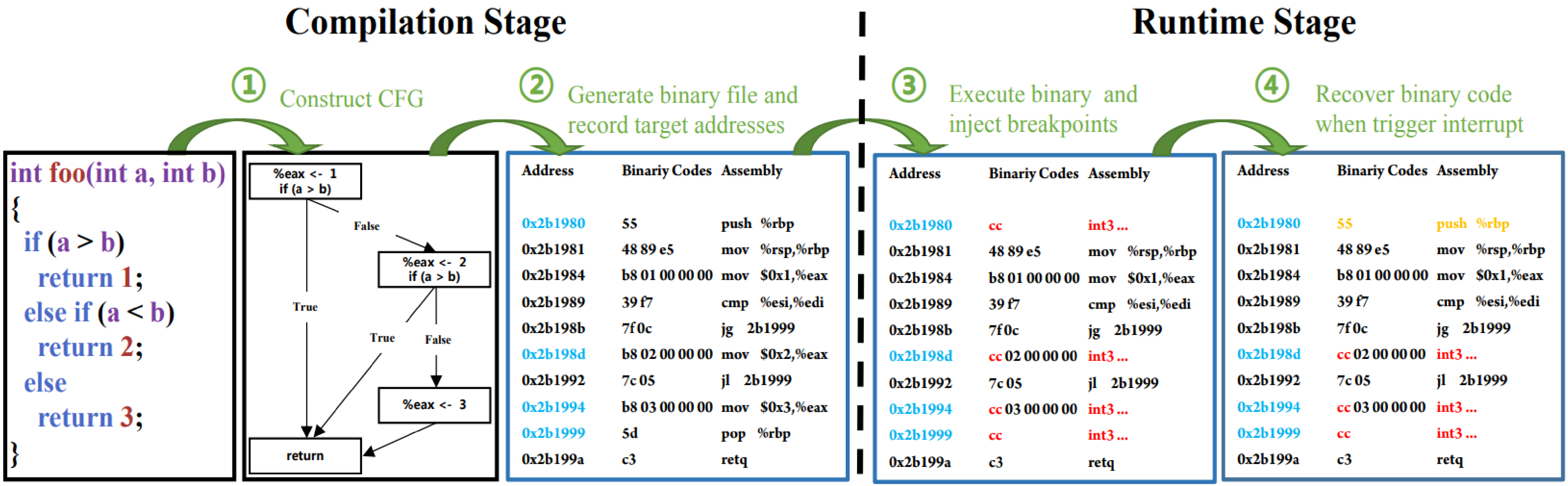


(a) code (b) basic-block level (c) edge level

Self-modifying Tracing

How to remove visited instrumentation points

Solution: self-modifying its instructions during fuzzing process



Binary-switching Scheduling

Key insight:

Estimate fuzzing efficiencies of diversely-instrumented binaries, and switch to high-efficiency binary at set interval

Problem:

How to estimate fuzzing efficiencies

Binary-switching Scheduling

For a binary, the efficiency at time period t is defined as

$$e_t = \frac{I_t}{T_t} = \frac{I_t}{M_t} * \frac{M_t}{T_t} = r_t * s$$

- I_t: the number of interesting seeds
- T_t: time spent on fuzzing
- M_t: the number of executions
- s: average execution speed (constant)

Binary-switching Scheduling

For a binary, the efficiency at time period t is defined as

$$e_t = \frac{I_t}{T_t} = \frac{I_t}{M_t} * \frac{M_t}{T_t} = \boxed{r_t} * s$$

- I_t: the number of interesting seeds
- T_t: time spent on fuzzing
- M_t: the number of executions
- s: average execution speed (constant)

Binary-switching Scheduling

Procedure:

1. Collect statistical data (i.e. the number of interesting seeds, the number of executions and the time spent on fuzzing) of each binary
2. Use empirical Bayesian to estimate interesting rate $\hat{r}_t$
3. Calculate the efficiency of each binary
4. Choose high-efficiency binary as optimal target

Binary-switching Scheduling

Specially, to smooth time-varying observed data, we leverage exponential smoothing to calculate the smoothed number of interesting seeds:

$$I_i = \begin{cases} I'_i & i = 1 \\ \gamma I'_i + (1-\gamma) I_{i-1} & i > 1 \end{cases}$$

Evaluation

1. Efficiency of Zeror

Project	average execution time for each test case (μs)				number of covered branches			
	AFL	AFL+INSTRIM	AFL+Untracer	AFL+Zeror	AFL	AFL+INSTRIM	AFL+Untracer	AFL+Zeror
boringssl	96.69	69.68	N/A	33.05	2661	**2694**	N/A	2549
c-ares	43.34	25.42	13.95	16.32	**57**	**57**	55	**57**
freetype2	44.68	25.17	25.13	20.33	8255	9268	7007	**10059**
guetzli	99.92	67.98	45.80	41.00	4757	4845	4748	**4987**
harfbuzz	149.82	80.36	66.06	55.73	8148	8048	7195	**9168**
json	145.82	100.03	64.33	98.39	1315	1333	1152	**1346**
lcms	97.71	70.92	44.18	63.96	2115	**2244**	1436	2077
libarchive	193.44	112.50	112.90	112.72	1208	1119	1082	**1618**
libjpeg	1469.47	668.96	261.30	337.36	2364	2564	2399	**2857**
libpng	15.34	5.48	5.27	7.54	1092	1096	1029	**1140**
libssh	638.00	340.52	309.62	309.29	**867**	**867**	**867**	**867**
libxml2	268.07	135.05	N/A	88.13	4063	4318	N/A	**4745**
llvm-libcxxabi	137.61	81.61	43.75	42.04	6488	6005	6000	**7012**
openssl-1.0.1f	3418.66	1998.27	N/A	1948.43	4748	6745	N/A	**7372**
openssl-1.0.2d	161.09	92.48	N/A	63.23	1825	**1828**	N/A	1769
openssl-1.1.0c	210.70	89.74	N/A	50.60	**1712**	1711	N/A	1658
openthread	145.51	91.17	64.80	85.16	3561	3537	3279	**3591**
pcre2	199.12	102.21	53.86	49.11	**6890**	6888	6597	**6890**
proj4	23.22	14.24	8.47	7.86	2541	2584	2347	**3886**
re2	640.24	391.97	260.19	235.40	4608	4647	4533	**4725**
sqlite	221.18	160.84	136.01	141.40	1892	**1997**	1986	1972
vorbis	96.14	58.08	36.45	25.48	2035	**2152**	1817	2079
woff2	31.55	20.12	11.80	8.67	2119	2152	1453	**2157**
wpantund	1921.02	2019.62	1544.89	1789.23	7959	7892	7802	**8781**
Zeror improvement	**+159.80%**	**+50.70%**	**-0.46%**		**+10.14%**	**+6.82%**	**+20.84%**	

Evaluation

1. Efficiency of Zeror

Table 3: Time to expose known bugs, ∞ denotes the fuzzer cannot expose the known bugs in 6 hours and the projects whose bugs can not be triggered by any fuzzer are removed.

Project	AFL	AFL+INSTRIM	AFL+Untracer	AFL+Zeror
c-ares	**8**	26	842	**8**
guetzli	∞	∞	16257	**6001**
json	**5**	**5**	**5**	**5**
lcms	20679	∞	11827	**10953**
llvm-libcxxabi	788	2197	2347	**709**
openssl-1.0.1f	**19**	**19**	∞	21
openssl-1.0.2d	8716	6877	∞	**6013**
pcre2	822	1375	6095	**439**
re2	∞	∞	∞	**8194**
woff2	3565	**1535**	∞	3260

Evaluation

2. Scalability of Zeror

Table 4: Time to expose known bugs, and the projects whose bugs cannot be triggered by them in 6 hours are removed.

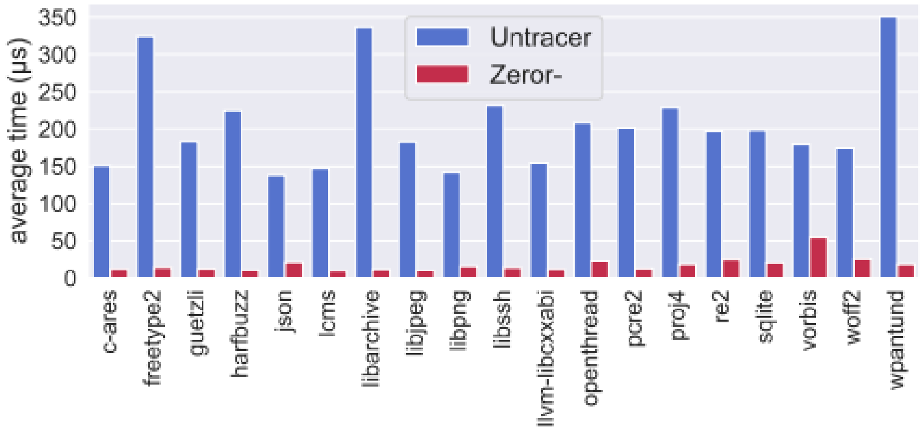
Project	MOPT	MOPT+Zeror
c-ares	**8**	**8**
json	**5**	**5**
llvm-libcxxabi	1818	**761**
openssl-1.0.1f	31	**21**
openssl-1.0.2d	1633	**1320**
pcre2	1944	**968**
woff2	3767	**3196**

Evaluation

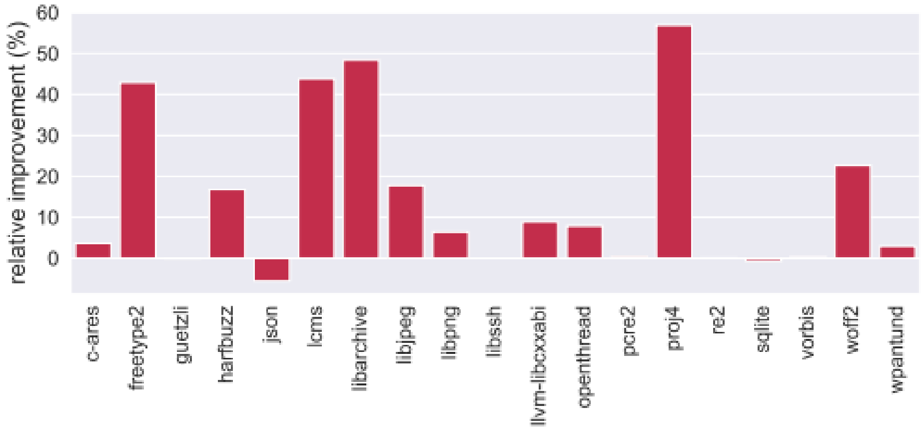
3. Evaluation of Individual Components

self-modifying tracing:

- 13.74x faster than Untracer when erasing instrumentation points
- helps fuzzer cover more branches compared with Untracer



(a) Average time taken for different methods to erase instrumentation points (lower is better).



(b) Relative covered branches improvement of Zeror- compared with Untracer.

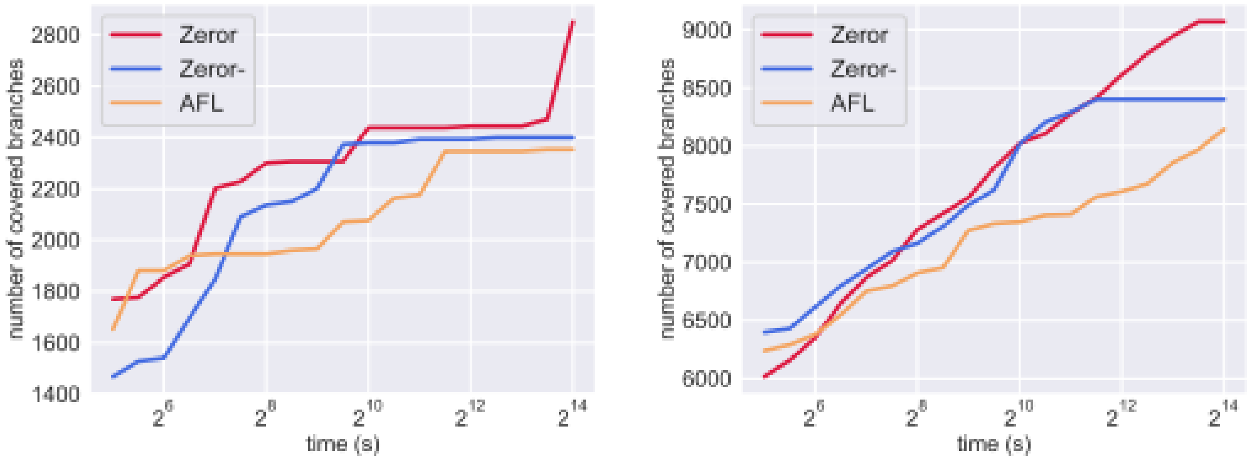
Figure 8: Comparison between Zeror- and Untracer.

Evaluation

3. Evaluation of Individual Components

binary-switching scheduling:

- help fuzzer cover more branches



(a) libjpeg (b) harfbuzz

Figure 9: Branches covered over time with different configurations. The x-axis is on a logarithmic scale.

Conclusion

We propose a novel speed-up fuzzing framework Zeror:

It is made up of two parts: (1) zero-overhead instrumentation (2) real-time scheduling.

It helps fuzzers speed up fuzzing process, further increase covered branches and discovered bugs.

It is easy to be complemented to other orthogonal fuzzing optimizations.

Thank You

If you have any questions, please send emails to
zcj18@mails.tsinghua.edu.com