ProFuzzer: On-the-fly Input Type Probing for Better Zero-day Vulnerability Detection

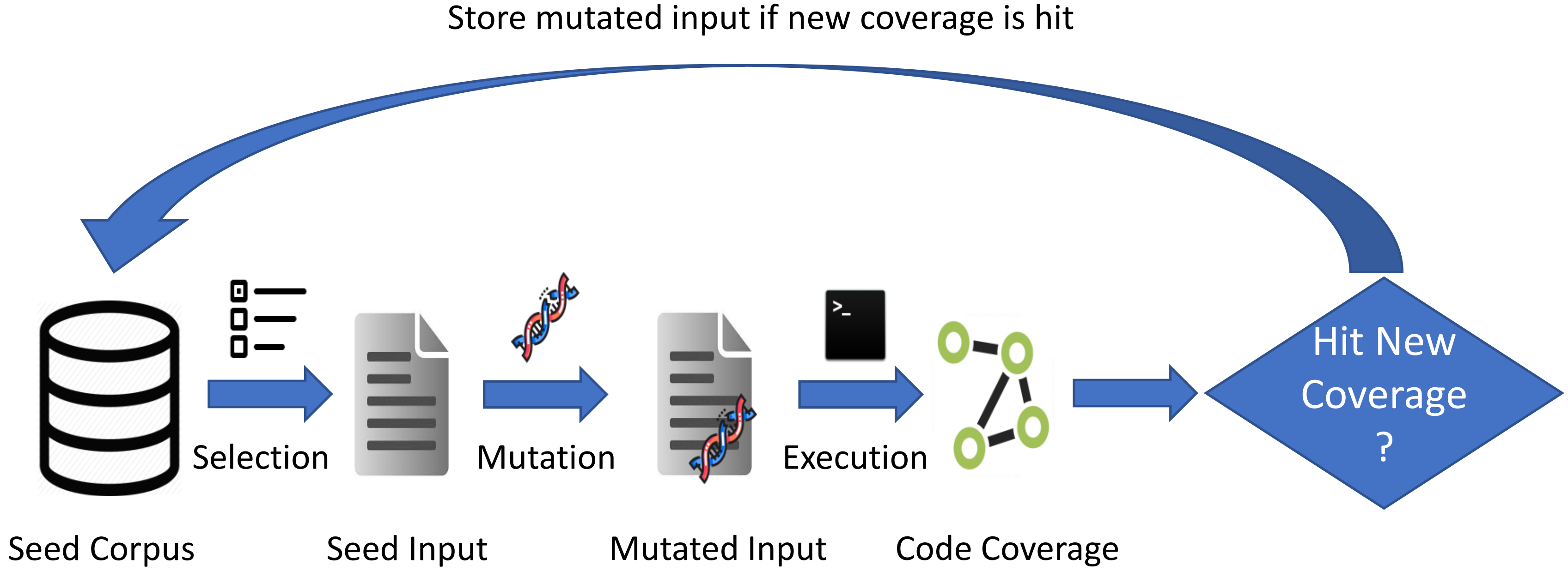
Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang,
Xiangyu Zhang, XiaoFeng Wang, Bin Liang

Email: {you58, ma229, xyzhang}@purdue.edu, {xw48, xw7}@indiana.edu,
{hjj, liangb}@ruc.edu.cn

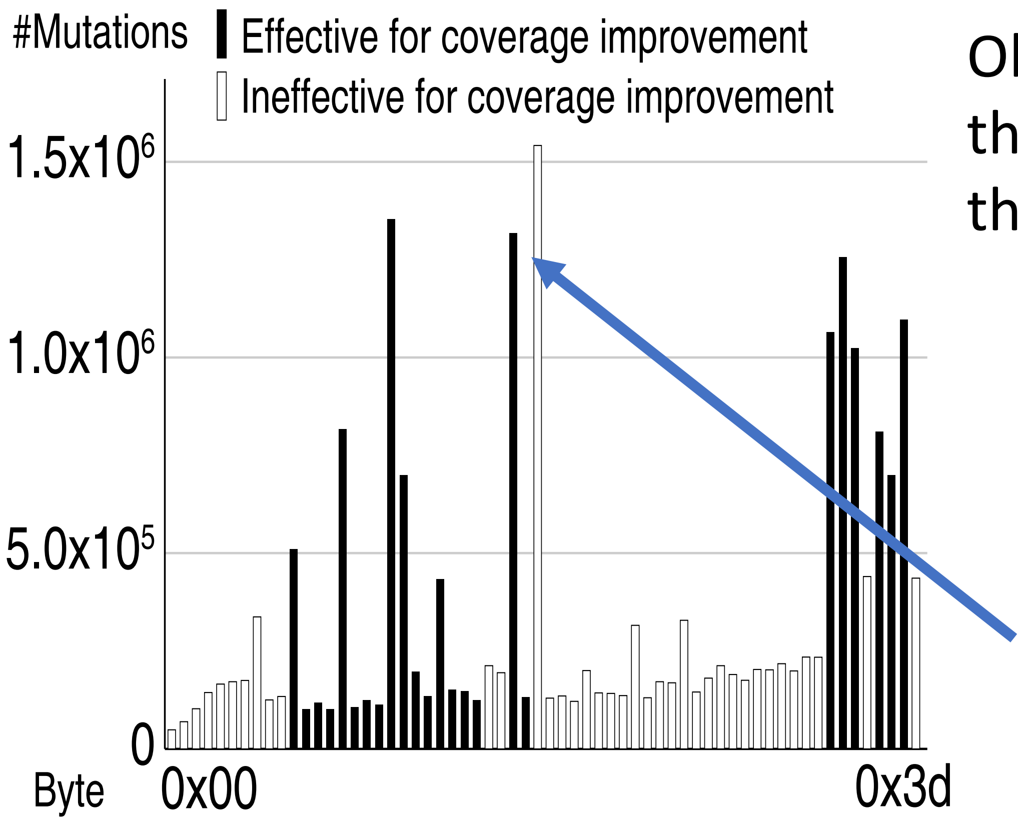
PURDUE UNIVERSITY INDIANA UNIVERSITY 中国人民大學 RENMIN UNIVERSITY OF CHINA

Mutation-based Fuzzing

- Starts from a set of valid input instances as seeds
- Continuously modify to explore various execution paths



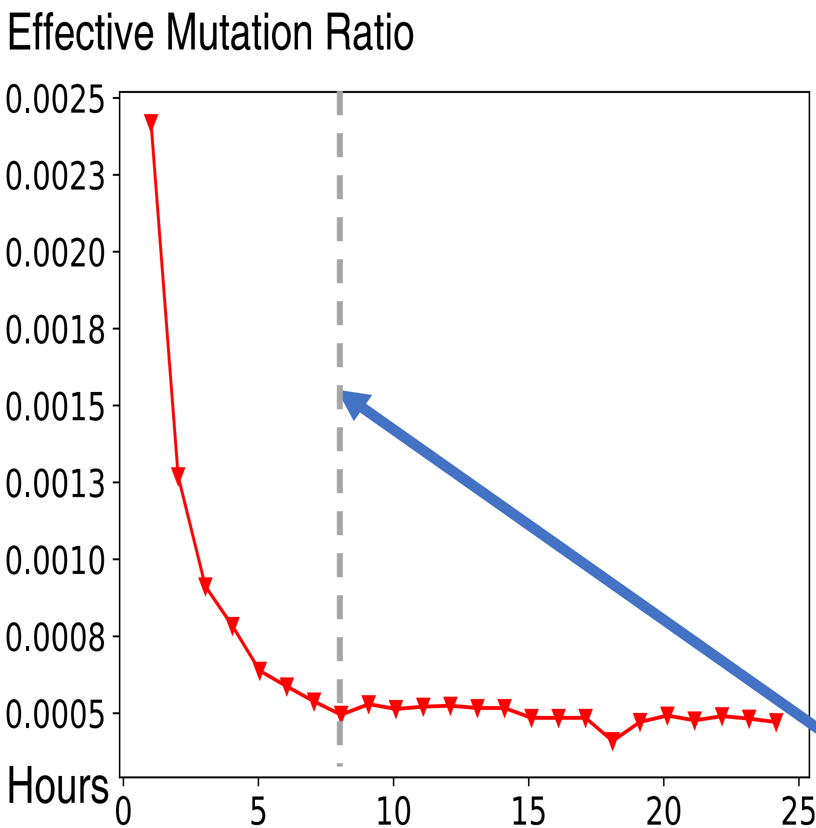
Effectiveness of AFL



Observation 1: More than 60% of the mutations are performed on the input bytes that are ineffective.

More than 1.5 million mutations are performed on the 31st byte (0x1F), which is *ineffective* for coverage improvement.

Effectiveness of AFL



Observation 2: effective mutation ratio (EMR) drops very quickly.

$$\text{EMR} = \frac{\text{\# mutations that increase coverage}}{\text{\# total mutations}}$$

Code coverage is hardly improved after 8 hours.

Existing Works

- Improve the breadth
  - Seed selection: Rebert et al. [SEC 14], Moonshine [SEC 18]
  - Seed prioritization: AFLFast [CCS 16], Steelix [FSE 17], FairFuzz [ASE 18]

- Improve the depth
  - Taint analysis: BuzzFuzz [ICSE 09], TaintScope [S&P 12], VUzzer [NDSS 17]
  - Symbolic execution: Driller [NDSS 16], QSYM [SEC 18], T-Fuzz [S&P 18]
  - Gradient-based search: Angora [S&P 18], NEUZZ [S&P 19]

ProFuzzer

- Basic idea: on-the-fly input structure understanding & utilizing
- Probe input types in a light-weight manner
  - Per-byte mutation observation
  - Field identification
  - Type discovery
- Leverage type information to guide further mutations
  - *Explore* valid values for better code coverage
  - *Exploit* specific values that may lead to a vulnerability
- Application-agnostic v.s. application-specific types
  - Application-agnostic: raw data, size, etc.
  - Application-specific: ip address, pdf data structure, etc.

Fuzzing-related Input Types

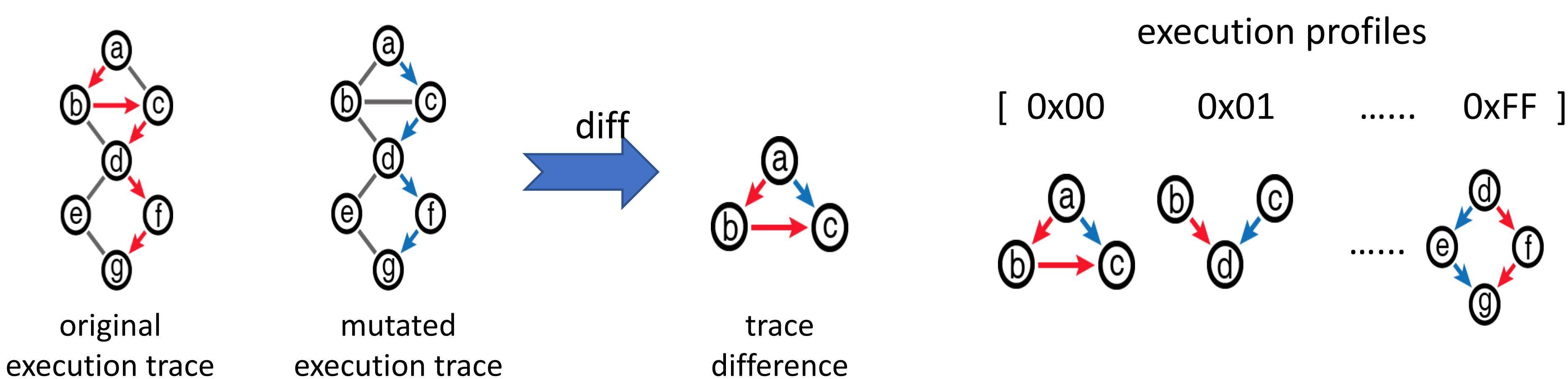
i. Assertion
ii. Raw Data
iii. Enumeration
iv. Offset
v. Size
vi. Loop Count

```
header->biBitCount = get2Bytes(IN);
switch (header->biBitCount) {
  case 0x08: bmp8toimage(pData, ...); break;
  case 0x10: bmp16toimage(pData, ...); break;
  case 0x18: bmp24toimage(pData, ...); break;
  case 0x20: bmp32toimage(pData, ...); break;
  default: exit_error();
}
```

```
header->bfOffBits  = get4Bytes(IN);
fseek(IN, header->bfOffBits, SEEK_SET);
if (fread(pData, ..., stride * header->biHeight, IN)
    != (stride * height)) exit_error();
```

Probing: observing per-byte mutation effect

```
            0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: FF 4D 3A 00 00 00 00 00 00 00 36 00 00 00 28 00
00000010h: 00 00 01 00 00 00 02 00 00 00 01 00 18 00 00 00
00000020h: 00 00 04 00 00 00 4F 00 00 00 4F 00 00 00 00 00
00000030h: 00 00 00 00 00 00 FF FF FF 00 FF FF FF 00
```

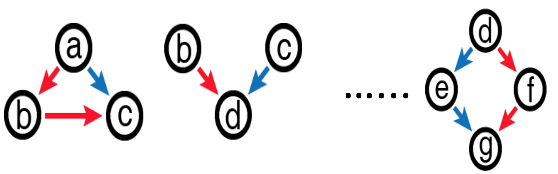


Field Identification: group consecutive bytes

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000h:	42	4D	3A	00	00	00	00	00	00	00	36	00	00	00	28	00
00000010h:	00	00	01	00	00	00	02	00	00	00	01	00	18	00	00	00
00000020h:	00	00	04	00	00	00	4F	00	00	00	4F	00	00	00	00	00
00000030h:	00	00	00	00	00	00	FF	FF	FF	00	FF	FF	FF	00		

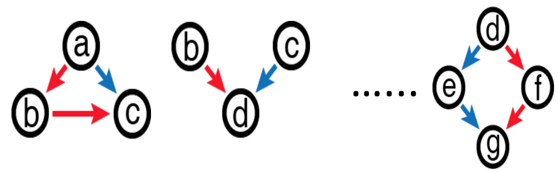
execution profile of byte 0x00

[0x00 0x01 ...... 0xFF]



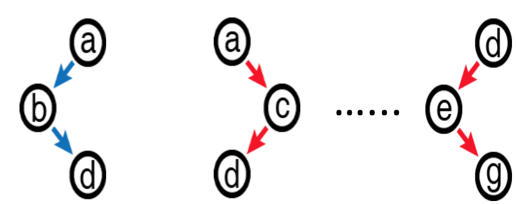
execution profile of byte 0x01

[0x00 0x01 ...... 0xFF]



execution profile of byte 0x02

[0x00 0x01 ...... 0xFF]



Field Identification: group consecutive bytes

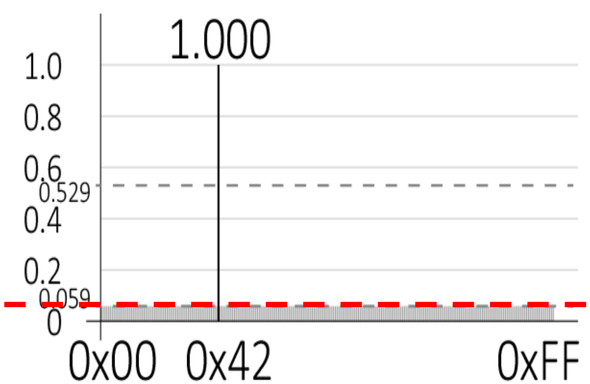
- Group bytes at offsets from *i* to *j* together as a field

if they share the same invalid execution profile (i.e., equal minimum similarity)

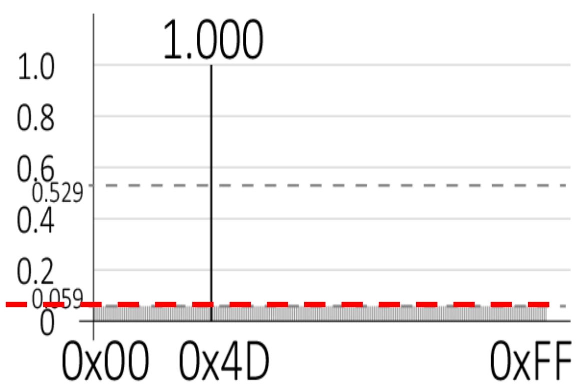
```
header->bfType = get2Bytes(IN);
if (header->bfType != 0x4d42) exit_error();
```

```
            0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 42 4D 3A 00 00 00 00 00 00 00 36 00 00 00 28 00
00000010h: 00 00 01 00 00 00 02 00 00 00 01 00 18 00 00 00
00000020h: 00 00 04 00 00 00 4F 00 00 00 4F 00 00 00 00 00
00000030h: 00 00 00 00 00 00 FF FF FF 00 FF FF FF 00
```

minimal similarity



profile similarity graph of byte 0x00

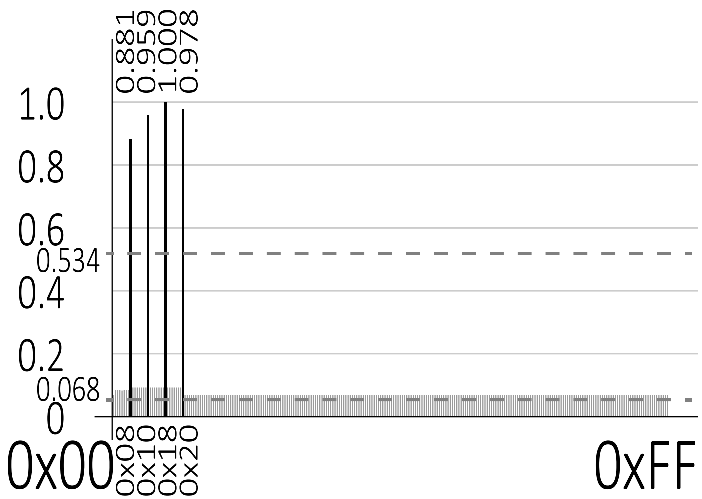


profile similarity graph of byte 0x01

Type Inference: determine type of each field

- Enumeration

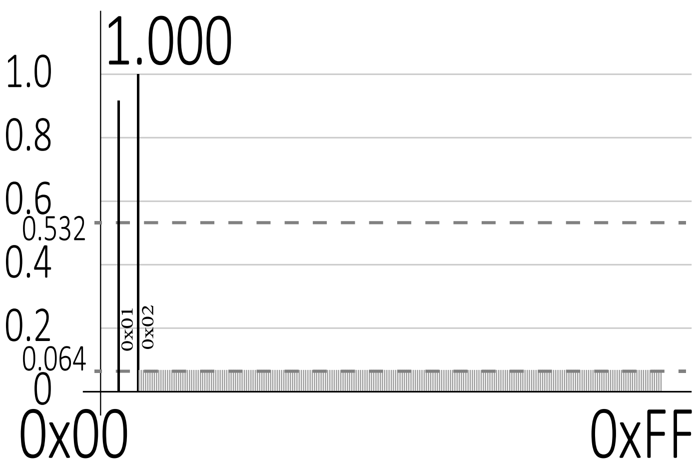
*If there exists a valid value set **VS**, such that: values in **VS** correspond to large similarity; other values correspond to small similarity.*



profile similarity graph of the 28th byte (0x1C)

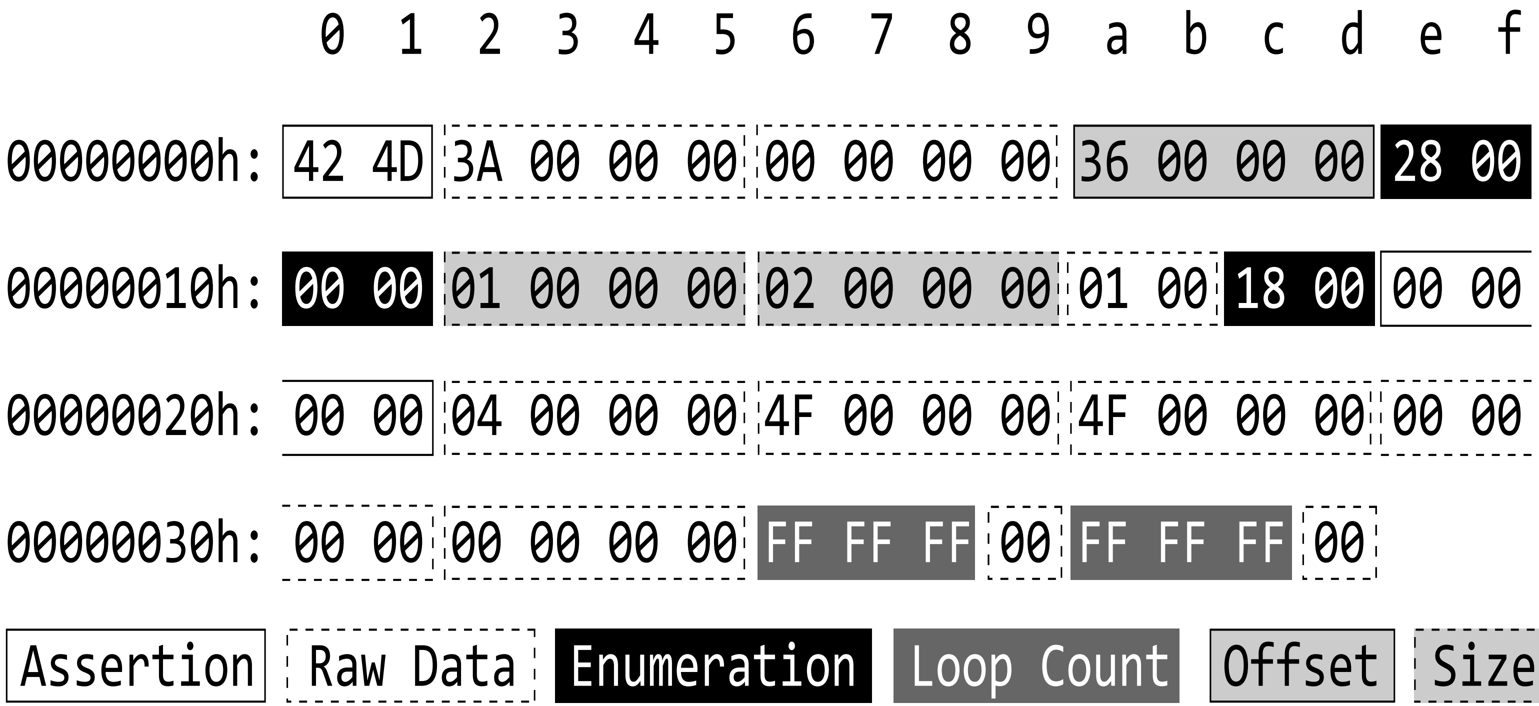
- Size

*If there exists a bound value **bv**, such that: values within **bv** correspond to large similarity; values beyond **bv** correspond to small similarity.*



profile similarity graph of the 22nd byte (0x16)

Type Inference: determine type of each field



By matching execution profiles with different feature patterns, the type of each input field is identified.

Type-guided Exploration (for better coverage)

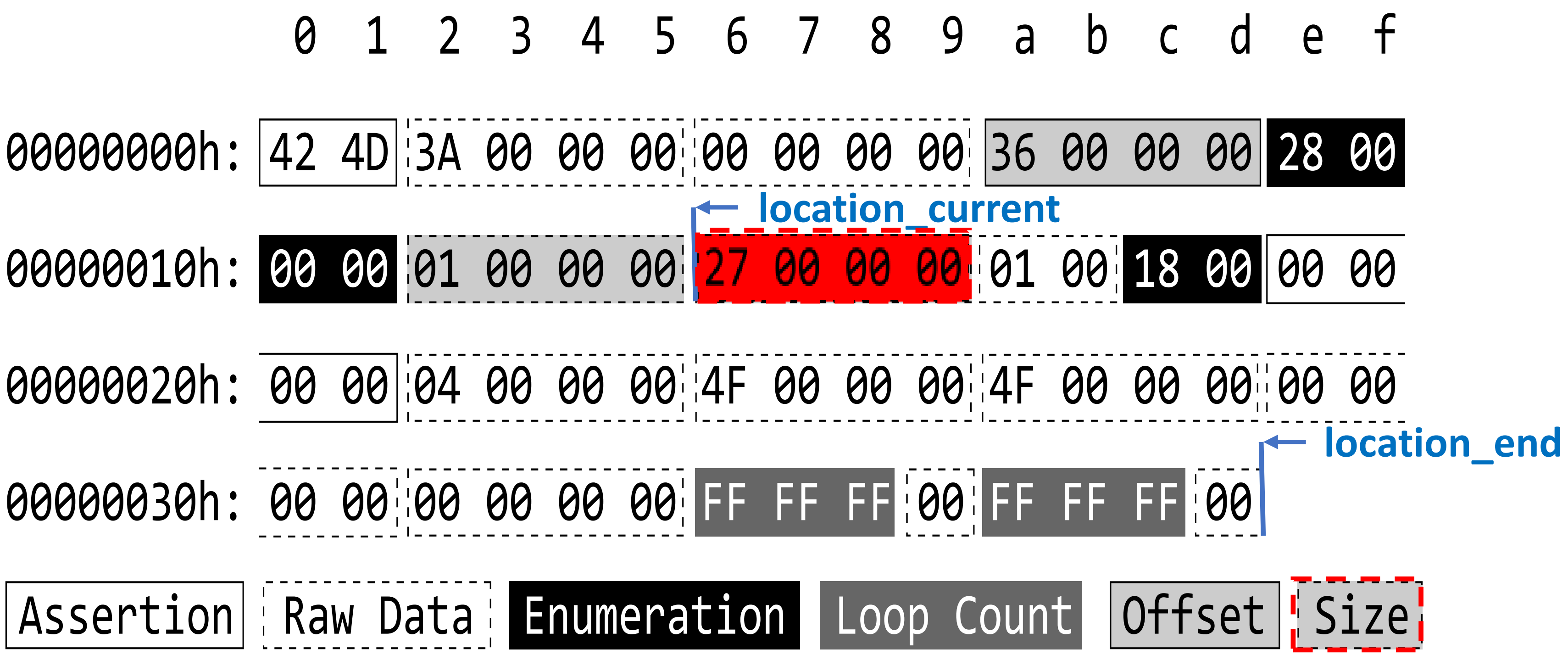
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000h:	42	4D	3A	00	00	00	00	00	00	00	36	00	00	00	28	00
00000010h:	00	00	01	00	00	00	03	00	00	00	01	00	18	00	00	00
00000020h:	00	00	04	00	00	00	4F	00	00	00	4F	00	00	00	00	00
00000030h:	00	00	00	00	00	00	FF	FF	FF	00	FF	FF	FF	00	00	

Assertion Raw Data Enumeration Loop Count Offset Size

Limit mutation to all the valid values of the field type.

For size field: increase its value by X and appends X bytes data

Type-guided Exploitation (for bug detection)



Exploit a set of special values that may lead to potential vulnerabilities.

location_end - location_current = 0x27

Evaluation

- Generality of Assumptions
- Input Size and Path Coverage
- Probing Accuracy
- Finding Zero-day Vulnerabilities
- Evaluation on Standard Benchmarks
- Exposing Known Vulnerabilities
- Performance

Probing Accuracy

Product	Actual	ProFuzzer			afl-analyze		
		Inferred	Wrong (FP*)	Missed (FN**)	Inferred	Wrong (FP*)	Missed (FN**)
exiv2	20	21	3 (14%)	0 (0%)	16	11 (69%)	15 (75%)
graphicsmagick	17	19	1 (5%)	2 (12%)	7	4 (57%)	14 (82%)
libtiff	20	23	2 (9%)	3 (15%)	17	9 (53%)	12 (60%)
openjpeg	17	17	1 (6%)	0 (0%)	9	4 (44%)	12 (71%)
libav	14	14	1 (7%)	0 (0%)	4	2 (50%)	12 (86%)
libming	14	14	0 (0%)	0 (0%)	3	1 (33%)	12 (86%)
mupdf	52	53	2 (4%)	1 (2%)	34	13 (38%)	31 (60%)
podofo	52	53	1 (2%)	2 (4%)	25	11 (44%)	38 (73%)
lrzip	39	39	0 (0%)	5 (13%)	30	3 (10%)	12 (31%)
zziplib	36	36	2 (6%)	0 (0%)	14	4 (29%)	26 (72%)

- ProFuzzer: 5.3% FP, 4.6% FN
- AFL-analysis: 42.7% FP, 69.6% FN

Finding Zero-day Vulnerabilities

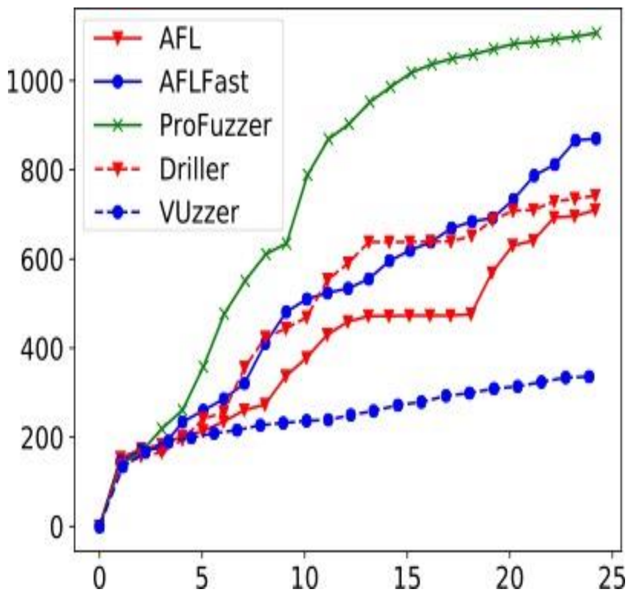
Category	**Product**	**SLOC**	**Bugs**	**CVEs**	**Fixes**
Image	exiv2	131,993	5	5	5
	graphicsmagick	299,186	2	1	1
	libtiff	82,484	8	1	1
	openjpeg	164,284	3	3	3
Audio & Video	libav	703,369	3	2	0
	libming	72,747	2	2	2
PDF	mupdf	102,824	1	1	1
	podofo	78,195	6	6	3
Compression	lrzip	19,098	3	3	3
	zziplib	12,898	8	6	8
Total	*10*	*1,667,078*	*42*	*30*	*27*

Evaluation on Standard Benchmarks

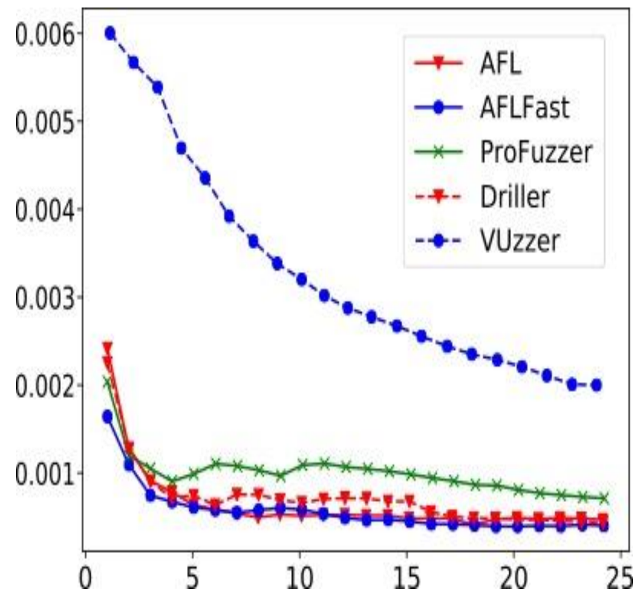
Program	Location	Reaching Time (hours)				
		ProFuzzer	AFL	AFLFast	Driller	VUzzer
guetzli	output_image.cc:398	0.83	3.64	2.37	3.73	8.60
json	fuzzer-..._json.cpp:50	0.05	0.02	0.04	0.12	4.26
lcms	cmsintrp.c:642	0.67	6.55	3.83	5.31	11.97
libarchive	archive_..._warc.c:537	1.34	7.88	6.92	6.74	14.42
libjpeg	jdmarker.c:659	11.68	T/O	T/O	T/O	T/O
libpng	png.c:1035	1.84	3.37	2.33	4.27	6.06
	pngread.c:757	0.03	0.01	0.01	0.02	0.17
	pngrutil.c:1393	7.63	T/O	T/O	T/O	T/O
vorbis	codebook.c:479	T/O	T/O	T/O	T/O	T/O
	codebook.c:407	T/O	T/O	T/O	T/O	T/O
	res0.c:690	11.76	T/O	T/O	T/O	T/O

- ProFuzzer reaches more target locations than other fuzzers
- ProFuzzer is 2.26 to 8.85 times faster than other fuzzers

Performance



Comparison on Path Coverage



Comparison on Effective Mutation Ratio

- ProFuzzer archives 27% ~ 227% more path coverage than other fuzzers
- ProFuzzer spends 53% ~ 79% less time to reach the same coverage
- ProFuzzer keeps relatively high effective mutation ratio

Closely Related Works

- Input structure reverse engineering
  - Tupni [CCS 08]: identify input bytes relations via symbolic execution
  - Reward [NDSS 10]: *propagates program type* through syscalls and instructions
  - Howard [NDSS 11]: analyze *memory access patterns* during program execution
- Field-aware fuzzing
  - Steelix [FSE 17] infers *magic value bytes* by intercepting string comparisons
  - TIFF [ACSAC 18] infers *program type* (e.g., int, string) via taint analysis
  - Angora [S&P 18] infers *shape and size* of input bytes via taint analysis
- Difference:
  - ProFuzzer adopts lightweight mechanism instead of heavyweight analysis
  - ProFuzzer infers application-agnostic and fuzzing-related types

Conclusion

- Leverage on-the-fly type learning to improve fuzzing
  - Probe input fields and types by observing the fuzzing process
  - *Explore* valid values for better code coverage
  - *Exploit* the values that could lead to an vulnerability

- Results:
  - Better performance on code coverage and vulnerability exposure
  - 42 zero-day vulnerabilities, 30 of which are assigned CVEs

Thank you!

Q&A