IEEE Security & Privacy 2023

UTopia

Unit Tests to Fuzzing https://github.com/Samsung/UTopia

UTOPIA: Automatic Generation of Fuzz Driver using Unit Tests

Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon,
Taesoo Kim, WooChul Shim, Yong Ho Hwang

bd.jeong, joonun.jang, hayoon.yi, jiin.moon, junsik1.kim, intae.jeon,
tsgates.kim, woochul.shim, yongh.hwang@samsung.com

Samsung Research Georgia Institute of Technology

Hurdles in Library Fuzzing : Scalability

Fuzz testing is an effective way to uncover bugs in libraries that may not be found through traditional testing techniques.


Taking the next step: OSS-Fuzz in 2023

February 1, 2023

Since launching in 2016, Google's free OSS-Fuzz code testing service has helped get over 8800 vulnerabilities and 28,000 bugs fixed across 850 projects. Today, we're happy

https://security.googleblog.com/2023/02/taking-next-step-oss-fuzz-in-2023.html

But, requires manually writing fuzz drivers.
: Time consuming and labor intensive

- #APIs X #Libs = ?
- Project even evolves! ← Not just a one-time task

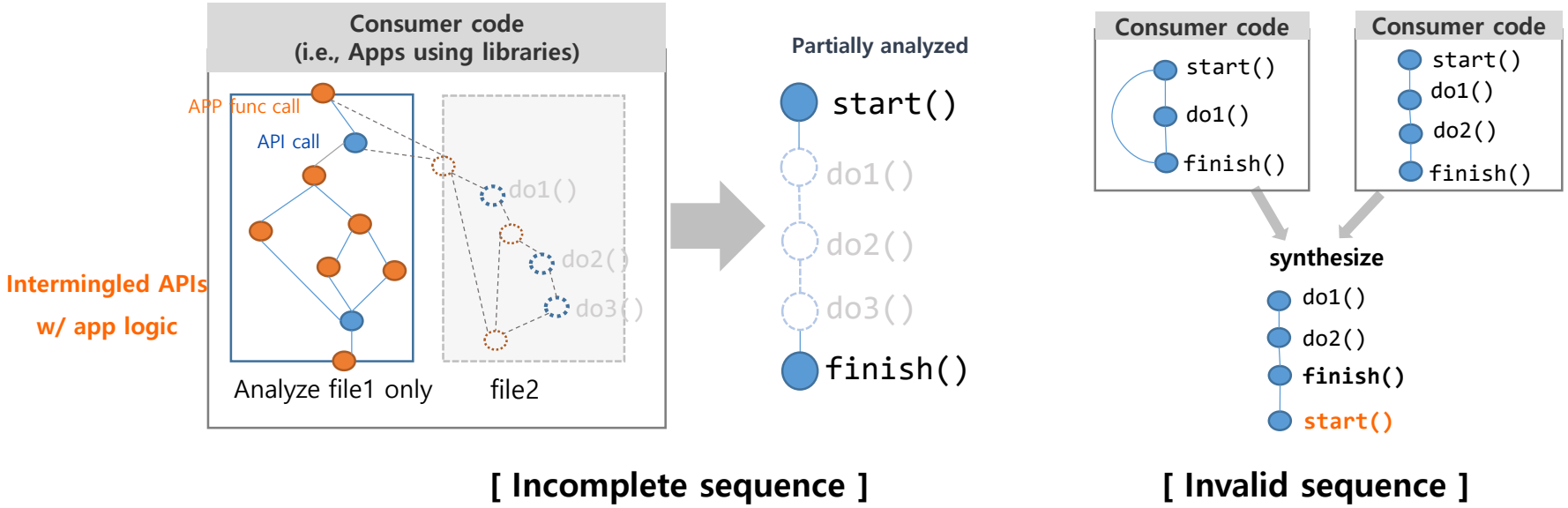
Manual human efforts for library fuzzing does not scale well.

Existing Work for Automatic Generation

Synthesize a sequence of APIs from consumer code by

1) inferring API dependences through the static analysis

2) or observing their uses at runtime



[Incomplete sequence]

[Invalid sequence]

Semantically invalid states

Root Cause of the Limitation: API Misuse

Using APIs properly means

- Valid API call sequences
- Valid API call parameters

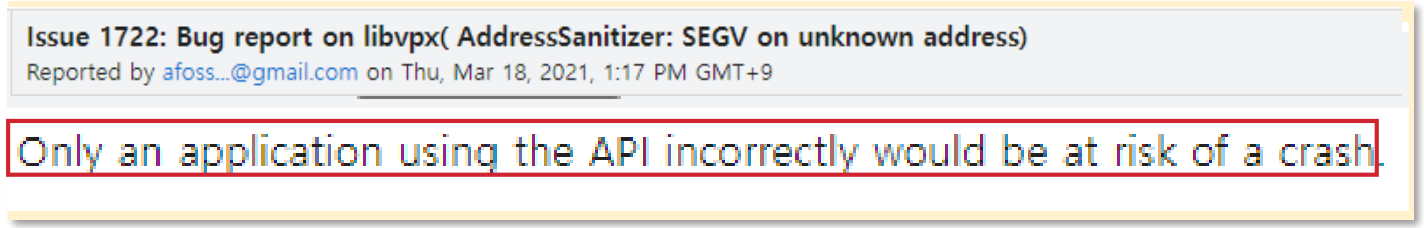
```
yr_initialize()
yr_compiler_create(&compiler)
yr_compiler_add_string(compiler, buffer, NULL)
yr_compiler_get_rules(compiler, &rules)
yr_rules_destroy(rules)
yr_compiler_destroy(compiler)
```

https://github.com/VirusTotal/yara/blob/master/tests/oss-fuzz/rules_fuzzer.cc

Adhering to valid API usage helps avoid

- exploring libraries in invalid and uninteresting states.
- producing spurious crashes.

  : unseen in end-to-end binary fuzzing



Challenges & Approach Ideas

C1: Valid API sequence

Key Idea 1: No Inference. Convert UT to fuzz driver.

```
bool unit_test() {
  struct A *a = CREATE(10);
  int sum = SUM(a, 1, 2);
```

```
bool fuzz_test() {
  struct A *a = CREATE(fuzz1);
  int sum = SUM(a, fuzz2, fuzz3);
```

C2: Fuzz input as a semantically valid API argument

Key idea2: Parameter attribute analysis + root-definition analysis

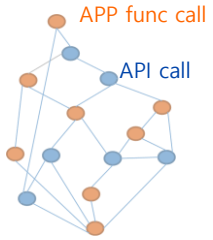
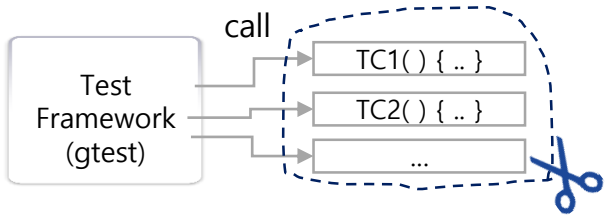
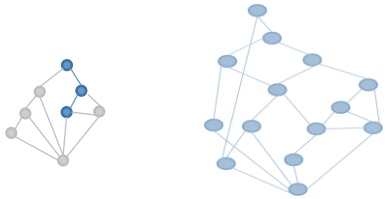
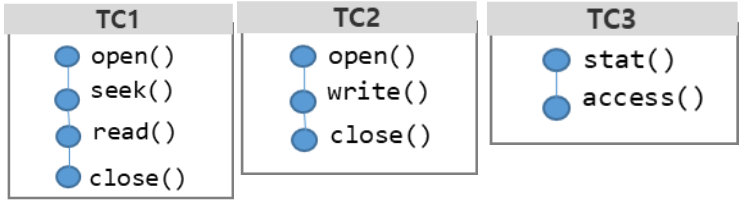
C3: UT specific challenges

- How to handle the assertion checks?

Key idea3: Explore several basic strategies to make choices

```
1 bool test() {
2   struct A *a = CREATE(10);
3   assert_neq(a, NULL); // if ignored, will crash on #4
4   int sum = SUM(a, 1, 2);
5   assert_eq(sum, 3);   // if enforced, can it reach #6?
6   do-something
```

Our Approach: Why UT instead of Consumer Code?

	Consumer code	UT
Code Extraction	 Program slicing: not practical	 **1. Simple: extract test functions only**
API uses	 Too small or too large	 **2. Carefully crafted APIs only**
Opportunity		**3. Increase the fuzzing adoption using UT**

Category	Total #	Unit Tested # (rate)	Fuzz Tested # (rate)
GitHub C/C++ (Top 200)	200	143 (72%)	38 (19%)
Android External	316	224 (71%)	61 (19%)
OSSFuzz	450	347 (77%)	450 (100%)

Detail: Fuzz Input w/ Valid Parameter Semantic

1. Give constraints to fuzz input.

```
void API(int *P) {
    *P = 0;
}
```

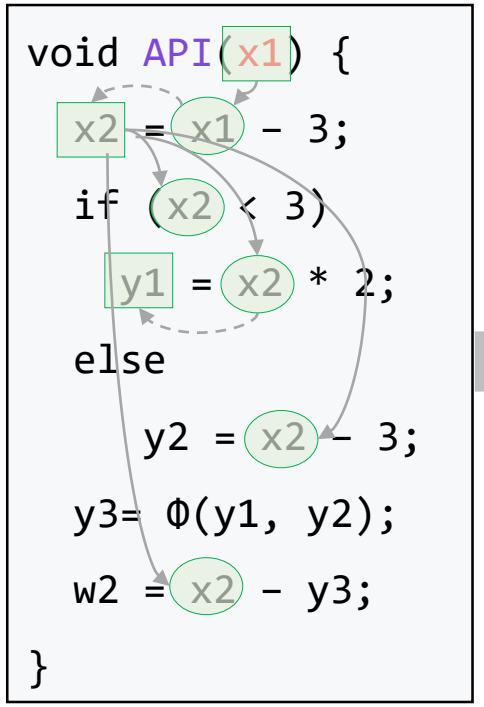
fuzzing an output param

→ **ineffective**

```
void API(int P) {
  void buf = malloc(P);
}
```

fuzzing malloc()

→ **undesirable crash**



Def-use chain construction

```
void API(int *P) {
    *P = 0;
}
```

Used as the operand of store instruction

```
void API(int P) {
    void *buf = malloc(P);
}
```

Argument flows into malloc()

Identify parameter attribute

Output: Don't fuzz

LoopCount: limit the maximum input

AllocSize: limit the maximum input

FilePath: Deliver fuzz input as the file content

Array ↔ Len

```
int *var1 = Fuzz.data();
API(var1, Fuzz.size());
```

Fuzz input constraint

< Data flow analysis on API parameters>

Detail: Fuzz Input w/ Valid Parameter Semantic

2. Assign fuzz input while maintaining valid parameter semantic.

```
bool unit_test() {
  int in = 3;
  int out = API1(in);
  API2(out);
  API3(in);
```

```
bool fuzz_test() {
  int in = 3;
  int out = API1(fuzz1); ?
  API2(fuzz2); ?
  API3(fuzz3); ?
```

invalid

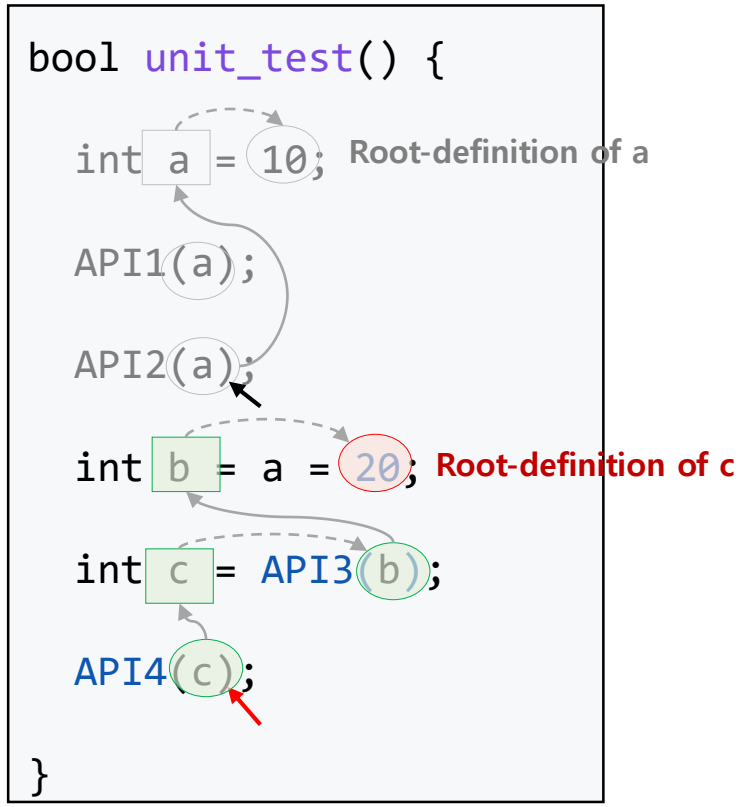
Valid parameter relationships between APIs are already well expressed in UT.
How to reflect the relationships to the fuzz input?

Detail: Root-def Analysis for Fuzz Input Assignment

2. Assign fuzz input while maintaining valid parameter semantic.

```
bool fuzz_test() {

  int a = fuzz_input_A;

  API1(a);

  API2(a);

  int b = a = fuzz_input_B;

  int c = API3(b);

  API4(c);

}
```

Assign fuzz input to the root-definitions



Following the use-def chain

< Root-definition Analysis for Fuzz Input Assignment >

Evaluation

Scalability of the generation of fuzz drivers :

5K working fuzz drivers generated from 55 OSS and Tizen open source projects

- 25 OSS projects: 2,715 fuzz drivers (1 fuzz driver in 15 per-core secs)
- 30 Tizen projects: 2,699 fuzz drivers

Fuzzing capability

123 bugs found

- 109 bugs in the OSS-Fuzz projects
- 14 in the Tizen projects

Fuzzing Evaluation Setup

- Libfuzzer w/ fork-mode + Asan
- Ran fuzz drivers with the seeds extracted from UT
- Results averaged over 10 fuzzing runs

Project	Bugs or CVEs	Category	Fuzzer
OpenCV	#21947	bof	readnetfromtensorflow_fuzzer.cc
OpenCV	#21852	bof	readnetfromtensorflow_fuzzer.cc
OpenCV	#21851	bof	
libaom	CVE-2021-30473	free	
libaom	CVE-2021-30474	bof	
libaom	CVE-2021-30475	nullchk	
uriparser	CVE-2021-46141	nullchk	
uriparser	CVE-2021-46142	nullchk	
libwebsockets	#2687	segfault	lws_upng_inflate_fuzzer.cpp
libhtp	#342	segfault	
libhtp	#343	segfault	
libphonenumber	#201466814	nullchk	
libphonenumber	#201470539	segfault	
libvpx	#1742	arith	
libvpx	#1722	free	
tesseract	#3583	bof	
tesseract	#3584	bof	
tesseract	#3586	segfault	
tesseract	#3694	arith	
vowpal-webbit	#3542	arith	
vowpal-webbit	#3543	arith	
wabt	#1793	segfault	
wabt	#1794	oom	
aosp/audio_utils	#206677585	arith	
tizen/libtbm	#254382	nullchk	

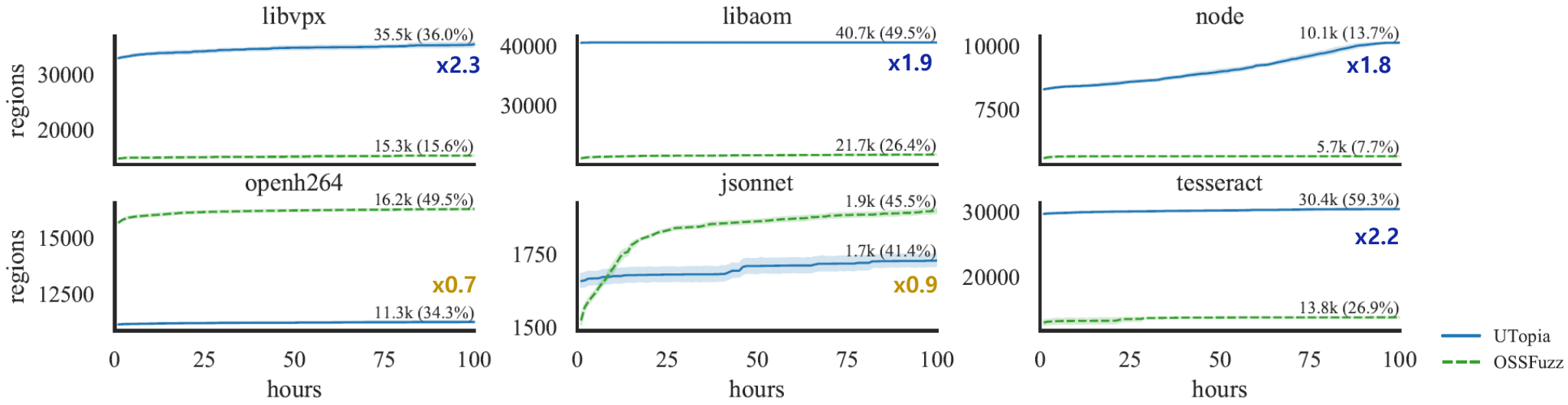
Evaluation : Fuzzing Effectiveness

Comparison with the manual fuzz drivers from OSS-Fuzz

100 per-core hours of fuzzing per project

	libvpx	libaom	Node.js	openh264	jsonnet	Tesseract
OSS-Fuzz	1 (7)	1 (5)	2 (32)	1 (7)	1 (5)	1 (9)
UTopia	**40 (43)**	**115 (109)**	**42 (60)**	**113 (132)**	**45 (6)**	**240 (356)**

 # Drivers (APIs tested)



* UTopia generated fuzz drivers from open264 and jsonnet test an internal function of API or an uninteresting API argument for logging.

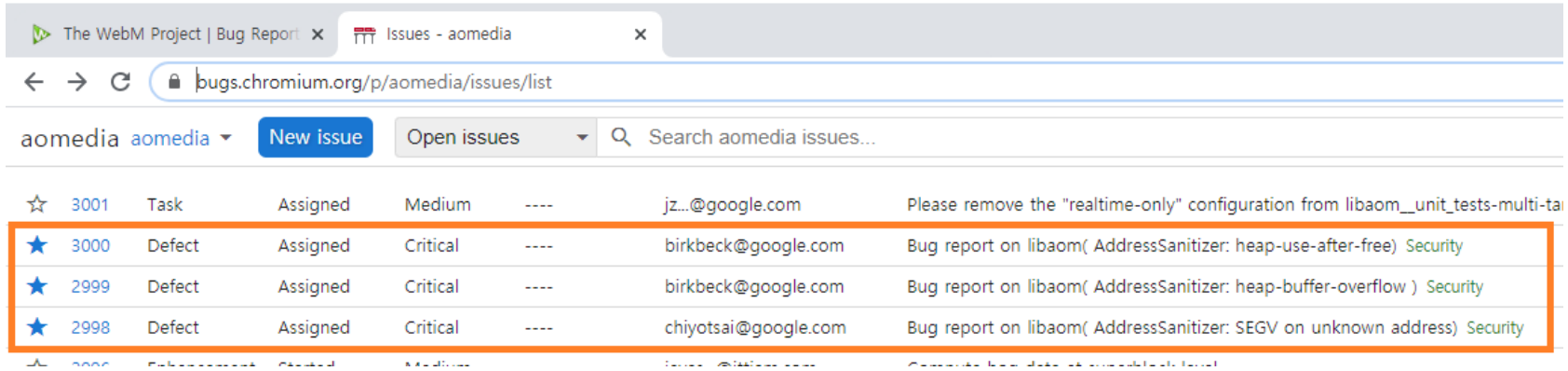
Evaluation : Fuzzing Effectiveness

Unique coverage of UTopia compared with UT

	Libvpx	libaom	Node.js	Openh264	jsonnet	Tesseract
max growth vs UT	x2.0	**x37.5**	x2.0	x1.3	x14.9	x3.2

1 per-core hours of fuzzing per fuzz driver

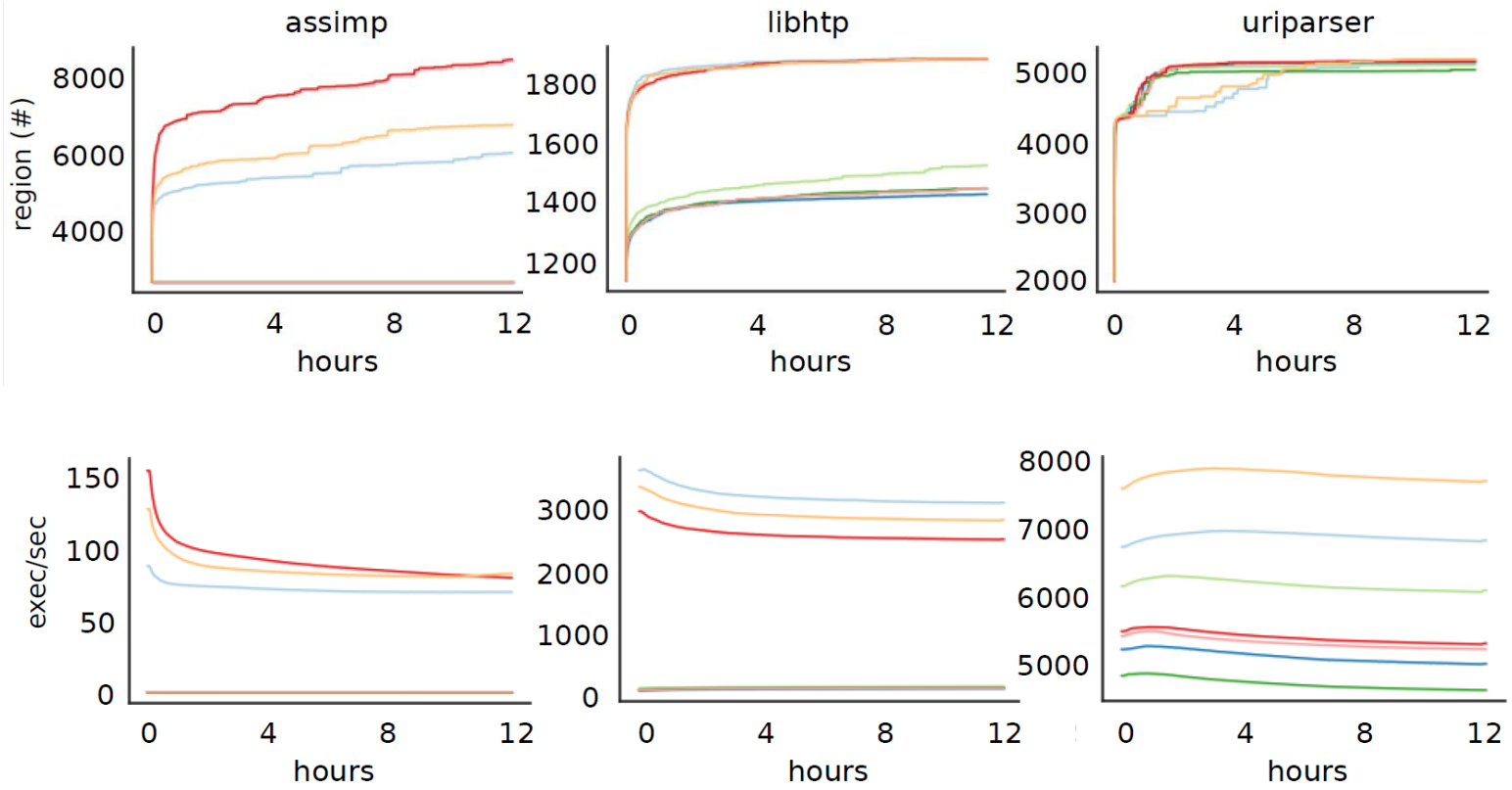
Uncovered 3yrs old bugs in libaom with new exploration based on UT



UTs are designed to check what developers expected to be correct, but fuzzers focus on what they didn't expect.
← Nicely bridged by UTopia.

Evaluation : Exploration of the Assertion

No silver bullet for all types of assertions



nullchk

```
struct A *a = CREATE(fuzz);
assert_neq(a, NULL);
```

obj_depend

```
INIT(&obj, fuzz);
assert_eq(obj.member, 3);
```

eterm : Keep all assert

nop : Ignore all

nop25 nop50 nop75

A single fuzz driver per project was selected and ran 10 times for 12 per-core hours

Limitations

Some UTs test internal function instead of APIs.

```
TEST_F(URLTest, Base6) {            // libnode TC
    char* input = autofuzz276;      // autofuzz276 = "/\nStrace:\n"
    char* base = autofuzz277;       // autofuzz277 = "file://1h\333\207eam`,ar"
    URL simple(input, strlen(input), base, strlen(base));  // crashes with SEGV
```

Insufficient error checks in UT make sense, but not in fuzzing

```
pkt = aom_codec_get_cx_data(&enc_, &iter);
// no check on pkt
if (pkt->kind == AOM_CODEC_CX_FRAME_PKT) {   // if pkt is NULL, it will crash
```

Non-conventional relations of parameters

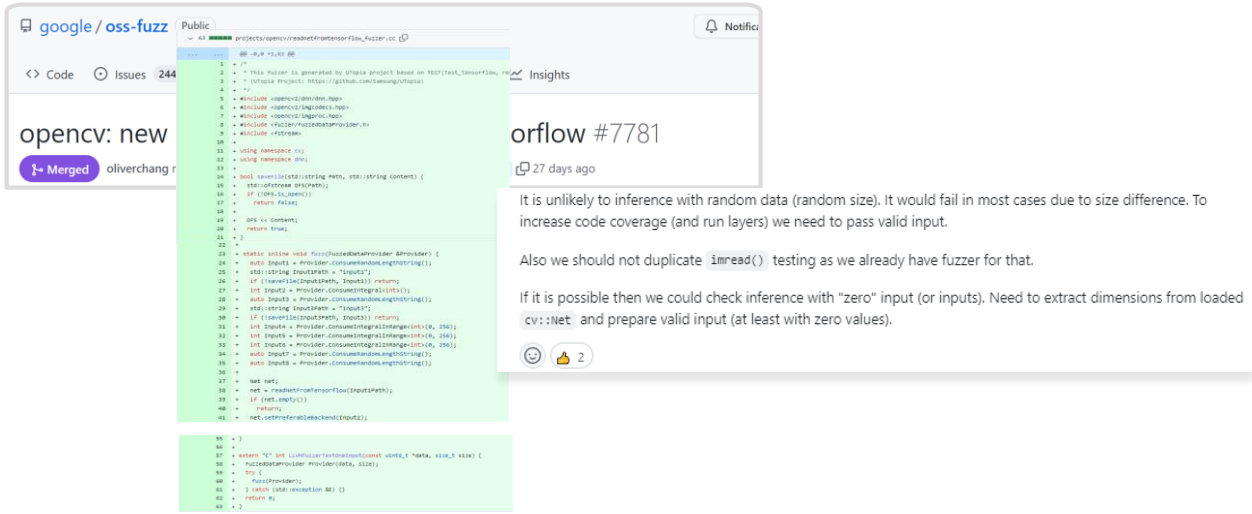
```
printf("Hello %s", input); // replace the first parameter with fuzz input?
```

How Is UTopia Being Utilized Now?

UTopia is open at https://github.com/Samsung/UTopia

Bug report with generated fuzz drivers

UTopia-generated fuzz drivers are merged to oss-fuzz with modification.



Initial submission

```
+ #include <opencv2/dnn/dnn.hpp>
+
+ using namespace cv;
+ using namespace dnn;
+
+ extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
+   try {
+     readNetFromTensorflow((const char*)data, size);
+   } catch (std::exception &e) {}
+   return 0;
+ }
```

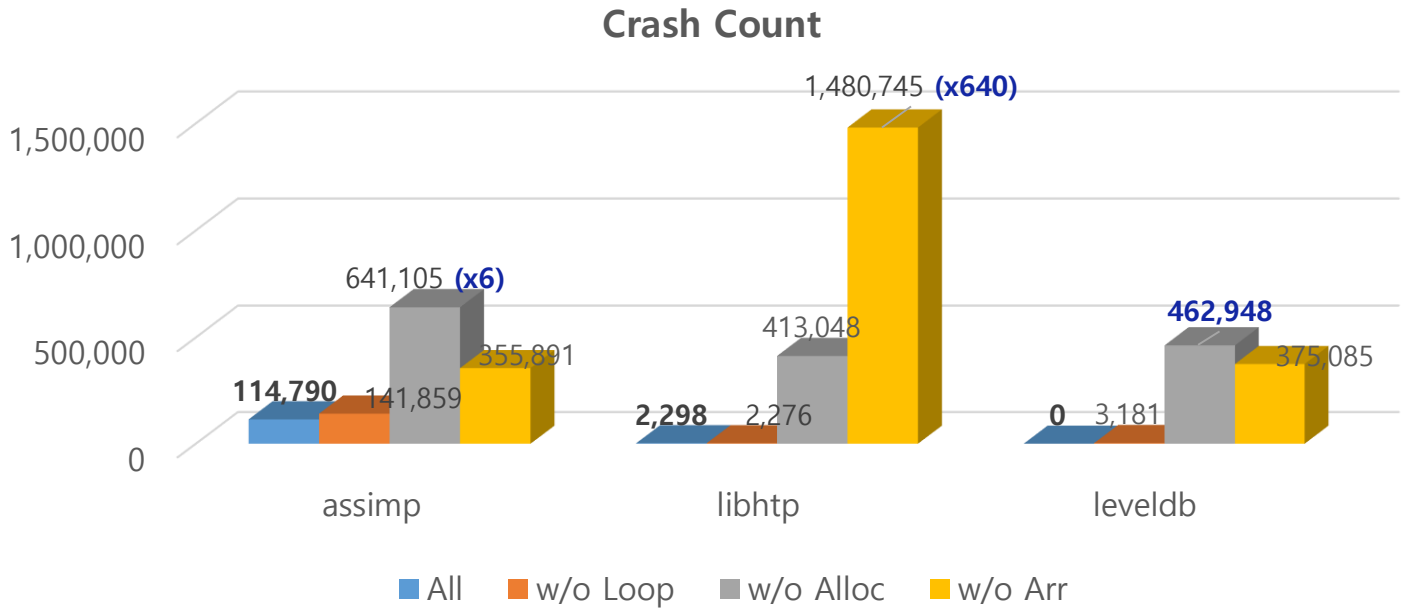
Merged after review with simplification

Q&A

APPENDIX

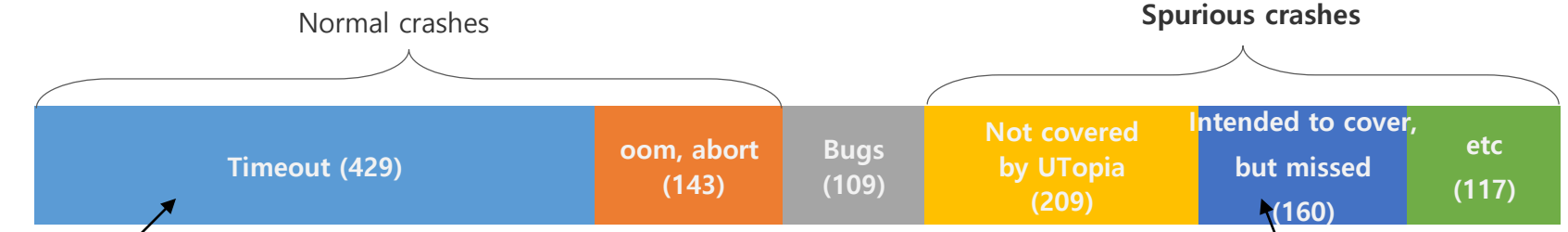
Evaluation : Mitigation of Spurious Crashes

Mitigation of undesirable crashes through static analysis



A single fuzz driver per project was selected and ran 10 times for 12 per-core hours.

Remaining crashes



1,167 unique crashes from the generated fuzz drivers using 25 OSS projects are collected.
Each fuzz driver ran for 1 per-core hr.

Due to the analysis failure with loop count attribute: nested loop, loop count multiplication, loop count identification failure in C++ object

Due to the analysis failure with array/len in complex loop or multi-dimension

Evaluation : UTopia-generated fuzz drivers

Target Library					Unit Tests				UTopia-Generated Fuzz Drivers								
									Analyzed Test Cases			Functions		Time(sec)			
Name	SR	LoC	BS	#eFn	TF	Cov.	TcCov.	#TC	#Oths.	#Ign.	#Gen.	#tested	#w/input	AT	GT	Cov./UCov	AG/MG
nodejs	O	3M	gn	3,065	G	11.8%	11.5%	124	0	83	41 (100%)	63	24	1,158	21	13.6%/ 3.1%	1.2/ 2.0
libaom	O	363K	cm	5,065	G	51.5%	48.4%	682	531	36	115 (100%)	109	98	18,290	7	54.6%/ 6.2%	1.0/ 37.5
assimp	O	356K	gn	5,055	G	45.5%	23.9%	449	0	199	250 (100%)	96	56	1,019	7	36.3%/ 13.9%	1.3/ 6.5
libvpx	O	248K	cm	1,446	G	47.6%	27.7%	373	315	18	40 (100%)	42	32	4,522	6	34.3%/ 6.5%	1.1/ 2.0
tesseract-ocr	O	158K	cm	3,650	G	62.4%	57.1%	477	165	72	240 (100%)	339	187	2,294	66	59.9%/ 2.9%	1.0/ 3.2
openh264	O	92K	cm	1,523	G	61.3%	33.6%	320	21	186	113 (100%)	133	94	4,521	8	34.3%/ 0.8%	1.0/ 1.3
libphonenumber	O	53K	cm	510	G	65.2%	63.7%	324	0	78	246 (100%)	152	97	4,675	9	65.2%/ 2.3%	1.0/ 3.0
wabt	O	47K	cm	1,034	G	24.9%	24.6%	190	0	110	80 (100%)	61	40	430	2	26.3%/ 1.8%	1.1/ 2.3
leveldb	O	21K	cm	397	G	87.1%	86.0%	218	0	43	175 (100%)	92	51	1,001	12	85.3%/ 1.0%	1.0/ 4.0
libhtp	O	20K	gn	386	G	73.6%	73.3%	339	3	0	336 (100%)	191	141	490	99	78.0%/ 5.8%	1.1/ 4.9
jsonnet	O	13K	cm	98	G	35.9%	35.9%	45	0	0	45 (100%)	6	4	16	2	41.3%/ 5.3%	1.2/ 14.9
uriparser	G	8K	cm	42	G	90.5%	88.7%	92	0	10	82 (100%)	50	50	80	2	92.1%/ 5.1%	1.0/ 14.2
mediapipe	G	225K	bz	2,237	G	47.0%	37.5%	524	4	321	199 (100%)	226	66	6,787	12	38.7%/ 1.3%	1.1/ 1.6
filament	G	64K	nj	5,948	G	32.8%	25.0%	292	0	170	122 (100%)	219	92	4,576	20	27.7%/ 2.7%	1.1/ 3.4
muduo	G	16K	cm	359	B	15.3%	9.9%	30	0	25	5 (100%)	11	5	32	1	11.0%/ 1.1%	1.3/ 1.3
vowpal_wabbit	G	81K	cm	1,383	B	20.3%	14.8%	224	0	155	69 (100%)	64	46	1,968	3	16.4%/ 1.6%	1.1/ 1.5
ledger	G	51K	cm	32	B	9.3%	9.3%	17	0	0	17 (100%)	32	10	410	1	10.5%/ 1.1%	1.2/ 1.6
cpuinfo	A	423K	cm	66	G	54.2%	15.6%	142	0	136	6 (100%)	1	1	2,068	6	16.2%/ 0.6%	1.0/ 2.3
minijail	A	16K	gn	162	G	54.1%	47.9%	189	0	30	159 (100%)	102	60	767	31	39.6%/ 1.3%	1.2/ 22.0
pthreadpool	A	12K	cm	54	G	69.3%	69.3%	291	0	1	290 (100%)	24	24	317	4	74.7%/ 5.4%	1.1/ 4.5
cpu_features	A	6K	cm	36	G	51.5%	9.43%	31	0	27	4 (100%)	5	3	23	11	9.6%/ 0.2%	1.2/ 1.7
puffin	A	5K	cm	92	G	83.6%	66.3%	44	0	17	27 (100%)	34	21	122	1	71.5%/ 5.2%	1.1/ 60.6
bsdiff	A	4K	gn	137	G	57.2%	43.4%	66	0	42	24(100%)	34	19	207	11	44.3%/ 2.0%	1.1/ 4.2
sfntly	A	23K	cm	897	G	48.7%	48.2%	23	0	7	16 (100%)	192	44	431	2	49.3%/ 1.4%	1.1/ 1.1
snappy	T	6K	gn	46	G	75.9%	75.9%	17	0	3	14 (100%)	14	12	112	1	79.5%/ 3.7%	1.1/ 2.3
Total	-	53M	-	33,720	-	-	-	5,523	1,039	1,769	2,715 (100%)	2,292	1,277	15.6hr	6min	-	-