ParmeSan: Sanitizer-guided Greybox Fuzzing

USENIX 2020

Sebastian Österlund
Vrije Universiteit Amsterdam

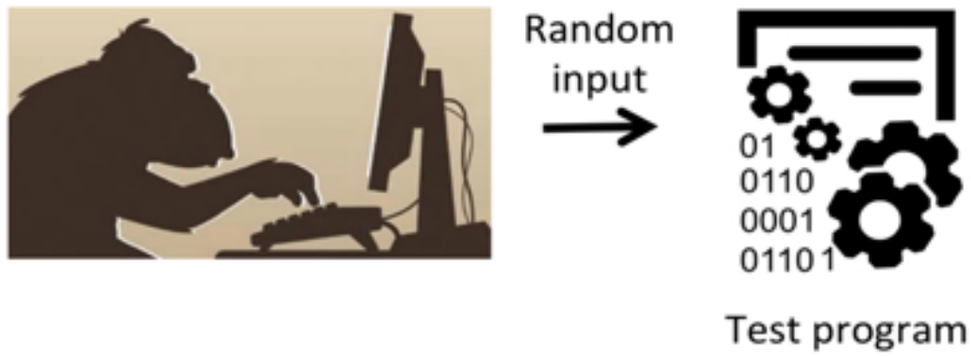
Kaveh Razavi
Vrije Universiteit Amsterdam

Herbert Bos
Vrije Universiteit Amsterdam

Cristiano Giuffrida
Vrije Universiteit Amsterdam

**some pages borrowed from Zheyu Ma*

background



- What is fuzzing: feed random inputs to tri
  as possible

- What is state-of-the-art of fuzzing research:

  - black-box fuzzing: totally random

  - white-box fuzzing: symbolic execution

  - gray-box fuzzing:

    - coverage-guided

    - directed fuzzing

    - heuristics: Dynamic data-flow analysis (DFA), Neural network, etc.

Contribution

- designs the first sanitizer-guided fuzzer using a two-stage directed fuzzing strategy to efficiently reach all the interesting targets.

- finds the same bugs as state-of-the-art coverage-guided and directed fuzzers in less time.

Motivation

- Coverage-guided Fuzzer:
  - Ideal: Code coverage is strongly correlated with bug coverage.
  - Reality: Code coverage is a huge **overapproximation** of bug coverage.
- Directed Fuzzer:
  - Ideal: Steering the program towards locations that are more likely to be affected by bugs
  - Reality: They **underapproximate** overall bug coverage.

ParmeSan: Sanitizer-guided Fuzzer

Overview

- Target Acquisition
- Dynamic Control Flow Graph (CFG)
- Sanitizer-guided Fuzzer

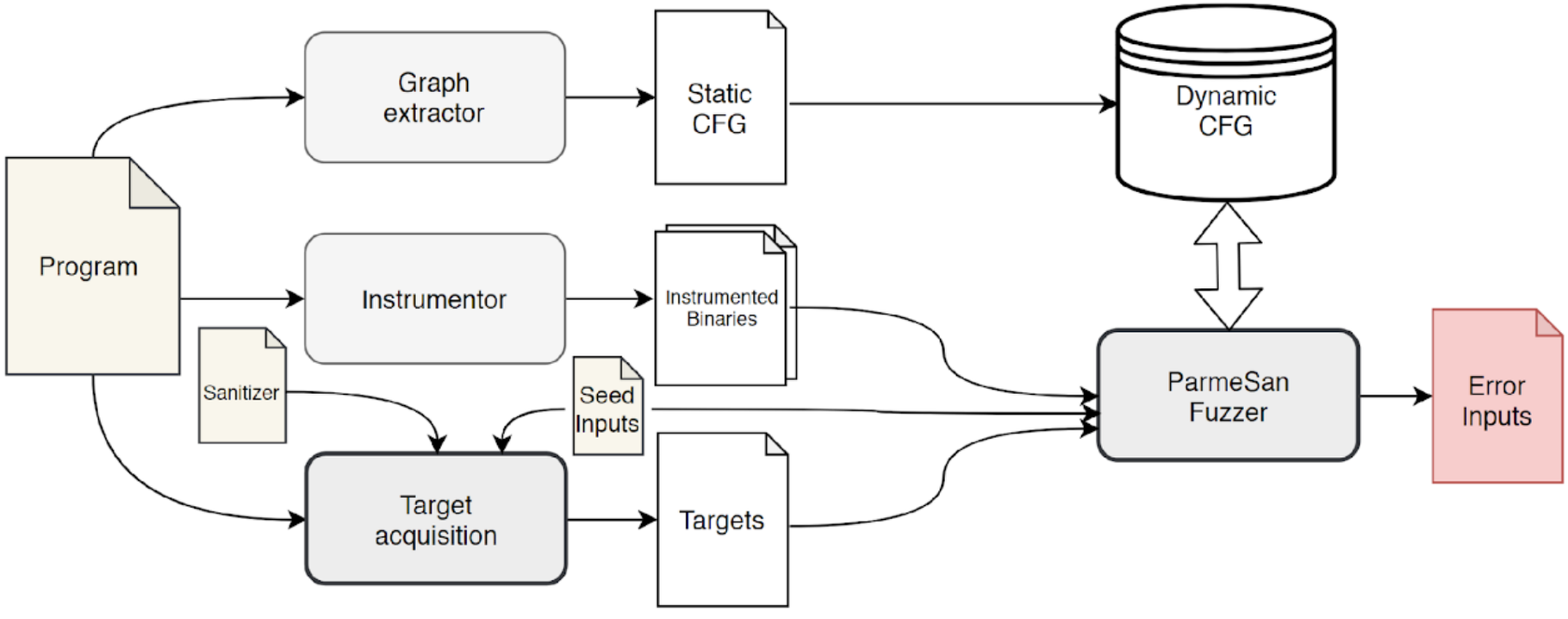


Figure 1: An overview of the ParmeSan fuzzing pipeline. The *target acquisition* step automatically obtains fuzzing targets. These targets are then fed to the ParmeSan fuzzer, which directs the inputs towards the targets by using the continuously updated *dynamic CFG*. The inputs to the pipeline consist of a target *program*, a *sanitizer*, and *seed inputs*.

Target Acquisition

- Statically compare Sanitizer-instrumented program and original program, instrumented points are target branch
  - Sanitizers instrument programs in two ways.
    - ✔ Update internal data structures (e.g., shadow memory)
    - ✔ Add a branch condition (e.g., ASan's out of bound access detection)

```
;... Non-sanitized
%4 = load i8*, i8** %2, align 8
%5 = getelementptr inbounds i8, i8* %4, i64 1
%6 = load i8, i8* %5, align 1
;..
```

```
; ... Sanitized with UBSan
%4 = load i8*, i8** %2, align 8
%5 = getelementptr inbounds i8, i8* %4, i64 1
%6 = ptrtoint i8* %4 to i64
%7 = add i64 %6,
%8 = icmp uge i64 %7, %6
%9 = icmp ult i64 %7, %6
%10 = select i1 true, i1 %8, i1 %9
br i1 %10, label %12, label %11

; <label>:11:                    ; preds = %1
call void @__ubsan_handle_pointer_overflow      (...)
br label %12

; ...
%17 = load i8, i8* %5, align 1
```

Target Acquisition

- Confirm sanitizer's ability to find real-world bugs
- Each kind of sanitizers target at one bug types

Prog	Bug	Type	Sanitizer (% non-target)					
			ASan		UBSan		TySan	
base64	LAVA-M	BO	✓	(5%)	✗	—	✗	—
who	LAVA-M	BO	✓	(9%)	✗	—	✗	—
uniq	LAVA-M	BO	✓	(15%)	✗	—	✗	—
md5sum	LAVA-M	BO	✓	(12%)	✗	—	✗	—
OpenSSL	2014-0160	BO	✓	(8%)	✗	—	✗	—
pcre2	-	UAF	✓	(7%)	✗	—	✗	—
libxml2	memleak	TC	✗	—	✗	—	✓	(80%)
libpng	oom	IO	✗	—	✓	(40%)	✗	—
libarchive	-	BO	✓	(17%)	✗	—	✗	—

Table 1: Bugs detected and percentage of branches that can be disregarded (i.e., are not on the path to an instrumented basic block) compared to coverage-oriented fuzzing. UAF= use-after-free, BO=buffer overflow, TC=type confusion, IO=integer overflow

Target Acquisition

- Target Pruning

  - Profile-guided pruning:
    Profile the target program and remove all the sanitizer checks on hot paths
  - Complexity-based pruning:
    Score functions based on how many instructions are added/modified by the sanitizer and mark targets that score higher than others as more interesting.

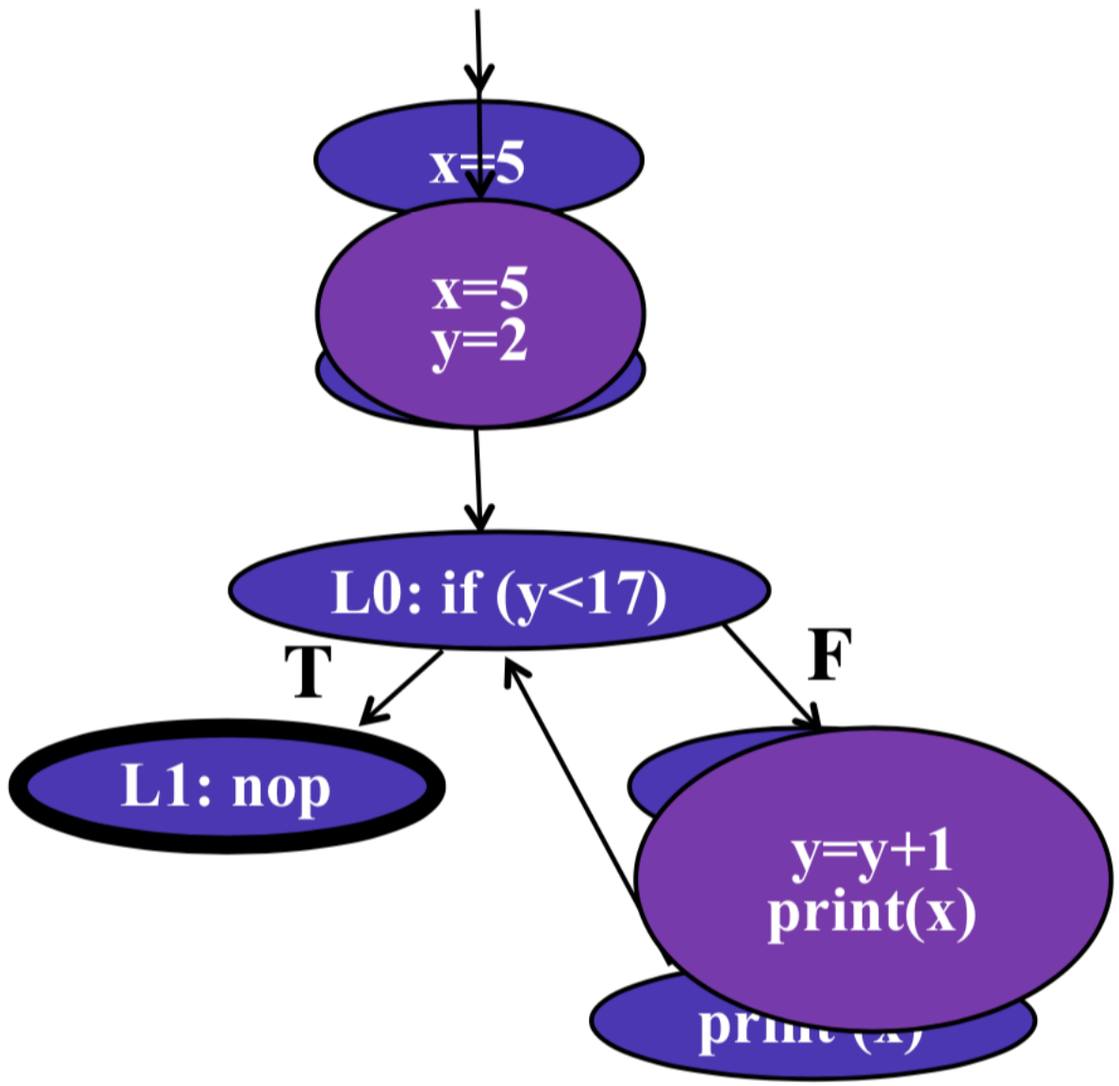
- Example

  - for base64 program in LAVA-M, top 3 targets are lava_get() , lava_set() , and emit_bug_reporting_address(), the first 2 triggers bugs

Dynamic CFG

- CFG construction
  - Start with the CFG that is statically generated by LLVM

```
      x=5
      y=2
L0: if (y< 17) goto L1
      y=y+1
      print (x)
      goto L0
L1: nop
```



Dynamic CFG

- CFG construction
  - Start with the CFG that is statically generated by LLVM
  - Adding edges as the program executes during fuzzing

```
typedef int(*fun_t)(int);
int foo(int a)
{
    printf("hello jack: %d\n", a);
    return a;
}
class CTargetObject
{
public:
    fun_t _fun;
};
int _tmain(int argc, _TCHAR* argv[])
{
    int i = 0;
    CTargetObject* o_array = new CTargetObject[5];
    for (i = 0; i < 1000; i++)
        o_array[i]._fun = foo;
    o_array[0]._fun(1);
    return 0;
}
```

Dynamic CFG

- CFG construction
  - Start with the CFG that is statically generated by LLVM
  - Adding edges as the program executes during fuzzing
  - Distance calculation:

    Use the number of conditionals between a starting point and the target

    Conditional Graph (CG)
- Distance Metric

$$d(c) = \begin{cases} 0 & \text{if } c \in Targets \\ \infty & \text{if } N(c) = \emptyset \\ \frac{(\sum_{n \in N(c)} d(n)^{-1})^{-1}}{|N(c)|} + 1 & \text{otherwise} \end{cases}$$

N(c): the set of successors of c with a path to at least one of the targets,

- Augmented with DFA

Sanitizer-guided Fuzzer

- End-to-end workflow
  - A short coverage-oriented exploration and tracing phase to get the CFG
  - A directed exploration phaseto reach the target basic blocks
  - An exploitation phase which gradually starts when any of the specified targets are reached.

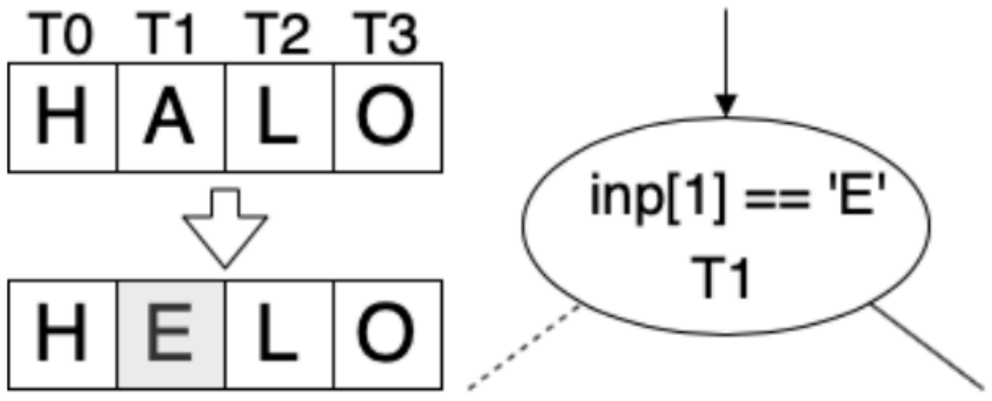


Figure 2: Example of DFA mutation. The taint label ($T1$) is recorded at a newly uncovered conditional, allowing the fuzzer to learn that the value should be either fixed to `E` or mutated further.

Sanitizer-guided Fuzzer

- Input Prioritization
  - Maintaining a queue of (input, condition)
    - The queue is sorted based on a tuple consisting of ***(runs, distance)***
    - ***runs*** is the number of times this entry has been popped from the queue
    - ***distance*** is the calculated distance of the conditional to our targets obtained by using our dynamic CFG.
    - Using the number of runs as the first key when sorting.
    - Mutate the selected seed (as provided by DFA)

Evaluation

- ParmeSan v.s. Other Directed Fuzzers

CVE	Fuzzer	Runs	*p*-val	Mean TTE
OpenSSL				
2014-0160	ParmeSan	30		5m10s
	HawkEye	—		—
	AFLGo	30	0.006	20m15s
Binutils				
2016-4487 2016-4488	ParmeSan	30		35s
	HawkEye	20		2m57s
	AFLGo	30	0.005	6m20s
2016-4489	ParmeSan	30		1m5s
	HawkEye	20		3m26s
	AFLGo	30	0.03	2m54s
2016-4490	ParmeSan	30		55s
	HawkEye	20		1m43s
	AFLGo	30	0.01	1m24s
2016-4491	ParmeSan	10		1h10m
	HawkEye	9		5h12m
	AFLGo	5	0.003	6h21m
2016-4492 2016-4493	ParmeSan	30		2m10s
	HawkEye	20		7m57s
	AFLGo	20	0.003	8m40s
2016-6131	ParmeSan	10		1h10m
	HawkEye	9		4h49m
	AFLGo	5	0.04	5h50m

Table 2: Reproduction of earlier results in crash reproduction in greybox fuzzers. We manually select the target and show the mean time-to-exposure.

- Target: Show the availability of DFA information alone improves directed fuzzing
- ParmeSan skips its target acquisition step
- Conclusion: ParmeSan significantly improves the TTE of bugs even for traditional directed fuzzing.

Evaluation

- ParmeSan v.s Coverage-guided Fuzzers

Prog	Type	Runs	AFLGo		NEUZZ		QSYM		Angora		ParmeSan	
boringssl	UAF	10	2281	2h32m	2520	1h20m	2670	3h20m	2510	45m	1850	25m
c-ares	BO	10	202	5s	275	3s	280	20s	270	1s	200	1s
freetype2	IO	5	✗	✗	✗	✗	✗	✗	57330	47h	49320	43h
pcre2	UAF	10	9023	25m	31220	16m	32430	1h20m	30111	15m	8761	8m
lcms	BO	10	1079	6m	2876	1m50s	3231	7m	2890	2m	540	41s
libarchive	BO	10	4870	1h12m	5945	1h20m	✗	✗	6208	22m	4123	13m
libssh	ML	10	365	3m10s	419	43s	631	2m32s	341	32s	123	50s
libxml2	BO	10	5780	51m	7576	25m	12789	2h5m	5071	20m	2701	11m
libxml2	ML	10	5755	30m	10644	19m	11260	1h10m	10580	20m	2554	17m
openssl-1.0.1f	BO	10	550	50m	814	10m12s	853	5h25m	793	5m	543	3m4s
openssl-1.0.1f	ML	10	1250	1m	717	40s	4570	23m	720	40s	709	37s
proj4	ML	10	82	7m30s	83	1m55s	86	10m5s	83	1m40s	80	1m26s
re2	BO	10	5172	47m	5178	50m	7610	2h	4073	21m	3267	12m35s
woff2	BO	10	91	45m	94	31m20s	98	41m	90	15m	83	8m
woff2	OOM	10	50	2m	50	22s	53	1m45s	50	20s	49	12s
Geomean diff			**+16%**	**+288%**	**+40%**	**+81%**	**+95%**	**+867%**	**+33%**	**+37%**		

- Target: ParmeSan finds bugs faster than coverage-guided fuzzers.
- Benchmark: Google fuzzer-test-suite
- Use ASan for ParmeSan's target acquisition step

Evaluation

- Sanitizer Impact

Bug	Type	Sanitizer	Targets	Covered	μTTE
CVE-2014-0160	BO	ASan	533	✓	5m
		UBSan	120	✗	6m
		TySan	5	✗	6m
CVE-2015-8317	BO	ASan	352	✓	10m
		UBSan	75	✗	50m
		TySan	30	✗	50m
pcre2	UAF	ASan	122	✓	10m
		UBSan	52	✗	20m
		TySan	12	✓	8m
freetype2	IO	ASan	437	✗	47h
		UBSan	48	✓	20h
		TySan	71	✗	>48h
CVE-2011-1944	IO	ASan	230	✓	30s
		UBSan	125	✓	20s
		TySan	8	✗	50s
CVE-2018-13785	IO	ASan	450	✗	11h
		UBSan	45	✓	32m
		TySan	31	✗	5h
libssh	ML	ASan	590	✗	31s
		UBSan	57	✗	33s
		TySan	13	✗	35s
		LSan	104	✓	25s
libxml	ML	ASan	352	✗	15m
		UBSan	75	✗	22m
		TySan	30	✗	25m
		LSan	191	✓	12m
openssl	ML	ASan	533	✗	40s
		UBSan	120	✗	50s
		TySan	5	✗	43s
		LSan	191	✓	32s
proj4	ML	ASan	729	✗	1m30s
		UBSan	170	✗	1m55s
		TySan	373	✗	2m10s
		LSan	43	✓	57s

Table 5: Bugs found by ParmeSan using different sanitizers in the analysis stage. ✓ in targets, bug found; ✗ not in targets, bug found; For the memory leak (ML) bugs we also show the performance of LeakSanitizer.

Evaluation

- Ability to detect new bugs

Prog	Version	Bugs	NEUZZ		QSYM		Angora		ParmeSan	
			1h	24h	1h	24h	1h	24h	1h	24h
OSS Fuzz [39]										
curl	54c622a	1	0	0	0	0	0	0	0	1
json-c	ddd0490	0	0	0	0	0	0	1	1	1
libtiff	804f40f3	1	0	0	0	0	0	1	1	1
libxml2	1fbcf40	2	0	0	0	0	0	1	1	2
libpcap	c0d27d0	1	0	0	0	0	0	1	1	1
OpenSSL	6ce4ff1	1	0	0	0	1	0	1	1	1
ffmpeg	9d92403	0	0	0	0	0	0	0	0	0
harfbuzz	b21c5ef	0	0	0	0	0	0	0	0	0
libpng	3301f7a1	0	0	0	0	0	0	0	0	0
Targets from prior work [3, 12, 32]										
jhead	3.03	2	0	2	0	2	2	2	2	2
pbc	0.5.14	37	9	9	2	12	10	29	23	37
protobuf-c	1.3.1	1	0	0	0	0	1	1	1	1

Table 6: New bugs found within 1h and 24h by ParmeSan and other state-of-the-art fuzzers. The version is denoted by either a version number or a commit id. In total ParmeSan found 47 new bugs.

Conclusion

- ParmeSan: Sanitizer-guided fuzzer.
- Directed target: Sanitizer-instrumented
- Fuzzing phase:
  - First: Construct a precise CFG dynamically
  - Second: DFA for fuzzing