ParmeSan

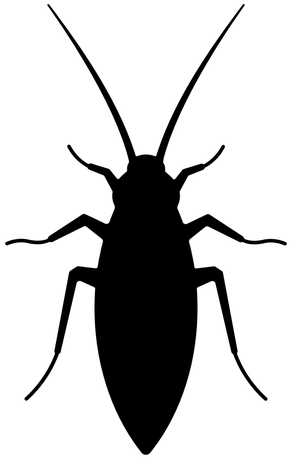
Sanitizer-guided Greybox Fuzzing

Sebastian Österlund, Kaveh Razavi, Herbert Bos, Cristiano Giuffrida

Vrije Universiteit Amsterdam

In fuzzing:
Coverage == Bugs?

Can we do better?



In fuzzing:

Cover

Actively direct the fuzzing towards bugs

Can we do better?

In fuzzing:

Covera

Actively direct the fuzzing towards bugs

Can we do better?

=> ~40% faster at finding bugs

How do we test for bugs?

How do we test for bugs?

Sanitizers

Sanitizers

- Run-time checks for bugs
- Add some instrumentation
- Plenty of these available:
  - ASan
  - UBSan
  - TySan
  - MyVeryNicheSan

```
;... 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
```

Sanitizers for fuzzing

Sanitizers for fuzzing

- Fuzzers get better -> harder to find new crashes

Sanitizers for fuzzing

- Fuzzers get better -> harder to find new crashes
- **Nowadays very common to use sanitizers when fuzzing**

Sanitizers for fuzzing

- Fuzzers get better -> harder to find new crashes
- **Nowadays very common to use sanitizers when fuzzing**
- Are able to catch most types of bugs

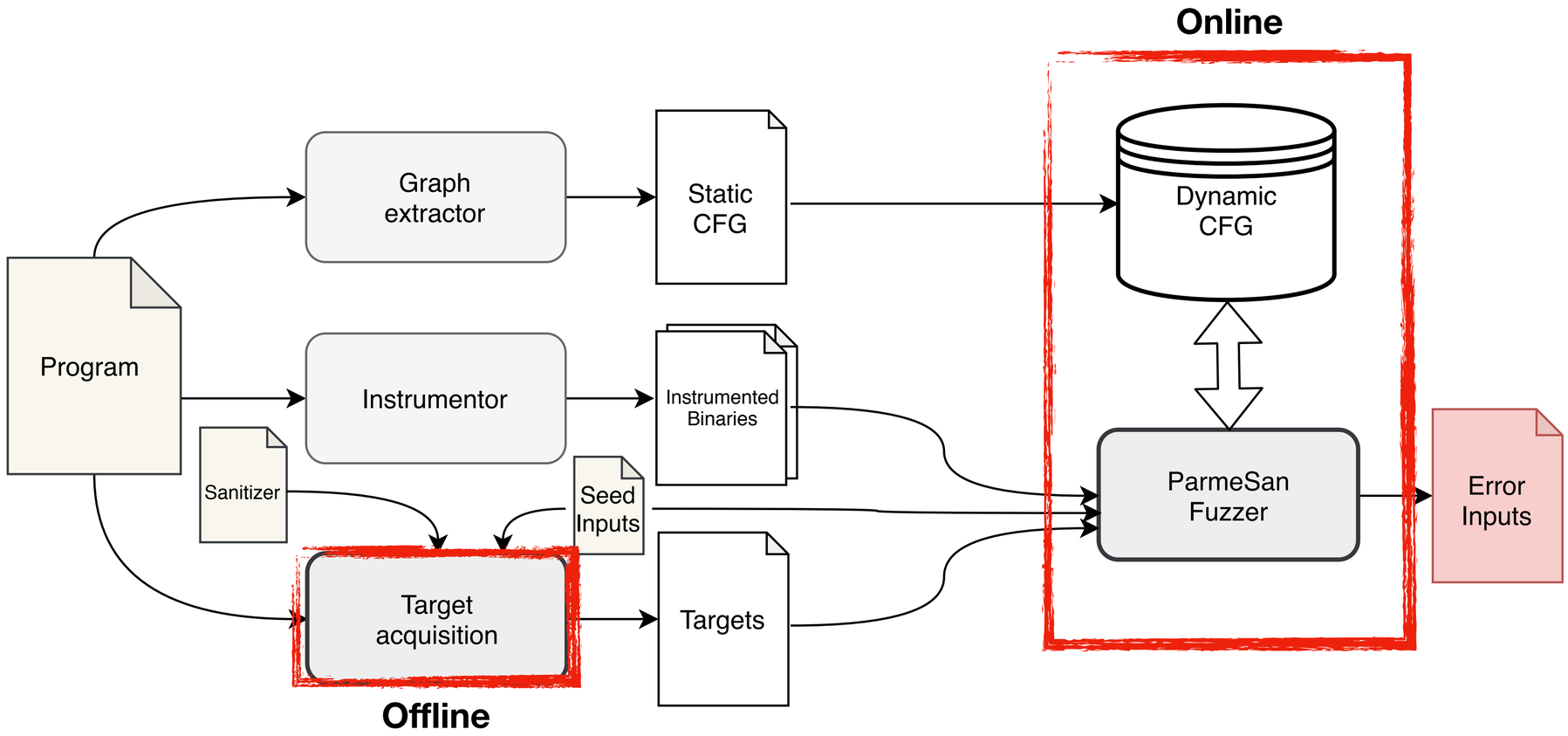
Prog	Bug	Type	Sanitizer (% non-target)					
			ASan		UBSan		TySan	
base64	LAVA-M	BO	✓	(5%)	✗	—	✗	—
who	LAVA-M	BO	✓	(9%)	✗	—	✗	—
uniq	LAVA-M	BO	✓	(15%)	✗	—	✗	—
md5sum	LAVA-M	BO	✓	(12%)	✗	—	✗	—
OpenSSL	2014-0160	BO	✓	(8%)	✗	—	✗	—
pcre2	-	UAF	✓	(7%)	✗	—	✗	—
libxml2	memleak	TC	✗	—	✗	—	✓	(80%)
libpng	oom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

Sanitizers for fuzzing

- Fuzzers get l
  crashes
- **Nowadays very common to use sanitizers when fuzzing**
- Are able to catch most types of bugs

Let's target these sanitizer checks!

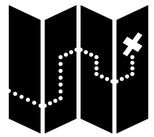
Prog	Bug	Type	Sanitizer (% non-target)					
			ASan		UBSan		TySan	
					✗	—	✗	—
					✗	—	✗	—
					✗	—	✗	—
					✗	—	✗	—
OpenSSL	2014-0160	BO	✓	(8%)	✗	—	✗	—
pcre2	-	UAF	✓	(7%)	✗	—	✗	—
libxml2	memleak	TC	✗	—	✗	—	✓	(80%)
libpng	oom	IO	✗	—	✓	(40%)	✗	—
libarchive	-	BO	✓	(17%)	✗	—	✗	—



ParmeSan

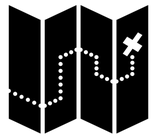
Pipeline

Target acquisition



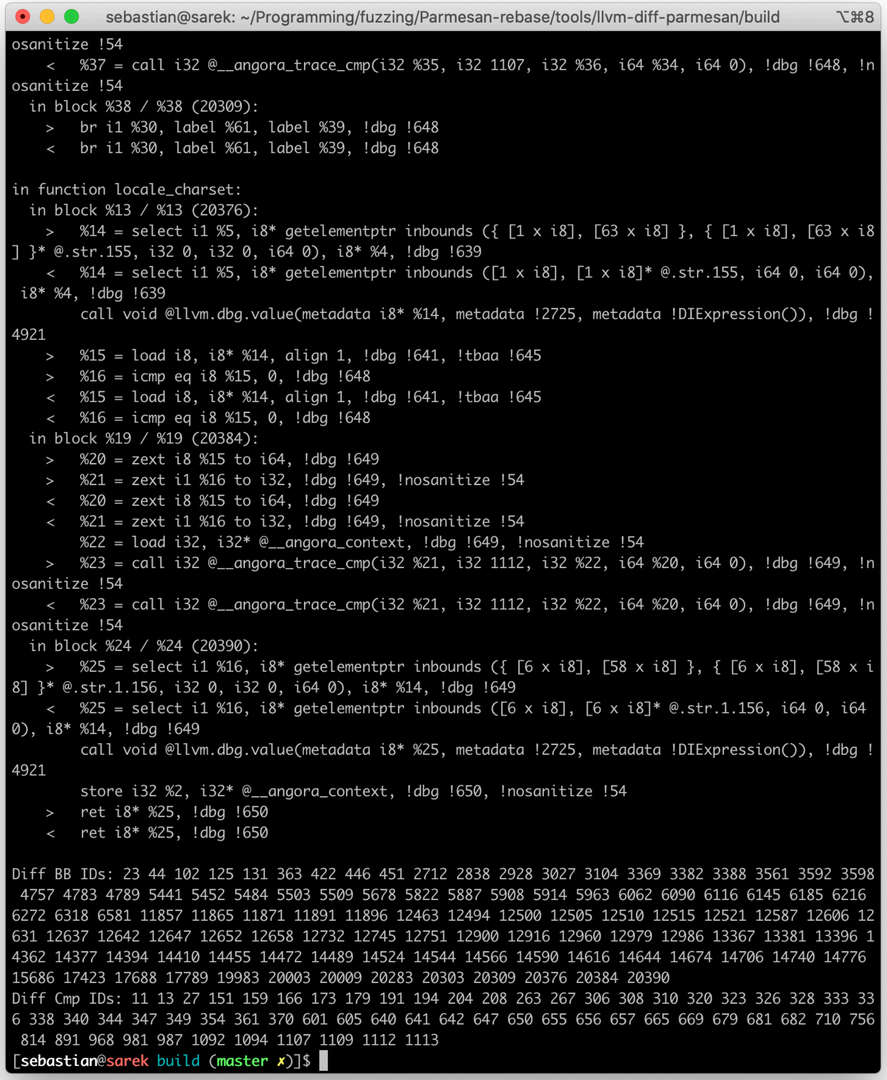
Target acquisition

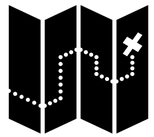
- **Target** branches where **sanitizers** add instrumentation



Target acquisition

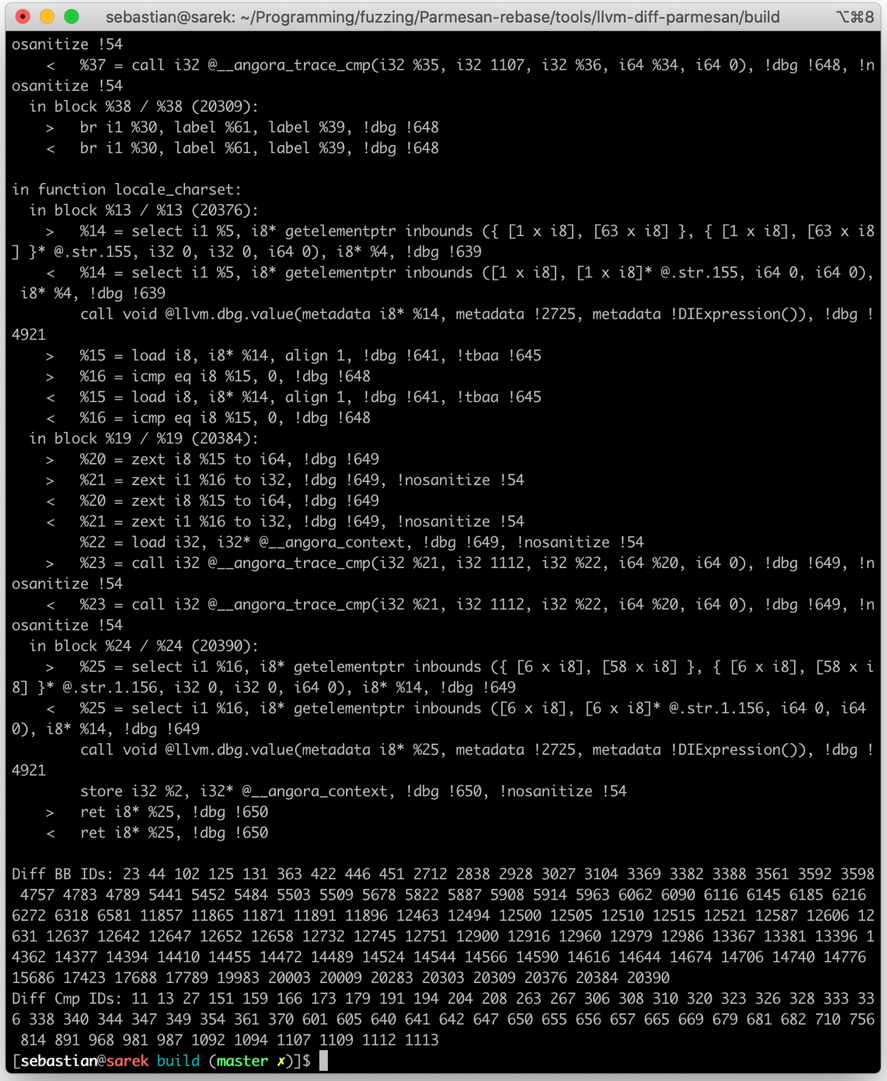
- **Target** branches where **sanitizers** add instrumentation
- **llvm-diff** to find locations



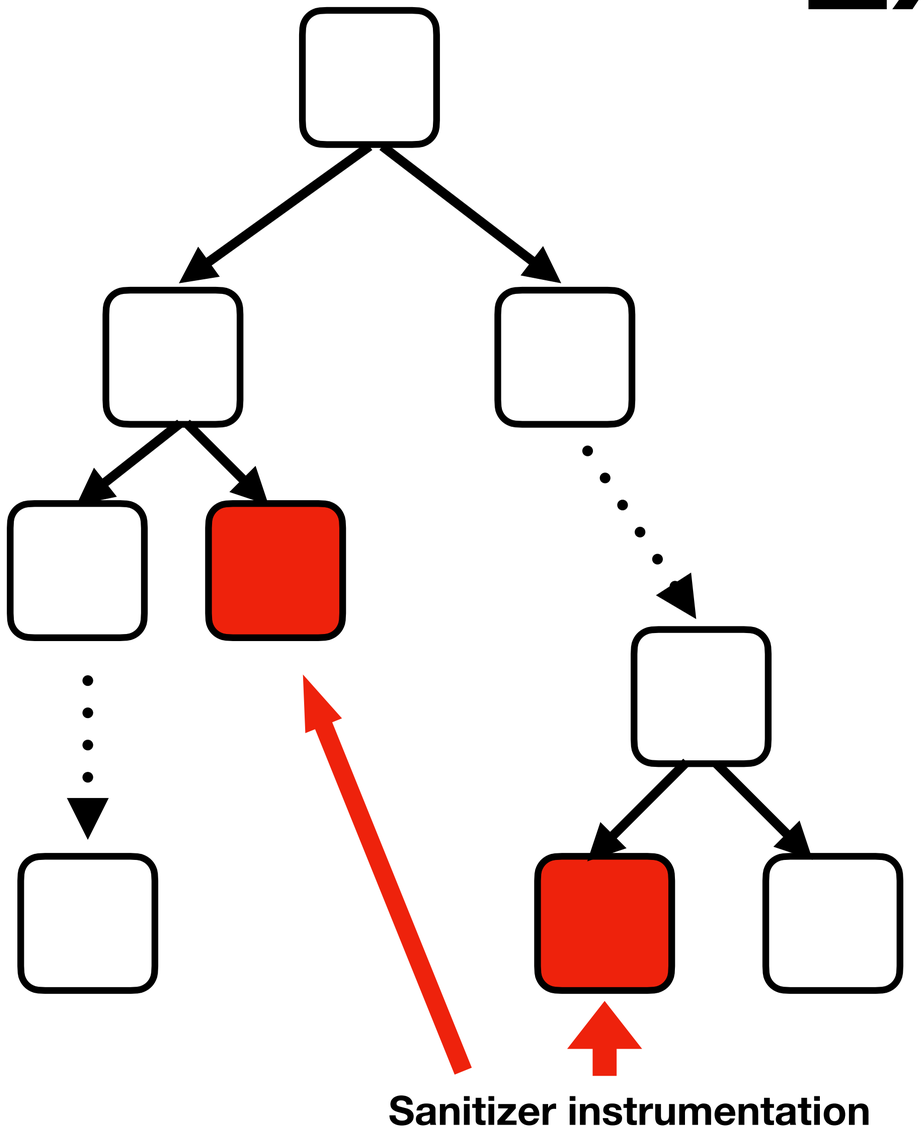


Target acquisition

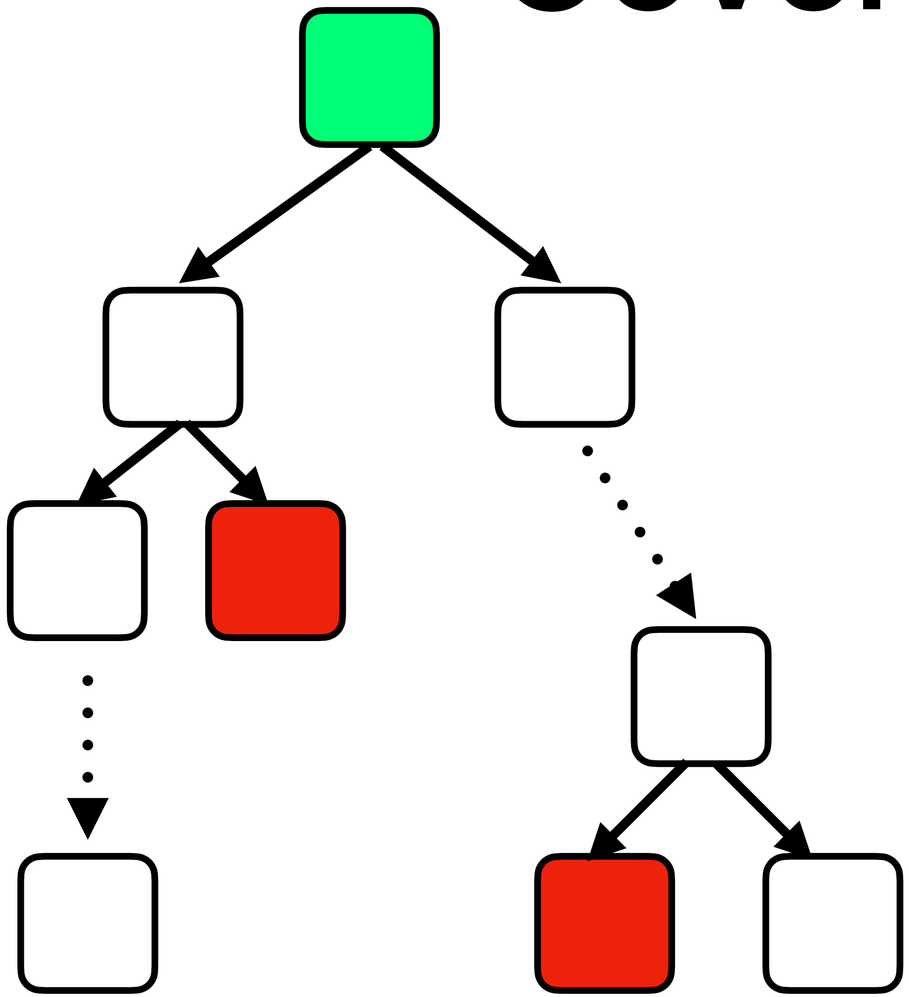
- **Target** branches where **sanitizers** add instrumentation
- **llvm-diff** to find locations
- Generic, works for **any** sanitizer



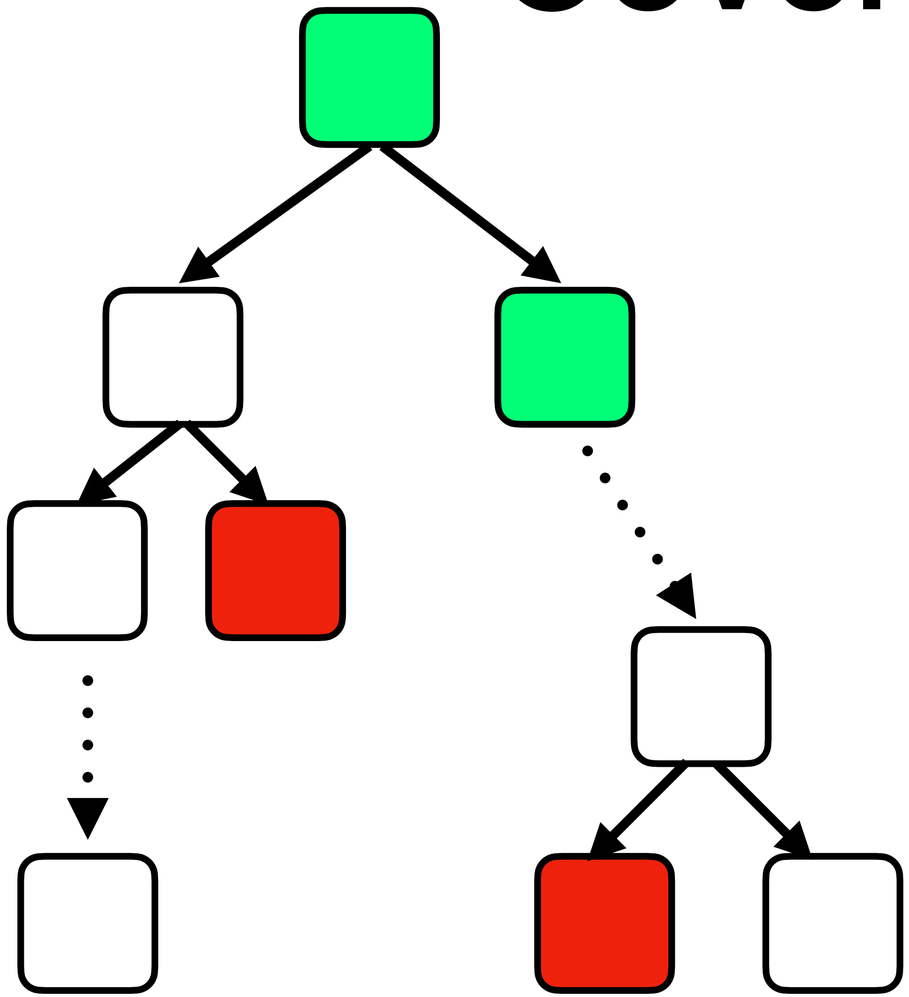
Example



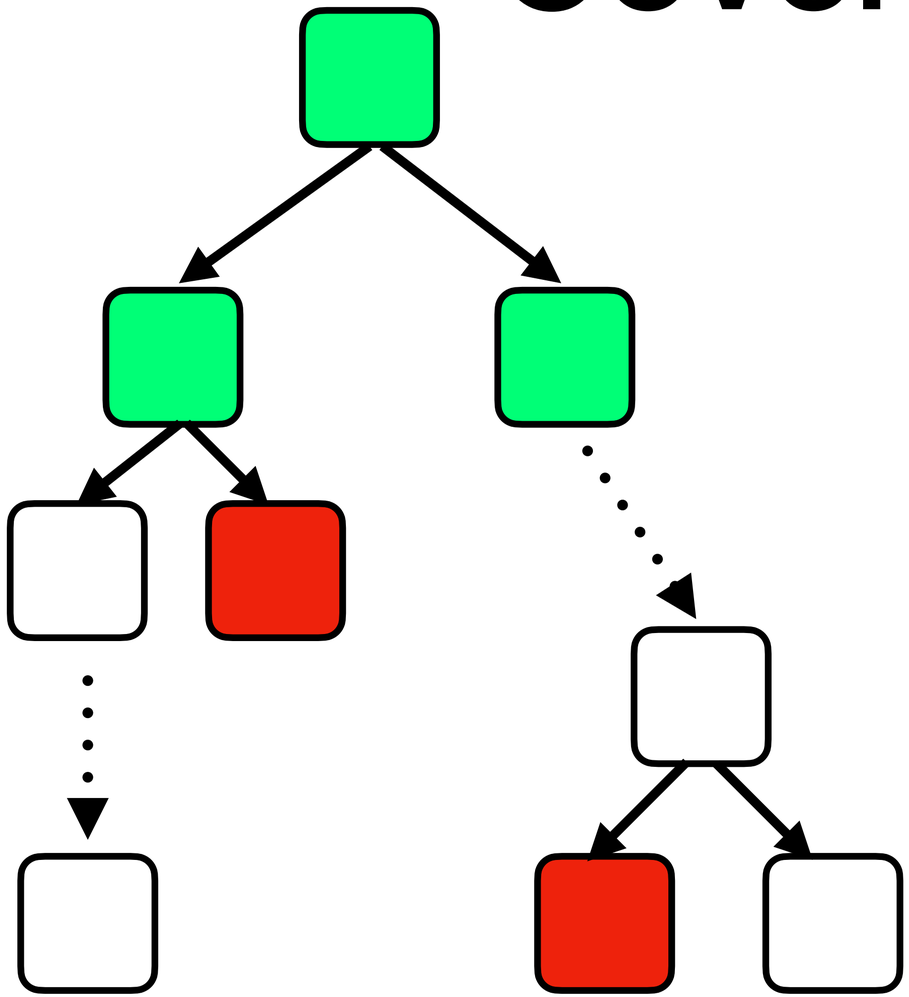
Coverage-guided



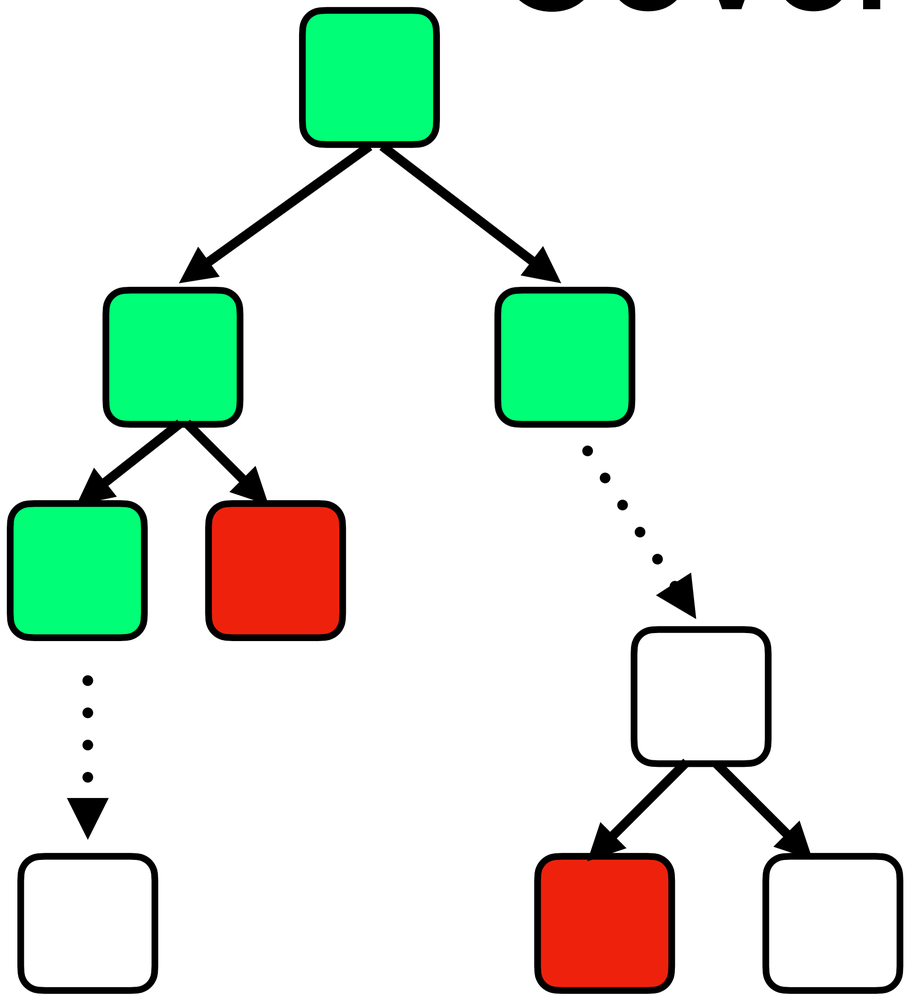
Coverage-guided



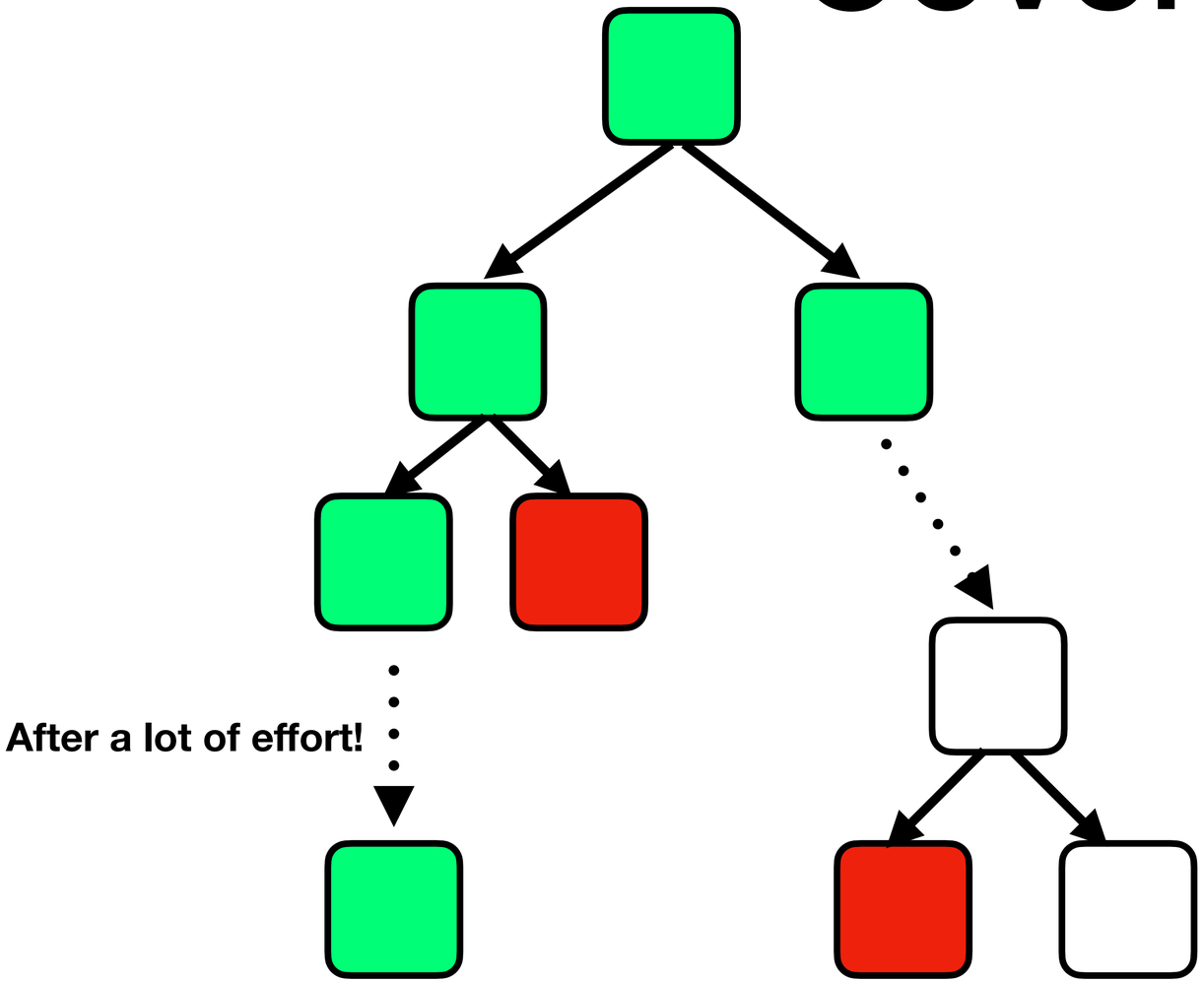
Coverage-guided



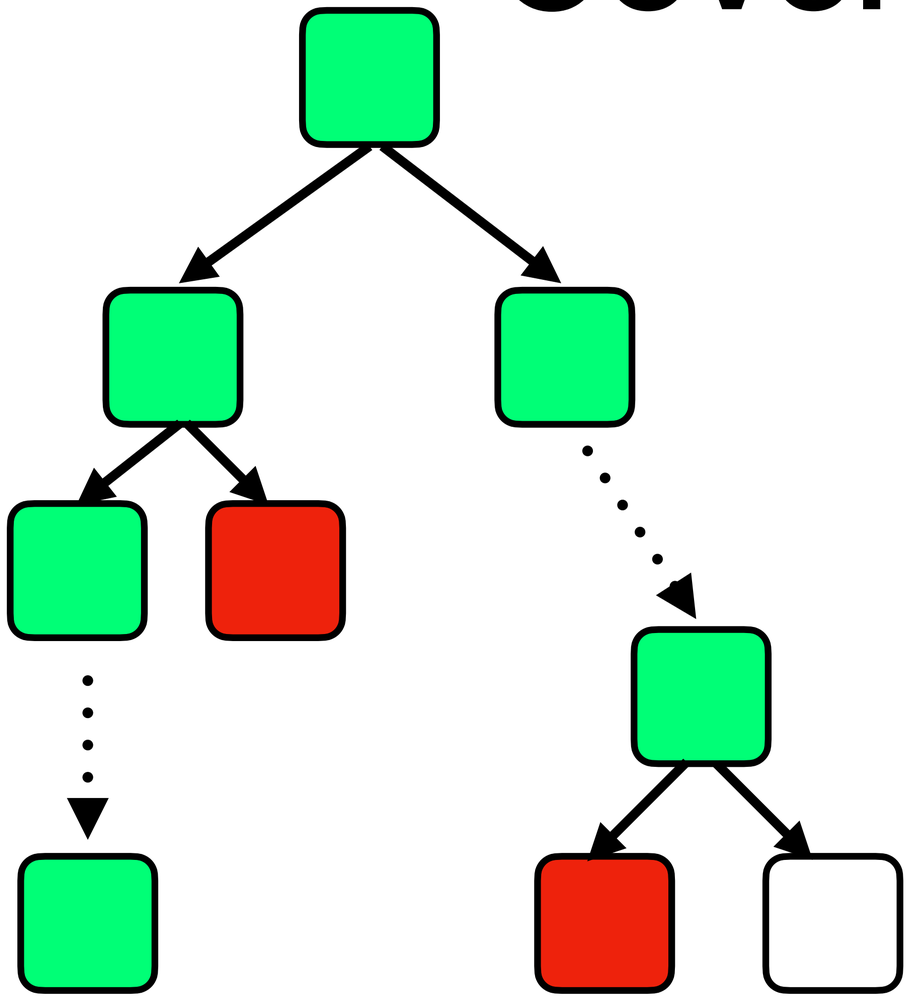
Coverage-guided



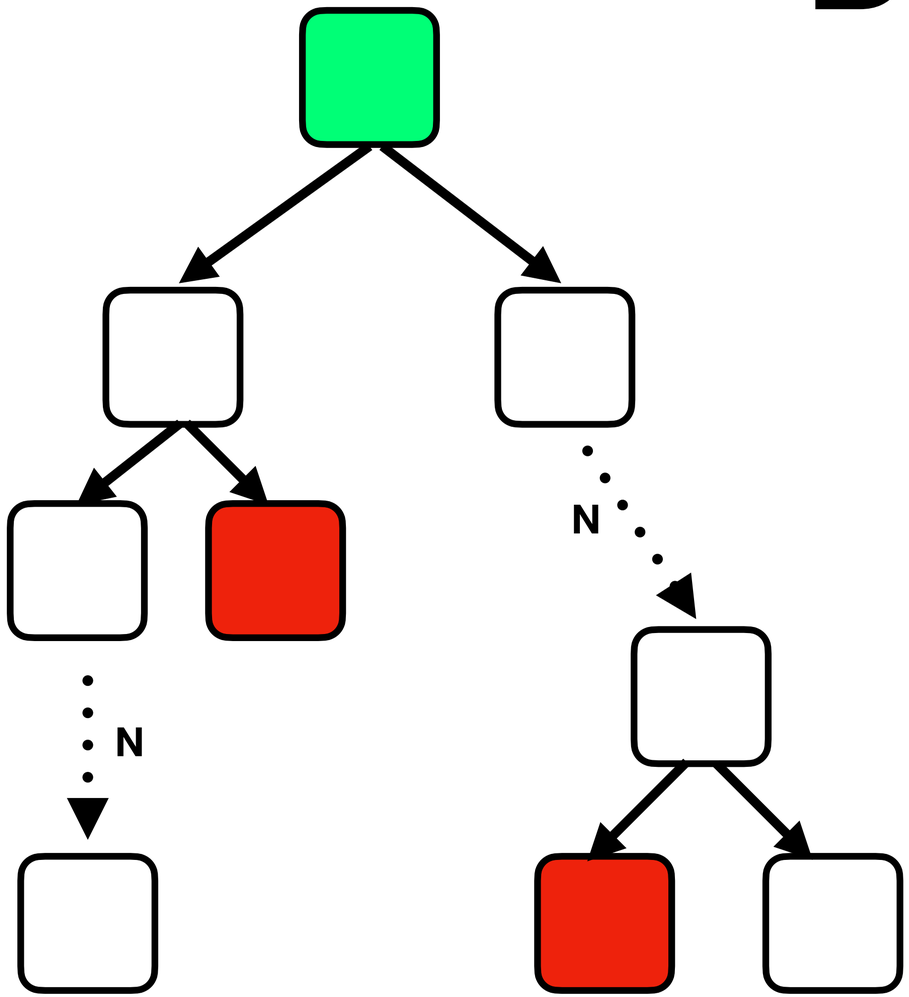
Coverage-guided



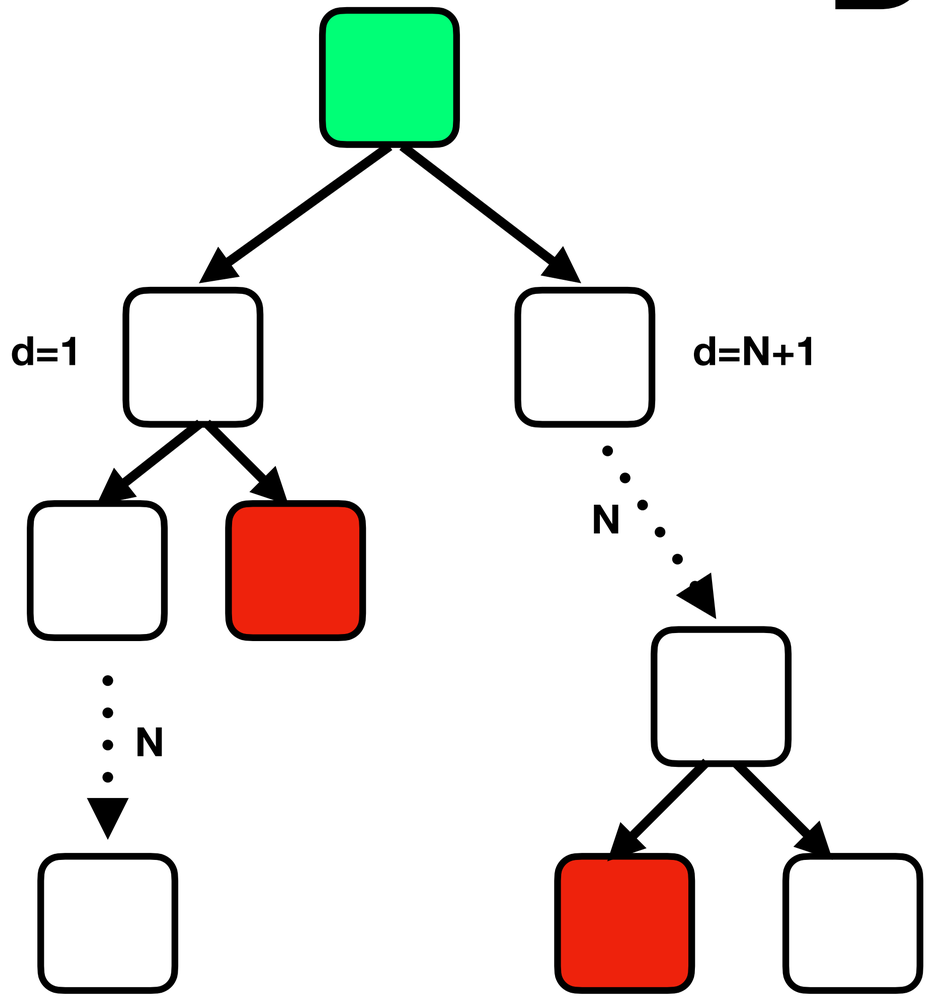
Coverage-guided



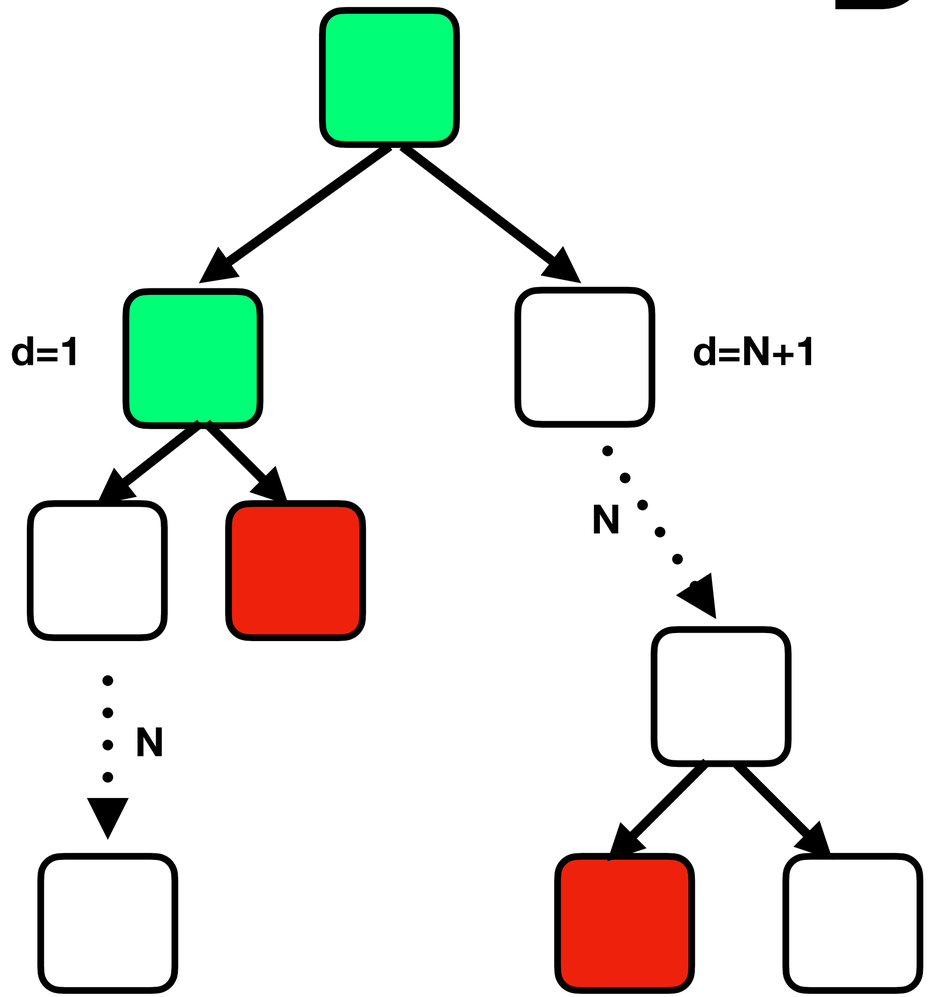
Directed



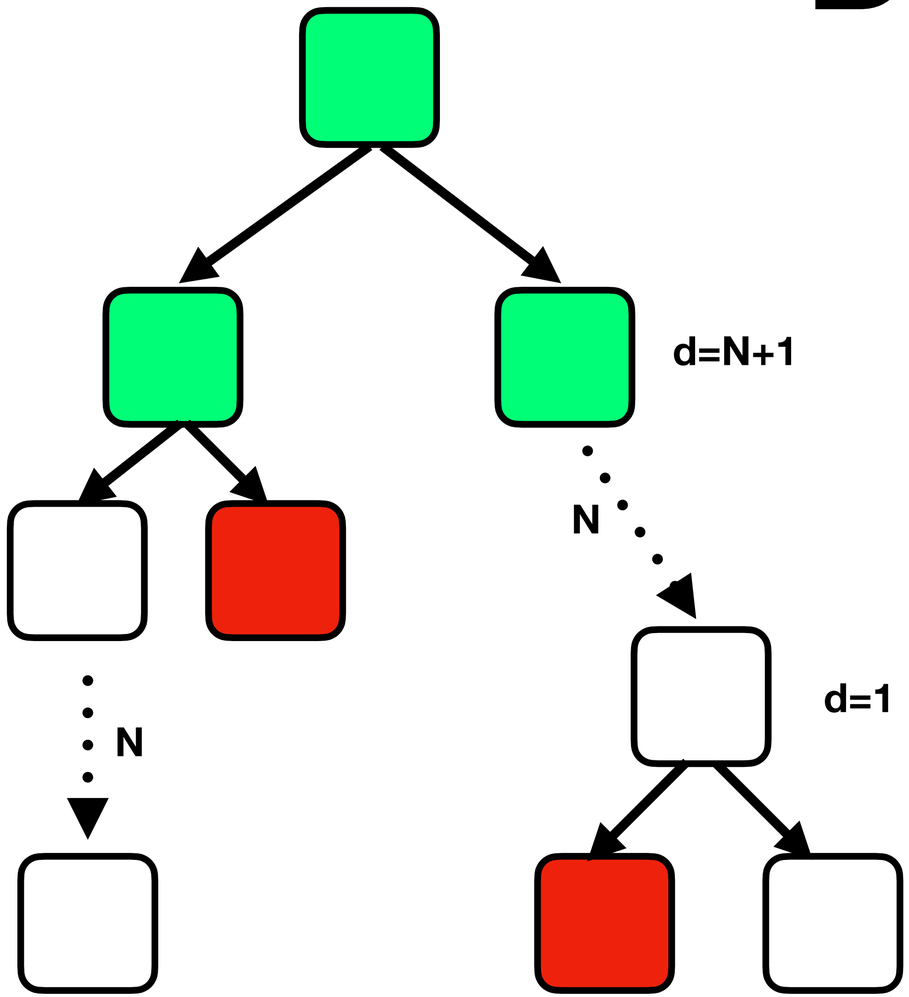
Directed



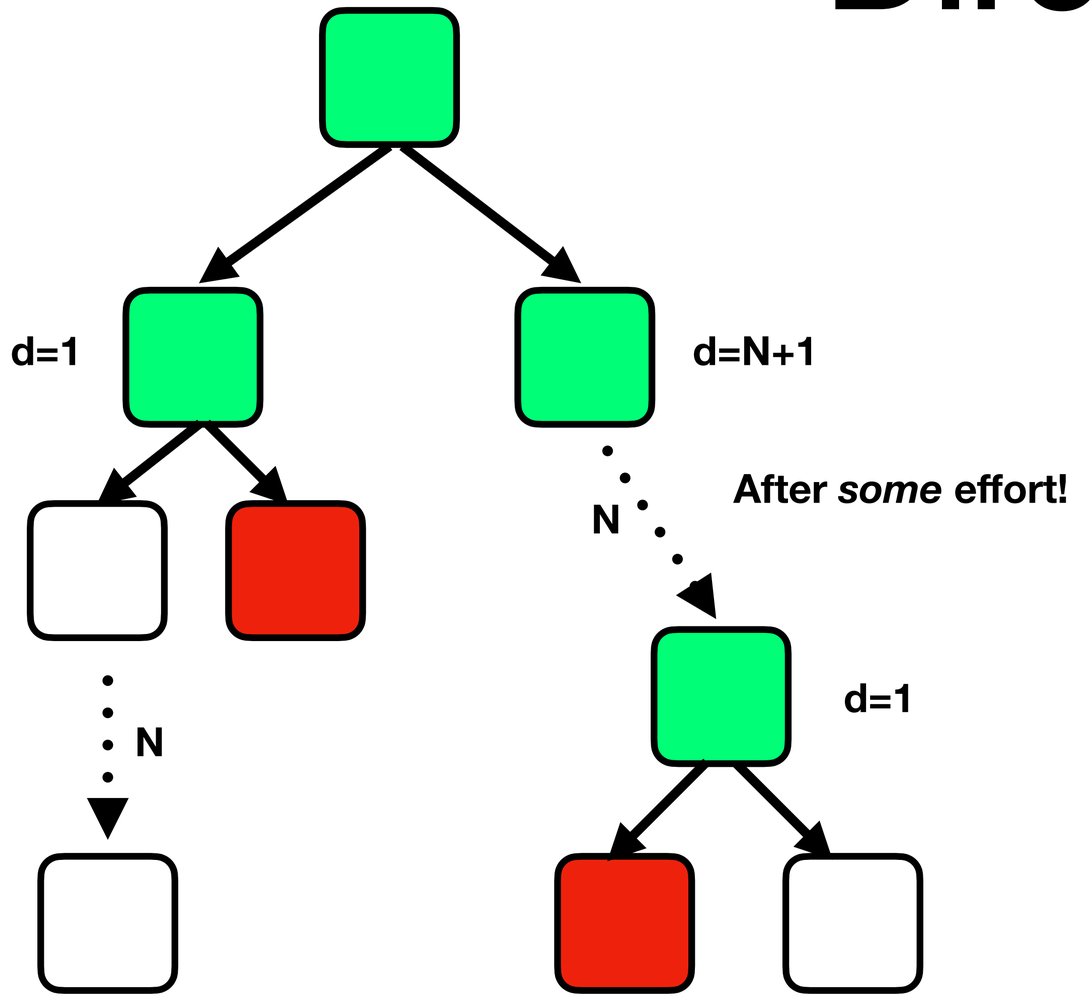
Directed



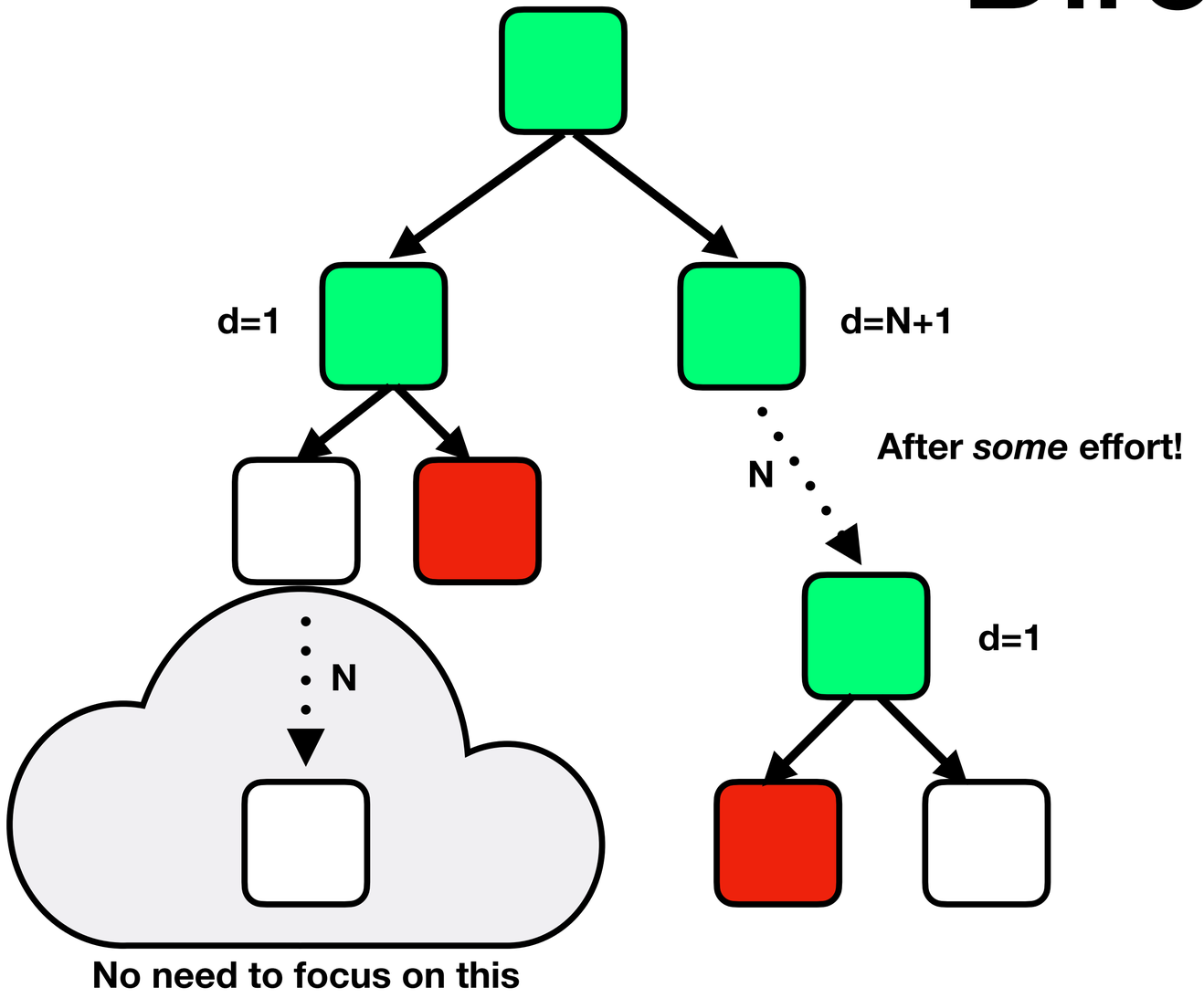
Directed



Directed



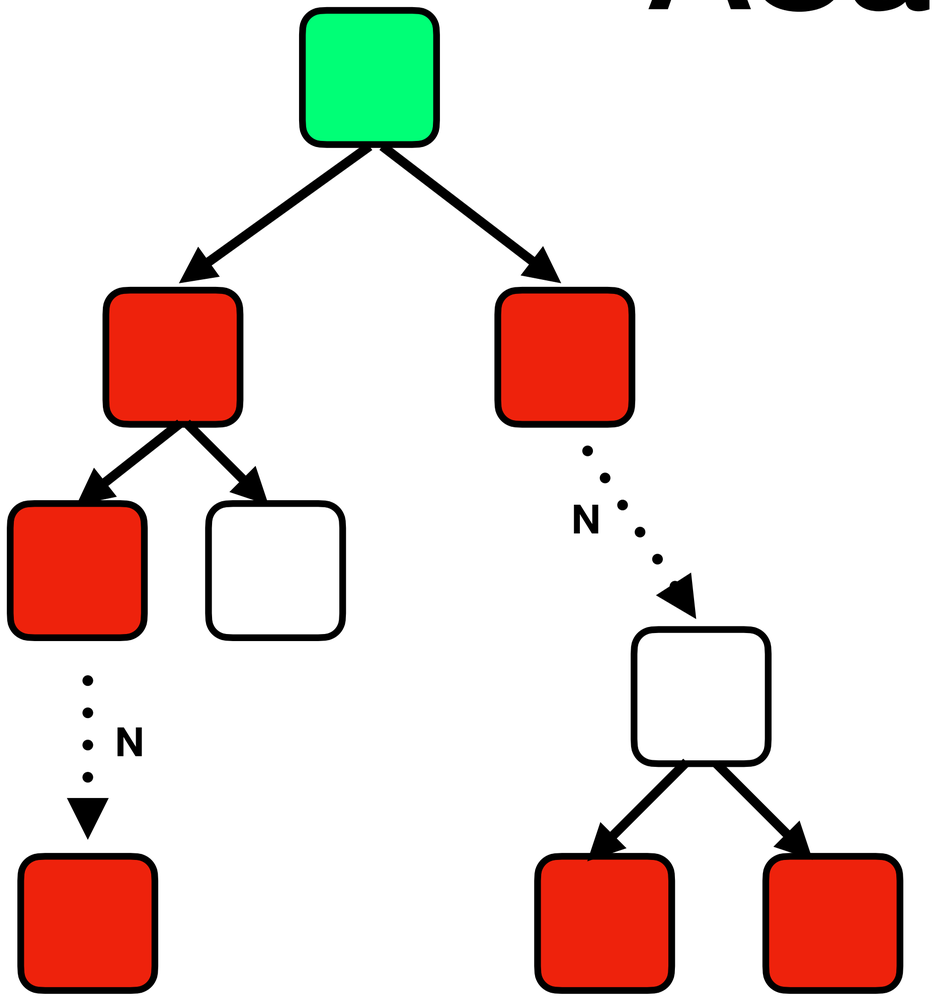
Directed



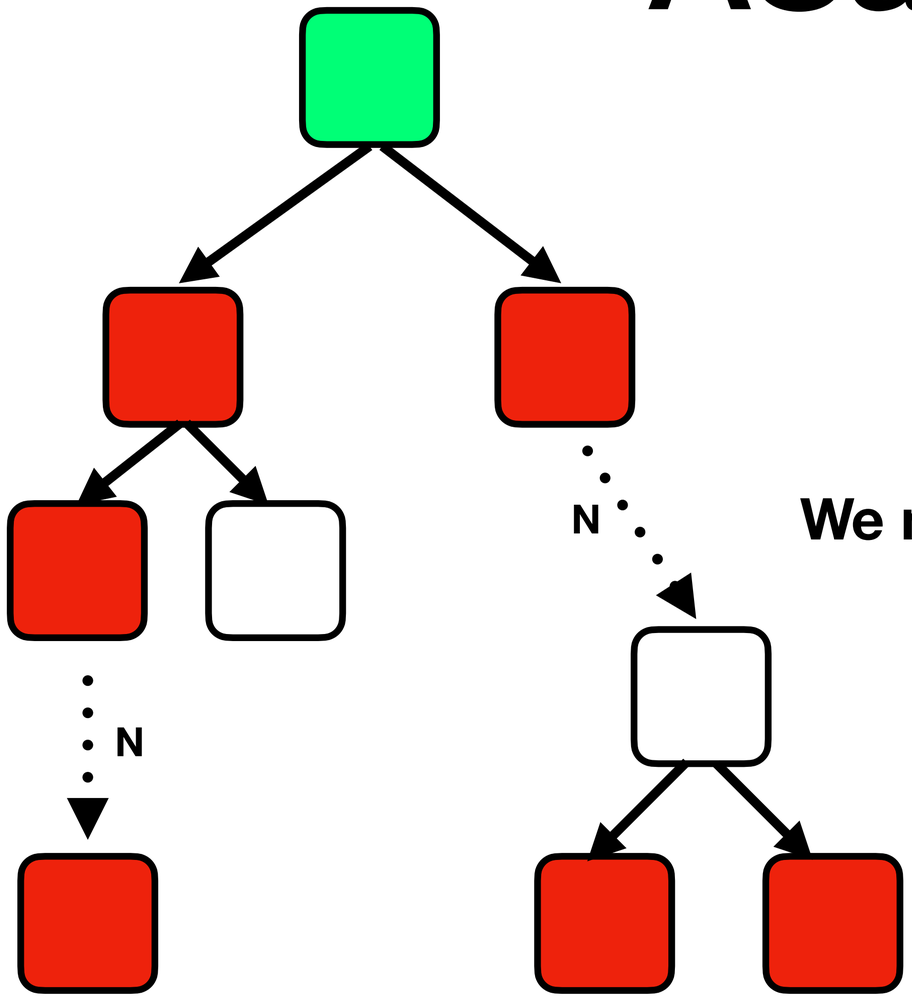
What about overly-eager sanitizers??

ehm… ASan

ASan targets



ASan targets



We might as well do coverage-guided fuzzing!!!

Target pruning

Target pruning

- **Profiling**-based pruning

Target pruning

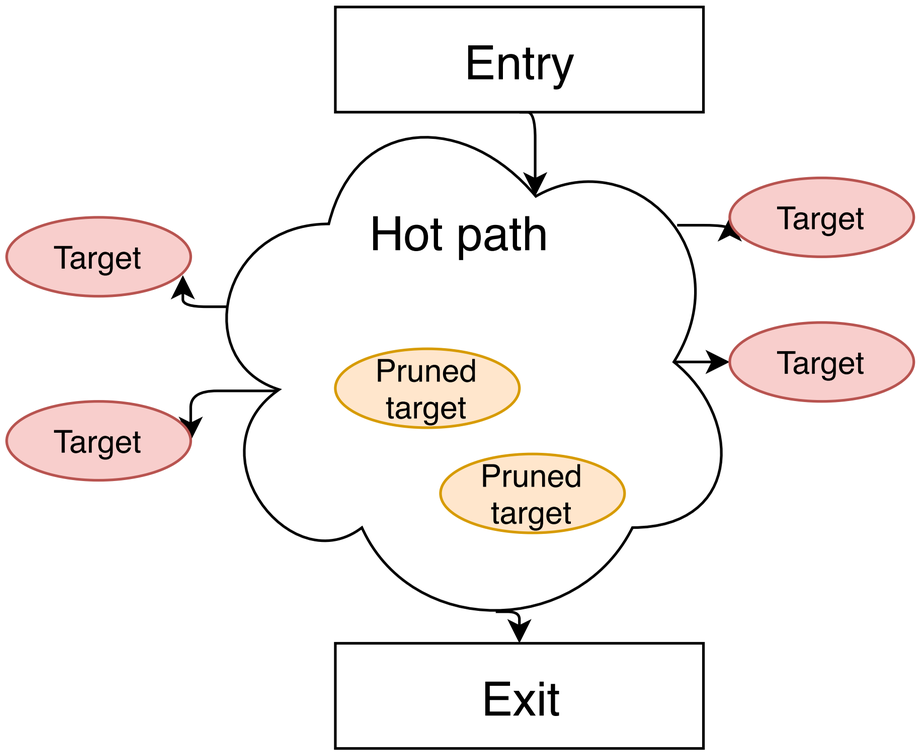
- **Profiling**-based pruning
  - **Intuition**: prefer “cold” code

Target pruning

- **Profiling**-based pruning
  - **Intuition**: prefer “cold” code
  - Hot paths are reached anyways

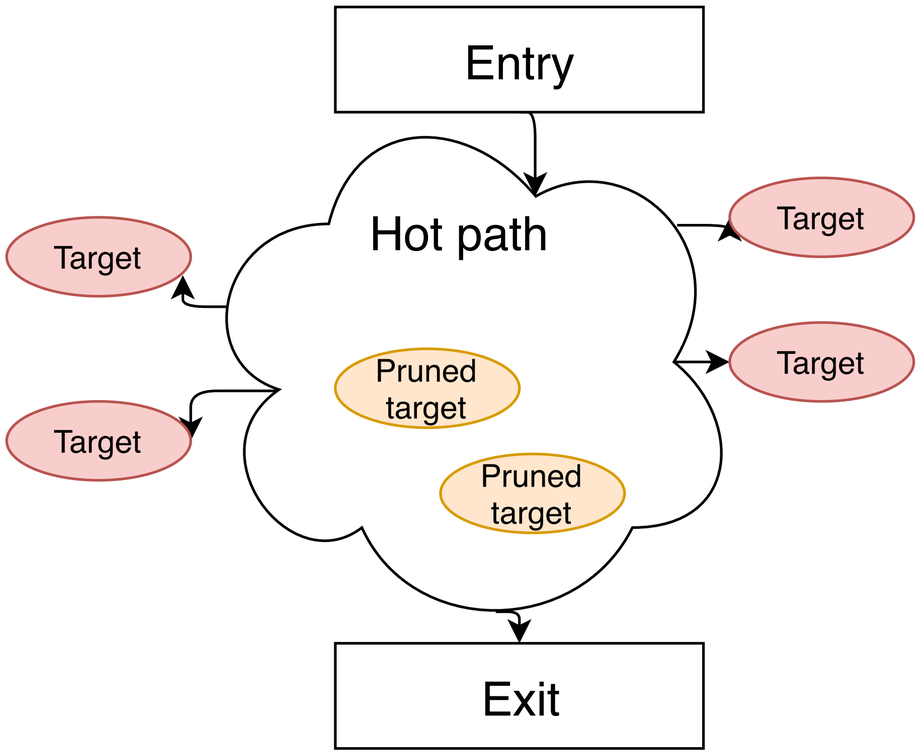
Target pruning

- **Profiling**-based pruning
  - **Intuition**: prefer “cold” code
  - Hot paths are reached anyways



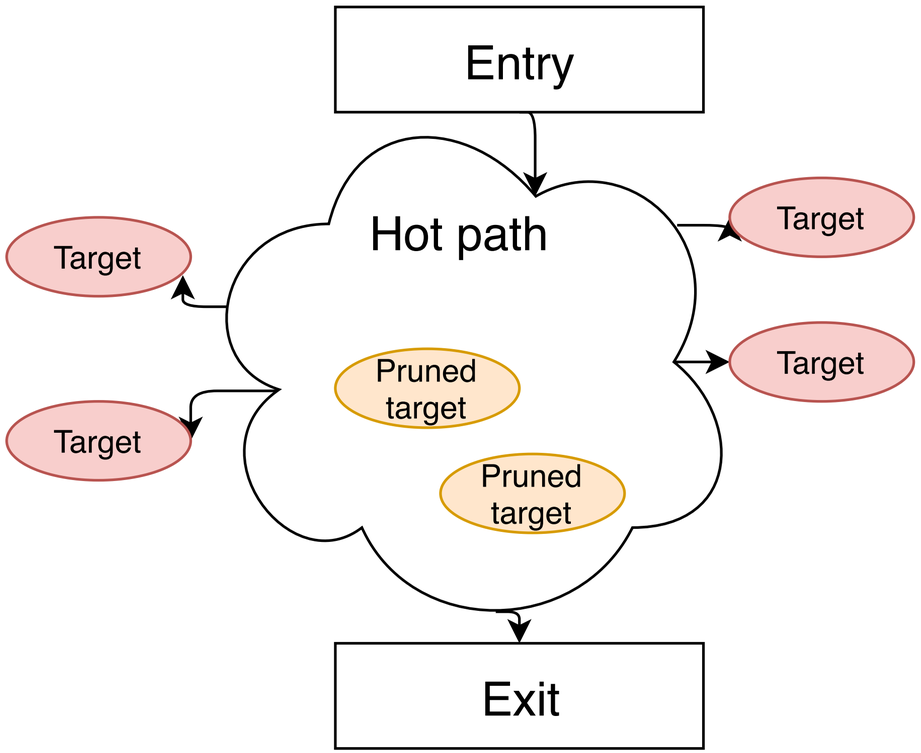
Target pruning

- **Profiling**-based pruning
  - **Intuition**: prefer “cold” code
  - Hot paths are reached anyways
- **Complexity**-based pruning



Target pruning

- **Profiling**-based pruning
  - **Intuition**: prefer “cold” code
  - Hot paths are reached anyways
- **Complexity**-based pruning
  - **Intuition**: more instrumentation -> more complex -> more likely to have bugs



Target pruning

- **Profiling**-based pruning
  - **Intuition**: prefer “cold” code
  - Hot paths are reached anyways
- **Complexity**-based pruning
  - **Intuition**: more instrumentation -> more complex -> more likely to have bugs

Target pruning

- **Profiling**-based pruning
  - **Intuition**: prefer “cold” code
  - Hot paths are reached anyways
- **Complexity**-based pruning
  - **Intuition**: more instrumentation -> more complex -> more likely to have bugs

Prog	Targets (pre-prune)	Targets (post-prune)
base64	1950	212
md5sum	1639	101
uniq	1832	193
who	2120	385

Target pruning

- **Profiling**-based pruning
  - **Intuition**: prefer “cold” code
  - Hot paths are reached anyways
- **Complexity**-based pruning
  - **Intuition**: more instrumentation -> more complex -> more likely to have bugs

Prog	Targets (pre-prune)	Targets (post-prune)
base64	1950	212
md5sum	1639	101
uniq	1832	193
who	2120	385

Prog	Type	Targets		
		ParmeSan	No c.b. pruning	No pruning
boringssl	UAF	51	51	253
c-ares	BO	21	21	36
freetype2	IO	730	950	8538
pcre2	UAF	1856	2051	21781
lcms	BO	95	98	785
libarchive	BO	273	340	1431
libssh	ML	55	45	229
libxml2	BO	670	751	5131
libxml2	ML	670	751	5131
openssl-1.0.1f	BO	43	39	304
openssl-1.0.1f	ML	43	39	304
proj4	ML	18	15	41
re2	BO	295	370	2129
woff2	BO	24	20	33
woff2	OOM	24	20	22
Geomean		**112** (+0%)	**108** (-3.5%)	**716** (+539%)

How to reach these targets?

Solve branches (using DFA) + distance prioritization

ParmeSan fuzzer

ParmeSan fuzzer

- Based on **Angora**

ParmeSan fuzzer

- Based on **Angora**
  - Contains a fuzzing queue of tuples *(conditional, seed)* sorted by *(#runs)*

ParmeSan fuzzer

- Based on **Angora**
  - Contains a fuzzing queue of tuples *(conditional, seed)* sorted by *(#runs)*
- **ParmeSan**

ParmeSan fuzzer

- Based on **Angora**
  - Contains a fuzzing queue of tuples *(conditional, seed)* sorted by *(#runs)*
- **ParmeSan**
  - Sorted by *(#runs, distance)*

ParmeSan fuzzer

- Based on **Angora**
  - Contains a fuzzing queue of tuples *(conditional, seed)* sorted by *(#runs)*
- **ParmeSan**
  - Sorted by *(#runs, distance)*
  - Distance is calculated using a *dynamic* CFG component

Dynamic CFG

Dynamic CFG

- Existing DGF use **static distance** instrumentation (AFLGo)

Dynamic CFG

- Existing DGF use **static distance** instrumentation (AFLGo)

Dynamic CFG

- Existing DGF use **static distance** instrumentation (AFLGo)
  - Targets/ paths **might change over time**

Dynamic CFG

- Existing DGF use **static distance** instrumentation (AFLGo)
  - Targets/ paths **might change over time**

Dynamic CFG

- Existing DGF use **static distance** instrumentation (AFLGo)
  - Targets/ paths **might change over time**
- ParmeSan allows for **augmenting** the CFG/ distance calculation at runtime

Dynamic CFG

- Existing DGF use **static distance** instrumentation (AFLGo)
  - Targets/ paths **might change over time**
- ParmeSan allows for **augmenting** the CFG/ distance calculation at runtime
- Fix targets of indirect calls

Evaluation

1. vs. **directed** fuzzers
   - Sprinkle on some data-flow analysis => better at reaching targets (**~290% faster**)
2. vs. **coverage-guided** fuzzers
   - Improves **latency** to expose bugs by **~40%**

Prog	Type	Runs	Mean. TTE				
			AFLGo (p)		Angora (p)		ParmeSan
Whole pipeline							
boringssl	UAF	30	2h32m	**0.004**	45m	**0.005**	25m
c-ares	BO	30	5s	**0.04**	1s	0.12	1s
freetype2	IO	5	✗	—	47h	**0.018**	43h
pcre2	UAF	30	25m	**0.006**	15m	**0.003**	8m
lcms	BO	30	6m	**0.002**	2m	**0.006**	41s
libarchive	BO	30	1h12m	**0.004**	22m	**0.001**	13m
libssh	ML	30	3m10s	**0.002**	32s	0.008	50s
libxml2	BO	30	51m	**0.007**	20m	**0.001**	11m
libxml2	ML	30	30m	**0.005**	20m	**0.001**	17m
openssl-1.0.1f	BO	30	50m	**0.003**	5m	**0.04**	3m4s
openssl-1.0.1f	ML	30	1m	**0.012**	40s	0.11	37s
proj4	ML	30	7m30s	**0.002**	1m40s	**0.03**	1m26s
re2	BO	30	47m	**0.002**	21m	**0.004**	12m35s
woff2	BO	30	45m	**0.004**	15m	**0.006**	8m
Geomean ParmeSan benefit			**288%**		**37%**		

Side-effect

Chosen sanitizer impacts the type of bugs found

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

Conclusion

- Off-the-shelf **sanitizers** already commonly used when fuzzing
- Try to **actively target** sanitizer instrumentation points
- Sprinkle on data-flow analysis and dynamic distance calculation to improve directed fuzzing
- Combine **automatic target acquisition** + these **improvements**
- => Find bugs **faster**

github.com/vusec/parmesan

s.osterlund@vu.nl

@sirmc

ParmeSan

Sanitizer-guided Greybox Fuzzing

Sebastian Österlund, Kaveh Razavi, Herbert Bos, Cristiano Giuffrida

Vrije Universiteit Amsterdam