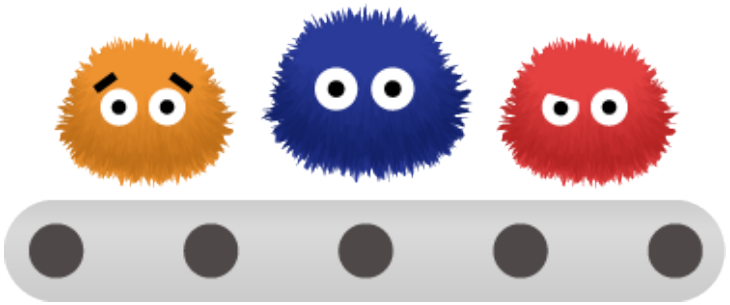


FuzzFactory: Domain-Specific Fuzzing with *Waypoints*

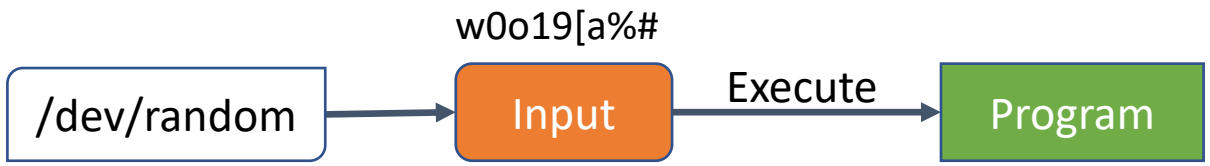
Rohan Padhye[1], Caroline Lemieux[1], Koushik Sen[1], Laurent Simon[2], Hayawardh Vijayakumar[2]

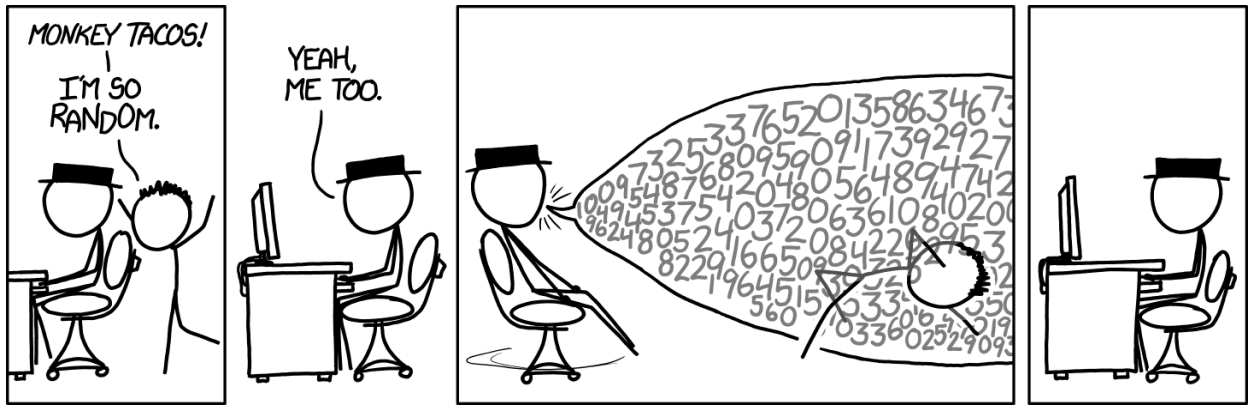
[1] UC Berkeley

[2] Samsung Research America

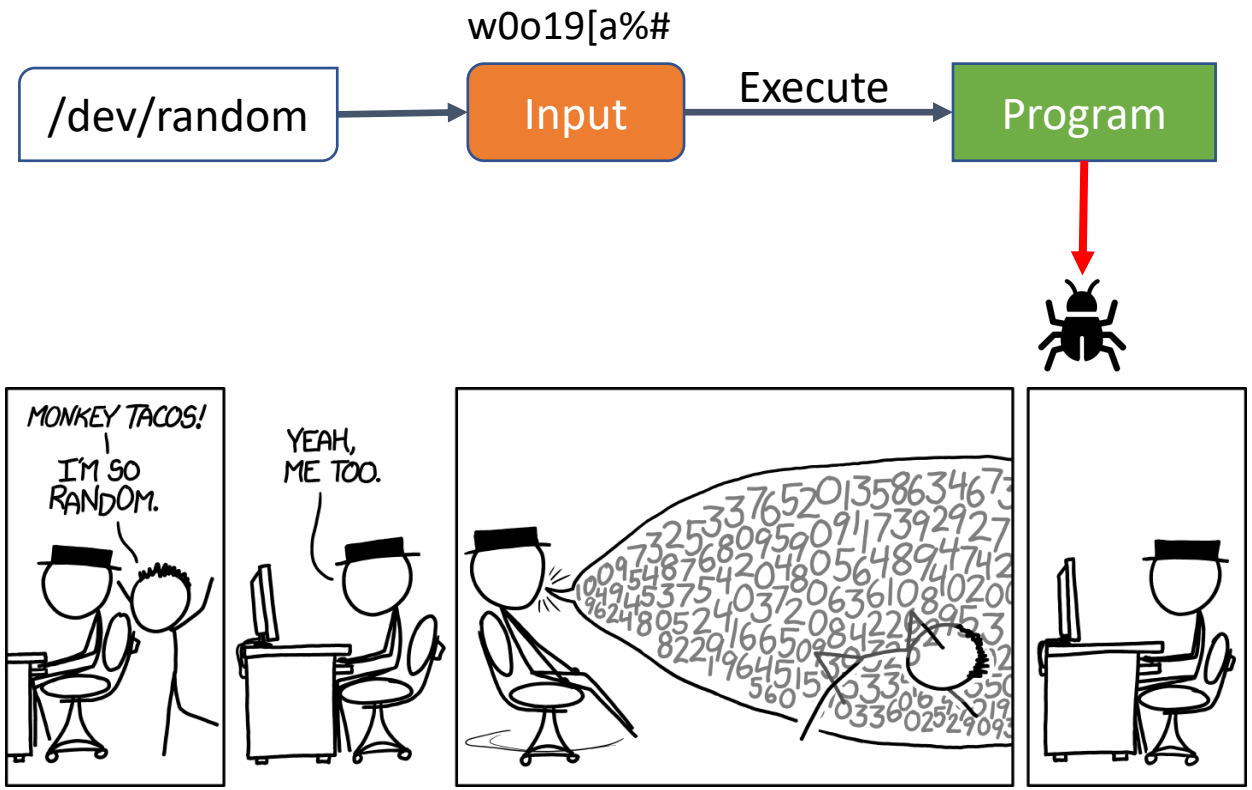
OOPSLA 2019

Fuzz Testing 101



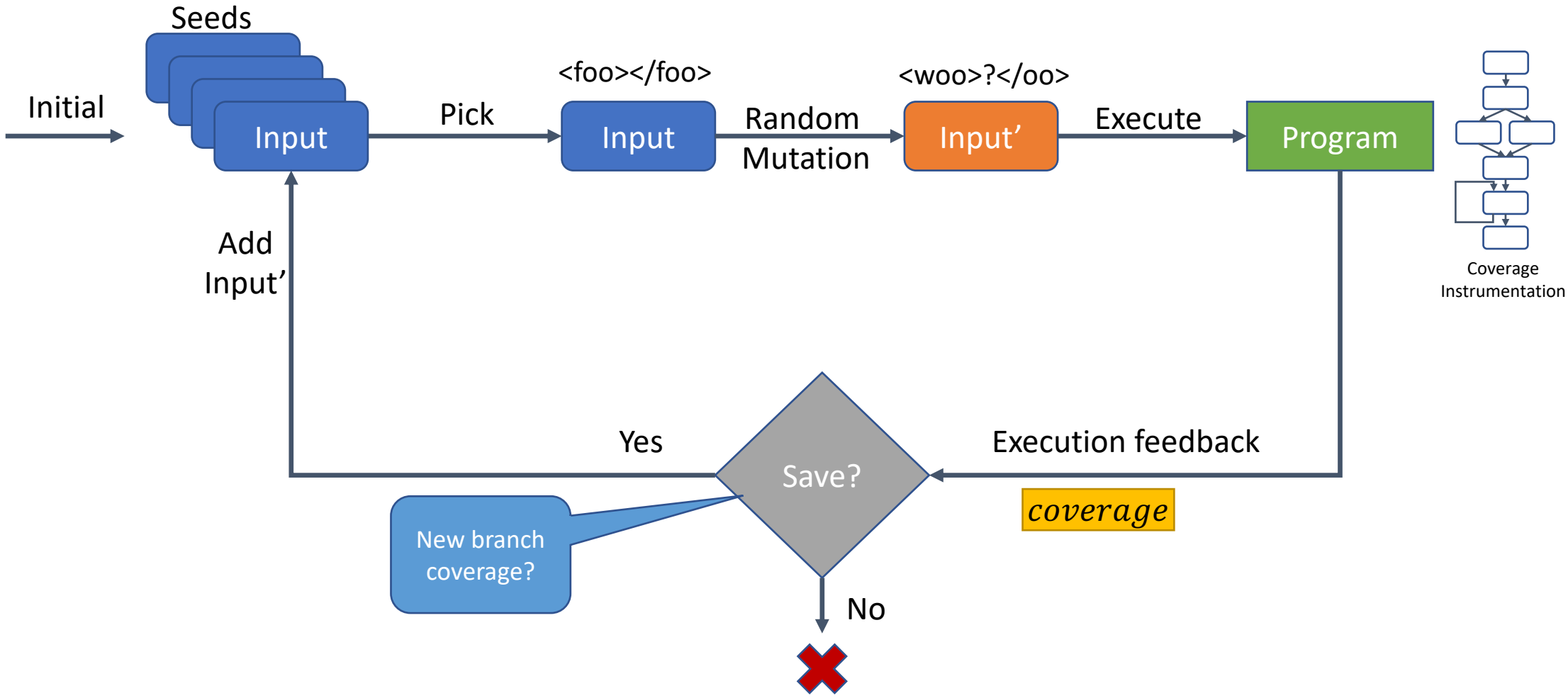


Fuzz Testing 101

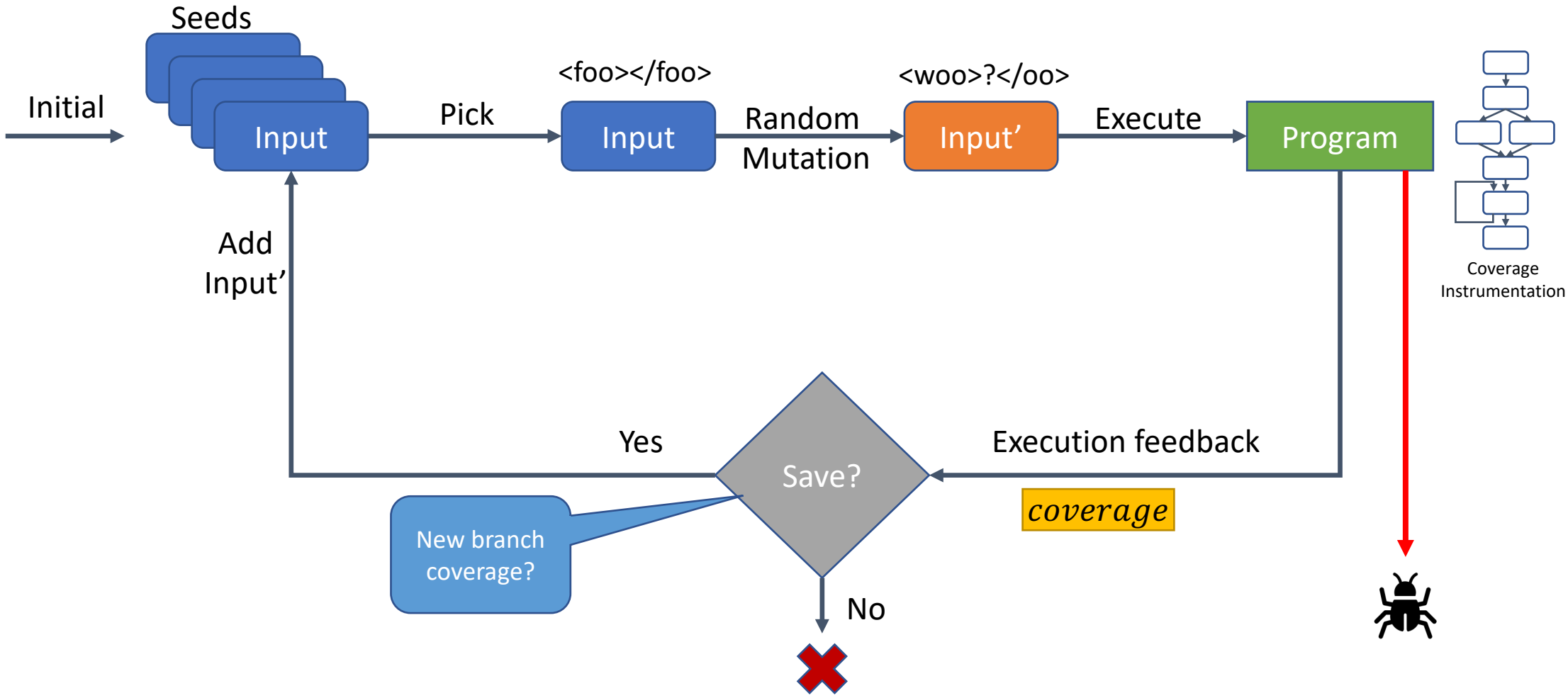


Bugs = Crashes (segfaults, aborts, etc.)

Coverage-Guided Fuzzing (e.g. AFL, libFuzzer)



Coverage-Guided Fuzzing (e.g. AFL, libFuzzer)



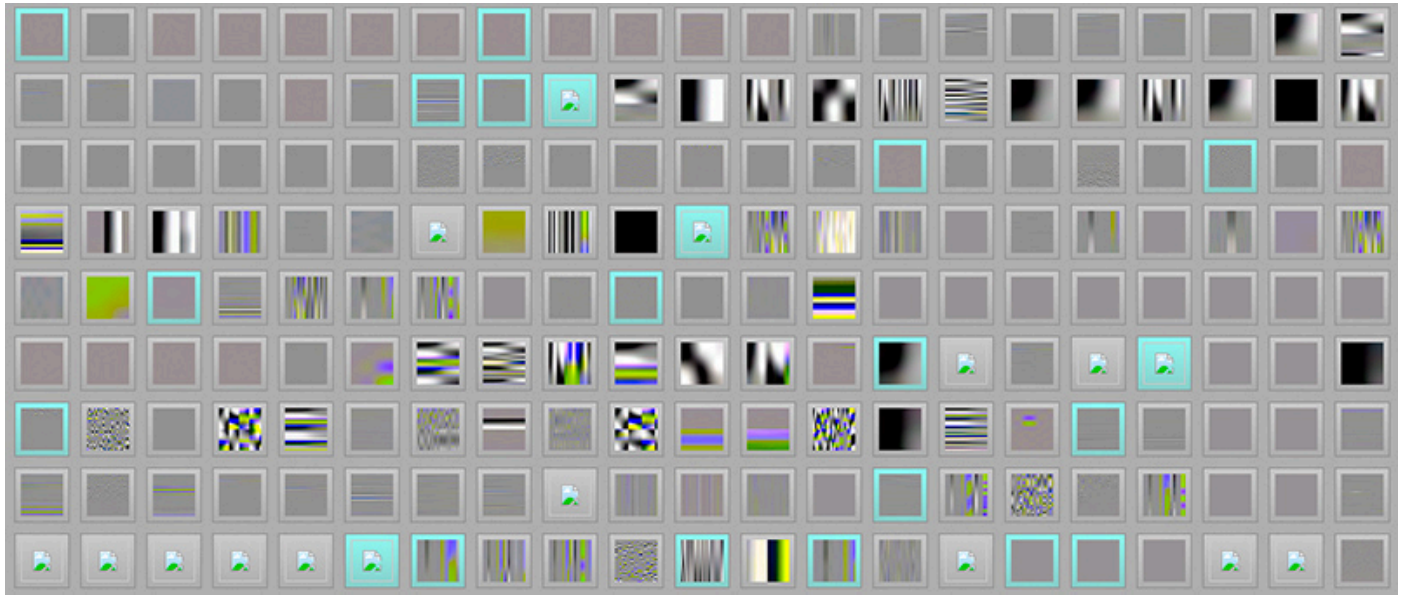
Coverage-Guided Fuzzing with AFL

November 07, 2014

Pulling JPEGs out of thin air

This is an interesting demonstration of the capabilities of afl; I was actually pretty surprised that it worked!

```
$ mkdir in_dir
$ echo 'hello' >in_dir/hello
$ ./afl-fuzz -i in_dir -o out_dir ./jpeg-9a/djpeg
```



Coverage-Guided Fuzzing with AFL

The bug-o-rama trophy case http://lcamtuf.coredump.cx/afl/

IJG jpeg [1]	libjpeg-turbo [1] [2]	libpng [1]
libtiff [1] [2] [3] [4] [5]	mozjpeg [1]	PHP [1] [2] [3] [4] [5] [6] [7] [8]
Mozilla Firefox [1] [2] [3] [4]	Internet Explorer [1] [2] [3] [4]	Apple Safari [1]
Adobe Flash / PCRE [1] [2] [3] [4] [5] [6] [7]	sqlite [1] [2] [3] [4]...	OpenSSL [1] [2] [3] [4] [5] [6] [7]
LibreOffice [1] [2] [3] [4]	poppler [1] [2]...	freetype [1] [2]
GnuTLS [1]	GnuPG [1] [2] [3] [4]	OpenSSH [1] [2] [3] [4] [5]
PuTTY [1] [2]	ntpd [1] [2]	nginx [1] [2] [3]
bash (post-Shellshock) [1] [2]	tcpdump [1] [2] [3] [4] [5] [6] [7] [8] [9]	JavaScriptCore [1] [2] [3] [4]
pdfium [1] [2]	ffmpeg [1] [2] [3] [4] [5]	libmatroska [1]
libarchive [1] [2] [3] [4] [5] [6] ...	wireshark [1] [2] [3]	ImageMagick [1] [2] [3] [4] [5] [6] [7] [8] [9] ...
BIND [1] [2] [3] ...	QEMU [1] [2]	lcms [1]

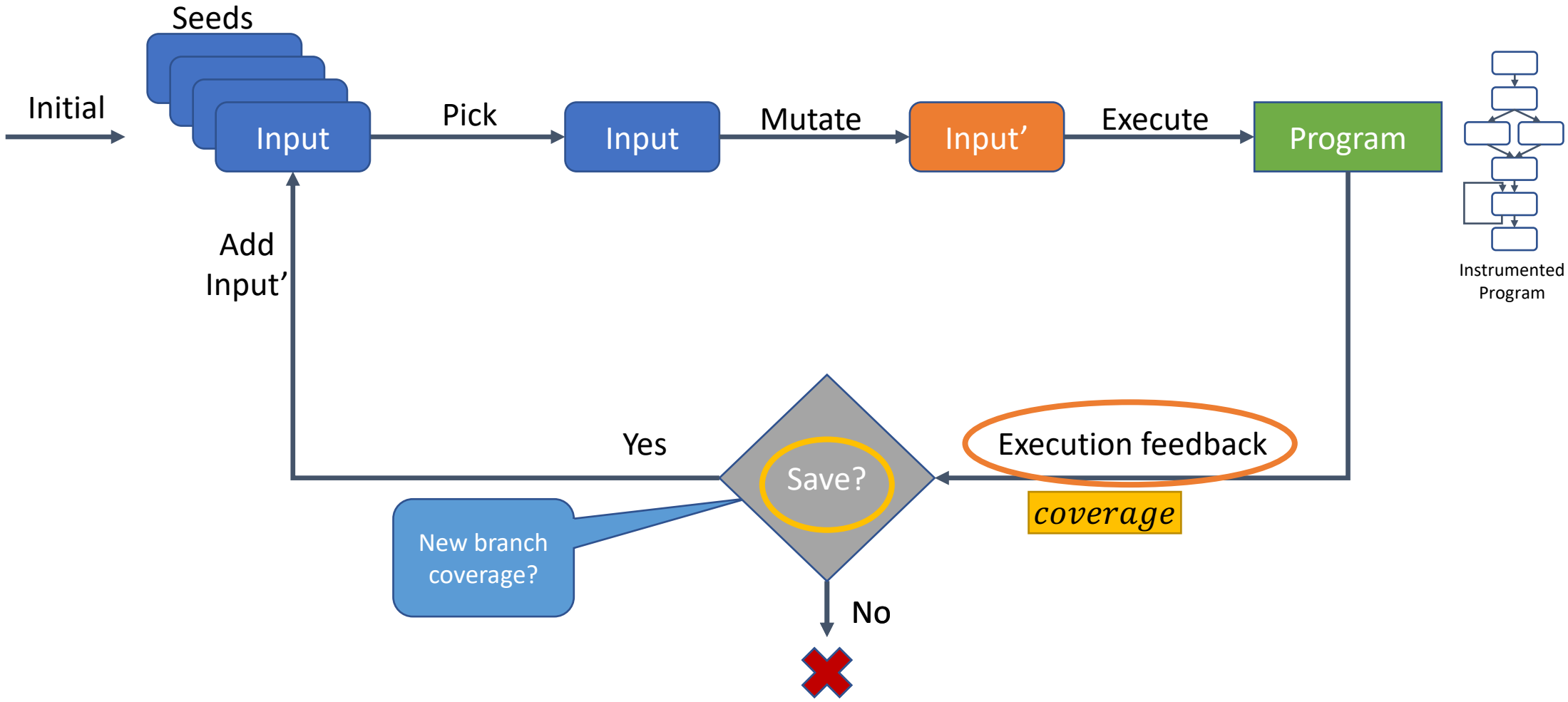
Oh So Many Fuzzers

- AFLFast [CCS 2016]
- Driller [NDSS 2016]
- AFLGo [CCS 2017]
- Vuzzer [NDSS 2017]
- Steelix [FSE 2017]
- SlowFuzz [CCS 2017]
- PerfFuzz [ISSTA 2018]
- FairFuzz [ASE 2018]
- Angora [IEEE S&P 2018]
- T-Fuzz [IEEE S&P 2018]
- NEUZZ [IEEE S&P 2019]
- Nautilus [NDSS 2019]
- Redqueen [NDSS 2019]
- Superion [ICSE 2019]
- MOPT [Usenix Sec 2019]
- GRIMOIRE [Usenix Sec 2019]
- MemFuzz [ICST 2019]
- Zest [ISSTA 2019]
- DifFuzz [ICSE 2019]
- AFLSmart [IEEE TSE 2019]
- FuzzChick [OOPSLA 2019]
- …

Oh So Many Fuzzers

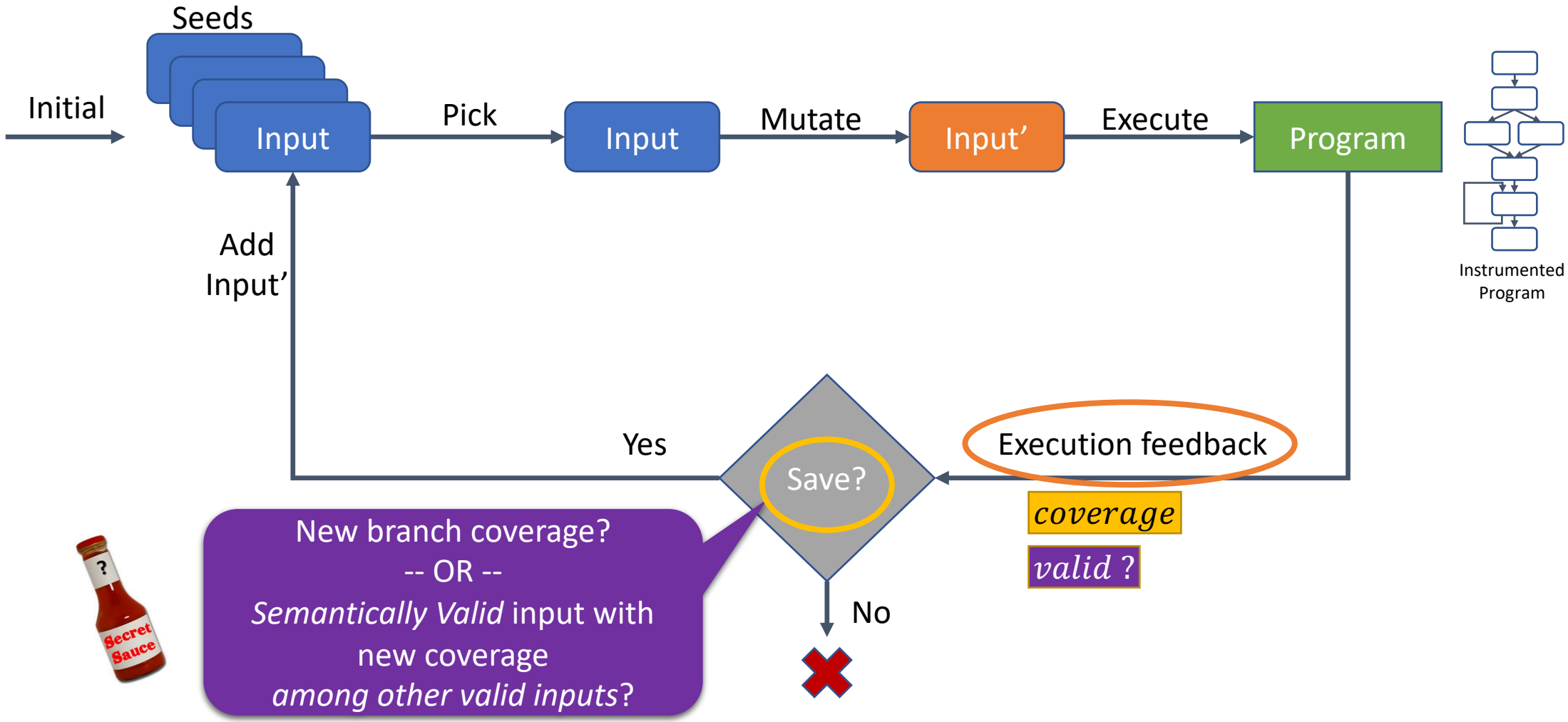
- AFLFast [CCS 2016]
- Driller [NDSS 2016]
- AFLGo [CCS 2017]
- Vuzzer [NDSS 2017]
- Steelix [FSE 2017]
- SlowFuzz [CCS 2017]
- PerfFuzz [ISSTA 2018]
- FairFuzz [ASE 2018]
- Angora [IEEE S&P 2018]
- T-Fuzz [IEEE S&P 2018]
- NEUZZ [IEEE S&P 2019]
- Nautilus [NDSS 2019]
- Redqueen [NDSS 2019]
- Superion [ICSE 2019]
- MOPT [Usenix Sec 2019]
- GRIMOIRE [Usenix Sec 2019]
- MemFuzz [ICST 2019]
- Zest [ISSTA 2019]
- DifFuzz [ICSE 2019]
- AFLSmart [IEEE TSE 2019]
- FuzzChick [OOPSLA 2019]
- …

Coverage-Guided Fuzzing (e.g. AFL, libFuzzer)



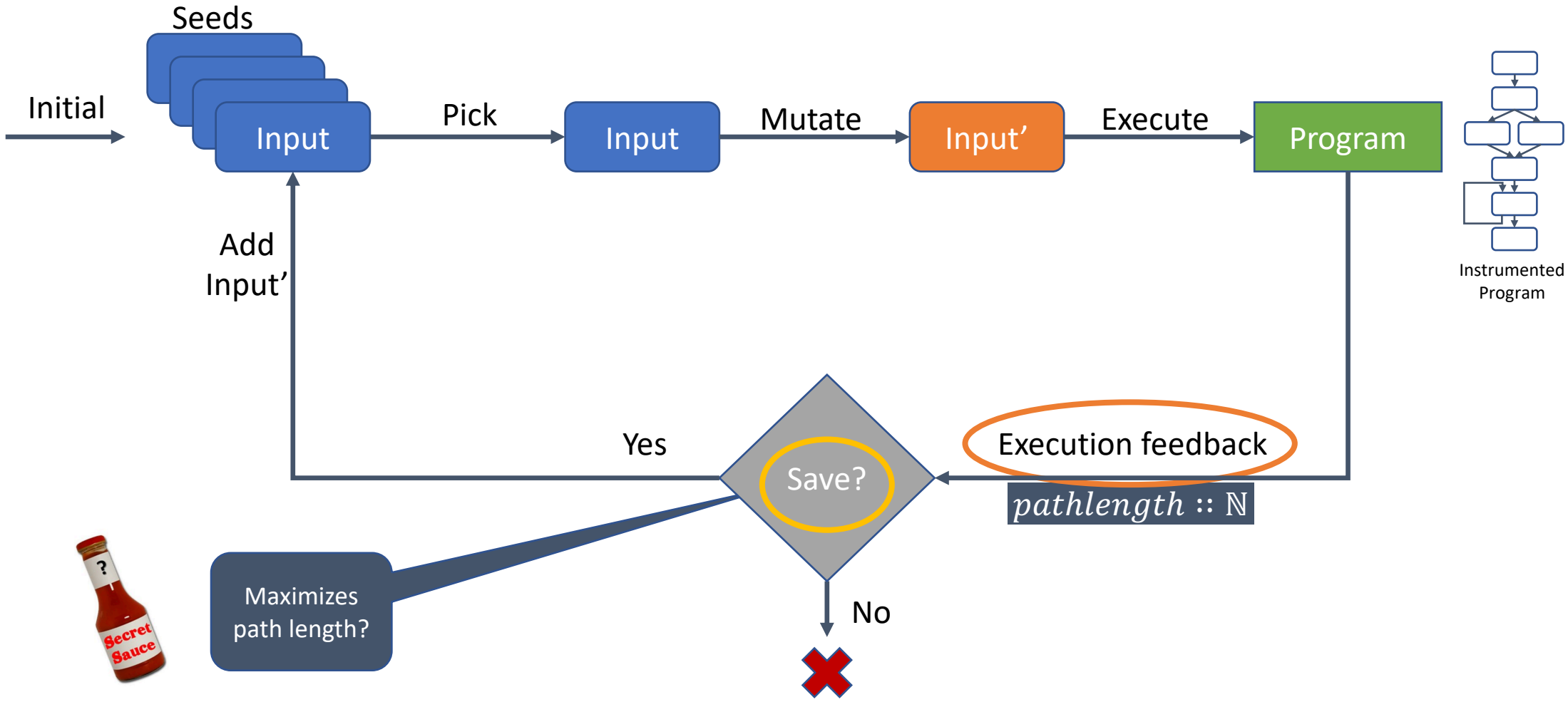
Zest's algorithm for Validity Fuzzing

[Padhye et al. ISSTA'19]



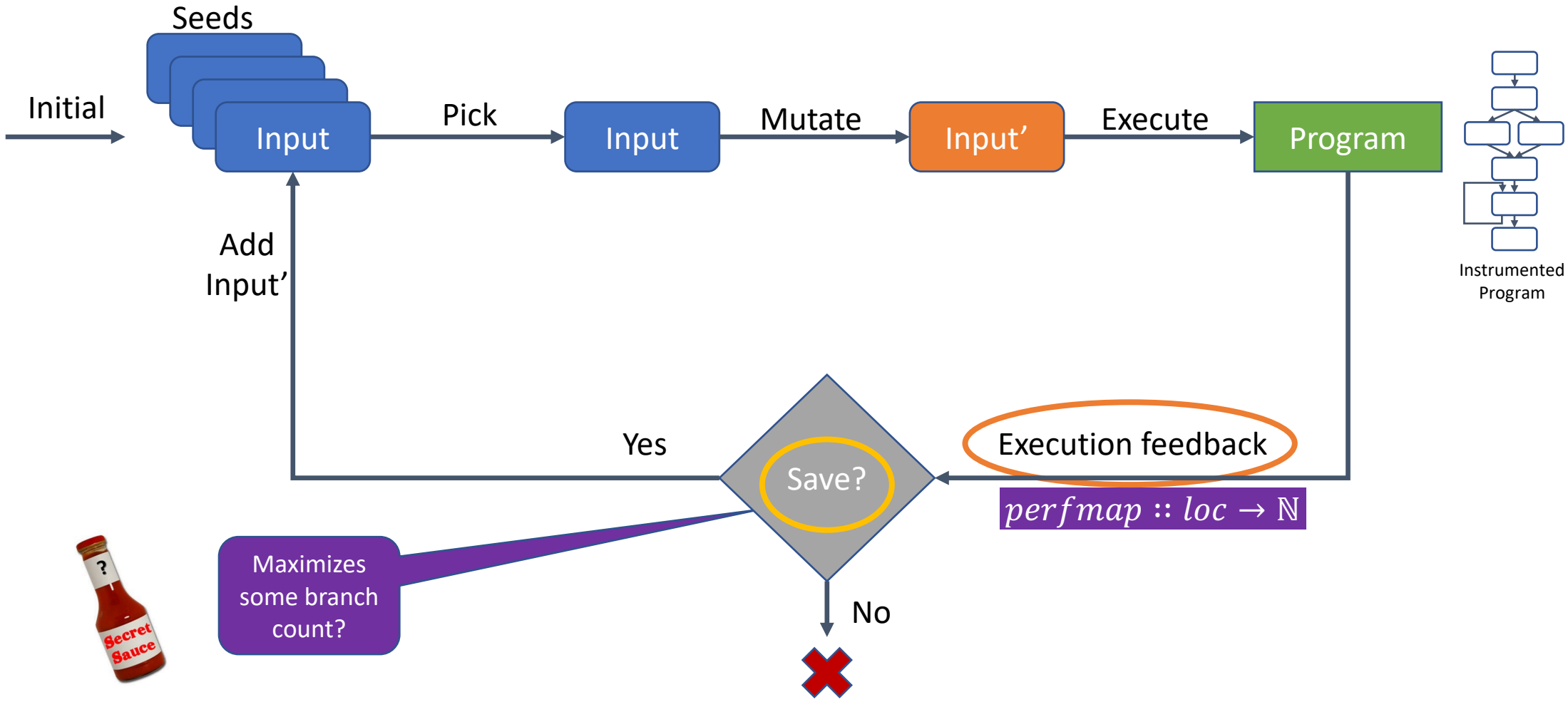
SlowFuzz: Find Inputs with Long Paths

[Petsios et al. CCS'17]



PerfFuzz: Maximize Branch Exec Counts

[Lemieux et al. ISSTA'18]



How to build a new fuzzer systematically?

Step 1:

Fork libFuzzer/AFL/JQF

Step 2:

???

SlowFuzz = libFuzzer ± 386 LoC [CCS'17]

PerfFuzz = AFL ± 312 LoC [ISSTA'18]

Zest = JQF ± 621 LoC [ISSTA'19]

How to combine fuzzers systematically?

e.g. Semantic validity (Zest) + worst-case performance (PerfFuzz)

Step 1:

???

Can we rapidly **create** and **combine** domain-specific fuzzers?

(without touching the search algorithm)

Factorize!

Zest [Padhye et al. 2018]

save if “increases coverage amongst valid inputs”

SlowFuzz [Petsios et al. 2017]

save if “increases path length”

PerfFuzz [Lemieux et al. 2018]

save if “maximizes branch exec counts”

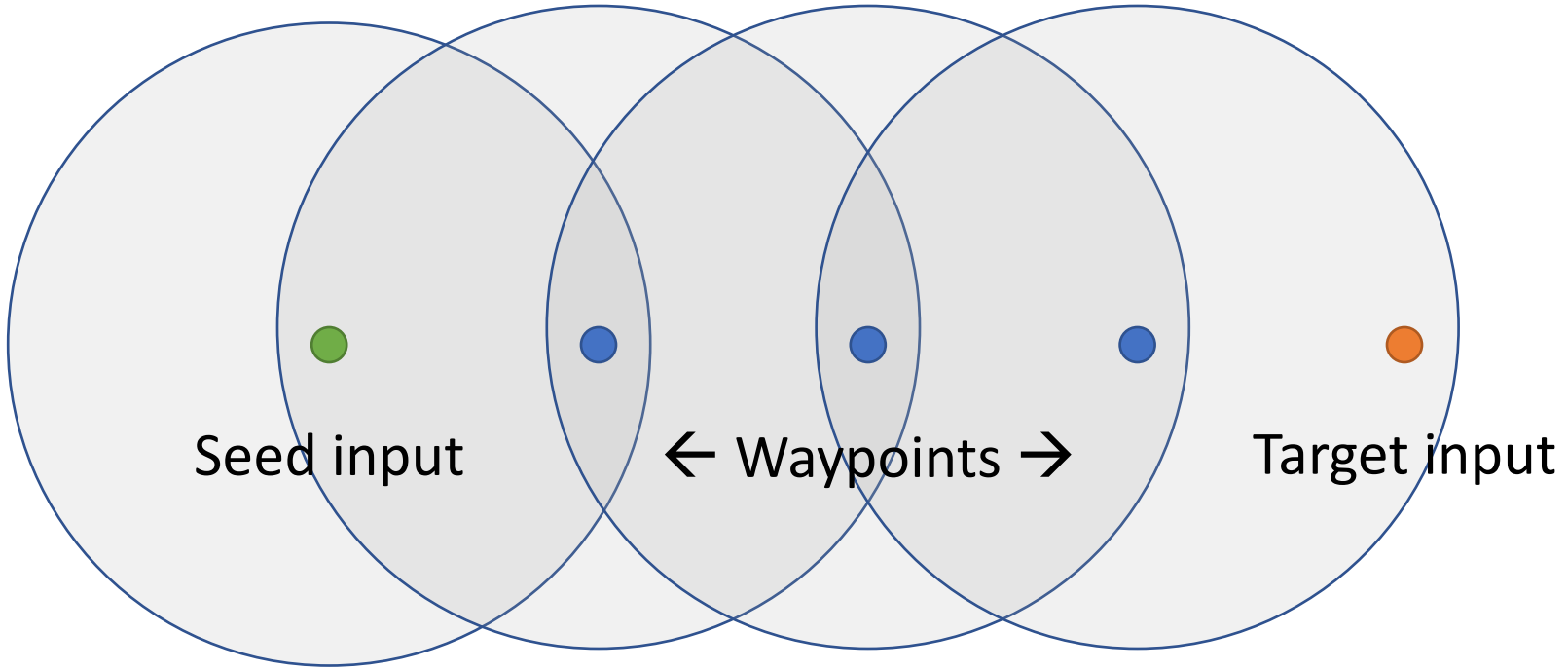
DifFuzz [Nilizadeh et al. 2019]

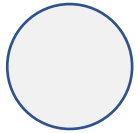
save if “leaks more info on the side channel”

MemFuzz [Coppik et al. 2019]

save if “accesses new input-dependent memory locations”

Common Strategy:
Select intermediate inputs
“Waypoints”





single-mutation
search space

$$is\_waypoint(i, S) :: I \times 2^I \rightarrow Bool$$

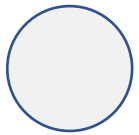
Given a new mutated input $i \in I$,
should i be saved to set of seeds S?

Our goal: Allow users to define $is\_waypoint$

$$is\_waypoint(i, S) = false$$

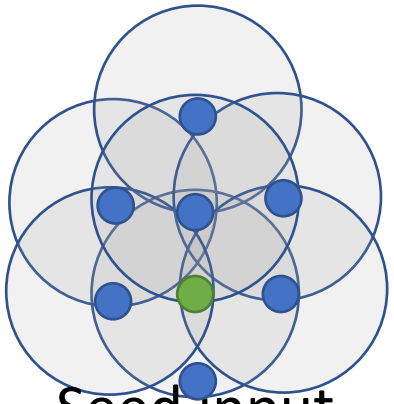
Seed input

Target input



single-mutation search space

$is\_waypoint(i, S) = true$

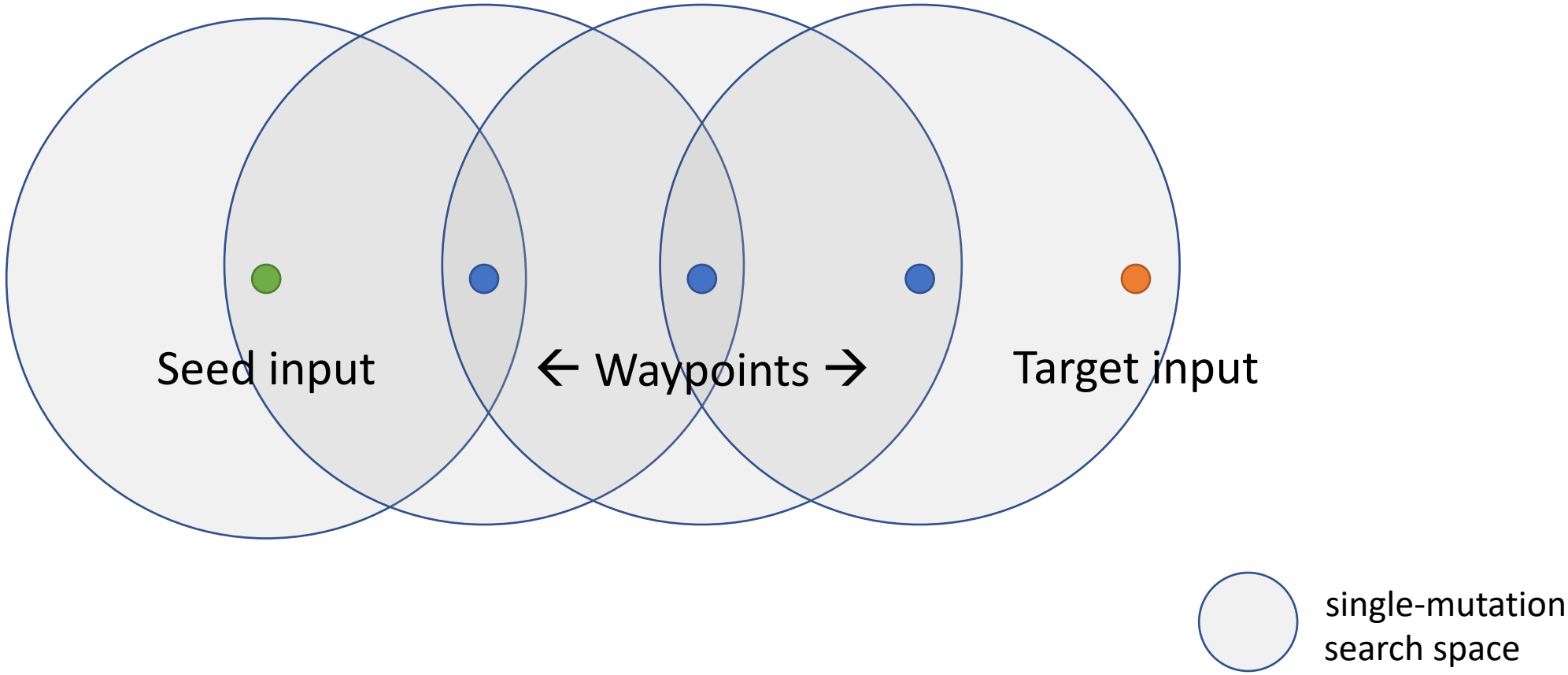


Seed input

Target input

single-mutation
search space

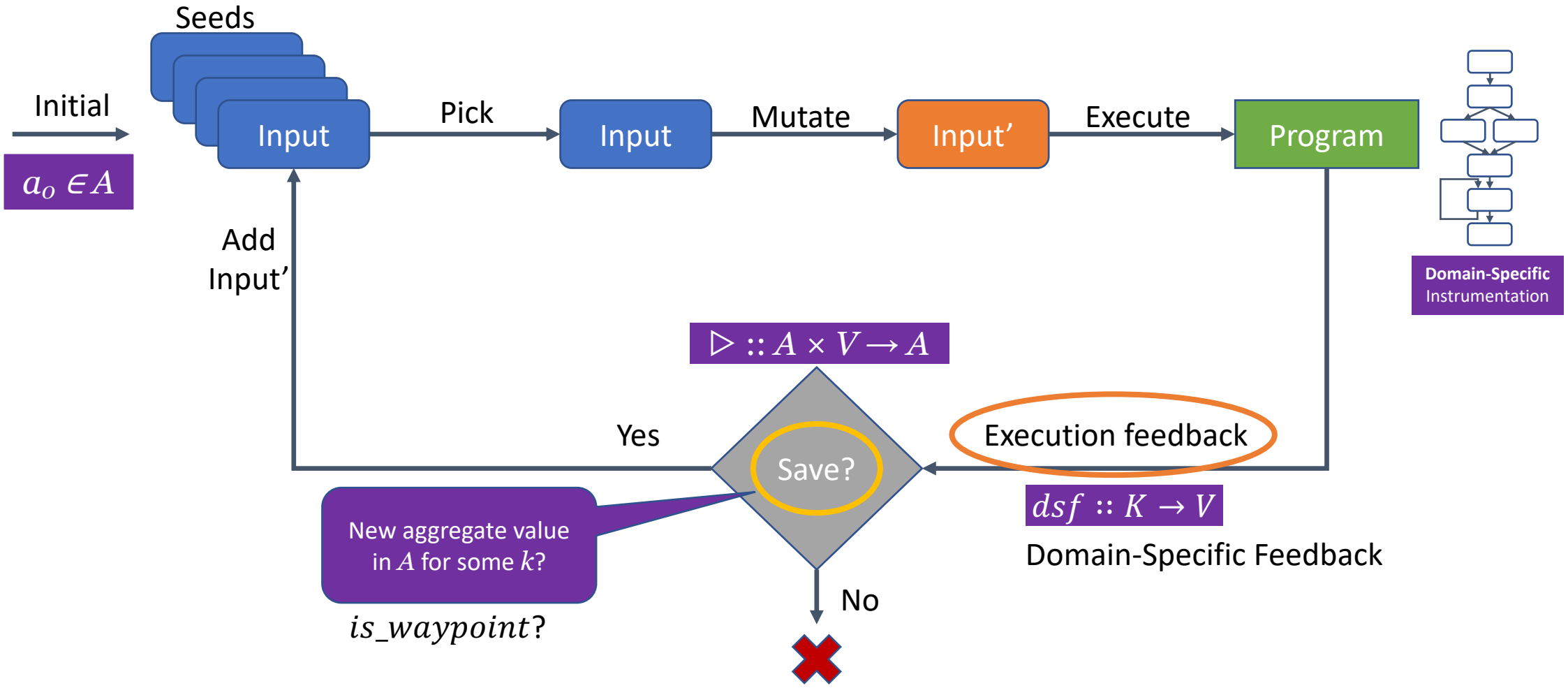
$is\_waypoint(i, S)$ = "Closer to my goal"



Q1: How to define $is\_waypoint(i, S)$ easily?

Q2: How to define $is\_waypoint(i, S)$ to ensure progress?

FuzzFactory: Domain-Specific Fuzzing



Q1: How to define $is\_waypoint(i, S)$ easily?

$dsf_i :: K \rightarrow V$

e.g. K = Program Locations, V = Exec Counts

e.g. PerfFuzz

k		**dsf_{i1}**		dsf_{i2}	
Loc_1		4		5	
Loc_2		2		1	

Q1: How to define $is\_waypoint(i, S)$ easily?

$dsf_i :: K \rightarrow V$

e.g. K = Program Locations, V = Exec Counts

$\triangleright :: A \times V \rightarrow A$ with $a_o \in A$

e.g. $\triangleright = \max$

$A = \mathbb{N}, a_o = 0$

e.g. PerfFuzz

k		dsf_{i1}		dsf_{i2}	
Loc_1		4		5	
Loc_2		2		1	

Q1: How to define $is\_waypoint(i, S)$ easily?

$dsf_i :: K \rightarrow V$

e.g. K = Program Locations, V = Exec Counts

$\triangleright :: A \times V \rightarrow A$ with $a_o \in A$

e.g. $\triangleright$ = max

$A = \mathbb{N}, a_o = 0$

e.g. PerfFuzz

k	▷	**dsf_{i1}**	▷	dsf_{i2}	▷
Loc_1	**0**	4		5	
Loc_2	**0**	2		1	

Q1: How to define $is\_waypoint(i, S)$ easily?

$dsf_i :: K \rightarrow V$

e.g. K = Program Locations, V = Exec Counts

$\triangleright :: A \times V \rightarrow A$ with $a_o \in A$

e.g. $\triangleright$ = max

$A = \mathbb{N}, a_o = 0$

e.g. PerfFuzz

k	▷	**dsf_{i1}**	▷	dsf_{i2}	▷
Loc_1	**0**	4	**4**	5	
Loc_2	**0**	2	**2**	1	

Q1: How to define $is\_waypoint(i, S)$ easily?

$dsf_i :: K \rightarrow V$

e.g. K = Program Locations, V = Exec Counts

$\triangleright :: A \times V \rightarrow A$ with $a_o \in A$

e.g. $\triangleright$ = max

$A = \mathbb{N}, a_o = 0$

e.g. PerfFuzz

k	▷	dsf_{i1}	▷	dsf_{i2}	▷
Loc_1	**0**	4	**4**	5	**5**
Loc_2	**0**	2	**2**	1	**2**

Q1: How to define $is\_waypoint(i, S)$ easily?

$dsf_i :: K \rightarrow V$

e.g. K = Program Locations, V = Exec Counts

$\triangleright :: A \times V \rightarrow A$ with $a_o \in A$

e.g. $\triangleright$ = max

$A = \mathbb{N}, a_o = 0$

$$is\_waypoint(i, S) \stackrel{\text{def}}{=} \exists k : \bigtriangleright_{s \in S} dsf_s(k) \quad != \quad \bigtriangleright_{s \in (S \cup \{i\})} dsf_s(k)$$

e.g. PerfFuzz

k	▷	dsf_{i1}	▷	dsf_{i2}	▷
Loc_1	**0**	4	**4**	5	**5**
Loc_2	**0**	2	**2**	1	**2**

Q1: How to define $is\_waypoint(i, S)$ easily?

$dsf_i :: K \rightarrow V$

e.g. K = Program Locations, V = Exec Counts

$\triangleright :: A \times V \rightarrow A$ with $a_o \in A$

e.g. $\triangleright$ = max

$A = \mathbb{N}, a_o = 0$

$$is\_waypoint(i, S) \stackrel{\text{def}}{=} \exists k : \bigtriangleright_{s \in S} dsf_s(k) \quad != \quad \bigtriangleright_{s \in (S \cup \{i\})} dsf_s(k)$$

New aggregate value in A for some k?

e.g. PerfFuzz

k	▷	dsf_{i1}	▷	dsf_{i2}	▷
Loc_1	**0**	4	**4**	5	**5**
Loc_2	**0**	2	**2**	1	**2**

Q1: How to define $is\_waypoint(i, S)$ easily?

$dsf_i :: K \rightarrow V$

e.g. K = Program Locations, V = Exec Counts

$\triangleright :: A \times V \rightarrow A$ with $a_o \in A$

$a \triangleright v = a \cup \{ \text{floor}(\log_2 v) \}$

$A = 2^{\mathbb{N}}, a_o = \emptyset$

e.g. AFL

k	▷	dsf_{i1}	▷	dsf_{i2}	▷
Loc_1	**Ø**	4	**{4}**	5	**{4}**
Loc_2	**Ø**	2	**{2}**	1	**{1,2}**

$$is\_waypoint(i, S) \stackrel{\text{def}}{=} \exists k : \left(\bigtriangleup_{s \in S} dsf_s(k) \right) \; != \; \left(\bigtriangleup_{s \in (S \cup \{i\})} dsf_s(k) \right)$$

New aggregate value in A for some k?

Q1: How to define $is\_waypoint(i, S)$ easily?

$dsf_i :: K \rightarrow V$

e.g. K = Program Locations, V = Exec Counts

$\triangleright :: A \times V \rightarrow A$ with $a_o \in A$

$a \triangleright v = a \cup \{ \text{floor}(\log_2 v) \}$

$A = 2^{\mathbb{N}}, a_o = \emptyset$

e.g. AFL

Q2: How to ensure progress?

$a \triangleright v \triangleright v = a \triangleright v$ (Idempotence)

$a \triangleright u \triangleright v = a \triangleright v \triangleright u$ (Order Insensitivity)

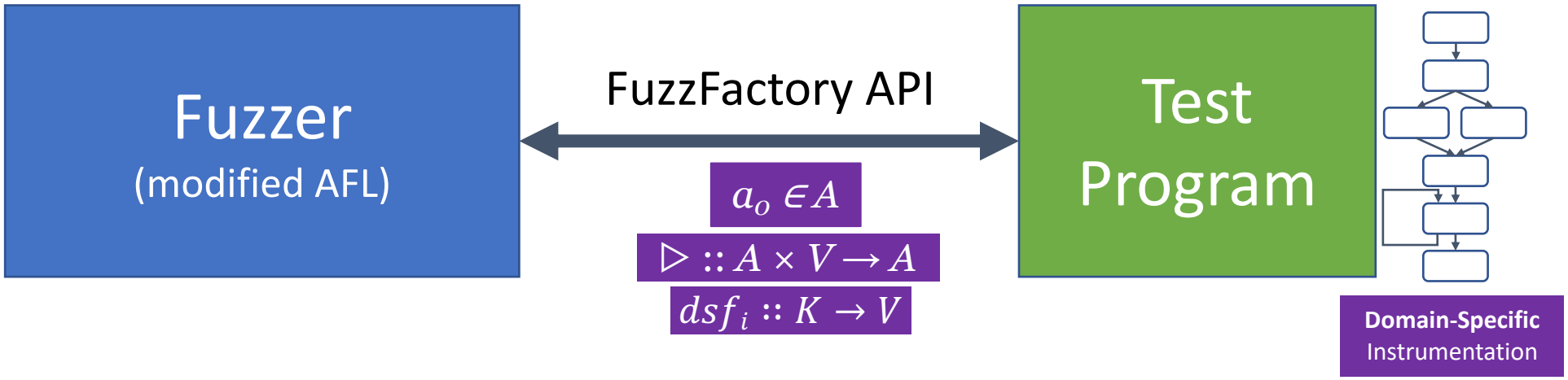
Theorem: $\triangleright$ is monotonic

$\therefore is\_waypoint \Leftrightarrow$ progress

$$is\_waypoint(i, S) \stackrel{\text{def}}{=} \exists k : \bigtriangleright_{s \in S} dsf_s(k) \quad != \quad \bigtriangleright_{s \in (S \cup \{i\})} dsf_s(k)$$

New aggregate value in A for some k?

Architecture of FuzzFactory



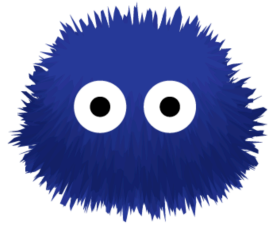
Option 1: Manually annotate test program to call APIs

Option 2: Auto-instrument test program to call APIs (e.g. LLVM, PIN)

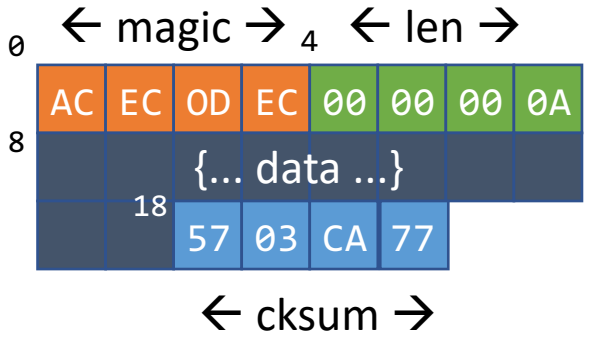
CMP: An example Domain-Specific Fuzzer

Built with FuzzFactory!

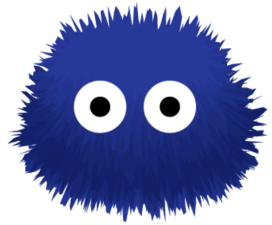
CMP: Blast through *hard* comparisons



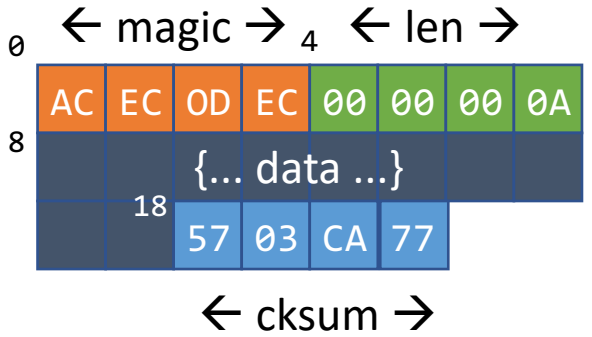
```
int32 magic = (int32) input[0..4];

if (magic != 0xACECODEC)
  return INVALID;

uint32 len = (uint32) input[4..8];
byte* data = (byte*) malloc(len);
if (sizeof(input) != len+12)
  return INVALID;

copy(data, input[8..8+len]);

int32 cksum = (int32) input[8+len..12+len];
int32 hashed = hash(data, len);

if (hashed != cksum)
  return INVALID;

// fun stuff here….
```



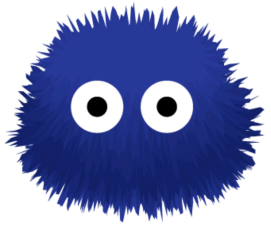
CMP: Blast through *hard* comparisons



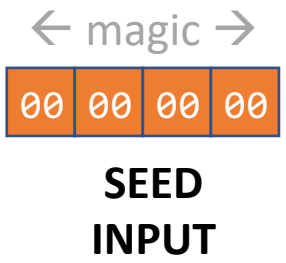
```
int32 magic = (int32) input[0..4];

if (magic != 0xACECODEC)
  return INVALID;

uint32 len = (uint32) input[4..8];
byte* data = (byte*) malloc(len);
if (sizeof(input) != len+12)
  return INVALID;

copy(data, input[8..8+len]);

int32 cksum = (int32) input[8+len..12+len];
int32 hashed = hash(data, len);

if (hashed != cksum)
  return INVALID;

// fun stuff here….
```



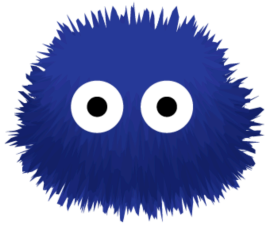
CMP: Blast through *hard* comparisons



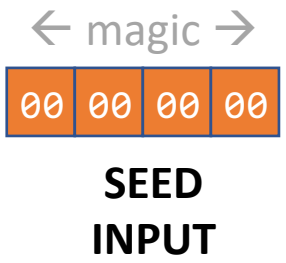
```
int32 magic = (int32) input[0..4];

if (magic != 0xACECODEC)
  return INVALID;

uint32 len = (uint32) input[4..8];
byte* data = (byte*) malloc(len);
if (sizeof(input) != len+12)
  return INVALID;

copy(data, input[8..8+len]);

int32 cksum = (int32) input[8+len..12+len];
int32 hashed = hash(data, len);

if (hashed != cksum)
  return INVALID;

// fun stuff here….
```



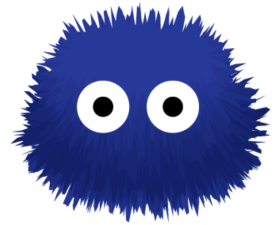
CMP: Blast through *hard* comparisons



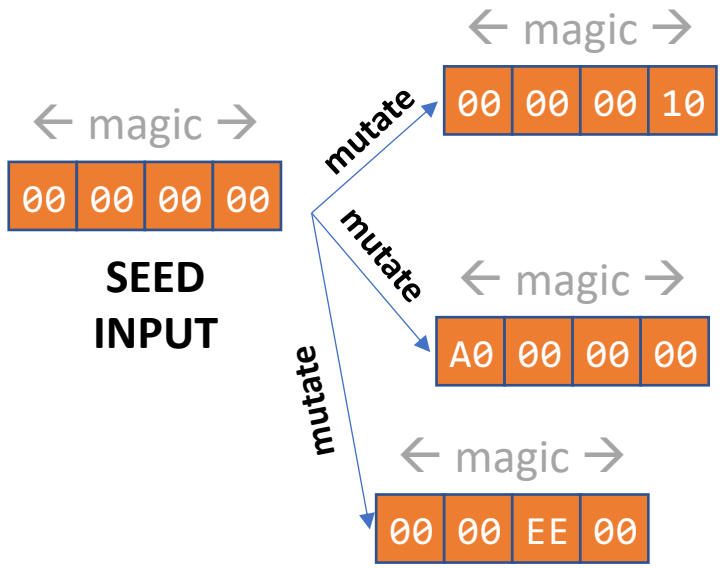
```
int32 magic = (int32) input[0..4];

if (magic != 0xACECODEC)
  return INVALID;

uint32 len = (uint32) input[4..8];
byte* data = (byte*) malloc(len);
if (sizeof(input) != len+12)
  return INVALID;

copy(data, input[8..8+len]);

int32 cksum = (int32) input[8+len..12+len];
int32 hashed = hash(data, len);

if (hashed != cksum)
  return INVALID;

// fun stuff here….
```



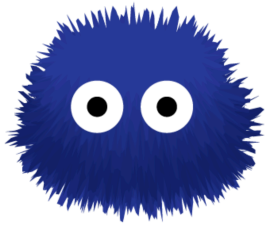
CMP: Blast through *hard* comparisons



```
int32 magic = (int32) input[0..4];

if (magic != 0xACECODEC)
  return INVALID;

uint32 len = (uint32) input[4..8];
byte* data = (byte*) malloc(len);
if (sizeof(input) != len+12)
  return INVALID;

copy(data, input[8..8+len]);

int32 cksum = (int32) input[8+len..12+len];
int32 hashed = hash(data, len);

if (hashed != cksum)
  return INVALID;

// fun stuff here….
```



CMP: Blast through *hard* comparisons



$$a \triangleright v := \min(a, v)$$

$$a_0 := 32$$

```
dsf_t cmp_dsf = fuzzfactory_new_domain(MIN, 32);
int32 magic = (int32) input[0..4];
cmp_dsf[0] = popcnt(magic ^ 0xACECODEC);
if (magic != 0xACECODEC)
  return INVALID;
```

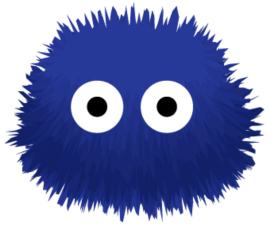
```
uint32 len = (uint32) input[4..8];
byte* data = (byte*) malloc(len);
cmp_dsf[1] = popcnt(sizeof(input) ^ len+12);
if (sizeof(input) != len+12)
  return INVALID;

copy(data, input[8..8+len]);

int32 cksum = (int32) input[8+len..12+len];
int32 hashed = hash(data, len);
cmp_dsf[2] = popcnt(hashed ^ cksum);
if (hashed != cksum)
  return INVALID;

// fun stuff here….
```

← magic →

00	00	00	00

SEED INPUT

CMP: Blast through *hard* comparisons

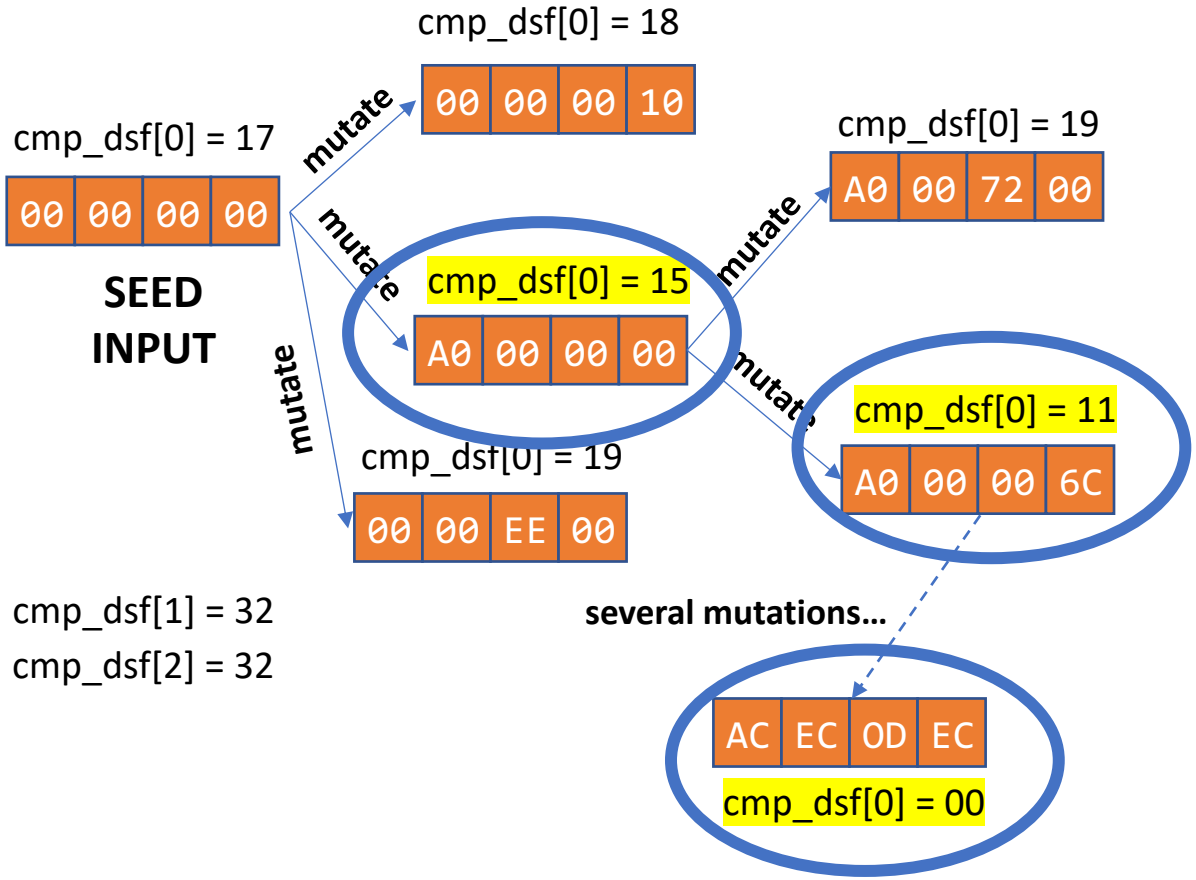


$$a \triangleright v := \min(a, v)$$

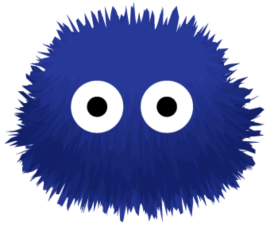
$$a_0 := 32$$

```
dsf_t cmp_dsf = fuzzfactory_new_domain(MIN, 32);
int32 magic = (int32) input[0..4];
cmp_dsf[0] = popcnt(magic ^ 0xACECODEC);
if (magic != 0xACECODEC)
  return INVALID;
```

```
uint32 len = (uint32) input[4..8];
byte* data = (byte*) malloc(len);
cmp_dsf[1] = popcnt(sizeof(input) ^ len+12);
if (sizeof(input) != len+12)
  return INVALID;

copy(data, input[8..8+len]);

int32 cksum = (int32) input[8+len..12+len];
int32 hashed = hash(data, len);
cmp_dsf[2] = popcnt(hashed ^ cksum);
if (hashed != cksum)
  return INVALID;

// fun stuff here….
```



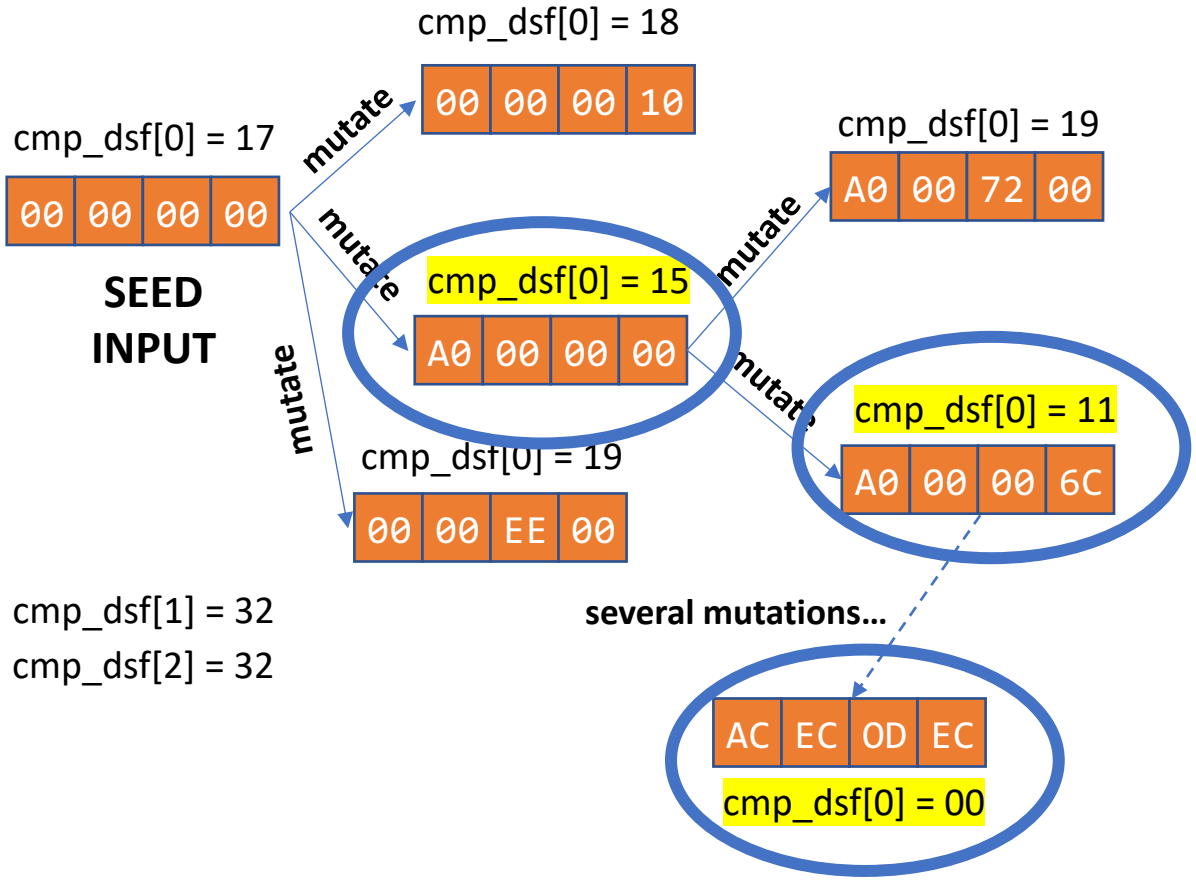
CMP: Blast through *hard* comparisons



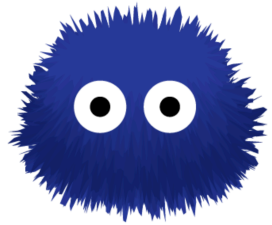
$$a \triangleright v := \min(a, v)$$

$$a_0 := 32$$

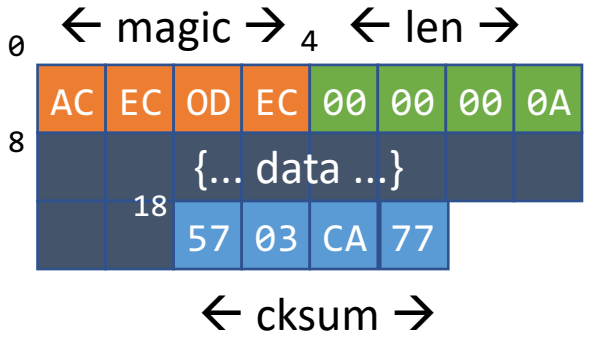
```
dsf_t cmp_dsf = fuzzfactory_new_domain(MIN, 32);
int32 magic = (int32) input[0..4];
cmp_dsf[0] = popcnt(magic ^ 0xACECODEC);
if (magic != 0xACECODEC)
  return INVALID;

uint32 len = (uint32) input[4..8];
byte* data = (byte*) malloc(len);
cmp_dsf[1] = popcnt(sizeof(input) ^ len+12);
if (sizeof(input) != len+12)
  return INVALID;

copy(data, input[8..8+len]);

int32 cksum = (int32) input[8+len..12+len];
int32 hashed = hash(data, len);
cmp_dsf[2] = popcnt(hashed ^ cksum);
if (hashed != cksum)
  return INVALID;

// fun stuff here….
```



CMP: Blast through *hard* comparisons



```
dsf_t cmp_dsf = fuzzfactory_new_domain(MIN, 32);
int32 magic = (int32) input[0..4];
cmp_dsf[0] = popcnt(magic ^ 0xACECODEC);
if (magic != 0xACECODEC)
  return INVALID;

uint32 len = (uint32) input[4..8];
byte* data = (byte*) malloc(len);
cmp_dsf[1] = popcnt(sizeof(input) ^ len+12);
if (sizeof(input) != len+12)
  return INVALID;

copy(data, input[8..8+len]);

int32 cksum = (int32) input[8+len..12+len];
int32 hashed = hash(data, len);
cmp_dsf[2] = popcnt(hashed ^ cksum);
if (hashed != cksum)
  return INVALID;

// fun stuff here….
```

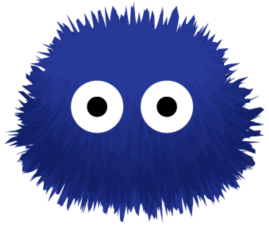


cmp_dsf[0] = 0

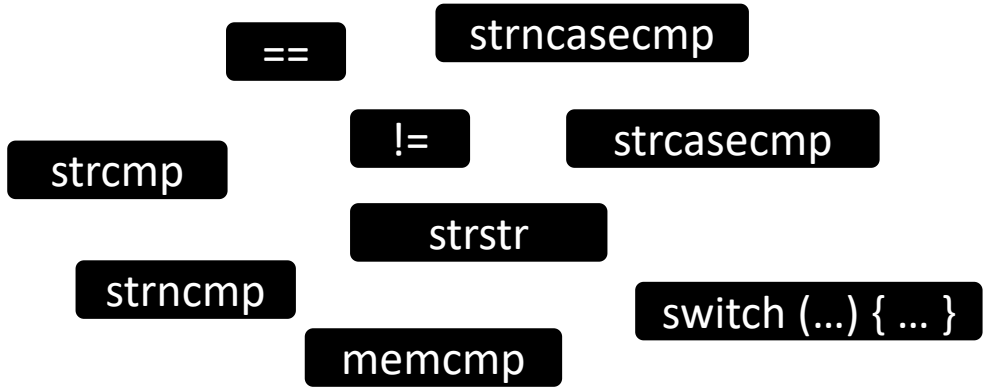
cmp_dsf[1] = 0

cmp_dsf[2] = 0

CMP: Blast through *hard* comparisons



LLVM Instrumentation Pass



Six LLVM-based Domains in FuzzFactory

Fuzzer	Keys	Values	Aggregation	LoC (C++)
Port of **SlowFuzz** [Petsios et al. ‘17]				
Port of **PerfFuzz** [Lemieux et al. ‘18]				
Validity Fuzzing [Padhye et al. ’19]				
Mem **Alloc** Fuzzing				
Cmp Fuzzing				
Incremental Fuzzing				

Six LLVM-based Domains in FuzzFactory

Fuzzer	Keys	Values	Aggregation	LoC (C++)
Port of **SlowFuzz** [Petsios et al. '17]	Singleton	Path length	max	
Port of **PerfFuzz** [Lemieux et al. '18]	Basic Blocks	Exec Counts	max	
Validity Fuzzing [Padhye et al. '19]	Basic Blocks	Exec Counts if Valid else 0	log-union (AFL-style bucketing)	
Mem **Alloc** Fuzzing	Locations invoking malloc()/calloc()	# of bytes allocated	max	
Cmp Fuzzing	==, strcmp, memcmp, switch, etc.	# of bits common between operands	max	
Incremental Fuzzing	Basic Block Transitions	Exec Counts	log-union (AFL-style bucketing)	

Six LLVM-based Domains in FuzzFactory

Fuzzer	Keys	Values	Aggregation	LoC (C++)
Port of **SlowFuzz** [Petsios et al. '17]	Singleton	Path length	max	**18**
Port of **PerfFuzz** [Lemieux et al. '18]	Basic Blocks	Exec Counts	max	**19**
Validity Fuzzing [Padhye et al. '19]	Basic Blocks	Exec Counts if Valid else 0	log-union (AFL-style bucketing)	**24**
Mem **Alloc** Fuzzing	Locations invoking malloc()/calloc()	# of bytes allocated	max	**29**
Cmp Fuzzing	==, strcmp, memcmp, switch, etc.	# of bits common between operands	max	**355**
Incremental Fuzzing	Basic Block Transitions	Exec Counts	log-union (AFL-style bucketing)	**146**

PERF: PerfFuzz in FuzzFactory (LLVM)

Instrumentation Pass

```
1  #include "fuzzfactory.hpp"
2  using namespace fuzzfactory;
3
4  class PerfFuzzFeedback : public DomainFeedback<PerfFuzzFeedback> {
5  public:
6      PerfFuzzFeedback(Module& M) : DomainFeedback<PerfFuzzFeedback>(M, "__afl_perf_dsf") { }
7
8      void visitBasicBlock (BasicBlock &bb) {
9          auto key = createProgramLocation(); // static random value
10
11         // Insert call to `dsf_increment(dsf_map, key, 1)`;
12         auto irb = insert_before(bb); // Get a handle to the LLVM IR builder
13         irb.CreateCall(DsfIncrementFunction, {DsfMapVariable, key, getConst(1)});
14     }
15 };
16
17 FUZZFACTORY_REGISTER_DOMAIN(PerfFuzzFeedback);
```

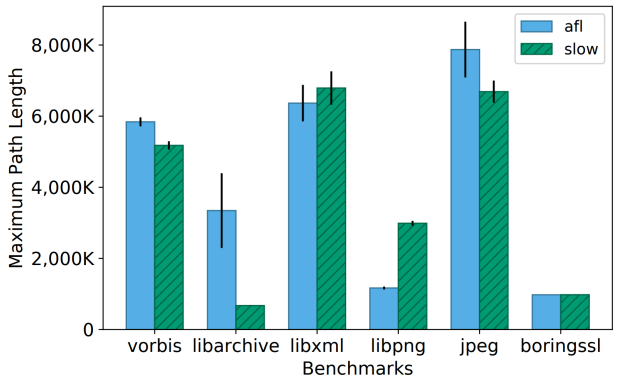
PERF: PerfFuzz in FuzzFactory (LLVM)

Instrumentation Pass

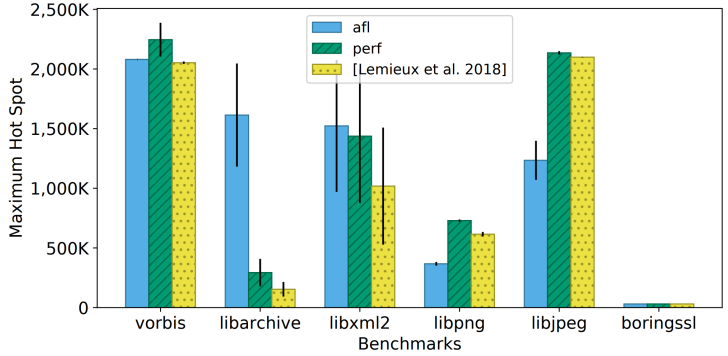
```
1  #include "fuzzfactory.hpp"
2  using namespace fuzzfactory;
3
4  class PerfFuzzFeedback : public DomainFeedback<PerfFuzzFeedback> {
5  public:
6      PerfFuzzFeedback(Module& M) : DomainFeedback<PerfFuzzFeedback>(M, "__afl_perf_dsf") { }
7
8      void visitBasicBlock (BasicBlock &bb) {
9          auto key = createProgramLocation(); // static random value
10
11         // Insert call to `dsf_increment(dsf_map, key, 1)`;
12         auto irb = insert_before(bb); // Get a handle to the LLVM IR builder
13         irb.CreateCall(DsfIncrementFunction, {DsfMapVariable, key, getConst(1)});
14     }
15 };
16
17 FUZZFACTORY_REGISTER_DOMAIN(PerfFuzzFeedback);
```

Evaluated on Google's Fuzzer Suite (6 progs)

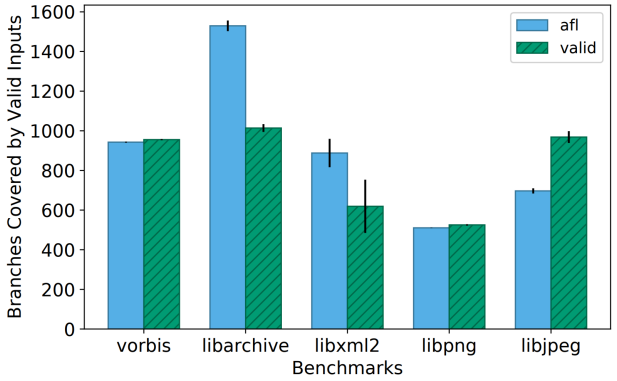
Port of **SlowFuzz**



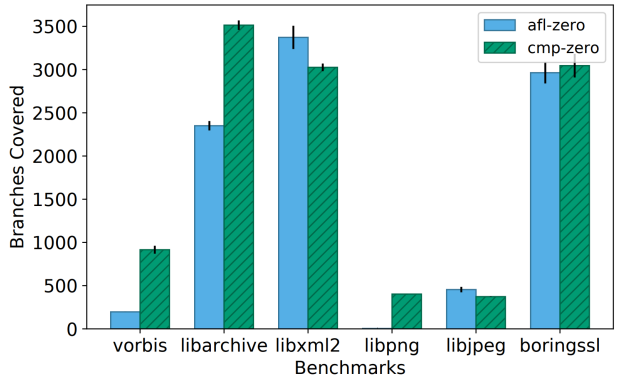
Port of **PerfFuzz**



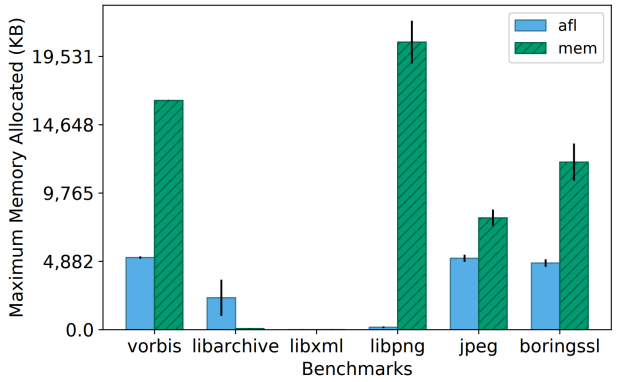
Validity Fuzzing



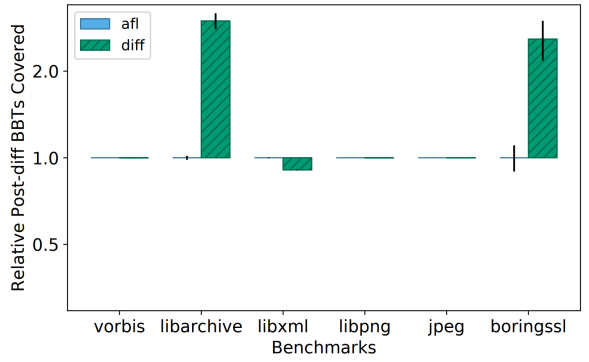
Cmp Fuzzing



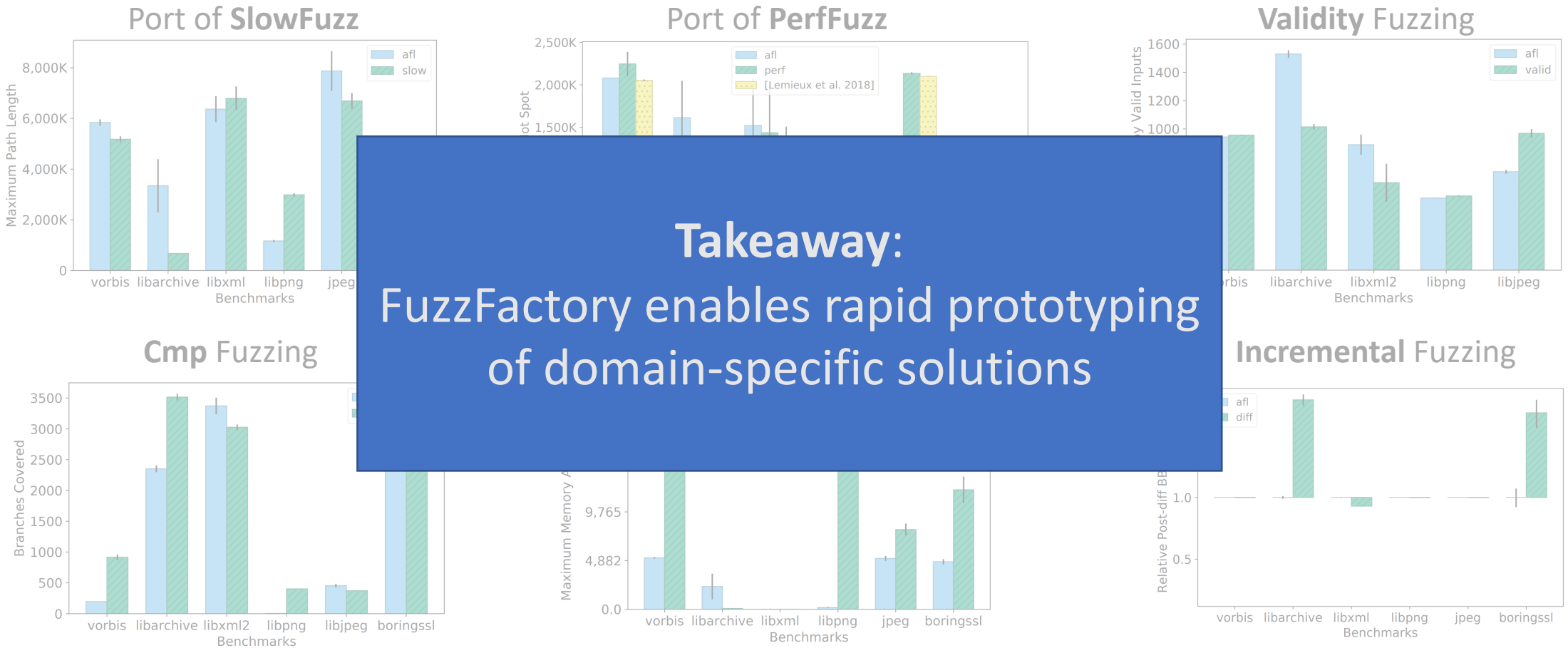
Mem **Alloc** Fuzzing



Incremental Fuzzing



Evaluated on Google's Fuzzer Suite (6 progs)

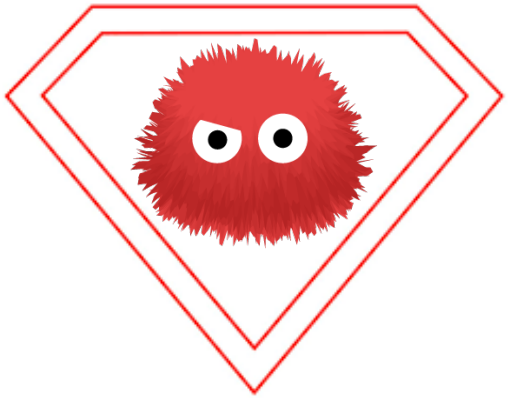


FuzzFactory enables **natural composition** of domains

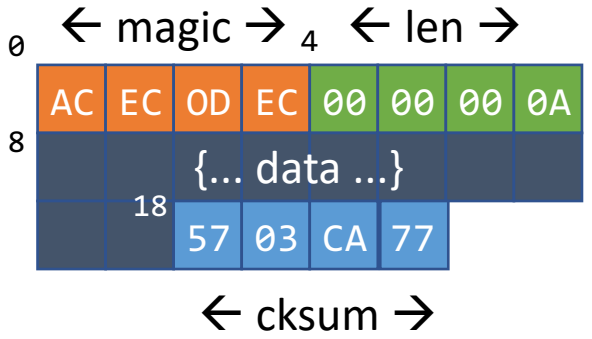
$$D = d_1 \circ d_2$$

$$is\_waypoint_D(i, S) = is\_waypoint_{d1}(i, S) \vee is\_waypoint_{d2}(i, S)$$

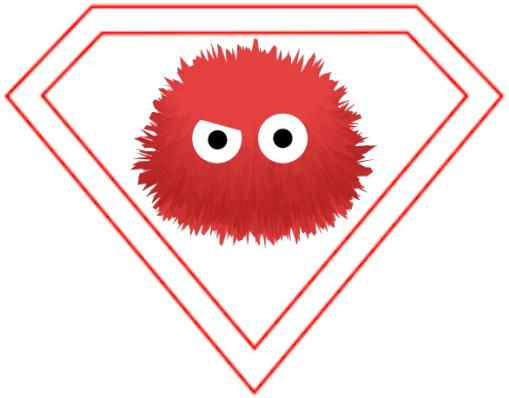
Super-Fuzzer: CMP ∘ MEM



```
dsf_t cmp_dsf = fuzzfactory_new_domain(MIN, 32);

int32 magic = (int32) input[0..4];
cmp_dsf[0] = popcnt(magic ^ 0xACECODEC);
if (magic != 0xACECODEC)
  return INVALID;

uint32 len = (uint32) input[4..8];

byte* data = (byte*) malloc(len);
cmp_dsf[1] = popcnt(sizeof(input) ^ len+12);
if (sizeof(input) != len+12)
  return INVALID;

copy(data, input[8..8+len]);

int32 cksum = (int32) input[8+len..12+len];
int32 hashed = hash(data, len);
cmp_dsf[2] = popcnt(hashed ^ cksum);
if (hashed != cksum)
  return INVALID;

// fun stuff here….
```



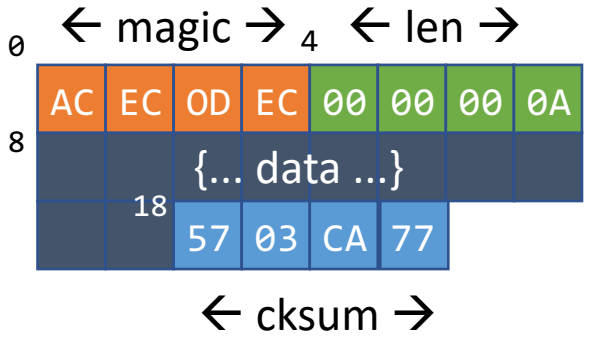
Super-Fuzzer: CMP ∘ MEM



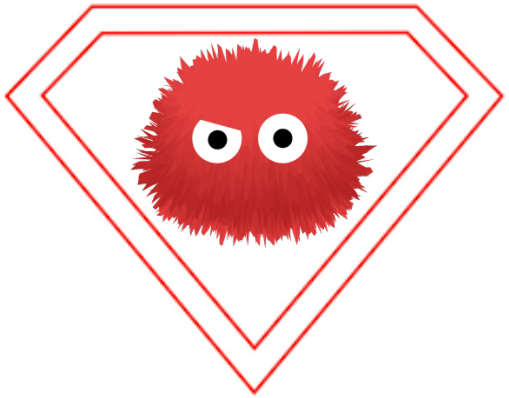
$$a \triangleright v := \max(a, v)$$
$$a_0 := 0$$

```
dsf_t cmp_dsf = fuzzfactory_new_domain(MIN, 32);
dsf_t mem_dsf = fuzzfactory_new_domain(MAX, 0);
int32 magic = (int32) input[0..4];
cmp_dsf[0] = popcnt(magic ^ 0xACECODEC);
if (magic != 0xACECODEC)
  return INVALID;

uint32 len = (uint32) input[4..8];
mem_dsf[0] += len;
byte* data = (byte*) malloc(len);
cmp_dsf[1] = popcnt(sizeof(input) ^ len+12);
if (sizeof(input) != len+12)
  return INVALID;

copy(data, input[8..8+len]);

int32 cksum = (int32) input[8+len..12+len];
int32 hashed = hash(data, len);
cmp_dsf[2] = popcnt(hashed ^ cksum);
if (hashed != cksum)
  return INVALID;

// fun stuff here….
```

Instrument all **mallocs**!



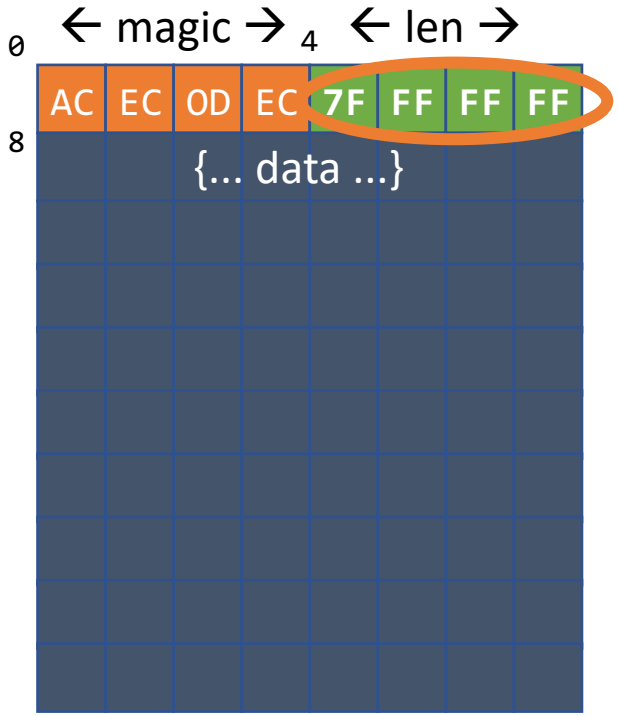
Super-Fuzzer: CMP ∘ MEM



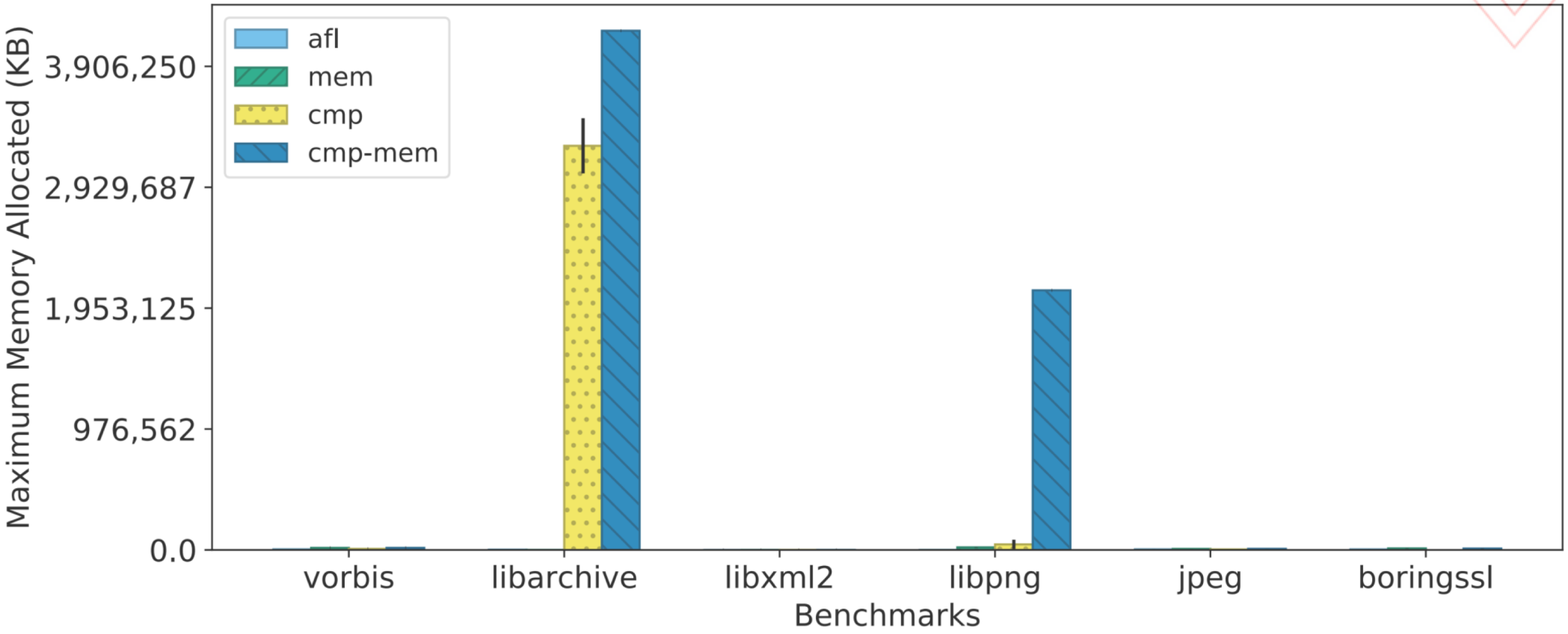
$$a \triangleright v := \max(a, v)$$
$$a_0 := 0$$

```
dsf_t cmp_dsf = fuzzfactory_new_domain(MIN, 32);
dsf_t mem_dsf = fuzzfactory_new_domain(MAX, 0);
int32 magic = (int32) input[0..4];
cmp_dsf[0] = popcnt(magic ^ 0xACECODEC);
if (magic != 0xACECODEC)
  return INVALID;

uint32 len = (uint32) input[4..8];
mem_dsf[0] += len;
byte* data = (byte*) malloc(len);
cmp_dsf[1] = popcnt(sizeof(input) ^ len+12);
if (sizeof(input) != len+12)
  return INVALID;

copy(data, input[8..8+len]);

int32 cksum = (int32) input[8+len..12+len];
int32 hashed = hash(data, len);
cmp_dsf[2] = popcnt(hashed ^ cksum);
if (hashed != cksum)
  return INVALID;

// fun stuff here….
```

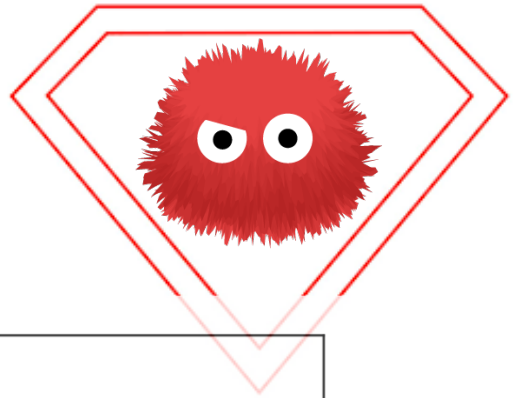
Instrument all **mallocs**!

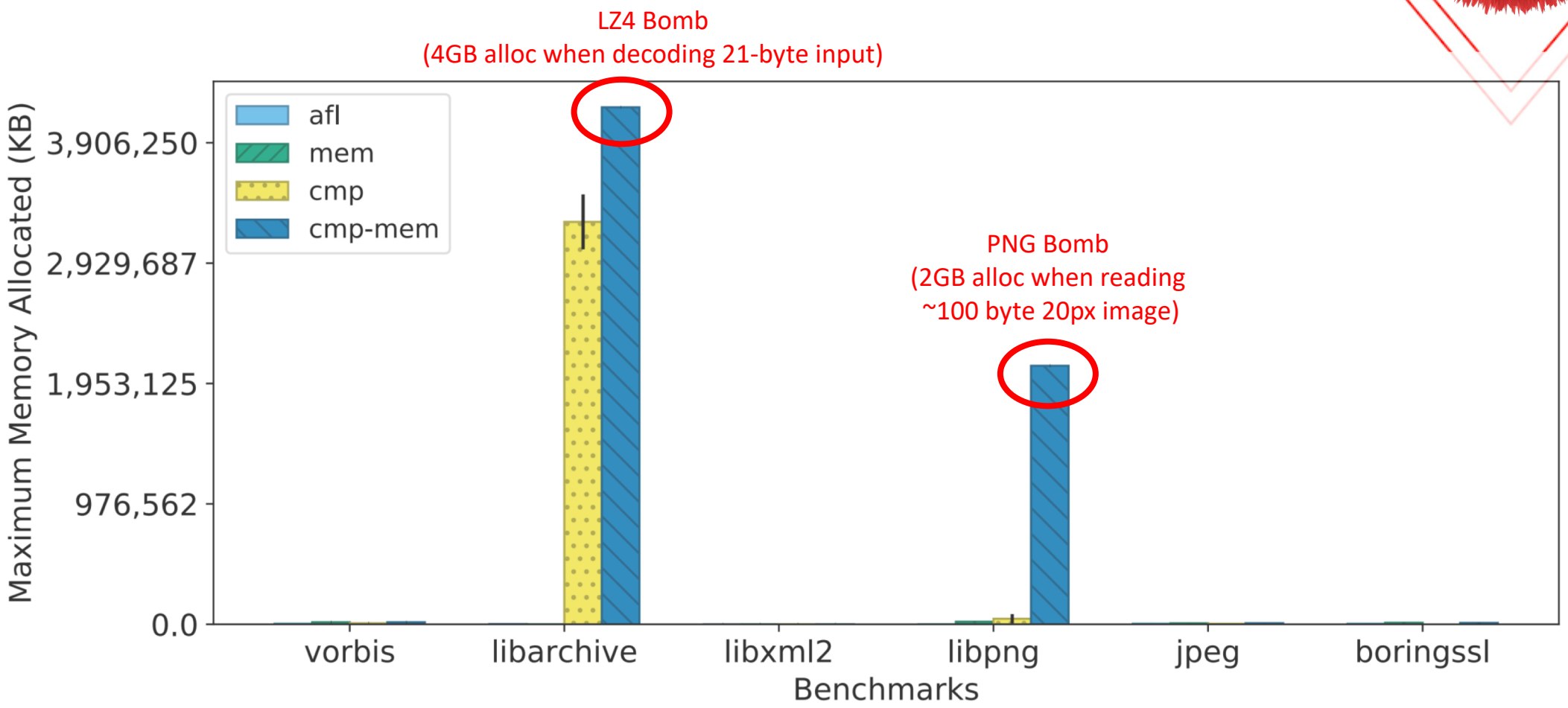


Super-Fuzzer: CMP ∘ MEM

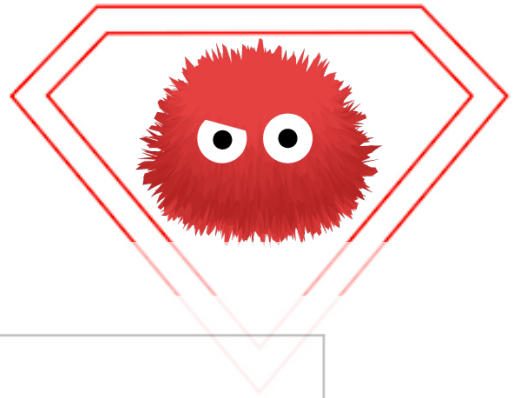


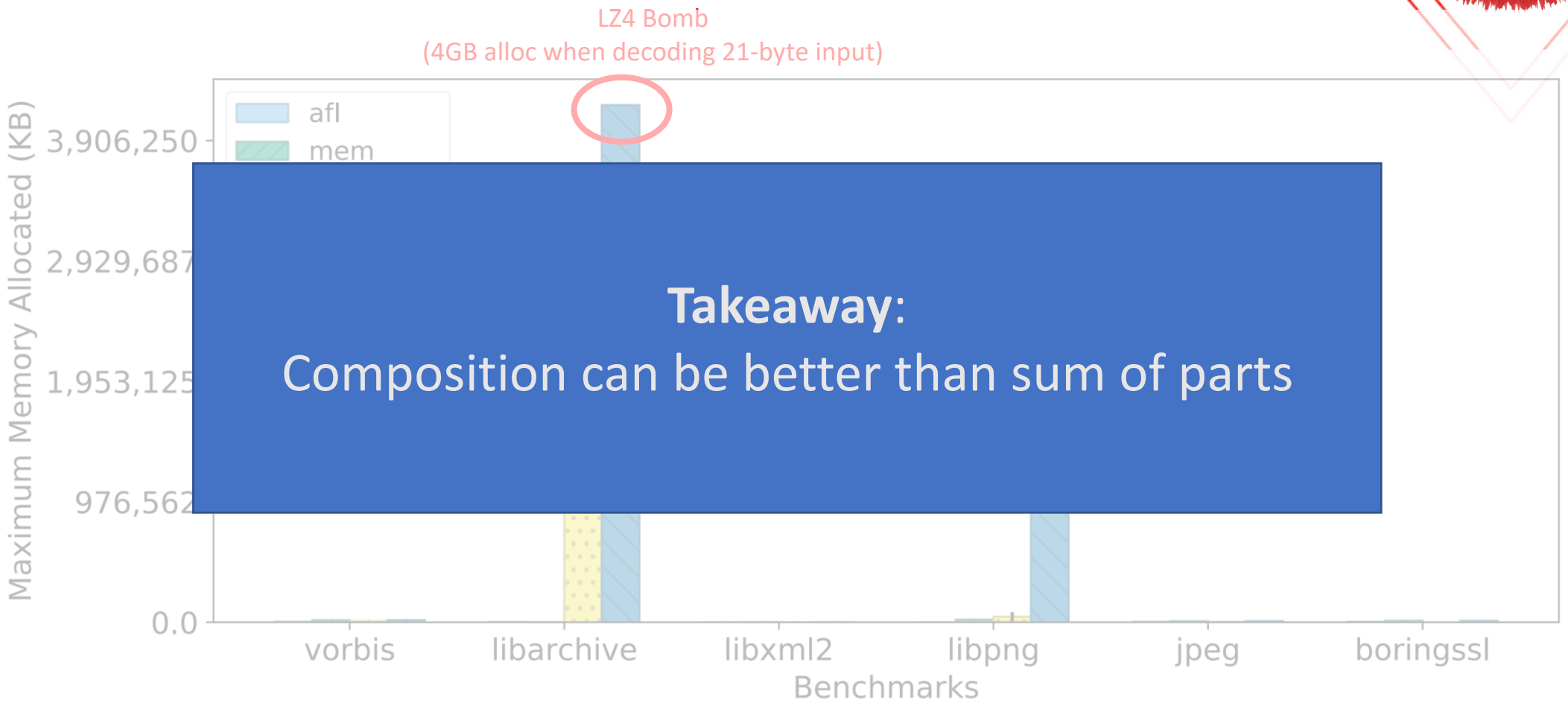
Super-Fuzzer: CMP ∘ MEM





Super-Fuzzer: CMP ∘ MEM





Takeaway:
Composition can be better than sum of parts

"FuzzFactory: Domain-Specific Fuzzing with Waypoints", R. Padhye, C. Lemieux, K. Sen, L. Simon, H. Vijayakumar. OOPSLA 2019

Summary

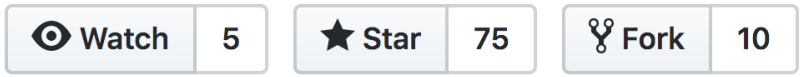
Effective fuzzing is domain-specific

Key abstraction: **Waypoints**

FuzzFactory: Rapid prototyping of waypoints-based fuzzers

FuzzFactory enables natural **composition**

https://github.com/rohanpadhye/FuzzFactory



Follow-up Work

Domain-Specific Fuzzing of ARM TrustZone via emulation

[USENIX Security ’20]

Domain-Specific Fuzzing of Big Data Analytics

(Work in progress)

Domain-Specific Fuzzing of Self-Driving Cars

(Early stages)