Performant Binary Fuzzing without Source Code using Static Instrumentation

Eric Pauley*, Danfeng Zhang‡, Gang Tan‡, Patrick McDaniel*

* University of Wisconsin–Madison (Work performed while at Penn State)

‡ Pennsylvania State University

IEEE CNS 2022

Approaches to Bug Finding

Unit/Functional Tests

- Manual effort required
- Full coverage is difficult (E.g., corner cases, memory safety)

Formal Verification

- Not possible for most projects

Automated Testcase Generation

- E.g., Fuzzing

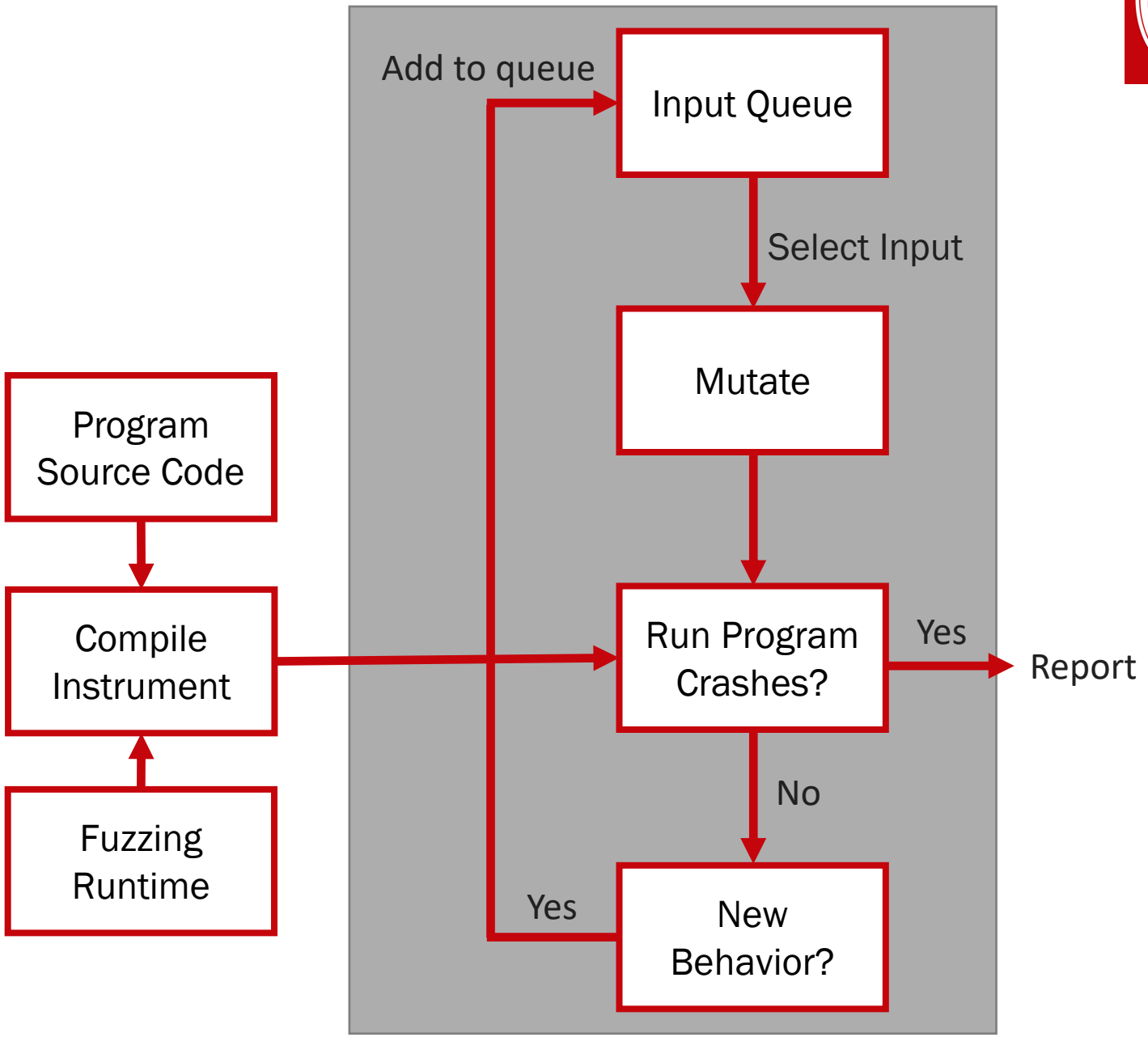
Greybox Fuzzers

Advantages:

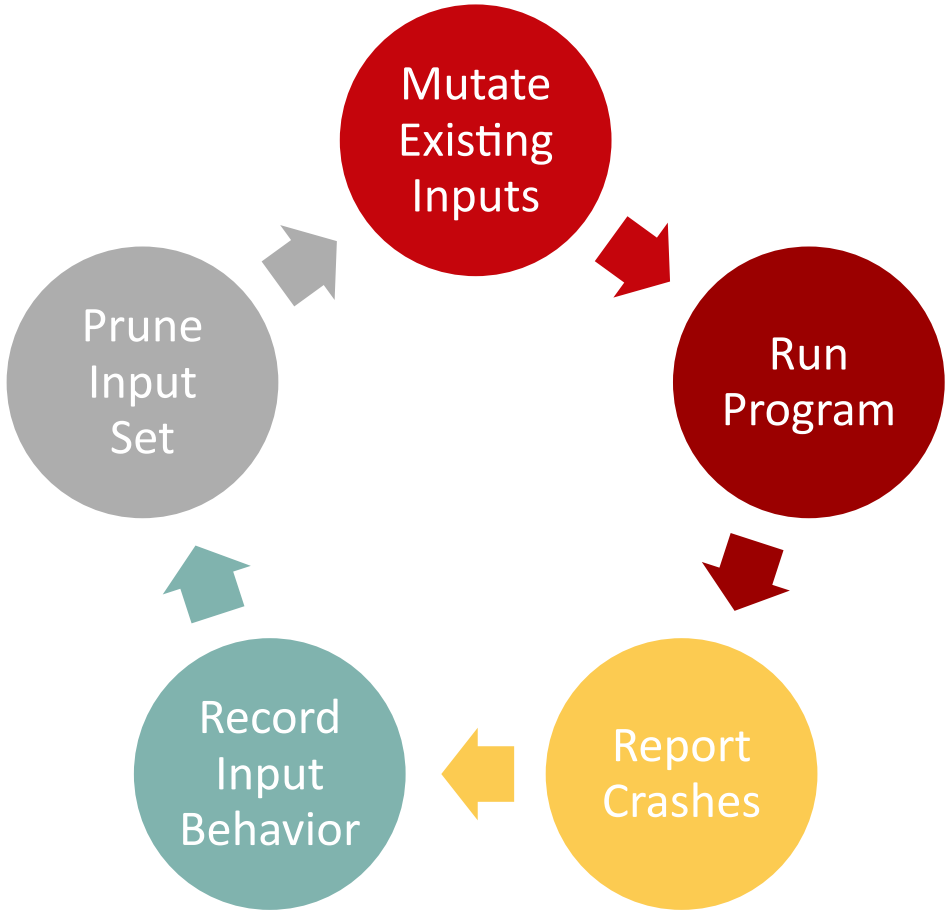
- Fully automatic
- Don’t rely on predictable input grammar

Disadvantages:

- Sensitive to initial inputs
- Highly probabilistic



How greybox fuzzers work



Mutation Strategies:

- Randomized mutation
- Magic Bytes
- Symbolic Execution
- Gradient Descent

Recording behavior:

- Control flow coverage
- Data flow
- Comparisons

Fuzzing Instrumentation

Source Code

```
int main(…) {
  unsigned int val = 0;
  fread(&val, 4, 1, stdin);

  if (val == 0x12345678)
    val = *(volatile int*)NULL;
  return val;
}
```

Compiler Frontend

Intermediate Representation

```
define i32 @main(i32, i8**) #0 {
  %9 = call i64 @fread(…)
  %11 = icmp eq i32 %10, 305419896
  br i1 %11, label %12, label %14

; <label>:12:
  %13 = load null, align 4
  store i32 %13, i32* %6, align 4
  br label %14

; <label>:14:
  %15 = load i32, i32* %6, align 4
  ret i32 %15
}
```

Is source code necessary?

Source Code

```
int main(...) {
  unsigned int val = 0;
  fread(&val, 4, 1, stdin);

  if (val == 0x12345678)
    val = *(volatile int*)NULL;
  return val;
}
```

Compilation

Lost:

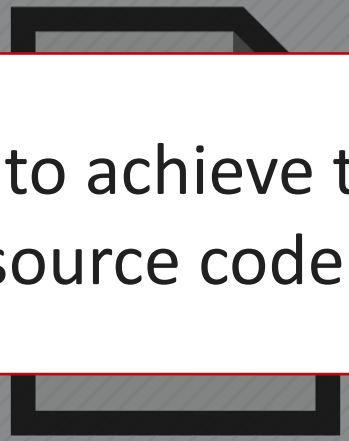
- Typing Information
- Control Flow
- Instruction Alignment

Is source code necessary?

Source Code

```
int main(...) {
  unsigned int val = 0;
  fread(&val, 4, 1, stdin
  if (val == 0x12345678
    val = *(volatile int*)
  return val;
}
```

Can we use ***binary instrumentation*** to achieve the same fuzzing techniques without source code?

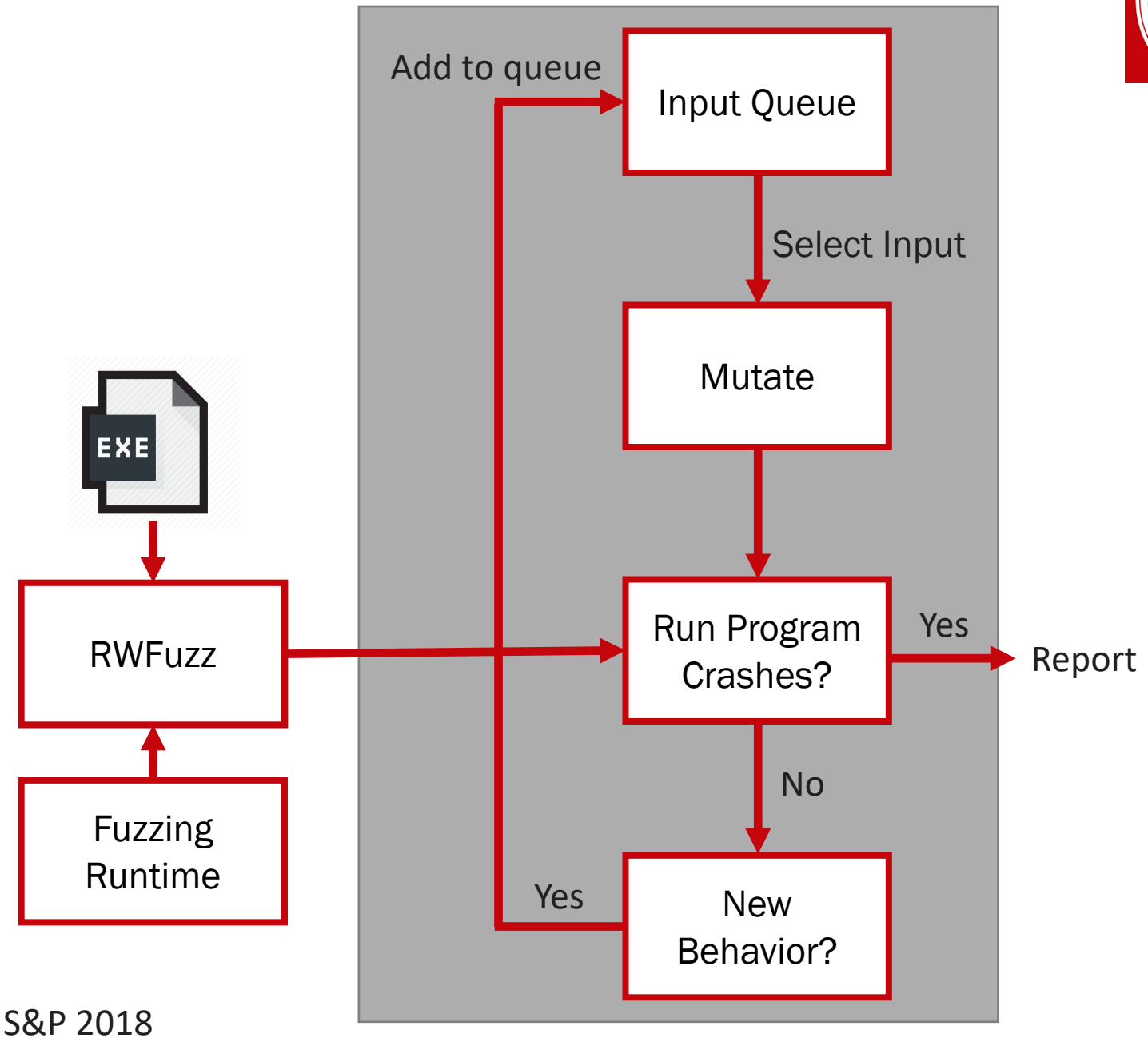


Lost:

- Typing Information
- Control Flow
- Instruction Alignment

What is RWFuzz?

- Instruments binaries for fuzzing
- Compatible with an existing fuzzer [1]



[1] Angora, Chen et al. S&P 2018

Binary Fuzzing Challenges

Coverage Measurement

- Basic Blocks/Instruction Alignment unavailable

Context Sensitivity

- Can't modify stack layout

Inferring Comparisons

- Comparisons span multiple instructions

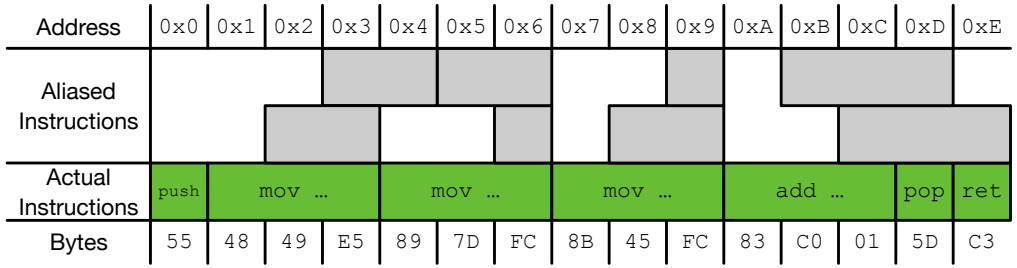
Instruction alignment

- Challenge: x86 instructions unaligned
  - Cannot determine alignment statically
- Solution: instrument everything
  - Leverage superset disassembly[1]
- Downside: still don't know control flow

```
1 int foo(int a)
2     return a + 1;
```

```
0 55          push rbp
1 48 89 e5    mov  rbp, rsp
4 89 7d fc    mov  [rbp-0x4], edi
7 8b 45 fc    mov  eax, [rbp-0x4]
A 83 c0 01    add  eax, 0x1
D 5d          pop  rbp
E c3          ret
```

(a) A simple C function `foo` compiled with GCC (`-O0`)



(b) Assembled binary. Actual instructions map to source assembly. Aliased instructions are valid instructions at other offsets.

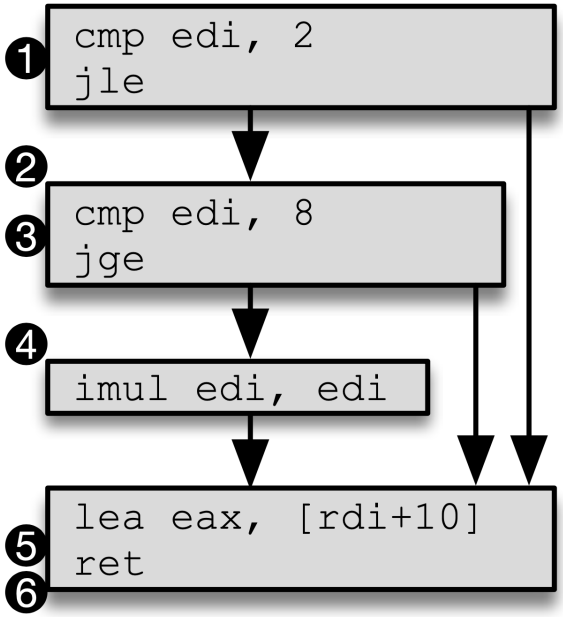
[1] Bauman et al. NDSS 2018

Coverage Measurement

- Challenge: control flow unavailable

- Solution: heuristic control flow instrumentation
  - Over-instrumenting is better than under-instrumenting

```
1 int foo(int a) {
2   if (a > 2)
3     if (a < 8)
4       a *= a;
5   return a + 10;
6 }
```



Context-sensitive coverage

- Crashes may only be discoverable in certain contexts
- Need to track call context to maximize coverage

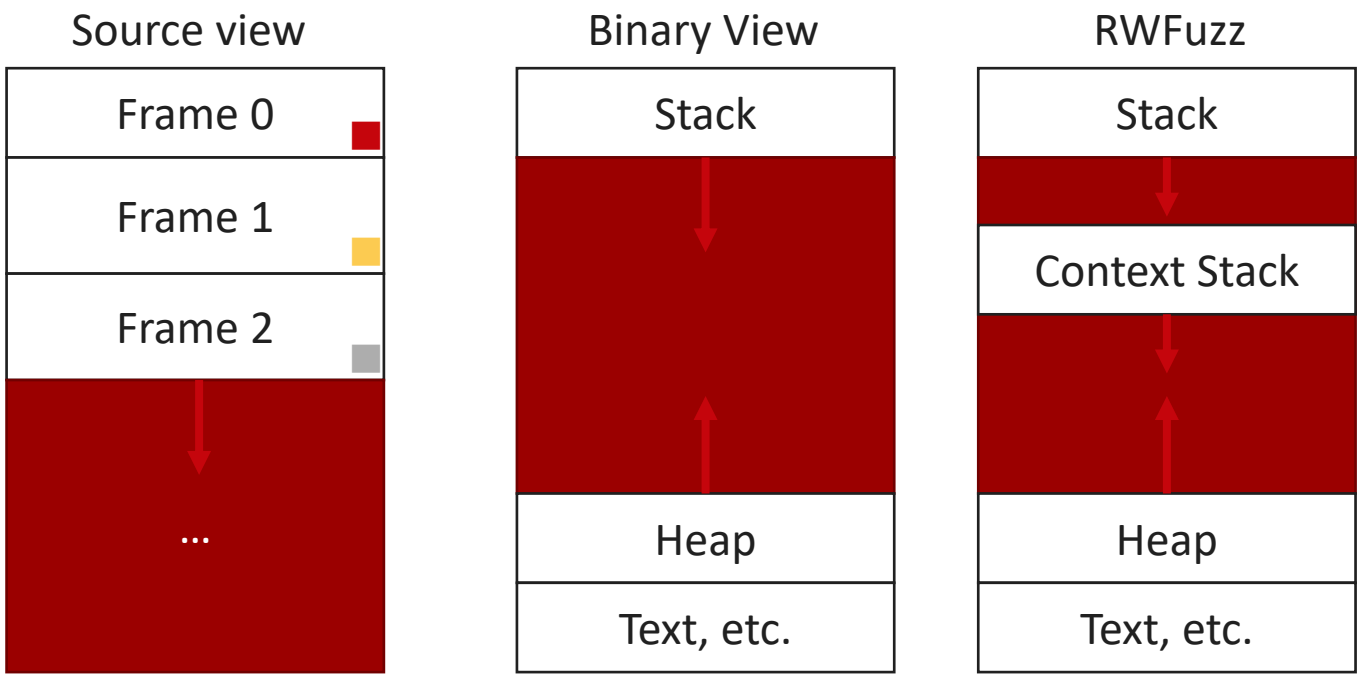
```
void foo(uint16_t a, uint16_t b) {
  if (a - b < 0x100)
    if (a < 0x6010)
      // Crash
}

int main(int argc, char** argv) {
  uint16_t a = read_uint16();

  foo(a, 0x7000);
  foo(a, 0x6000);
}
```

Input	a – b [1]	a –b [2]	Coverage	Cov (With Ctx)
0x7050	0x50	0x1050	■+■	■+■
0x6050	0xF050	0x50	■+■	■+■
0x6005	0xF005	0x5	■+■+■+✗	■+■+■+✗

Context-sensitive coverage: Shadow Stack



Inferring Comparisons

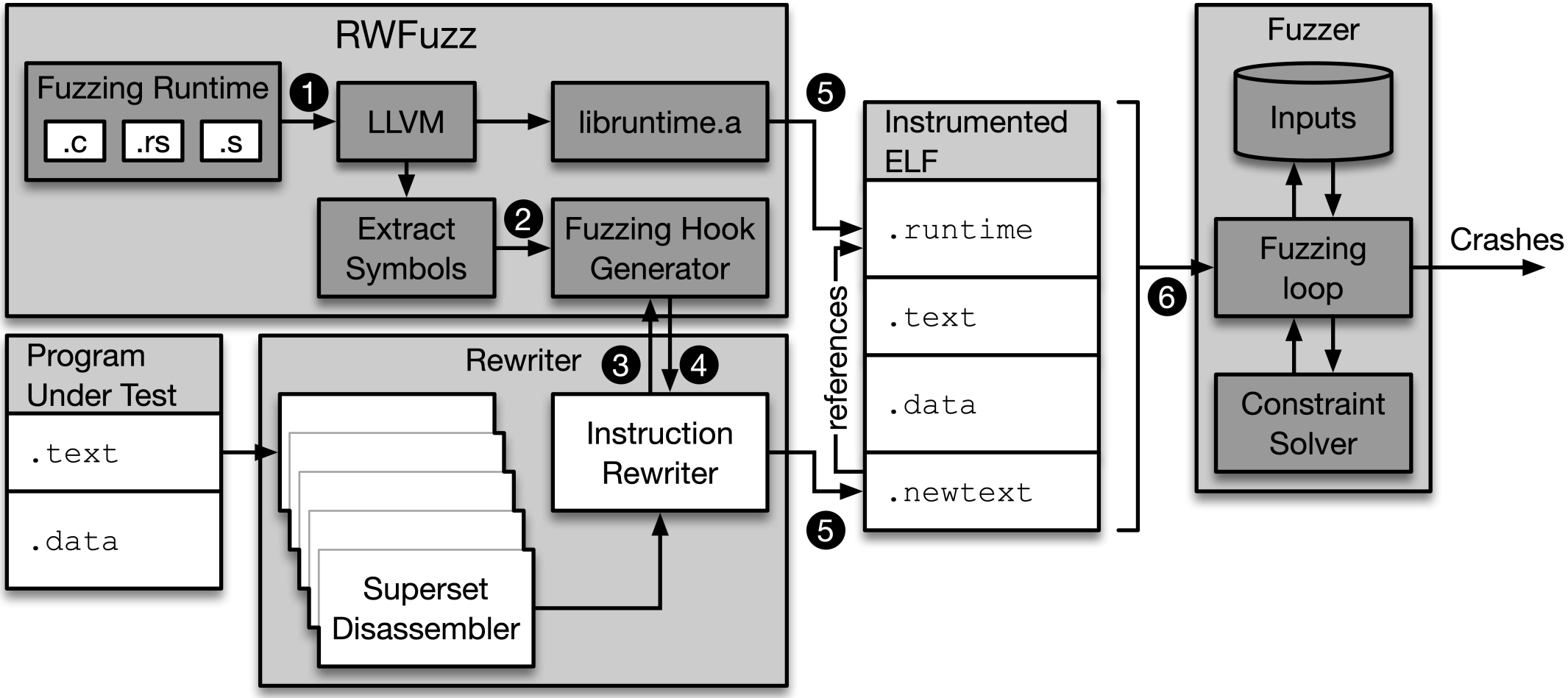
- Challenge: x86-64 comparisons are performed in multiple instructions
- Solution: mirror processor state in fuzzing runtime

Compare	Conditional
• Reads inputs • Outputs all results • Populates FLAGS	• Reads FLAGS • Determines comparison used • Changes control flow

```
1 int min(int a, int b) {
2   if (a > b)
3       return b;
4   return a;
5 }
```

```
0 39 f7        cmp    edi,esi # Compare
2 89 f0        mov    eax,esi
4 0f 4e c7     cmovle eax,edi # Conditional
7 c3           ret
```

Instrumenting programs with RWFuzz



Evaluation

Evaluation Overview

Functional Test

- Evaluation on manually-generated bugs

Performance Evaluation

- Comparison against source-code fuzzers
- Synthetic bug corpus [1]

[1] Dolan-Gavitt et al. (S&P 2016)

Manually-created bugs

```
1 int main(int argc, char **argv) {
2   char buf[10];
3   gets(buf);
4   return buf[0] != NULL;
5 }
```

(a) `simple` - A buffer overrun can be caused by calling `gets`

```
1 int main(int argc, char **argv) {
2   unsigned int val = 0;
3   fread(&val, 4, 1, stdin);
4
5   if (val == 0x12345678)
6     val = *(volatile int *)NULL;
7   return val;
8 }
```

(b) `magic` - A specific input causes a null-pointer exception

```
1  __attribute__((noinline)) volatile
2  int foo(unsigned int a, unsigned int b) {
3    if (a - b < 0x1000)
4      if (a < 0x60000100)
5        *(volatile int *)NULL;
6    return 1;
7  }
8
9  int main(int argc, char **argv) {
10   unsigned int a = 0;
11   unsigned int ret = 0;
12   fread(&a, 4, 1, stdin);
13
14   ret += foo(a, 0x59239472);
15   ret += foo(a, 0x70000000);
16   ret += foo(a, 0x80000000);
17   ret += foo(a, 0x90000000);
18   ret += foo(a, 0xa0000000);
19   return ret;
20 }
```

(c) `context` - The bug is only triggered in the first call to `foo`. Note that integers are unsigned.

Results on manually-created bugs

Program	Time to find crash with each fuzzer (s)			
	RWFUZZ	Angora	LAF-Intel	QAFL
simple	2.7	5.6	0.2	0.3
magic	1.8	0.9	38.9[1]	–
context	4.0	3.6	–	–
undef	1.9	–	–	–

Figure: Time taken by each fuzzer to find manually-generated bugs

```
1 __attribute__((noinline))
2 int foo(unsigned int a, unsigned int b) {
3   if (a - b < 0x1 && a < 0x60000100)
4     return *(int *)(a - b);
5   return 1;
6 }
7
8 // Same as in `context`
9 int main(int argc, char **argv) {...}
```

(d) `undef` - The bug may be optimized out by some compilers

Evaluation on Synthetic Bugs

Table: Bugs found on LAVA-M in 1 hour

Program	Number of bugs found				
	RWFuzz	Angora	LAF-Intel	QAFL	RetroWrite[1]
`base64`	45	43	42	0	2
`md5sum`	59	56	6	0	0
`uniq`	29	29	16	0	2
`who`	258	258	2	0	0

[1] Median bugs found over 5 trials of 24 hours [12].

[12] Dinesh et al. (S&P 2022)

Performance overhead of RWFuzz

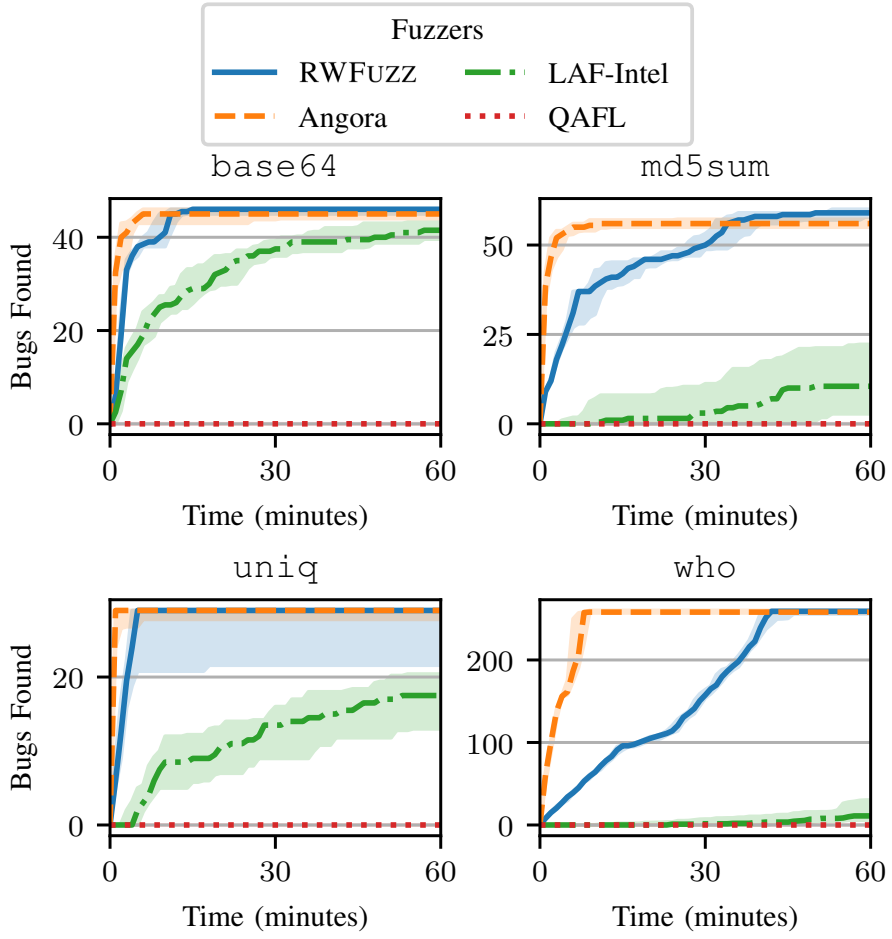


Figure: bugs found over time by four fuzzers

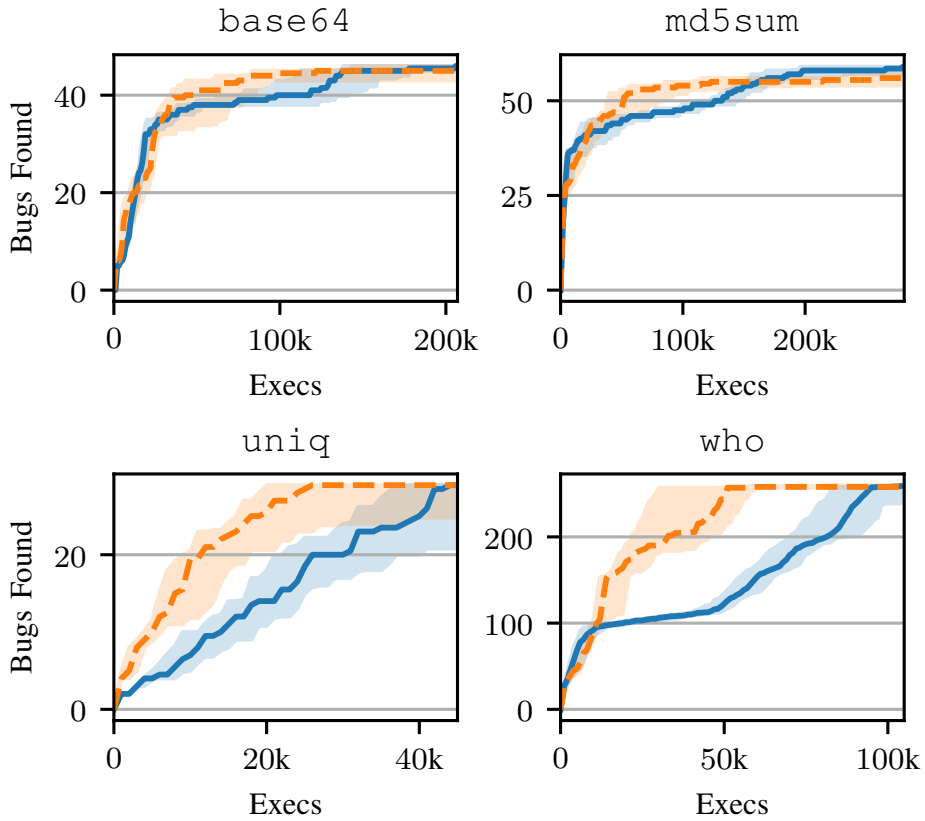


Figure: Bugs found per program execution

Thank you!

@EricPauley_

epauley@cs.wisc.edu

pauley.me/rwfuzz