Fine-Grained Coverage-Based Fuzzing

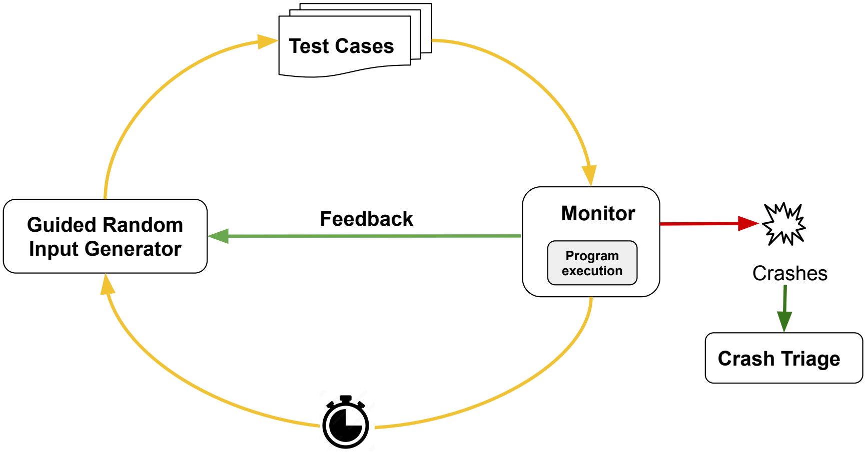
Bernard Nongpoh, Marwan Nour, Michaël Marcozzi, Sébastien Bardin

Université Paris-Saclay, CEA, List
Palaiseau, Paris Metropolitan Area, France
first.last@cea.fr

International Fuzzing Workshop (FUZZING) 2022
San Diego, CA, USA

April 24, 2022

Feedback-based Fuzzing Process



- Feedback $\longrightarrow$ Branch Coverage

Problem

Branch Coverage is a shallow metric

Goals

- Support **fine-grained coverage metrics** within state-of-the-art fuzzers
- Leverage decades of **software engineering** research on defining such metrics
- Make them available in existing fuzzers **out-of-the-box** (no fuzzer modification)

Contributions

- Developed a Clang pass to **annotate** C code with fine-grained coverage criteria
- Preliminary evaluation using two state-of-art fuzzers, namely **AFL++** and **QSYM** on the LAVA-M benchmarks

Motivating Example

```c
void check(int current_temp,char *data[] ){

if(current_temp>=50)
    {
    // Deal with appliance running outside
    // the allowed temperature limit
    ...
        // The bug triggers a detectable crash
        // only when current_temp==50
        // and when rare specific values
        // are present in data
    }
}
```

$\Longrightarrow$

```c
void check(int current_temp,char *data[] ){
...
if (current_temp>=50 != current_temp>50) {
}
...

if(current_temp>=50)
    {
    // Deal with appliance running outside
    // the allowed temperature limit
    ...
        // The bug triggers a detectable crash
        // only when current_temp==50
        // and when rare specific values
        // are present in data
    }
}
```

A **buggy** program checking if an appliance is running outside its allowed temperature range

Our Approach

Making test objectives explicit in the code of the fuzzed program with **labels**

```
statement_1;
if(x==y && a<b)
   {...};
statement_3;
```

⟹

```
statement_1;
//l-1: x==y
//l-2: x!=y
//l-3: a<b
//l-4: a>=b
if(x==y && a<b)
   {...};
statement_3;
```

Condition Coverage (CC)

```
statement_1;
//l-1: x==y && a<b
//l-2: x!=y && a<b
//l-3: x==y && a>=b
//l-4: x!=y && a>=b
if(x==y && a<b)
   {...};
statement_3;
```

Multiple Condition Coverage (MCC)

```
statement_1;
x=a+b;
statement_3;
```

⟹

```
statement_1;
x=a*b;
statement_3;
```

Arithmetic Operator Replacement (AOR) Mutant

```
statement_1;
//l-1: (a+b)!=(a*b)
x=a+b;
statement_3;
```

Weak Mutation Coverage (WM)

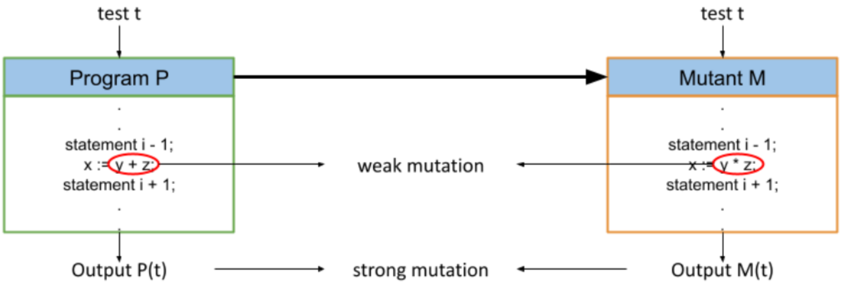
Code Coverage Criteria

```
if(A && B){
    //Statement-1
}
else{
    //Statement-2
}
```

Combination	DC	CC	DCC	MCC
$A \&\& B$	✓	✗	✓	✓
$\bar{A} \&\& B$	✓	✓	✗	✓
$A \&\& \bar{B}$	✗	✓	✗	✓
$\bar{A} \&\& \bar{B}$	✗	✗	✓	✓

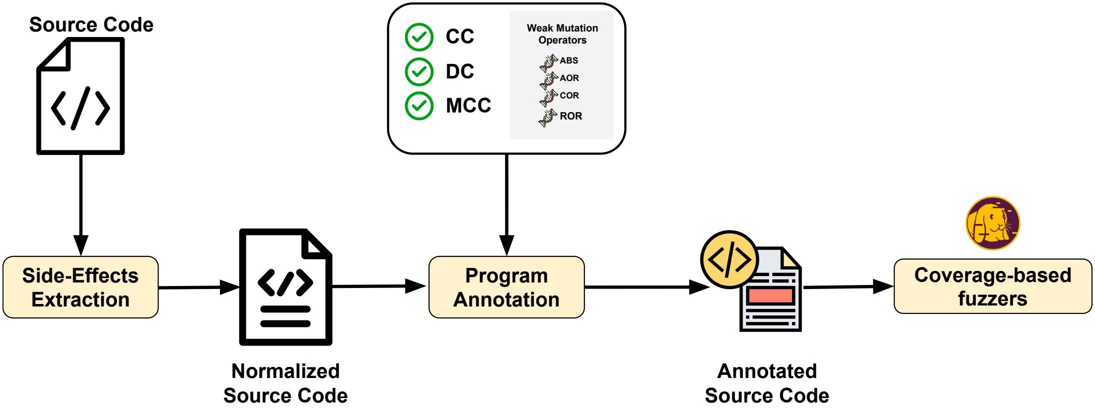
- Decision Coverage (DC)
- Condition Coverage (CC)
- Decision Condition Coverage (DCC)
- Multiple Condition Coverage (MCC)

Mutation Coverage



- **Strong Mutation Coverage (SM)**: A Mutant M is covered/killed by a test t if the outputs of $p(t)$ and $M(t)$ differ from each other.
- **Weak Mutation Coverage (WM)**: A Mutant M is covered/killed by a test t if the internal states of $P(t)$ and $M(t)$ differ from each other right after the mutated location.

Fine-grained Coverage-based Fuzzing



Workflow

Side-Effects

Output: **ab** $\neq$ Output: **aabbab**

```
statement_1;
//l-1: print("a")
//l-2: !print("a")
//l-3: print("b")
//l-4: !print("b")
if(print("a") && print("b"))
   {...};
statement_3;
```

Program with labels for CC

$\Longrightarrow$

```
statement_1;
if (print("a")) {}
if (!print("a")) {}
if (print("b")) {}
if (!print("b")) {}
if(print("a") && print("b"))
   {...};
statement_3;
```

Annotated program for CC

Examples of Side-Effects Extraction (1)

```c
int foo();
....
if(foo()){ }
```

$\Longrightarrow$

```c
int foo();
....
int temp = foo();
if(temp){ }
```

Side-effect in an atomic condition at a decision point

Examples of Side-Effects Extraction (2)

```c
int foo();
....
if(x>0 && foo()){ }
```

$\Longrightarrow$

```c
if(x>0){
int temp = foo();
if(temp){}
else
goto label_1;
}
label_1:{
}
```

Side-effect in the second atomic condition of a lazy boolean operator (AND case)

Examples of Side-Effects Extraction (3)

```c
int foo();
....
if(x>0 || foo()){ }
```

$\Longrightarrow$

```c
if(x>0){
goto label_1;
}
else{
int temp = foo();
if(temp)
label_1:{}
else{}
}
```

Side-effect in the second atomic condition of a lazy boolean operator (OR case)

Program Annotation

```
statement_1;
//l-1: x==y
//l-2: x!=y
//l-3: a<b
//l-4: a>=b
if(x==y && a<b)
    {...};
statement_3;
```

Program with labels for CC

$\Longrightarrow$

```
if (x==y) {}
if (x!=y) {}
if (a<b) {}
if (a>=b) {}
if(x==y && a<b)
    {...};
statement_3;
```

Annotated program for CC

- Off-the-shelf fuzzers will be able to handle **Condition/Branch** Coverage out-of-the-box. **Thanks to This!**

Research Questions

- **RQ1**: Is our code annotation tool **effective** and **useful**?
  - Is it easy to use and does out-of-the-box **integration** with existing fuzzers work well?
  - Can it **scale** to real-world applications?
- **RQ2**: Does our fine-grained approach allow to **improve** over the baseline state-of-art fuzzers?

Experimental Evaluation

- **Objective**: Fuzzer performance with and without label instrumentation (MCC and WM)
- Coverage-based fuzzers: **AFL++** and **QSYM**
- Infrastructure
  - Intel Skylake CPU, with 192GB memory RAM and 72 logical cores running at 2.6GHz.
- Time budget of **24 hours** (repeated 5 times)
- Benchmark: **LAVA-M Benchmark** Suite

Preliminary Experimental Evaluation

Application	LOC	MCC	Weak Mutation				Total
			ABS	AOR	COR	ROR	
uniq	494	204	61	7	18	45	335
base64	255	26	56	7	6	51	146
md5sum	663	125	113	20	24	79	361
who	622	170	180	15	30	19	414

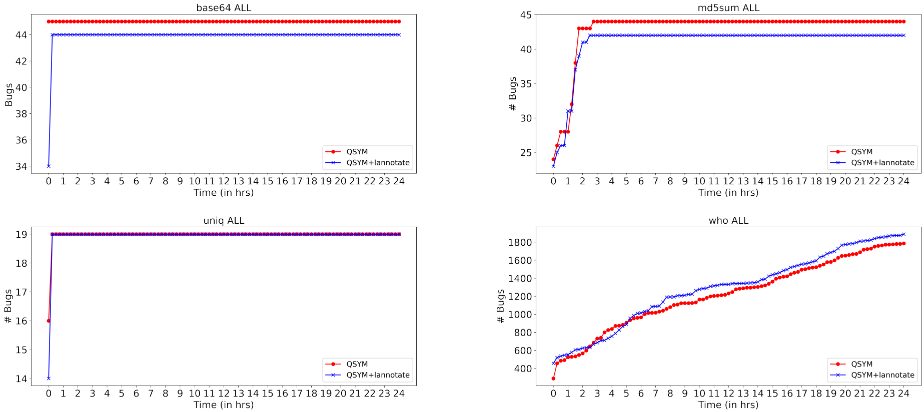
Number of **labels** on LAVA-M benchmark suite

Preliminary Experimental Evaluation

	AFL++	AFL++ + Iannot	QSYM	QSYM + Iannot
uniq	1	1	19	19
base64	0	2	45	44
md5sum	0	0	44	42
who	1	1	1734	1887

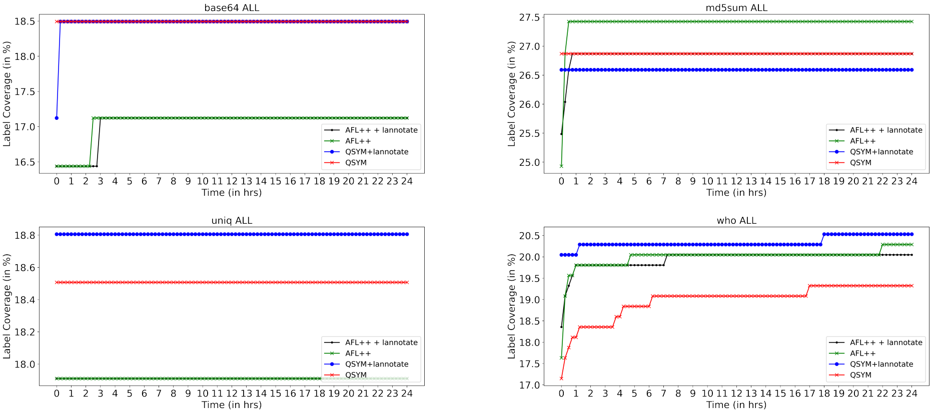
Bugs found on the LAVA-M benchmark by fuzzers

Time to Bug



Average number of bugs found by QSYM and QSYM+*lannot* vs time

Label Coverage



Cumulative label coverage (in %) vs time

Conclusion

- Borrow well-established research over fine-grained code coverage criteria and provide off the shelf support in popular fuzzers
- Making test objectives (defined by fine-grained metrics) explicit as new branches in the target program
- Preliminary evaluation on the four **LAVA-M benchmark suite**
- Tested on two fuzzers: **AFL++** and **QSYM**
- Preliminary findings:
  - On average, 100 more bugs being discovered in total
  - Bugs being uncovered faster during the fuzzing process
  - Label coverage improvement in two applications

Future Direction

- Pruning out **infeasible** labels
- Test on **Magma** (ground-truth benchmark) and **real-world** applications
- Investigate the effect of each coverage criteria on the **fuzzing performance** separately
- Evaluation on **standard metrics** as suggested by the fuzzing community (edge coverage)
- Investigate the **overhead** introduce by labelling in fuzzer throughput

"Making current fuzzer support fine-grained coverage metrics out-of-the-box"

BINSEC

Preprint available at **https://binsec.github.io/**

Follow us on Twitter

@BinsecTool, @BernardNongpoh, @__M4rwan, @michaelmarcozzi

The team is looking for Ph.D. Students and PostDoc
Visit **https://binsec.github.io/** for more information