Sapienza
Università di Roma

Fuzzing Symbolic Expressions

Emilio Coppa

The work presented today appeared at ICSE 2021 (conference) and COSE 2021 (journal).

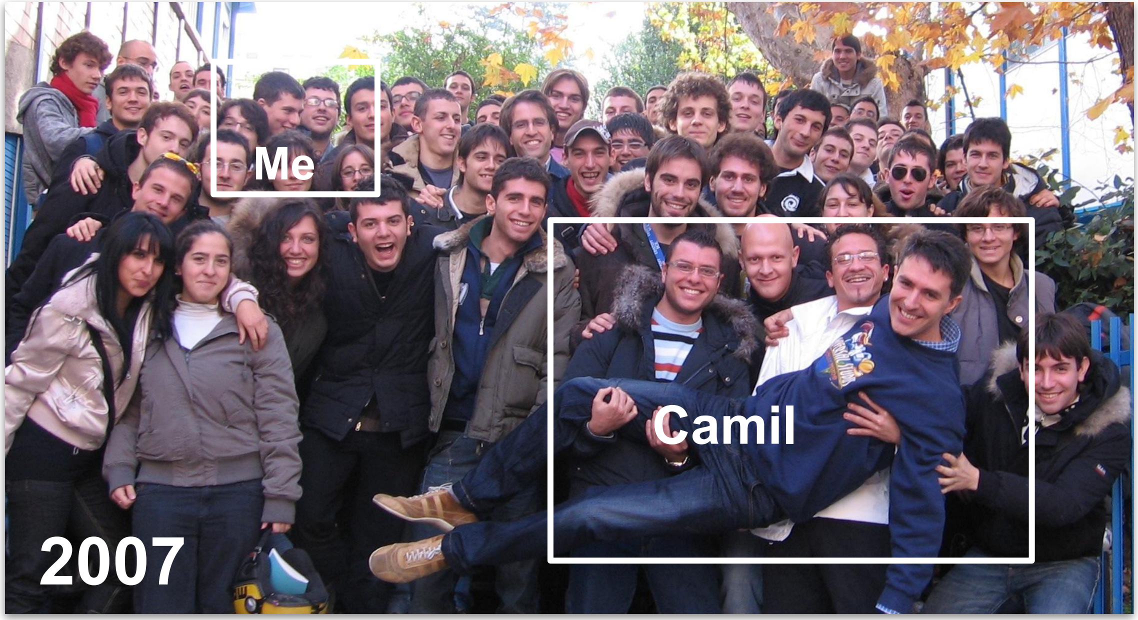
Co-authors: Luca Borzacchiello and **Camil Demetrescu**

The work presented today appeared at ICSE 2021 (conference) and COSE 2021 (journal).
Co-authors: Luca Borzacchiello and **Camil Demetrescu**

Unfortunately, Camil passed away in April 2022

- Full professor at Sapienza University of Rome
- A brilliant researcher (algorithms, program analyses)
- One of the best teachers. Students loved him.

He was my (best) teacher…



He was my thesis advisor (both BSc and MSc)…

He was my co-author for 10+ years and 20+ papers

PLDI 2012 @ China

[my first paper]

He was my friend…

It was an honour for me to work with Camil.

Symbolic execution is the coolest program analysis ever met!
However:

- hard to implement
- …a lot of (non trivial) scalability issues!

When instead considering fuzzing:

- simple(r) to implement
- quite effective in practice
  (see OSS-Fuzz results)

Symbolic execution is the coolest program analysis ever met!
However:

- hard to implement
- …a lot of (non trivial) scalability issues!

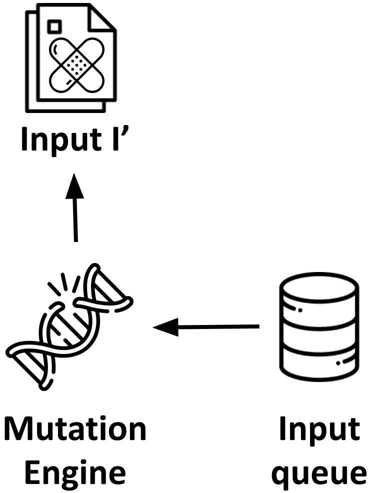
When instead considering fuzzing:

- simple(r) to implement
- quite effective in practice (see OSS-Fuzz results)

Coverage-guided Fuzzing

1. Pick an input from the queue

2. Mutate it:
   - random bit-flips
   - random substitutions
   - random… things

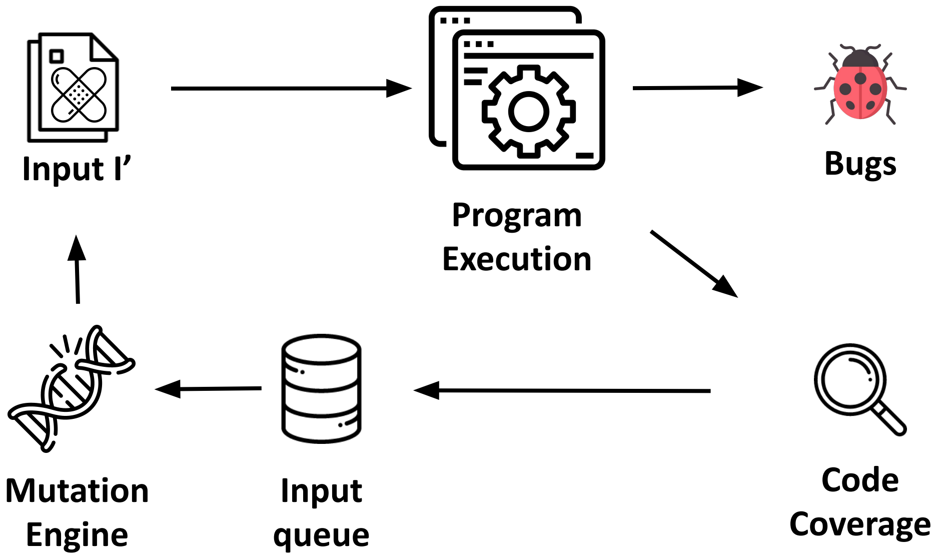


Coverage-guided Fuzzing (2)

3. Run the program
   - Look for crashes
   - Track code coverage

4. If new coverage: keep mutating that input!

5. Repeat from (1) for a trillion of times



Can we reduce the number of (wasted) attempts?

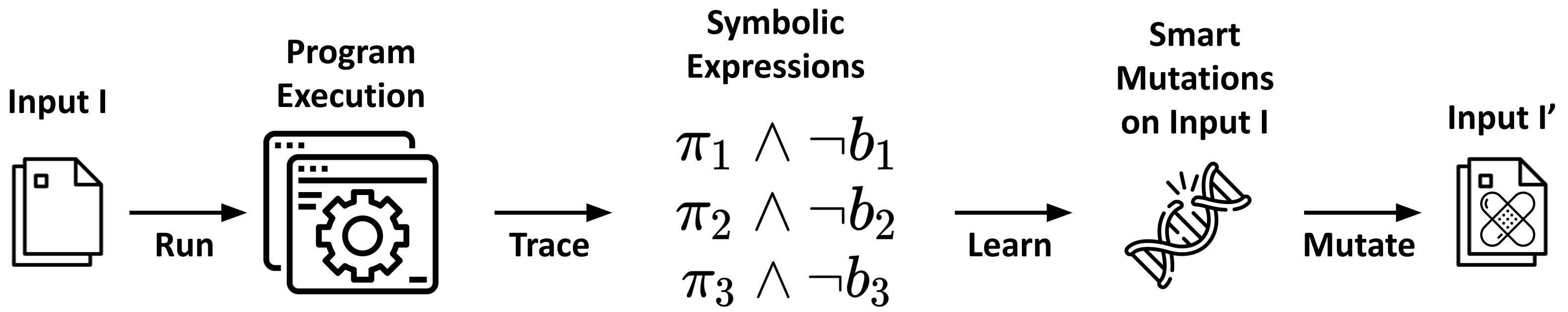
Can we reduce the number of (wasted) attempts?

Some works have used taint analysis to understand which bytes to mutate.

Can we reduce the number of (wasted) attempts?

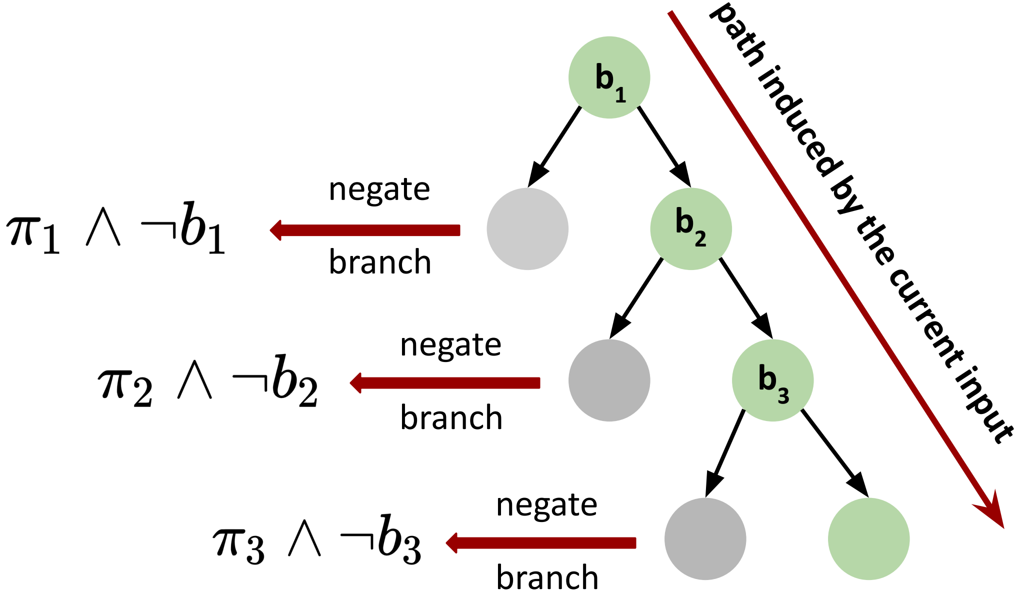
Some works have used taint analysis to understand which bytes to mutate.

What if we build symbolic expressions to **learn** how to **mutate** the **input**?


Input I
Run
Program Execution
Trace
Symbolic Expressions
$\pi_1 \wedge \neg b_1$
$\pi_2 \wedge \neg b_2$
$\pi_3 \wedge \neg b_3$
Learn
Smart Mutations on Input I
Mutate
Input I'

Trace? Concolic Execution!

A dynamic twist of symbolic execution: execute the program over an input and build expressions along the path, negating branch conditions to generate new inputs



Pros:

- driven by one input: no need for a solver to go on in the exploration
- exploit concrete state when hard to reason symbolically

Cons

- for each input, rebuild expressions [recent works significantly reduced this cost, see, e.g., SymCC, Fuzzolic, SymQEMU, SymSan]

Observations

$$\pi_i \wedge \neg b_i$$

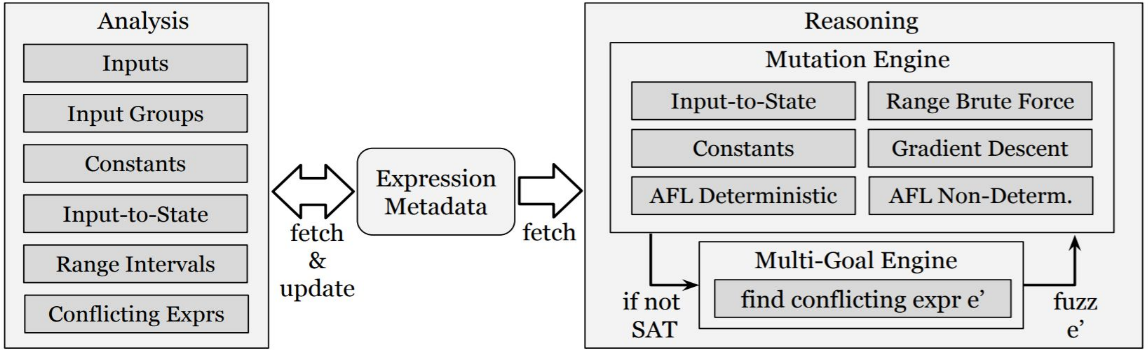
1. π_i is satisfied by the input I that has induced the execution

2. to learn how to mutate input I, we should look at $\neg b_i$

3. if we change some bytes in input I, then π_i may become unsatisfied. Hence we should do it carefully… but fuzzing is often lucky… we may expect to be lucky as well.

Fuzzy-SAT: Learn and Mutate

Given a branch query $\neg b \wedge \pi$ and the input I, mutate the bytes of the input trying to solve $\neg b$ while keeping π satisfiable.

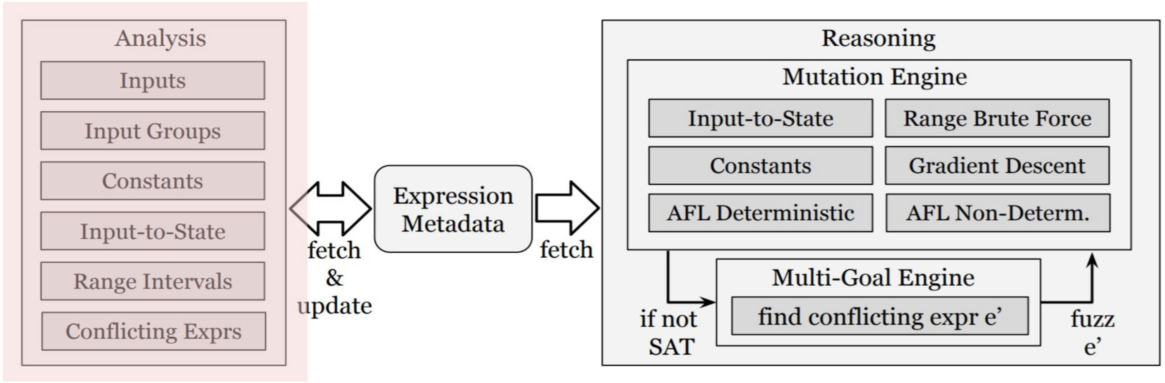


Fuzzy-SAT Architecture

Fuzzy-SAT: Learn and Mutate

Given a branch query $\neg b \wedge \pi$ and the input I, mutate the bytes of the input trying to solve $\neg b$ while keeping π satisfiable. Two stages:

- **Analysis**: learn from the symbolic expressions added to π

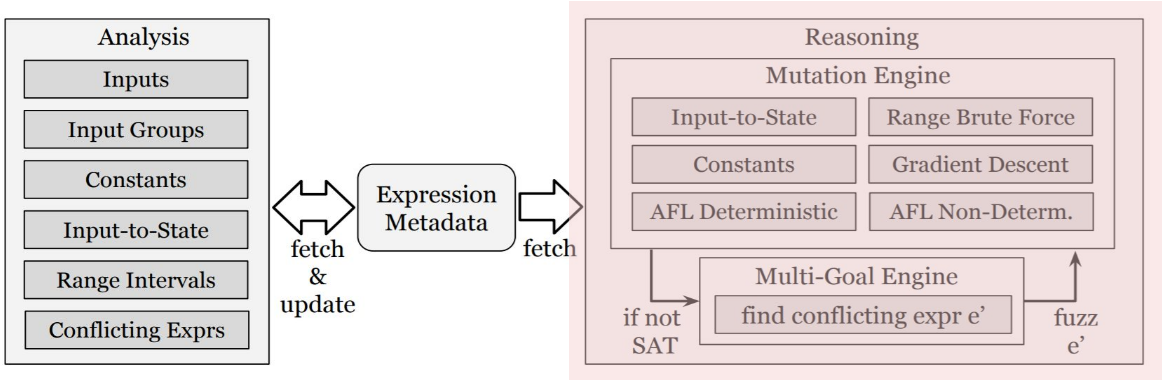


Fuzzy-SAT Architecture

Fuzzy-SAT: Learn and Mutate

Given a branch query $\neg b \wedge \pi$ and the input I, mutate the bytes of the input trying to solve $\neg b$ while keeping π satisfiable. Two stages:

- **Analysis**: learn from the symbolic expressions added to π
- **Reasoning**: use the acquired knowledge to apply simple but fast mutations to the input



Fuzzy-SAT Architecture

Analysis Stage (simplified)

Learn from the constraints added to π

- Detect *input groups*
- Detect *constants*
- Detect *ITS expressions*
- Detect *range constraints*

Analysis Stage (simplified)

Learn from the constraints added to π

Input Group: input symbols that are used together in the expression, and that never mix their bits

- Detect *input groups*
- Detect *constants*
- Detect *ITS expressions*
- Detect *range constraints*

$$\boxed{i_0 \mathbin{+\!\!+} i_1} + 10 = 42$$

$$\boxed{i_0} - \boxed{i_1} = 0$$

Analysis Stage (simplified)

Learn from the constraints added to π

- Detect *input groups*
- Detect *constants*
- Detect *ITS expressions*
- Detect *range constraints*

Collect the constants within the symbolic expression

$$i_0 \cdot \boxed{10} = \boxed{30}$$

$$i_0 \mathbin{+\!\!+} i_1 + \boxed{10} = \boxed{42}$$

Analysis Stage (simplified)

Learn from the constraints added to π

Detect expressions that contains an *input-to-state* [1] relation, i.e., a comparison of an input group with raw bytes

- Detect *input groups*
- Detect *constants*
- Detect *ITS expressions*
- Detect *range constraints*

$$(i_0 << 8) | i_1 = 42$$

$$i_0 + i_1 > 1$$

[1] C. Aschermann et al. "*REDQUEEN: fuzzing with input-to-state correspondence*". NDSS 2019

Analysis Stage (simplified)

Learn from the constraints added to π

- Detect *input groups*
- Detect *constants*
- Detect *ITS expressions*
- Detect *range constraints*

Detect patterns where a constraint sets an upper or lower bound to an input group:

$$(i_0 \mathbin{+\!\!+} i_1) + 0\mathrm{xAAAA} <_{unsigned} 0\mathrm{xBBBB}$$

$$\downarrow$$

$$i_0 \mathbin{+\!\!+} i_1 \in [0, 0\mathrm{x}1110] \cup [0\mathrm{x}5556, 0\mathrm{xFFFF}]$$

Reasoning Stage (simplified)

Mutate the bytes of the seed, trying to keep π *satisfiable*

- Mutation Engine
  - Input-to-State
  - Range brute-force
  - Gradient descent
  - AFL det. and non-det.
- Multi-Goal Engine

Reasoning Stage (simplified)

Mutate the bytes of the seed, trying to keep π *satisfiable*

- Mutation Engine
  - Input-to-State
  - Range brute-force
  - Gradient descent
  - AFL det. and non-det.
- Multi-Goal Engine

If the query has been tagged as input-to-state by the analysis stage, substitute the raw bytes in the input group

$$(i_0 << 8)|i_1 = 42$$

$$\downarrow$$

$$i_0 \leftarrow 0$$
$$i_1 \leftarrow 42$$

Reasoning Stage (simplified)

Mutate the bytes of the seed, trying to keep π *satisfiable*

- Mutation Engine
  - Input-to-State
  - Range brute-force
  - Gradient descent
  - AFL det. and non-det.
- Multi-Goal Engine

If an expression contains only *one* input group that has a *small* interval associated with it, brute force all the possible values

$i_0 \mathbin{+\!\!\!+} i_1 \in [0, 9]$ Result of *range analysis*

$i_0 \mathbin{+\!\!\!+} i_1 + 0\mathrm{xABAD} = 0\mathrm{xCAFE}$ Query

Reasoning Stage (simplified)

Mutate the bytes of the seed, trying to keep π *satisfiable*

- Mutation Engine
  - Input-to-State
  - Range brute-force
  - Gradient descent
  - AFL det. and non-det.
- Multi-Goal Engine

Reduce the query to a minimization problem, and use gradient descent to solve it

$$(i_0 + i_1) - 10 > (i_2 + i_3) + 5$$

$$\Big\downarrow{}^{1}$$

$$(i_2 + i_3) + 5 - ((i_0 + i_1) - 10) < 0$$

[1] The implementation also takes into account the wrap around!

Reasoning Stage (simplified)

Mutate the bytes of the seed, trying to keep π *satisfiable*

- Mutation Engine
  - Input-to-State
  - Range brute-force
  - Gradient descent
  - AFL det. and non-det.
- Multi-Goal Engine

Apply deterministic and non-deterministic transformations inspired by two mutation stages of AFL

- AFL transformations include bit-flips, addition and subtraction with small constants, etc.
- Differently from AFL:
  - our mutations are applied only to the bytes involved in the branch condition
  - multi-byte mutations are considered only in the presence of multi-byte input groups

Multi-Goal Engine (simplified)

Looking only at the branch condition is too restrictive:

```c
int f(char i1, char i2) {
  assert ( i1 == i2 );
  if ( i2 == 1 )
    return 0;
  return 1;
}
```

$\text{input} \leftarrow \{i_1 = 0, i_2 = 0\}$

$\pi \leftarrow i_1 = i_2$

$\neg b \leftarrow i_2 = 1$

$\pi \wedge \neg b$ *cannot* be solved by mutating only i_2

Multi-Goal Engine (simplified)

Looking only at the branch condition is too restrictive:

```
int f(char i1, char i2) {
  assert ( i1 == i2 );
  if ( i2 == 1 )
    return 0;
  return 1;
}
```

$$\text{input} \leftarrow \{i_1 = 0, i_2 = 0\}$$
$$\pi \leftarrow i_1 = I_2$$
$$\neg b \leftarrow i_2 = 1$$

$\pi \wedge \neg b$ *cannot* be solved by mutating only i_2

The **multi-goal engine** employs a *greedy approach* to solve this problem. Assuming that the reasoning engine solved $\neg b$:

- It checks whether $\neg b$ has *conflicting constraints* in π
- It tries to *solve* the conflicting constraints without modifying the bytes involved in $\neg b$

Implementation

Fuzzy-SAT:

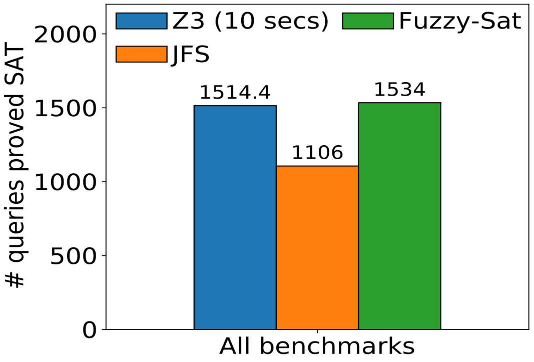
- C library
- Operates on Z3 expressions
- Integration in:
  - QSYM
  - Fuzzolic
  - SymQEMU
  - SymCC

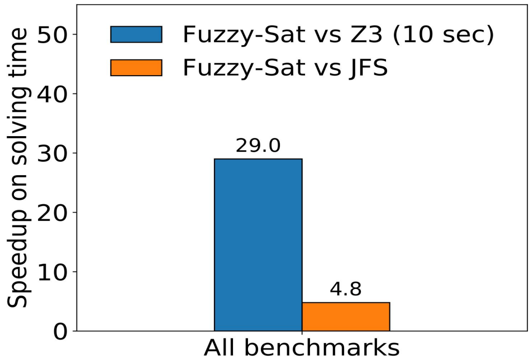
Fuzzy-SAT is available at
https://season-lab.github.io/fuzzolic/

Can Fuzzy-SAT actually solve queries generated by concolic executors?

Evaluation #1: Fuzzy-SAT vs Z3 vs JFS

Queries collected with QSYM using 12 real-world programs.





- **Fuzzy-SAT vs Z3**: Fuzzy-SAT may solve some queries that are not solved within the timeout by Z3. But is true also the opposite! Moreover, Fuzzy-SAT is designed to “fail fast” on too complex queries.

- **Fuzzy-SAT vs JFS**: Fuzzy-SAT may solve more queries than JFS likely due to the knowledge acquired during the analysis stage.

- Overall, Fuzzy-SAT was able to solve more queries than Z3/JFS, while also being faster.

Does Fuzzy-SAT only generate inputs that are already produced by traditional fuzzers?
Does it actually make the difference?

Evaluation #2: Code Coverage

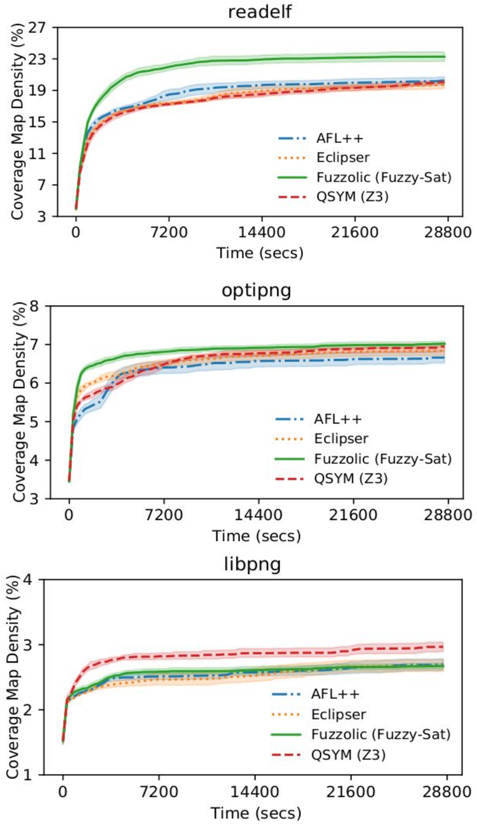
Fuzzolic (our concolic executor) with Fuzzy-SAT against:

- AFL++
- QSYM using Z3
- Eclipser

Concolic executors (Fuzzolic, QSYM) used in a hybrid fuzzing setup.

8H experiments, **Fuzzolic with Fuzzy-SAT** reached:

- an higher coverage in 7/12 programs
- a comparable coverage in 4/12 programs
- a lower coverage in 1/12 programs



Concluding remarks

Limitations (future directions?):

- It does not support the theory of arrays [ABV] and floating points.
- Not yet a clue on how to alternate effectively Fuzzy-SAT and Z3.

What we have learned:

- There are reasons why fuzzing is effective.
- Building symbolic expressions is expensive but we can learn a lot from them.
- JFS and Fuzzy-SAT are just first steps. Another recent step: JIGSAW @ IEEE SP 22

Thank you

I am open to research collaborations!

Let us have a chat if you wish :)