Hybrid Fuzzing

Luca Borzacchiello

$ whoami

- Participant of 1st CyberChallenge.it (2017)
- 3rd year Ph.D. Student @ Sapienza
- Research interests:
  - Program Analysis
  - Symbolic Execution
  - Fuzzing

Outline

1. Symbolic Execution
2. Concolic Execution
3. Fuzzing and Hybrid Fuzzing
4. Fuzzolic

Symbolic Execution

Symbolic Execution

Program analysis technique pioneered by J. C. King in **1976**

Programming Languages — B. Wegbreit, Editor

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

1. Introduction

The large-scale production of reliable programs is one of the fundamental requirements for applying computers to today's challenging problems. Several techniques are used in practice; others are the focus of current research. The work reported in this paper is directed at assuring that a program meets its requirements even when formal specifications are not given. The current technology in this area is basically a testing technology. That is, some small sample of the data that a program is expected to handle is presented to the program. If the program is judged to produce correct results for the sample, it is assumed to be correct. Much current work [11] focuses on the question of how to choose this sample.

Symbolic Execution

Program analysis technique pioneered by J. C. King in **1976**

Programming Languages — B. Wegbreit, Editor

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

1. Introduction

The large-scale production of reliable programs is one of the fundamental requirements for applying computers to today's challenging problems. Several techniques are used in practice; others are the focus of current research. The work reported in this paper is directed at assuring that a program meets its requirements even when formal specifications are not given. The current technology in this area is basically a testing technology. That is, some small sample of the data that a program is expected to handle is presented to the program. If the program is judged to produce correct results for the sample, it is assumed to be correct. Much current work [11] focuses on the question of how to choose this sample.

Key ideas:

- Each input in the program is associated with a symbol
- Each symbol represents a *set of values*
- Instructions in the program generates *expressions*

$$\text{int i} \quad \mapsto \quad \alpha_i$$

$$\alpha_i \in [0, 2^{32} - 1]$$

$$\text{i * 2 + 5} \quad \mapsto \quad 2 \cdot \alpha_i + 5$$

Symbolic Execution - Example Straight-line

```
int32_t foo(int32_t a) {
  a = a + 10;
  a = a * 2;
  return a;
}
```

Symbolic Execution - Example Straight-line

$$a \leftarrow \alpha_a$$

```
int32_t foo(int32_t a) {
  a = a + 10;
  a = a * 2;
  return a;
}
```

Symbolic Execution - Example Straight-line

```c
int32_t foo(int32_t a) {
  a = a + 10;
  a = a * 2;
  return a;
}
```

$$
\begin{tikzcd}
a \leftarrow \alpha_a \arrow[d] \\
a \leftarrow \alpha_a + 10
\end{tikzcd}
$$

Symbolic Execution - Example Straight-line

```
int32_t foo(int32_t a) {
   a = a + 10;
   a = a * 2;
   return a;
}
```

$$a \leftarrow \alpha_a$$

$$\downarrow$$

$$a \leftarrow \alpha_a + 10$$

$$\downarrow$$

$$a \leftarrow (\alpha_a + 10) * 2$$

Symbolic Execution - Example Straight-line

```c
int32_t foo(int32_t a) {
  a = a + 10;
  a = a * 2;
  return a;
}
```

$$a \leftarrow \alpha_a$$

$$\downarrow$$

$$a \leftarrow \alpha_a + 10$$

$$\downarrow$$

$$a \leftarrow (\alpha_a + 10) * 2$$

Easy! No branches!

Symbolic Execution - Example Branch

What happens at branches?

Symbolic Execution - Example Branch

What happens at branches?

```
int32_t abs(int32_t a) {
  if (a > 0)
    return a;
  return -a;
}
```

Symbolic Execution - Example Branch

What happens at branches?

$$a \leftarrow \alpha_a$$

```
int32_t abs(int32_t a) {
  if (a > 0)
    return a;
  return -a;
}
```

Symbolic Execution - Example Branch

What happens at branches?

```
int32_t abs(int32_t a) {
  if (a > 0)
    return a;
  return -a;
}
```

$$a \leftarrow \alpha_a$$

$$a \leftarrow \alpha_a \qquad \pi \leftarrow \alpha_a > 0$$

$$a \leftarrow \alpha_a \qquad \pi \leftarrow \alpha_a \leq 0$$

The execution state is *split* in two states that models either outcome of the branch

Symbolic Execution - Example Branch

What happens at branches?

```
int32_t abs(int32_t a) {
  if (a > 0)
    return a;
  return -a;
}
```

$$a \leftarrow \alpha_a$$

$$a \leftarrow \alpha_a \qquad\qquad a \leftarrow \alpha_a$$

$$\boxed{\pi \leftarrow \alpha_a > 0} \qquad \boxed{\pi \leftarrow \alpha_a \leq 0}$$

The execution state is *split* in two states that models either outcome of the branch

Each state has a *path constraint* that defines its condition of validity

Symbolic Execution - Symbolic Formulas

Now What?

Symbolic Execution - Symbolic Formulas

Now What?

We can use an **SMT Solver** to *reason* on the generated formulas: e.g., we can check whether the dividend of a division can be zero

```
[...]
return a / b;
```

$$
\begin{aligned}
a &\leftarrow \alpha_a / 7 \\
b &\leftarrow \alpha_a + \alpha_b \\
\pi &\leftarrow \alpha_a * \alpha_b > 1
\end{aligned}
$$

Is satisfiable?

$\alpha_a + \alpha_b = 0 \land \pi$ → SMT Solver

32-bit bit-vectors

Symbolic Execution - Symbolic Formulas

Now What?

We can use an **SMT Solver** to *reason* on the generated formulas: e.g., we can check whether the dividend of a division can be zero

```
[...]
return a / b;
```

$$a \leftarrow \alpha_a/7$$
$$b \leftarrow \alpha_a + \alpha_b$$
$$\pi \leftarrow \alpha_a * \alpha_b > 1$$

Is satisfiable?

$$\alpha_a + \alpha_b = 0 \land \pi$$

32-bit bit-vectors

SMT Solver

yes!

$$\alpha_a = \texttt{0xfffd4000}$$
$$\alpha_b = \texttt{0x2c000}$$

Symbolic Execution - Example

```
1.   void foobar(int a, int b) {
2.       int x = 1, y = 0;
3.       if (a != 0) {
4.           y = 3+x;
5.           if (b == 0)
6.               x = 2*(a+b);
7.       }
8.       assert(x-y != 0);
9.   }
```

Symbolic Execution can find **all** the inputs that makes the assertion at line 8 fail

image credit

Symbolic Execution - Final Remarks

In general, *pure static* symbolic execution hardly scales on real-world programs:

- Path explosion
- Hard-to-solve constraints
- Symbolic memory accesses
- Emulation time
- …

Static symbolic executors:

Concolic Execution

Concolic Execution

Dynamic flavor of symbolic execution (**conc**rete + symb**olic**)

Key idea:

- Choose a concrete path (i.e., an input for the program)
- Run concretely the program
- Build symbolic expressions *alongside* concrete values
- Check *conditions* on the chosen path

Concolic Execution - Example

```
int32_t foo(int32_t a) {
  a = a + 1;
  if (a > 5)
    if (a < 8)
      a = a / 2;
  return a;
}
```

Concolic Execution - Example

$$a \leftarrow 20$$ concrete value (seed)

$$a \leftarrow \alpha_a$$

$$\pi \leftarrow \top$$

```
int32_t foo(int32_t a) {
   a = a + 1;
   if (a > 5)
      if (a < 8)
         a = a / 2;
   return a;
}
```

Concolic Execution - Example

```
int32_t foo(int32_t a) {
  a = a + 1;
  if (a > 5)
    if (a < 8)
      a = a / 2;
  return a;
}
```

$$\begin{array}{l} a \leftarrow 20 \\ a \leftarrow \alpha_a \\ \pi \leftarrow \top \end{array}$$

$\downarrow$

$$\begin{array}{l} a \leftarrow 21 \\ a \leftarrow \alpha_a + 1 \\ \pi \leftarrow \top \end{array}$$

Concolic Execution - Example

```
int32_t foo(int32_t a) {
  a = a + 1;
  if (a > 5)
    if (a < 8)
      a = a / 2;
  return a;
}
```

$$
\begin{array}{l}
a \leftarrow 20 \\
a \leftarrow \alpha_a \\
\pi \leftarrow \top
\end{array}
$$

$$
\begin{array}{l}
a \leftarrow 21 \\
a \leftarrow \alpha_a + 1 \\
\pi \leftarrow \top
\end{array}
$$

$$
\begin{array}{l}
a \leftarrow 21 \\
a \leftarrow \alpha_a + 1 \\
\pi \leftarrow \alpha_a > 5
\end{array}
$$

Concolic Execution - Example

```
int32_t foo(int32_t a) {
  a = a + 1;
  if (a > 5)
    if (a < 8)
      a = a / 2;
  return a;
}
```

$$\begin{aligned} a &\leftarrow 20 \\ a &\leftarrow \alpha_a \\ \pi &\leftarrow \top \end{aligned}$$

$$\begin{aligned} a &\leftarrow 21 \\ a &\leftarrow \alpha_a + 1 \\ \pi &\leftarrow \top \end{aligned}$$

$$\boxed{\pi \leftarrow \alpha_a \leq 5}$$

$$\begin{aligned} a &\leftarrow 21 \\ a &\leftarrow \alpha_a + 1 \\ \pi &\leftarrow \alpha_a > 5 \end{aligned}$$

We can use the *negated* branch condition to generate a new input that covers the other outcome of the branch

Concolic Execution - Example

```
int32_t foo(int32_t a) {
  a = a + 1;
  if (a > 5)
    if (a < 8)
      a = a / 2;
  return a;
}
```

$$a \leftarrow 20 \\ a \leftarrow \alpha_a \\ \pi \leftarrow \top$$

$$a \leftarrow 21 \\ a \leftarrow \alpha_a + 1 \\ \pi \leftarrow \top$$

$$\pi \leftarrow \alpha_a \leq 5$$

$$a \leftarrow 21 \\ a \leftarrow \alpha_a + 1 \\ \pi \leftarrow \alpha_a > 5$$

$$a \leftarrow 21 \\ a \leftarrow \alpha_a + 1 \\ \pi \leftarrow \alpha_a \geq 8 \wedge \alpha_a > 5$$

Concolic Execution - Example

```c
int32_t foo(int32_t a) {
  a = a + 1;
  if (a > 5)
    if (a < 8)
      a = a / 2;
  return a;
}
```

$$\begin{array}{l} a \leftarrow 20 \\ a \leftarrow \alpha_a \\ \pi \leftarrow \top \end{array}$$

$$\downarrow$$

$$\begin{array}{l} a \leftarrow 21 \\ a \leftarrow \alpha_a + 1 \\ \pi \leftarrow \top \end{array}$$

$$\searrow$$

$$\pi \leftarrow \alpha_a \leq 5$$

$$\begin{array}{l} a \leftarrow 21 \\ a \leftarrow \alpha_a + 1 \\ \pi \leftarrow \alpha_a > 5 \end{array}$$

$$\downarrow$$

$$\begin{array}{l} a \leftarrow 21 \\ a \leftarrow \alpha_a + 1 \\ \pi \leftarrow \alpha_a \geq 8 \wedge \alpha_a > 5 \end{array}$$

$$\boxed{\pi \leftarrow \alpha_a < 8 \wedge \alpha_a > 5}$$

Concolic Execution - Final Remarks

Pros wrt *static* symbolic execution

- No need to call the solver at branches if we are not interested in generating a new input: less queries
- The implementation can be *very* fast if we compile the instrumentation (e.g., using LLVM, PIN, QEMU)

Cons wrt *static* symbolic execution

- To explore a branch that is not taken by the seed, you need to re-execute from the beginning

NOTE: path explosion is still an issue!

Hybrid Fuzzing

Fuzzing

- State-of-the-art technique for automatic test-case generation and vulnerability detection
- Based on random mutations of a *pool* of inputs with the goal of finding *crashes* in the target application

Fuzzing

- State-of-the-art technique for automatic test-case generation and vulnerability detection
- Based on random mutations of a *pool* of inputs with the goal of finding *crashes* in the target application

very effective in practice

Fuzzing

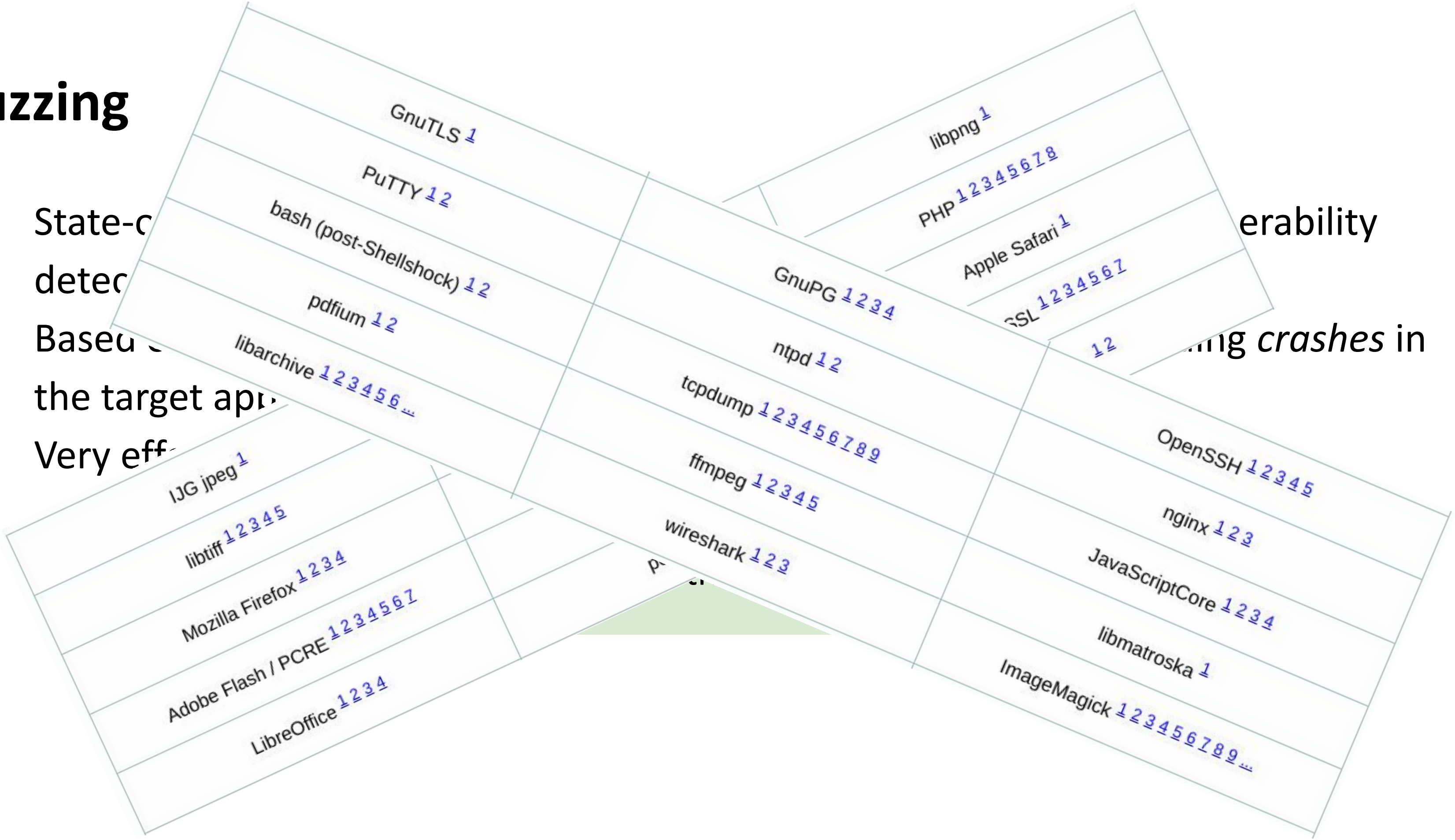
- State-of-the-art technique for autom… erability detection
- Based on random mut… ing *crashes* in the target appl…
- Very eff…

…tive in practice

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

Fuzzing

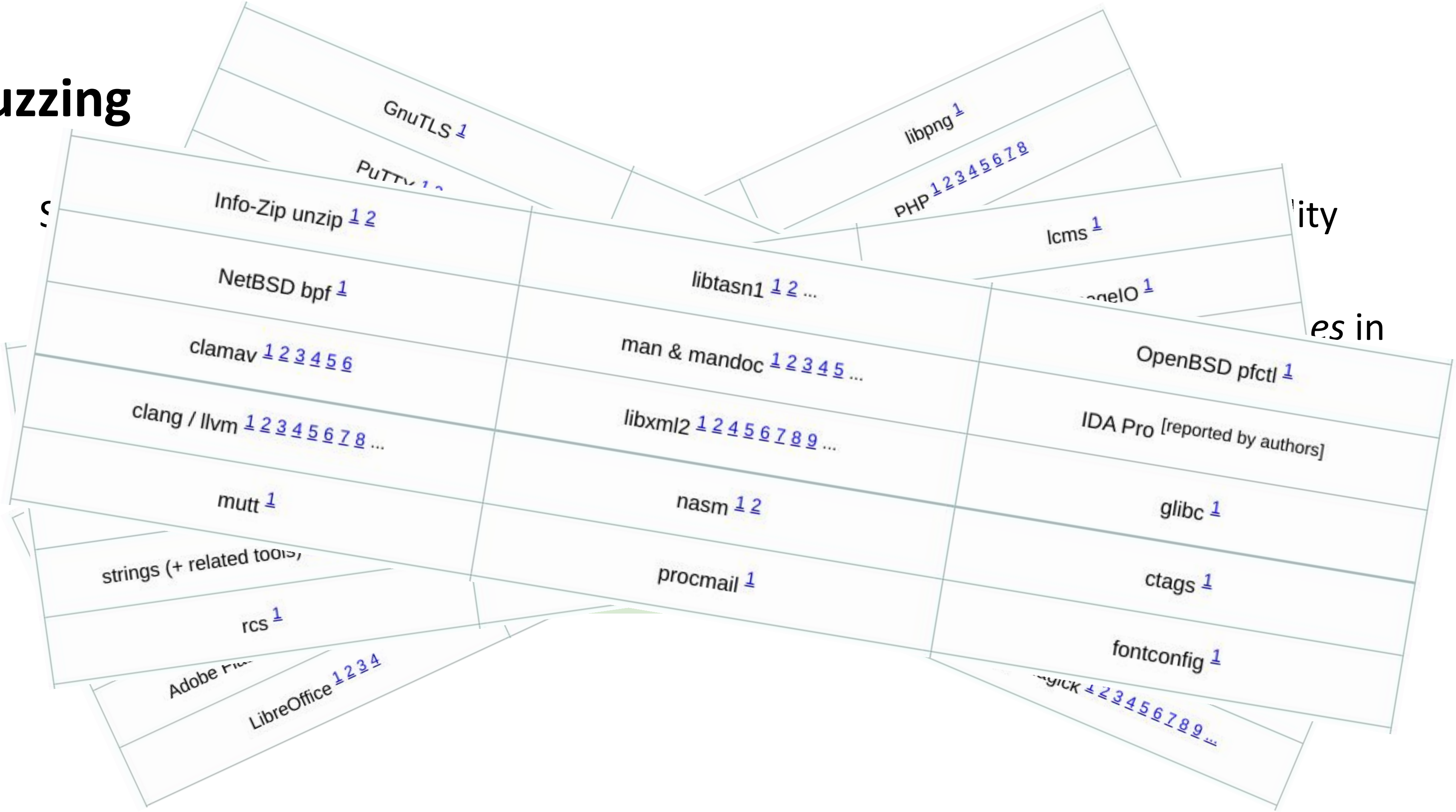
- State-c … erability detec…
- Based … ng *crashes* in the target app…
- Very eff…



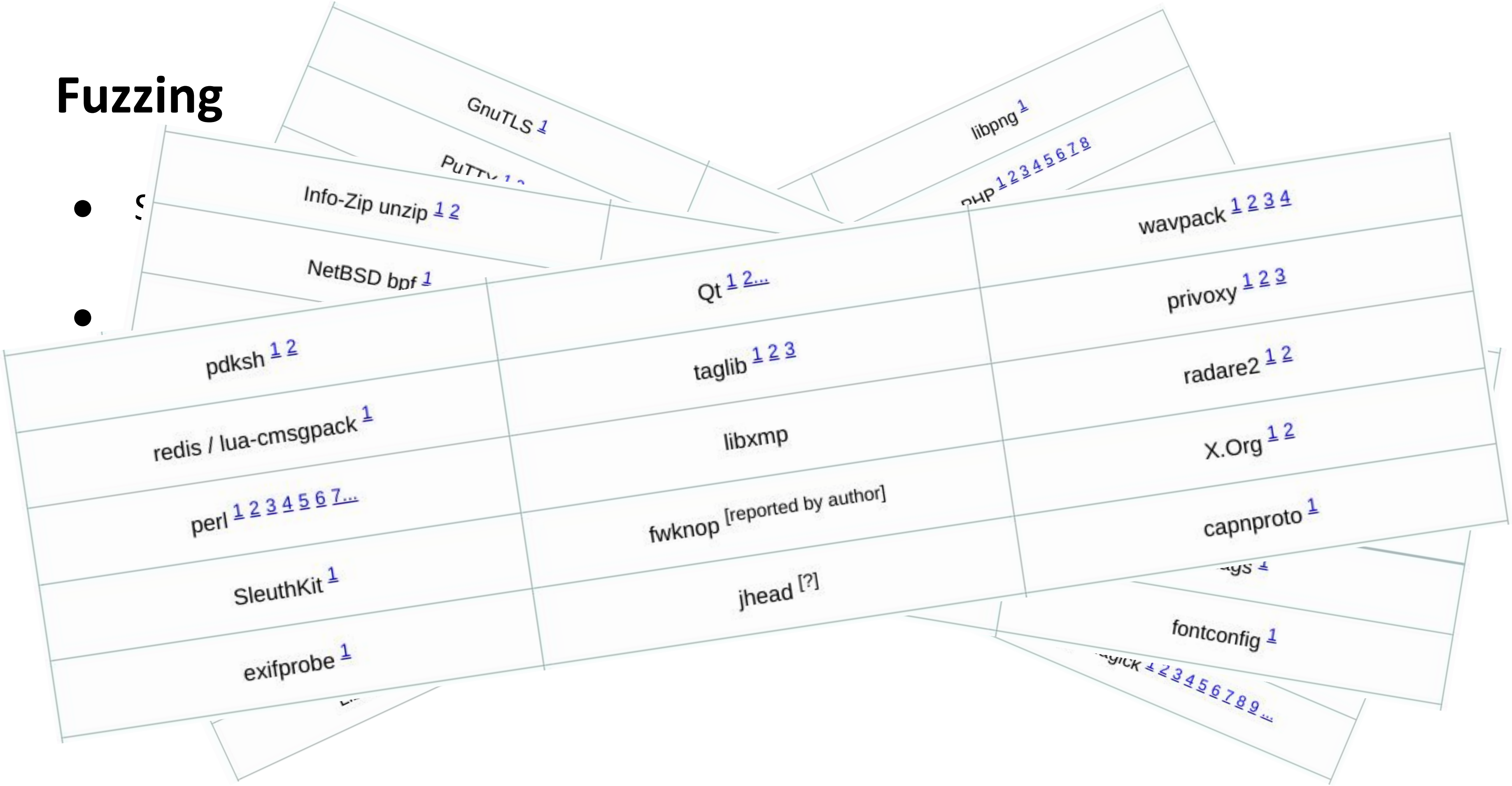
Fuzzing

- State-c ... ity detec
- Base ... *es* in
-

Fuzzing


GnuTLS 1
libpng 1
PHP 1 2 3 4 5 6 7 8
lcms 1
Info-Zip unzip 1 2
NetBSD bpf 1
clamav 1 2 3 4 5 6
clang / llvm 1 2 3 4 5 6 7 8 ...
mutt 1
libtasn1 1 2 ...
man & mandoc 1 2 3 4 5 ...
libxml2 1 2 4 5 6 7 8 9 ...
nasm 1 2
procmail 1
OpenBSD pfctl 1
IDA Pro [reported by authors]
glibc 1
ctags 1
fontconfig 1
strings (+ related tools)
rcs 1
LibreOffice 1 2 3 4

Fuzzing


GnuTLS 1
libpng 1
PHP 1 2 3 4 5 6 7 8
Info-Zip unzip 1 2
NetBSD bpf 1
wavpack 1 2 3 4
Qt 1 2...
privoxy 1 2 3
pdksh 1 2
taglib 1 2 3
radare2 1 2
redis / lua-cmsgpack 1
libxmp
X.Org 1 2
perl 1 2 3 4 5 6 7...
fwknop [reported by author]
capnproto 1
SleuthKit 1
jhead [?]
fontconfig 1
exifprobe 1

Fuzzing

Still, it has some drawbacks:

- Struggles in overcoming checks against *magic numbers* or *checksums* (road-blocks)
- Typically needs a *reasonable* set of input seeds

Fuzzing

Still, it has some drawbacks:

- Struggles in overcoming checks against *magic numbers* or *checksums* (road-blocks)
- Typically needs a *reasonable* set of input seeds

Can we combine fuzzing and symbolic execution to take the best from the two worlds?

Fuzzing vs Symbolic Execution

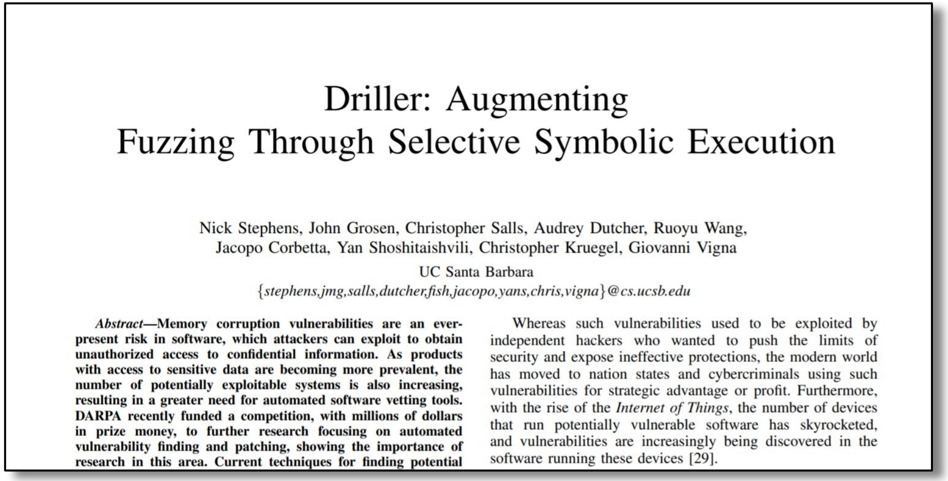
```
x = input()

def recurse(x, depth):
  if depth == 2000
    return 0
  else {
    r = 0;
    if x[depth] == "B":
      r = 1
    return r + recurse(x
[depth], depth)

if recurse(x, 0) == 1:
  print "You win!"
```

Fuzzing Wins

```
x = int(input())
if x >= 10:
    if x^2 == 152399025:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

Symbolic Execution Wins

Driller



Driller: Augmenting
Fuzzing Through Selective Symbolic Execution

Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang,
Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna
UC Santa Barbara
{stephens,jmg,salls,dutcher,fish,jacopo,yans,chris,vigna}@cs.ucsb.edu

***Abstract*—Memory corruption vulnerabilities are an ever-present risk in software, which attackers can exploit to obtain unauthorized access to confidential information. As products with access to sensitive data are becoming more prevalent, the number of potentially exploitable systems is also increasing, resulting in a greater need for automated software vetting tools. DARPA recently funded a competition, with millions of dollars in prize money, to further research focusing on automated vulnerability finding and patching, showing the importance of research in this area. Current techniques for finding potential**

Whereas such vulnerabilities used to be exploited by independent hackers who wanted to push the limits of security and expose ineffective protections, the modern world has moved to nation states and cybercriminals using such vulnerabilities for strategic advantage or profit. Furthermore, with the rise of the *Internet of Things*, the number of devices that run potentially vulnerable software has skyrocketed, and vulnerabilities are increasingly being discovered in the software running these devices [29].

First work (2016) on fuzzing improved with concolic execution

Key Ideas:

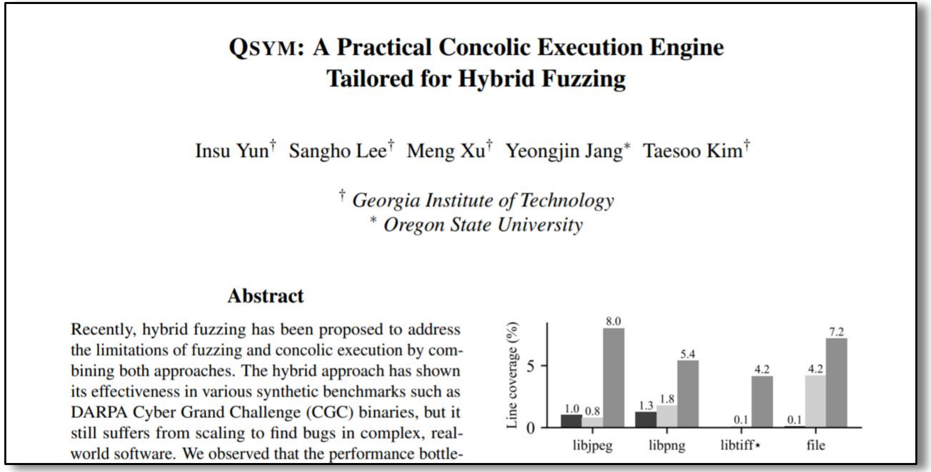
- Launch the fuzzer (AFL) for some time
- When the fuzzer *get stuck* (i.e., is unable to generate a new input), run the concolic executor and trying to generate a new input

Issues:

- The concolic executor is slow (interpreted, and written in python)
- The interaction between the fuzzer and the concolic executor is *coarse grained*

QSYM

Hybrid fuzzer and concolic executor based on Intel PIN (2018)



QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing

Insu Yun† Sangho Lee† Meng Xu† Yeongjin Jang* Taesoo Kim†

† *Georgia Institute of Technology*
* *Oregon State University*

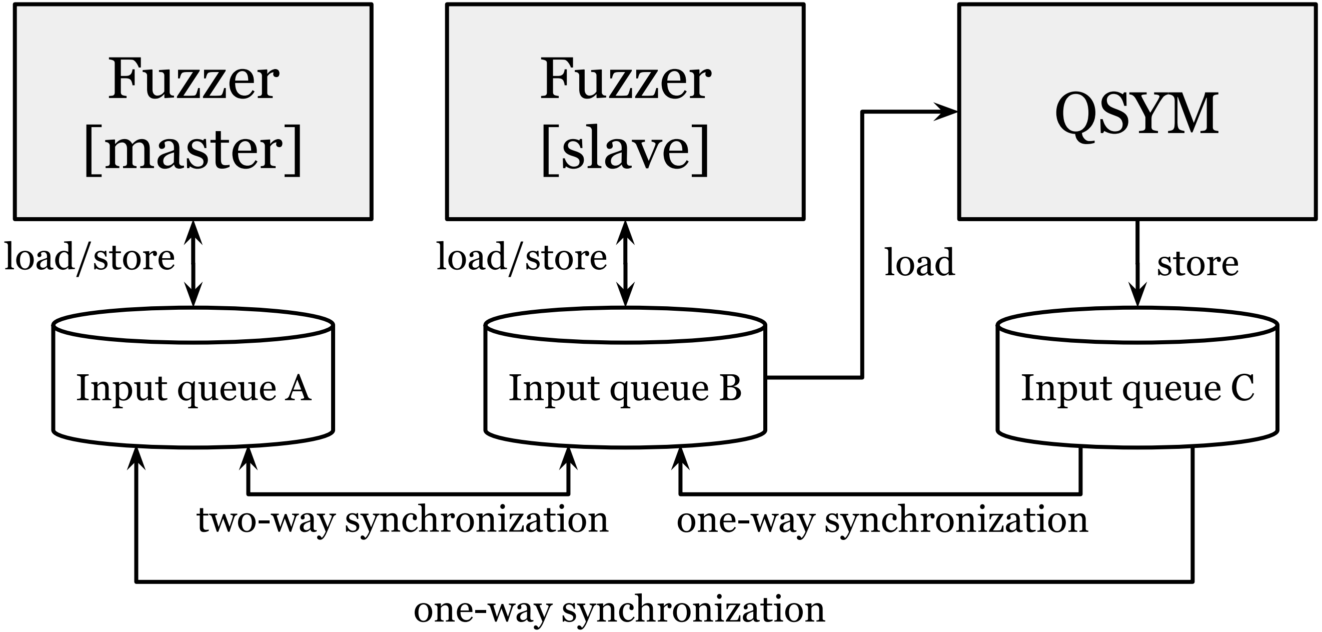
Abstract

Recently, hybrid fuzzing has been proposed to address the limitations of fuzzing and concolic execution by combining both approaches. The hybrid approach has shown its effectiveness in various synthetic benchmarks such as DARPA Cyber Grand Challenge (CGC) binaries, but it still suffers from scaling to find bugs in complex, real-world software. We observed that the performance bottle-

Key Ideas:

- JIT the instrumentation at runtime using PIN (no interpretation)
- The concolic executor runs in parallel with the fuzzer
- Optimizations: *linearization* and *optimistic solving*

QSYM - Parallel Setup



SymCC

Hybrid fuzzer and concolic executor based on LLVM (2020)

ARTIFACT EVALUATED usenix ASSOCIATION PASSED

Symbolic execution with SymCC: Don't interpret, compile!

Sebastian Poeplau
EURECOM

Aurélien Francillon
EURECOM

Abstract

A major impediment to practical symbolic execution is speed, especially when compared to near-native speed solutions like fuzz testing. We propose a compilation-based approach to symbolic execution that performs better than state-of-the-art implementations by orders of magnitude. We present SymCC, an LLVM-based C and C++ compiler that builds concolic execution right into the binary. It can be used by software

therefore fewer bugs detected per invested resources. Several challenges are commonly identified, one of which is slow code execution: Yun et al. have recently provided extensive evidence that the execution component is a major bottleneck in modern implementations of symbolic execution [45]. We propose an alternative execution method and show that it leads to considerably faster symbolic execution and ultimately to better program coverage and more bugs discovered.

Key Ideas:

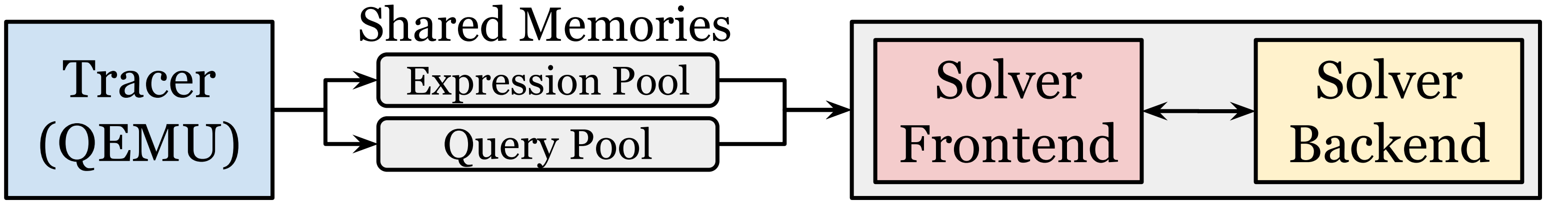
- Very fast instrumentation added at *compilation* time with an LLVM pass
- Parallel setup (as in QSYM)
- *Function models* to inject symbolic data
- Requires source code and recompilation of all components (or misses symbolic propagation)

Fuzzolic

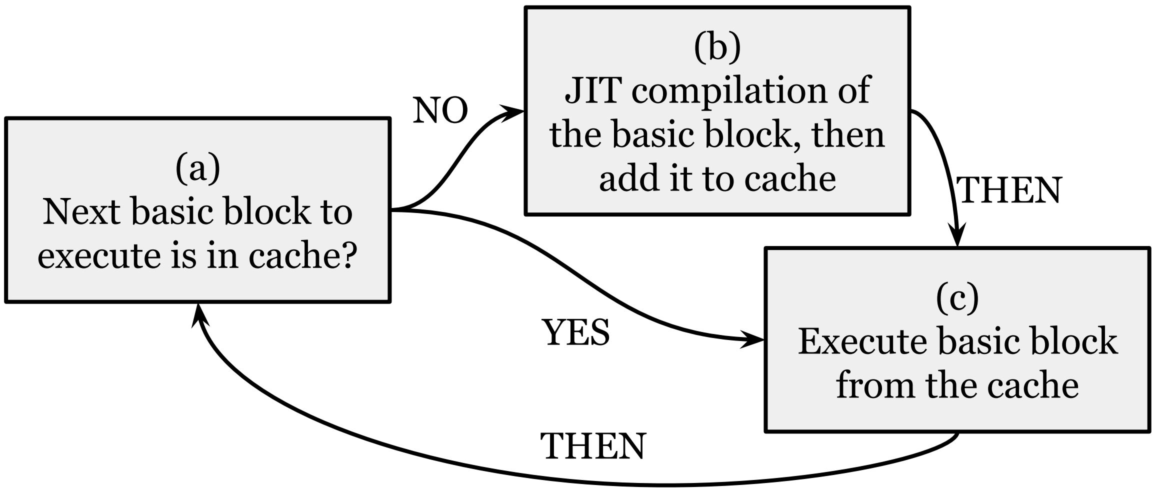
Fuzzolic

Fast concolic executor based on QEMU (JIT-ed instrumentation)

- Developed independently and in parallel with SymQEMU (another concolic executor based on QEMU)
- Same parallel setup of QSYM
- Two solver backends:
    - Z3 SMT solver
    - FuzzySAT: approximate solver



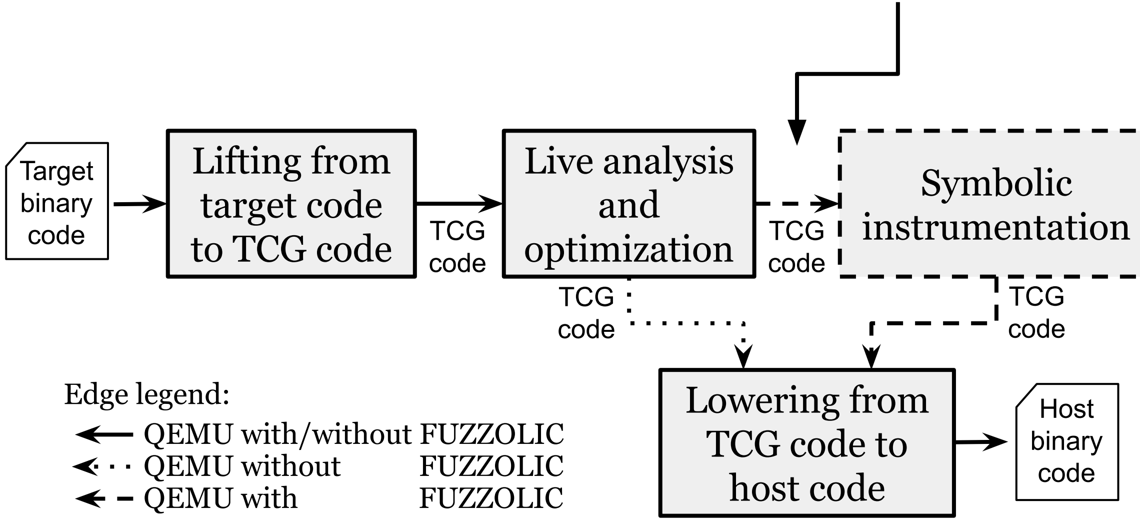
QEMU - JIT Workflow



Note: during JIT compilation, QEMU translate native code
to the intermediate language TCG

Fuzzolic - Instrumentation

Fuzzolic instruments the program in TCG



Fuzzolic - Instrumentation

Two instrumentation methods:

- *Inline instrumentation (simplified)*

```
mov rbx, rax
```

x64

Fuzzolic - Instrumentation

Two instrumentation methods:

- *Inline instrumentation (simplified)*

x64		tcg
`mov rbx, rax`	→	`mov_i64 tcg_reg_rbx, tcg_reg_rax`

Fuzzolic - Instrumentation

Two instrumentation methods:

- *Inline instrumentation (simplified)*

```
mov rbx, rax          →    mov_i64 tcg_reg_rbx, tcg_reg_rax
    x64                                     tcg
```

```
mov_i64 tcg_sym_reg_rbx, tcg_sym_reg_rax
mov_i64 tcg_reg_rbx, tcg_reg_rax
```

instrumented (inline)

Fuzzolic - Instrumentation

Two instrumentation methods:

- *Inline instrumentation (simplified)*

```
mov rbx, rax          →          mov_i64 tcg_reg_rbx, tcg_reg_rax
    x64                                          tcg
```

```
mov_i64 tcg_sym_reg_rbx, tcg_sym_reg_rax
mov_i64 tcg_reg_rbx, tcg_reg_rax
```

instrumented (inline)

- *Helper instrumentation (simplified)*

```
call fuzzolic_mov_reg, tcg_sym_reg_rax, tcg_sym_reg_rbx
mov_i64 tcg_reg_rbx, tcg_reg_rax
```

instrumented (helper)

Fuzzolic - TCG Helpers

Unfortunately, while TCG is architecture-independent, it uses many *helpers* for particular instructions

- x86 is full of those helpers (e.g., vector instructions)

- Fuzzolic has hard-written models for *some* of those helpers, but many are missing (e.g., floating point instructions)

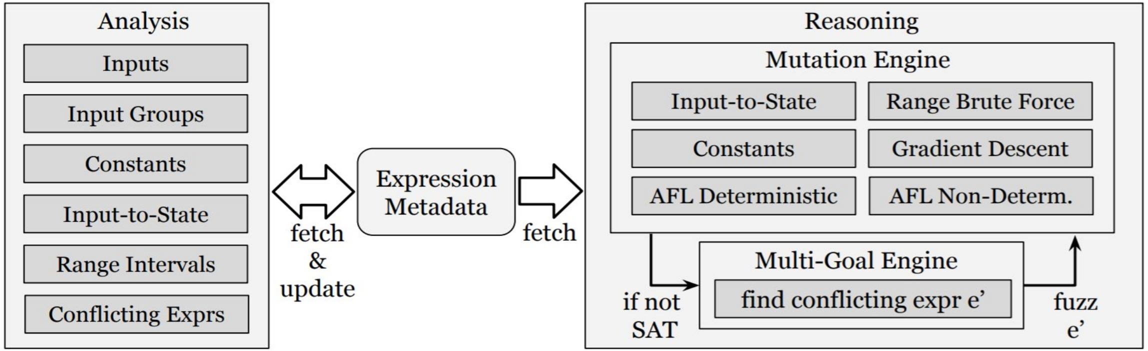
Fuzzy-SAT - An Approximate Solver

Key Ideas:

- Given a branch query of a concolic executor ($\neg b \wedge \pi$), the seed that has driven the concolic exploration satisfies by design π
- Fuzzing transformations has been proven effective in overcoming branch conditions
- While SMT solvers offer a rich set of solving primitives, most concolic executors (e.g., QSYM) are built on top of a few but essential primitives

Fuzzy-SAT - An Approximate Solver

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

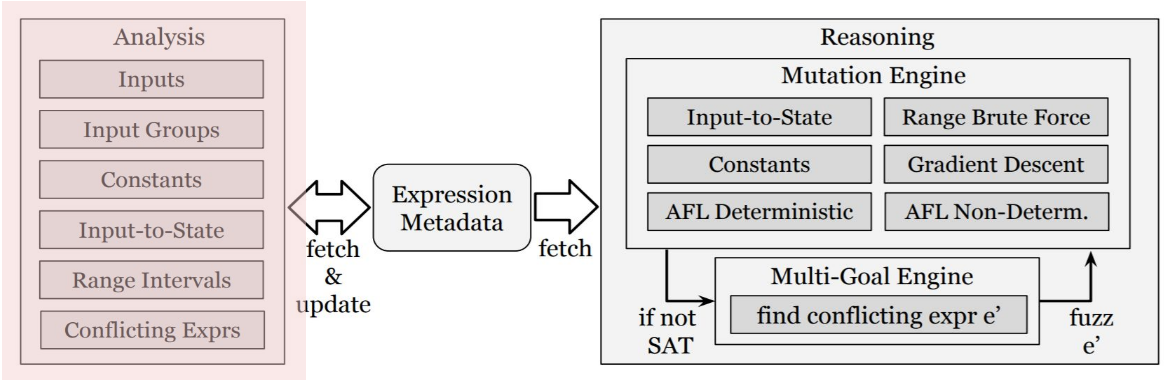


Fuzzy-SAT Architecture

Fuzzy-SAT - An Approximate Solver

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

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

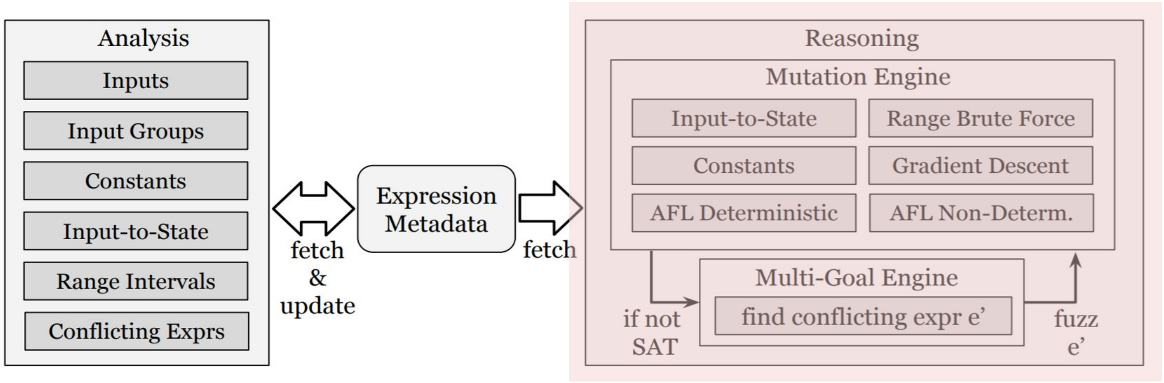


Fuzzy-SAT Architecture

Fuzzy-SAT - An Approximate Solver

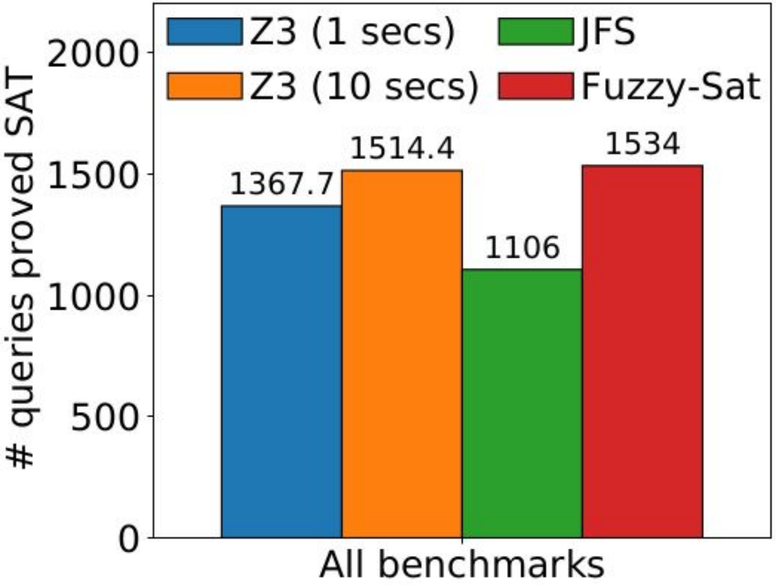
Given a branch query $\neg b \wedge \pi$ and the seed, mutate the bytes of the seed trying to solve $\neg b$ while keeping π satisfiable.

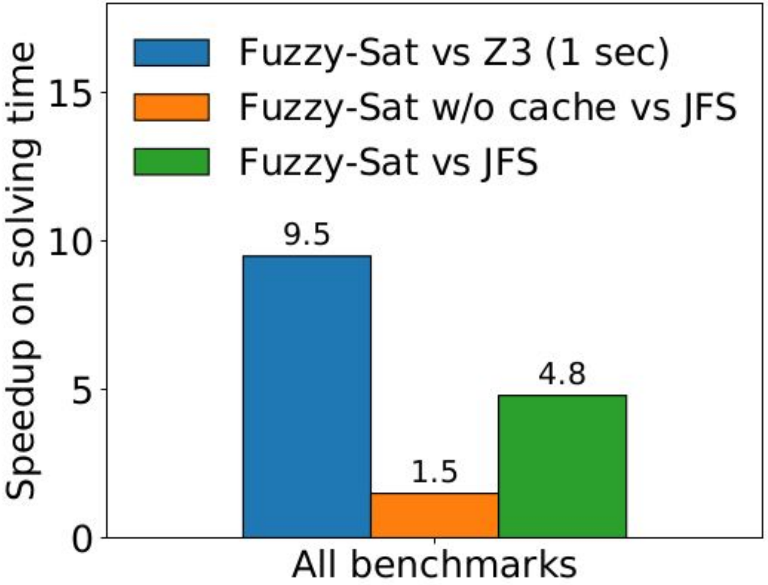
- **Analysis**: learn from the symbolic expressions added to π
- **Reasoning**: use the acquired knowledge to apply simple but fast transformations to the seed



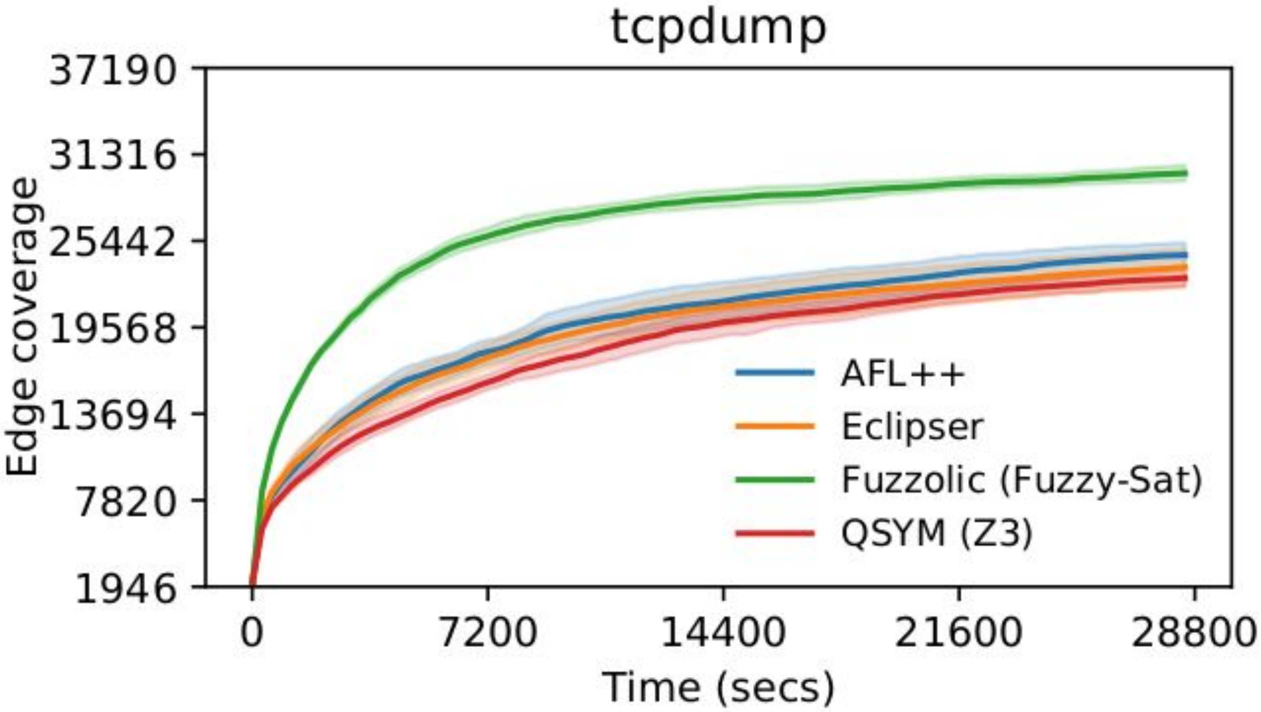
Fuzzy-SAT Architecture

Fuzzy-SAT - Performance





Fuzzolic - Performance



DEMO

Thanks!
Questions?