T-Fuzz: Fuzzing by Program Transformation

Hui Peng[1], Yan Shoshitaishvili[2], Mathias Payer[1]

[1] Purdue University

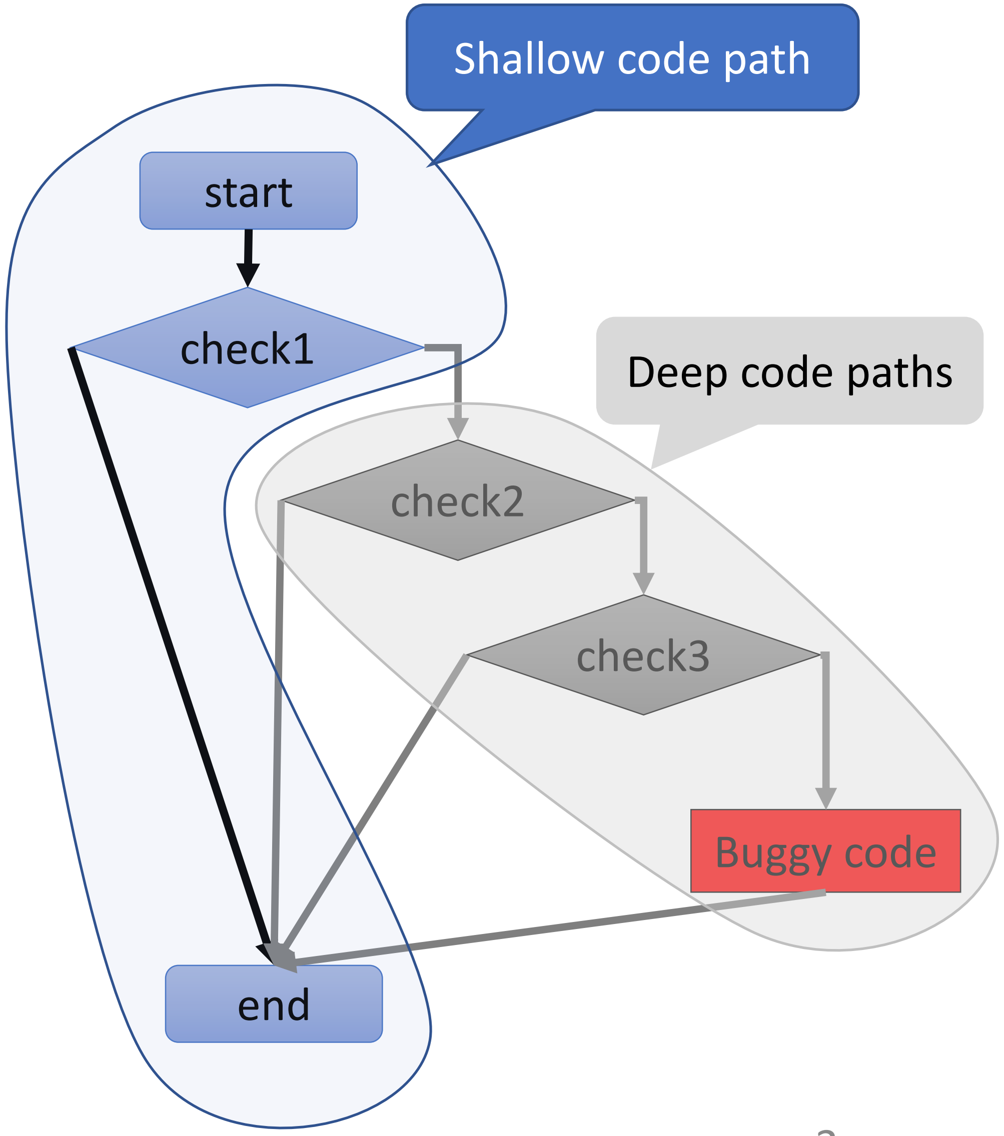
[2] Arizona State University

Fuzzing as a bug finding approach

- Fuzzing is finding more and more CVEs
- Vendors use it as proactive defense measure: OSS-Fuzz
- Hackers use it as first step in exploit development

Challenges for fuzzers

- Challenges
  - Shallow coverage
  - Hard to find “deep” bugs
- Root Cause
  - Fuzzer-generated inputs cannot bypass complex sanity checks in the target program



Existing approaches & their limitations

- Existing approaches focus on ***input generation***
  - Driller (concolic execution)
  - VUzzer (taint analysis, data & control flow analysis)
- Limitations
  - High overhead
  - Not scalable
  - Not able to bypass “hard” checks
    - Checks on checksum, crypto-hash values

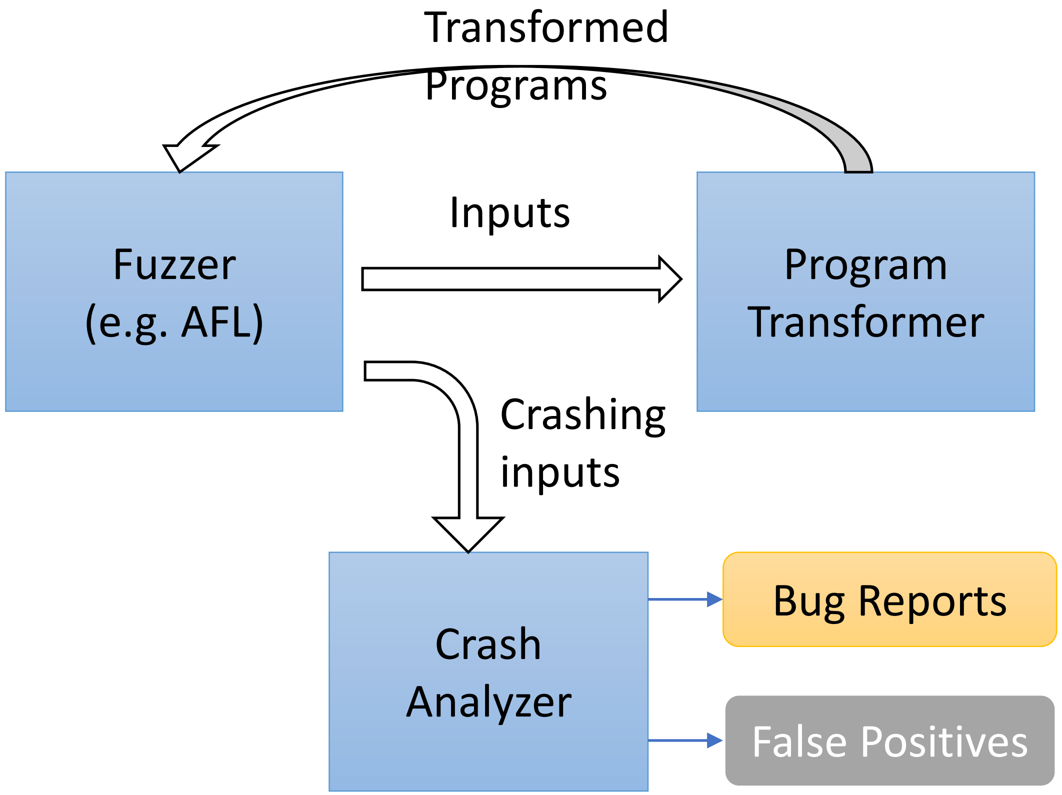
Insight: some checks are non-critical

- Some sanity checks are not intended to prevent bugs
- ***<u>Non-Critical Checks (NCC)</u>***
  - E.g., check on magic values, checksums, hashes
- Removing NCCs won't incur erroneous bugs
- Removal of NCCs simplifies fuzzing

```
void main() {
  int fd  = open(...);
  char *hdr = read_header(fd);
  if (strncmp(hdr, "ELF", 3) == 0) {
    // main program logic
    // ...
  } else {
    error();
  }
}
```

T-Fuzz: fuzzing by program transformation

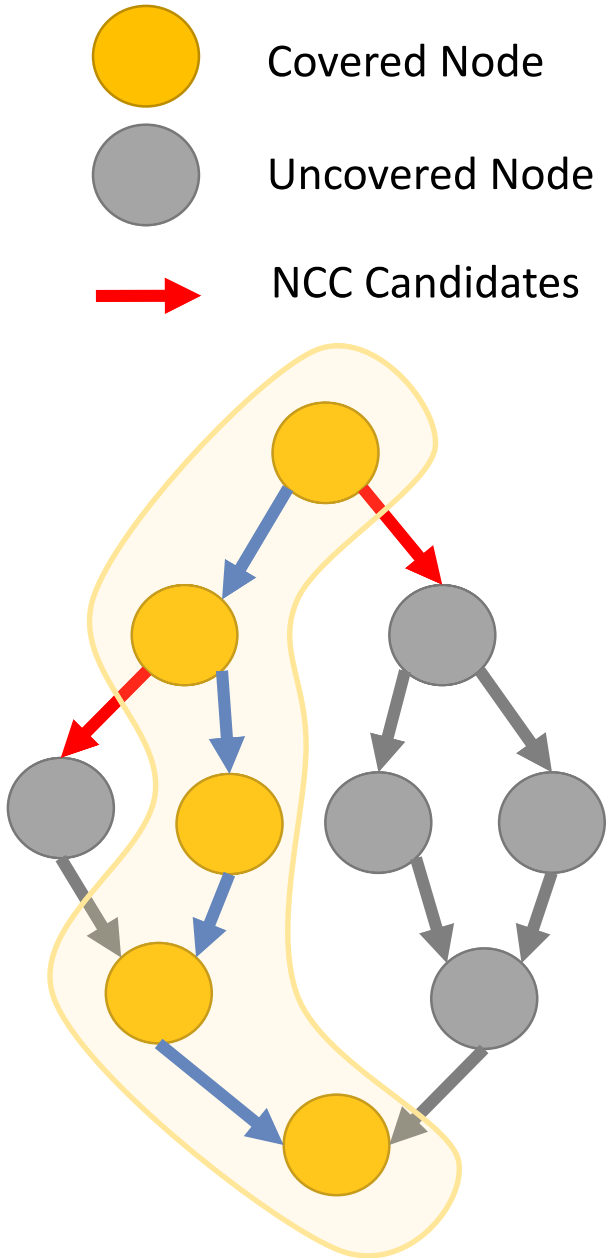
- Fuzzer generates inputs
- When Fuzzer gets stuck,
  Program Transformer:
  - Detects ***NCC candidates***
  - Transforms program
- Crash Analyzer verifies crashes
- Repeat



T-Fuzz design

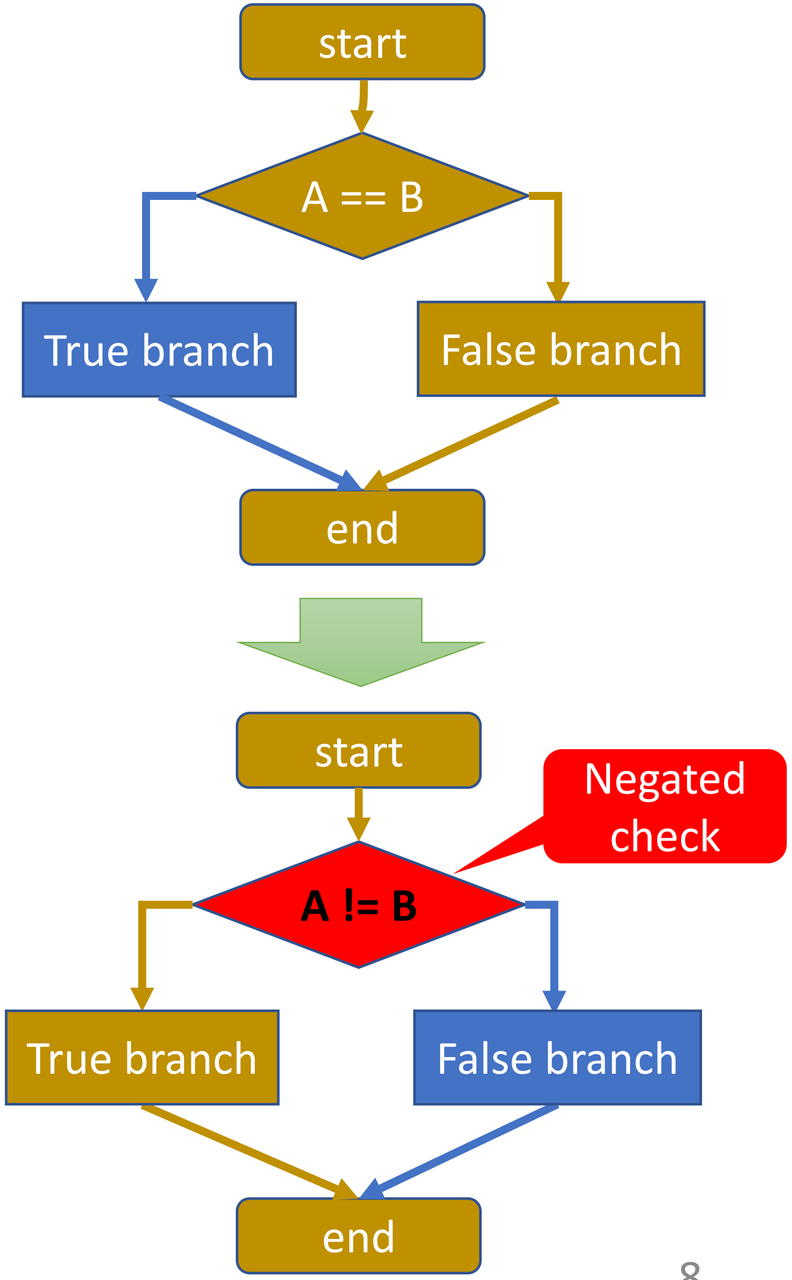
Detecting NCC candidates

- Approximate NCCs as the edges connecting covered/uncovered nodes in the CFG
- Overapproximate, *may contain false positive*
- Lightweight and simple to implement
  - dynamic tracing

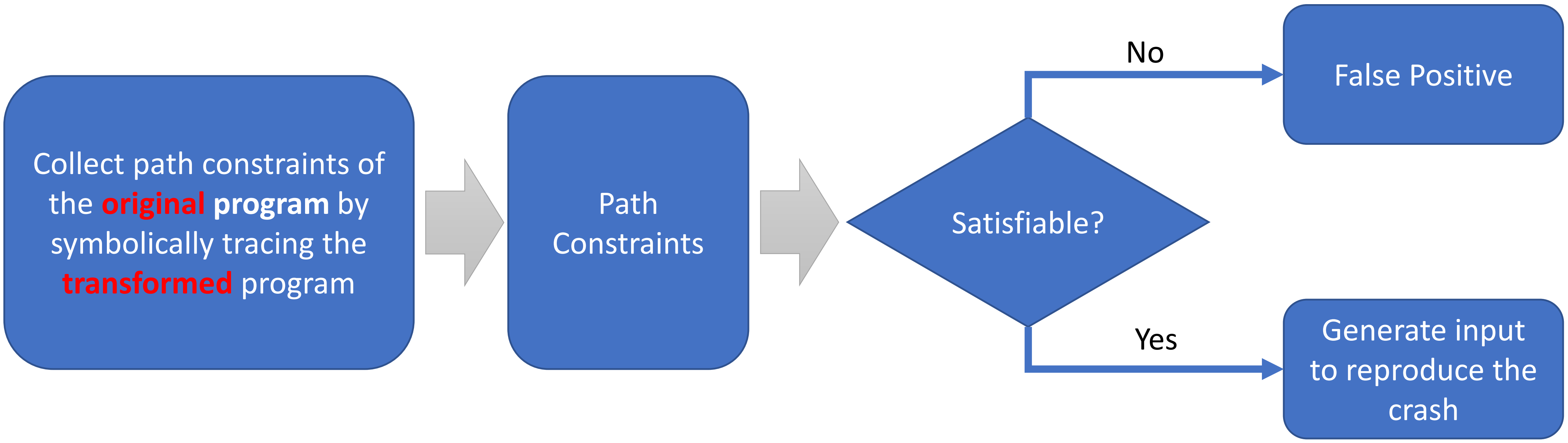


Program Transformation

- **Goal**: disable NCCs
- Our approach: **negate NCCs**
  - Easy to implement: static binary rewriting
  - Zero runtime overhead in target program
  - The CFG of the program stays the same
  - Traces of the transformed program map to the original one
  - Path constraints of the original program can be recovered



Filtering out false positives & reproducing bugs



Example 1

Collected path constraints:

{x > 0, y == 0xdeadbeef}

SAT

True BUG

Un-negating

```
int main (){
  int x = read_input();
  int y = read_input();
  if (x > 0) {
    if (y == 0xdeadbeef)
      bug();
  }
}
```

Original Program

```
int main (){
  int x = read_input();
  int y = read_input();
  if (x > 0) {
    if (y != 0xdeadbeef)
      bug();
  }
}
```

Transformed Program

Negated check

Example 2

UNSAT

False BUG

```c
int main (){
  int i = read_input();

  if (i > 0) {
    func(i);
  }
}

void func(int i) {
  if (i <= 0) {
    bug();
  }
  //...
}
```

Original Program

```c
int main (){
  int i = read_input();

  if (i > 0) {
    func(i);
  }
}

void func(int i) {
  if (i > 0) {
    bug();
  }
  //...
}
```

Transformed Program

path constraints:

{i > 0; i <= 0}

Un-negating

Negated check

Limitations of T-Fuzz (1)

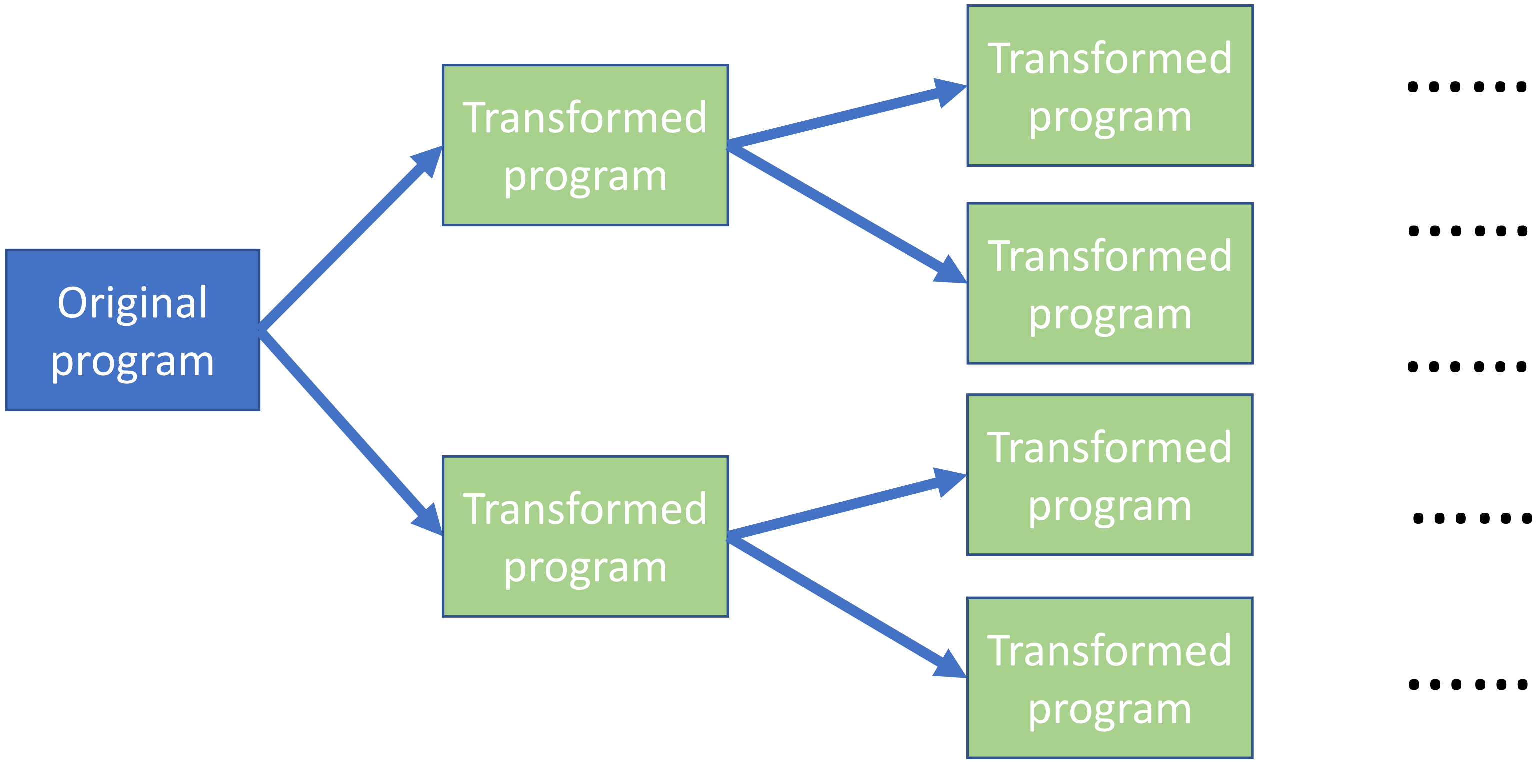
- False crashes may hinder discovery of true bugs (L1)

```
FILE *fp = fopen(...);
if (fp != NULL) {
        // False crash
        fread(fp, ...);
        // ...
        // true bug
        bug();
}
```

Example: false crash hindering discovery of true bug

Limitations of T-Fuzz (2)

- Transformation explosion (L2)
  - Analogous to path explosion issue in symbolic execution

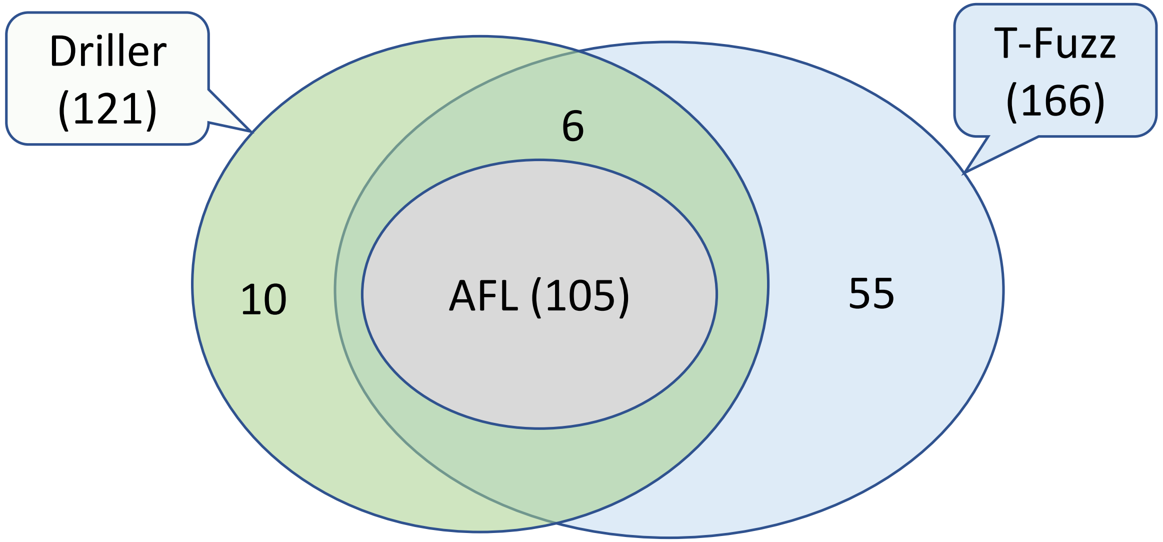


Evaluation

- DARPA CGC dataset
- LAVA-M dataset
- 4 real-world programs

CGC dataset

- Improvement over Driller/AFL: **55 (45%)/61 (58%)**
- T-Fuzz is defeated by Driller in 10
  - due to false crashes (L1) in 3
  - due to transformation explosion (L2) in 7



Method	# of bugs
AFL	105
Driller	121
T-Fuzz	166
Driller - AFL	16
T-Fuzz - AFL	61
T-Fuzz - Driller	55
Driller - T-Fuzz	10

LAVA-M dataset

- T-Fuzz performs well given conditions favorable for VUzzer and Steelix
- T-Fuzz outperforms VUzzer and Steelix for “hard” checks
- T-Fuzz was defeated by Steelix due to transformation explosion in who
- T-Fuzz found 1 unintended bug in who

Program	Total # of bugs	VUzzer	Steelix	T-Fuzz
base64	44	17	43	43
unique	28	27	24	26
md5sum	57	1	28	49
who	2136	50	194	95*

Real-world programs

- Widely evaluated in related work
- T-Fuzz detected far more (verfified) crashes than AFL
- T-Fuzz found 3 new bugs

Program + library	AFL	T-Fuzz
pngfix + libpng (1.7.0)	0	11
tiffinfo + libtiff (3.8.2)	53	124
magick + ImageMagicK (7.0.7)	0	2
pdftohtml + libpoppler (0.62.0)	0	1

Conclusion & future work

- Fuzzers are limited by coverage and unable to find "deep" bugs
- T-Fuzz extend fuzzing by "mutating" the target program as well
- **Experimental results show that T-Fuzz is more effective than state-of-art fuzzers**
  - T-Fuzz has improvement over Driller/AFL by 45%/58%
  - T-Fuzz was able to trigger bugs guarded by "hard" checks
  - T-Fuzz found new bugs: 1 in LAVA-M dataset and 3 in real world programs
- Future work
  - Improve transformation strategies
  - Improve filtering of false positives

https://github.com/HexHive/T-Fuzz