T-Fuzz: Fuzzing by Program Transformation

Hui Peng¹, Yan Shoshitaishvili², Mathias Payer¹
Fuzzing as a bug finding approach

➢ Fuzzing is highly effective in finding bugs (CVEs)
➢ Developers use it as proactive defense measure: OSS-Fuzz, MSRD
➢ Analysts 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**
  - AFL improvements (searching for constants, corpus generation)
  - Driller (selective concolic execution)
  - VUzzer (taint analysis, data & control flow analysis)

- Limitations
  - High overhead
  - Not scalable
  - Unable to bypass “hard” checks
    - Checksum values
    - Crypto-hash values
Insight: some checks are non-critical

- Some checks are not intended to prevent bugs
- **Non-Critical Checks (NCC)**
  - E.g., checks on magic values, checksum, hashes
- Removing NCCs won’t incur erroneous bugs
- Removal of NCCs simplifies fuzzing

```c
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
- Repeats
- Crash Analyzer verifies crashes in the original program
Detecting NCCs (1)

➢ Precisely detecting NCCs is hard
➢ Precise approach
  ○ Leveraging control and data flow analysis techniques
  ○ Slow and unscalable
➢ Imprecise approach
  ○ Approximate NCCs as the checks fuzzer cannot bypass
  ○ May result in false positives due to imprecision
Detecting NCCs (2)

- Approximate NCCs as edges connecting covered and uncovered nodes in CFG
- Over approximate, *may contain false positive*
- Lightweight and simple to implement
  - Dynamic tracing
Program Transformation (1)

➢ **Goal**: disable NCCs

➢ **Possible options**

  ◦ Source rewriting & recompilation
    ■ Complexity involved with mapping between binary and source code
    ■ Compilation results in overhead
  ◦ Static instrumentation
    ■ Error prone
  ◦ Dynamic instrumentation
    ■ High overhead
Our approach: **negate NCCs**
- Easy to implement: static binary rewriting
- Zero runtime overhead in resulting target program
- The CFG of program stays the same
- Trace in transformed program maps to original program
- Path constraints of original program can be recovered
Filtering out false positives & reproducing bugs

Collect paths constraints of the original program by symbolically tracing the transformed program with crashing input

Path constraints

Satisfiable?

False Positive

Generate input to reproduce the crash in original program
Example 1

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

**Original Program**

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

**Transformed Program**

Collected path constraints

\{ x > 0, y == 0xdeadbeef \}

**SAT**

True BUG

un-negating

Negated check
Example 2

Original Program

```c
int main (){
    int i = read_input();
    if (i > 0) {
        func(i);
    }
}

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

Transformed Program

```c
int main (){
    int i = read_input();
    if (i > 0) {
        func(i);
    }
}

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

Collected path constraints

\{ i > 0, i <= 0 \}

UNSAT False BUG

un-negating

Negated check
Comparison with other SE based approaches (1)

➢ Pure symbolic execution, e.g., KLEE
  ○ Explores all possible code paths, tracking input constraints
  ○ Path explosion issue, especially in the presence of loops
    ■ Each branch doubles the number of code paths
  ○ Very high resource requirement
  ○ Theoretically beautiful, limited practical use
Comparison with other SE based approaches (2)

➢ Concolic execution, e.g., CUTE
  o Guided by concrete inputs
  o Following a single code path, collects constraints for new code paths by flipping conditions
  o Reduced resource requirements
  o Total number of explored **symbolic** code paths remains exponential
Combining fuzzing with concolic execution (Driller)

- Fuzzing explores code paths as much as possible
- When fuzzing gets “stuck”, concolic execution explores new code paths using fuzzer generated inputs
- Limitations
  - “SE & constraints solving” slows down fuzzing
  - Not able to bypass “hard” checks
Comparison with other SE based approaches (4)

- SE is decoupled from fuzzing
- SE only applied to detected crashes
- In case of “hard” checks, T-Fuzz still detects the guarded bug, though cannot verify it
T-Fuzz limitation: false crashes (L1)

➢ False crashes may hinder true bug discovery

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

Example: false crash hindering discovery of true bug
T-Fuzz limitation: transformation explosion (L2)

➢ Analogous to path explosion issue in symbolic execution
T-Fuzz limitation: Crash Analyzer (1)

Conflicting constraints result from checks on the same input cause UN SAT.

FILE *fp = fopen(...);
// injected bug in lava-m dataset
fread(fp + lava_get(123) * (lava_get(123) == 0x12345678), ...);

int lava_get(int bug_num) {
    if (lava_vals[bug_num] == 0x12345678) {
        printf("triggered bug %d\n", bug_num);
    }
    return lava_vals[bug_num];
}

Transformed Program

FILE *fp = fopen(...);
// injected bug in lava-m dataset
fread(fp + lava_get(123) * (lava_get(123) != 0x12345678), ...);

int lava_get(int bug_num) {
    if (lava_vals[bug_num] == 0x12345678) {
        printf("triggered bug %d\n", bug_num);
    }
    return lava_vals[bug_num];
}

Collected path constraints

\{ lava_123 == 0x12345678, lava_123 != 0x12345678 \}
T-Fuzz limitation: Crash Analyzer (2)

➢ Unable to verify non-termination (endless loop) detections
   ○ Tracing won’t terminate

➢ Overhead is still high
   ○ Size of program trace (collecting constraints)
   ○ Size of collected path constraints set (constraints solving)
Implementation

➢ Fuzzer: shellphish fuzzer (python wrapper of AFL)
➢ Program Transformer
  ○ angr tracer
  ○ radare2
➢ Crash Analyzer
  ○ angr
➢ 2K LOC (python) + a lot of hackery in angr
Evaluation

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

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

<table>
<thead>
<tr>
<th>Method</th>
<th># bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFL</td>
<td>105</td>
</tr>
<tr>
<td>Driller</td>
<td>121</td>
</tr>
<tr>
<td>T-Fuzz</td>
<td>166</td>
</tr>
<tr>
<td>Driller - AFL</td>
<td>16</td>
</tr>
<tr>
<td>T-Fuzz - AFL</td>
<td>61</td>
</tr>
<tr>
<td>T-Fuzz - Driller</td>
<td>55</td>
</tr>
<tr>
<td>Driller - T-Fuzz</td>
<td>10</td>
</tr>
</tbody>
</table>
LAVA-M dataset

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Total # of bugs</th>
<th>VUzzer</th>
<th>Steelix</th>
<th>T-Fuzz</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64</td>
<td>44</td>
<td>17</td>
<td>43</td>
<td>43</td>
</tr>
<tr>
<td>unique</td>
<td>28</td>
<td>27</td>
<td>24</td>
<td>26</td>
</tr>
<tr>
<td>md5sum</td>
<td>57</td>
<td>1</td>
<td>28</td>
<td>49</td>
</tr>
<tr>
<td>who</td>
<td>2136</td>
<td>50</td>
<td>194</td>
<td>95*</td>
</tr>
</tbody>
</table>
Real-world programs

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

<table>
<thead>
<tr>
<th>Program + library</th>
<th>AFL</th>
<th>T-Fuzz</th>
</tr>
</thead>
<tbody>
<tr>
<td>pngfix + libpng (1.7.0)</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>tiffinfo + libtiff (3.8.2)</td>
<td>53</td>
<td>124</td>
</tr>
<tr>
<td>magick + ImageMagicK (7.0.7)</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>pdftohtml + libpoppler (0.62.0)</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Case study: CROMU_00030 (from CGC dataset)

```c
void main() {
    int step = 0;
    Packet packet;
    while (1) {
        memset(packet, 0, sizeof(packet));
        if (step >= 9) {
            char name[5];
            int len = read(stdin, name, 128);
            printf("Well done, %s\n", name);
            return SUCCESS;
        }
        read(stdin, &packet, sizeof(packet));
        if (strcmp((char *)&packet, "1212") == 0) {
            return FAIL;
        }
        if (compute_checksum(&packet) != packet.checksum) {
            return FAIL;
        }
        if (handle_packet(&packet) != 0) {
            return FAIL;
        }
        step ++;
    }
}
```

Stack Buffer overflow bug

C1: check on magic values

C2: check on checksum

C3: authenticate user info
How the bug was found by T-Fuzz

```c
void main() {
    int step = 0;
    Packet packet;
    while (1) {
        memset(packet, 0, sizeof(packet));
        if (step >= 9) {
            char name[5];
            int len = read(stdin, name, 128);
            printf("Well done, %s\n", name);
            return SUCCESS;
        }
        read(stdin, &packet, sizeof(packet));
        if (strcmp((char *)&packet, "1212") == 0) {
            return FAIL;
        }
        if (compute_checksum(&packet) != packet.checksum) {
            return FAIL;
        }
        if (handle_packet(&packet) != 0) {
            return FAIL;
        }
        step ++;
    }
}
```

Total time to find the bug: ~4h
Manually verified
Demo - T-Fuzz finding bugs in LAVA-M’s uniq
Current status

➢ Program transformation
  ○ No support to transform shared libraries
  ○ Jump tables are not supported
    ■ switch … case statements, complex if … else if … statements

➢ Crash Analyzer
  ○ Scalability issues for large programs
  ○ Lack of environmental modelling (syscall, libc functions) in angr
Future work

➢ Improve precision of NCCs
  ○ Use some static analysis to, e.g., underestimate NCCs

➢ Improve mutation of target program
  ○ Add support for mutating jump tables
  ○ Add support for mutating shared libraries

➢ Improve Crash Analyzer
  ○ Add environmental modelling to better support real-world programs
  ○ Crash Analyzer
    ■ Reduce tracing time: eager concolic execution
    ■ Reduce memory consumption: keep track of only one program state
    ■ rewrite the core of angr using C/C++ (?)
Conclusion

➢ Fuzzers are limited by coverage and unable to find “deep” bugs
➢ T-Fuzz extends fuzzing by mutating both inputs and target program
➢ **T-Fuzz outperforms state-of-art fuzzers**
  ○ T-Fuzz had improvement over Driller/AFL by 45%/58%
  ○ T-Fuzz triggered bugs guarded by “hard” checks
  ○ T-Fuzz found new bugs: 1 in LAVA-M dataset and 3 in real-world programs

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