T-Fuzz: Fuzzing by Program Transformation

Hui Peng\textsuperscript{1}, Yan Shoshitaishvili\textsuperscript{2}, Mathias Payer\textsuperscript{1}

Purdue University\textsuperscript{1}, Arizona State University\textsuperscript{2}

A PRESENTATION BY

MD SHAMIM HUSSAIN AND NAFIS NEEHAL
Outline

• Brief forecast
• Background
• Key Contributions
• T-Fuzz Design
• T-Fuzz Limitations
• Experimental Results
• Concluding Remarks
• References
Fuzzing

- Randomly generates input to discover bugs in the target program
  - Highly effective for discovering vulnerabilities
  - Standard technique in software development to improve reliability and security

- Roughly two types of fuzzers - generational and mutational
  - Generational fuzzers require strict format for generating input
  - Mutational fuzzers create input by randomly mutating or by random seeds
Challenges in Fuzzing

- Shallow coverage
- “Deep Bugs” are hard to find
- Reason – fuzzers get stuck at complex sanity checks

Fig 1: Fuzzing limitations
Existing Approaches and Limitations

• Existing approaches that focus on input generation
  ◦ AFL (Feedback based)
  ◦ Driller (Concolic execution based)
  ◦ VUzzer (Taint analysis based)
  ◦ etc.

• Limitations
  ◦ Heavyweight analysis (taint analysis, concolic execution)
  ◦ Not scalable (concolic execution)
  ◦ Gets stuck on complex sanity checks (e.g. checksum values)
  ◦ Slow progress for multiple sanity checks (feedback based)
Motivation for T-Fuzz

• Hard to bypass complex sanity checks only by input mutation
• Can be easily bypassed by transforming the program
• Same bugs also exist in transformed program
• Detect bugs in the transformed program and analyze their validity
Motivation (Cont.)

• Non-Critical Checks (NCC)
  • Checks on magic values, checksum, hashes

• Detect and bypass NCCs to continue fuzzing

Fig 2: Transforming program to continue fuzzing
T-Fuzz: A novel mutational fuzzing technique

• Uses a mutational fuzzer off-the-shelf for input mutation
• Bypasses complex sanity checks in the program by program transformation
• Lightweight dynamic tracing during fuzzing process instead of heavyweight symbolic analysis
• Removes false-positives by a post-processing (symbolic execution-based analysis)
Key Contributions

• Transformational fuzzing for effective bug finding

• Automatic detection of sanity checks, program transformation and reproduction of bugs in the original program

• Detects more bugs than existing state-of-the-art approaches on benchmark datasets

• Found new bugs in real world programs
Overview of T-Fuzz

- Fuzzer generates inputs guided by coverage
  - Gets stuck at complex sanity checks
- T-Fuzz detects when fuzzer is stuck
  - Detects NCC candidates
  - Negates NCC candidates (program transformation)
- Continue fuzzing on transformed programs (in FIFO order)
- Crashes are recorded
- Crash analyzer filters out false positives

Fig 3: Overview of T-Fuzz
• Precise detection would require techniques like dataflow analysis
  ◦ Would slow down fuzzing process

• Solution – defer precision for speed
  ◦ Verification is done post-mortem

• Imprecise approach (Chosen Approach)
  ◦ Work with a superset of possible NCCs
    ◦ At points where the fuzzer gets stuck
    ◦ Filter out “true” bugs from crashes by symbolic analysis
T-Fuzz Design: Detecting NCC Candidates (Cont.)

- Detection is based on the CFG
- Dynamic tracing for fuzzer generated inputs
  - Accumulate the covered nodes
- NCC candidate edges – originates from covered node – leads to uncovered node
  - Over-approximation
  - But lightweight and simple
- Prune – obviously undesired ones
  - e.g. edges leading to program end

Fig 4: Detecting NCC Candidates
T-Fuzz Design: Program Transformation

- Target is to bypass NCCs

- Options
  - Static Instrumentation
    - Introduces complexity
    - CFG changes
  - Dynamic Instrumentation
    - High overhead
  - Negate NCCs (Chosen Approach)
T-Fuzz Design: Program Transformation (Cont.)

• Transformation by negating NCCs
  ◦ Each basic block contains one jump instruction
  ◦ Negate the condition of the jump instruction
  ◦ Add transformed programs to a FIFO queue

• Advantages
  ◦ 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

Fig 5: Program Transformation
T-Fuzz Design: Filtering False-Positives

1. **Crashing Input**
2. Run Symbolic Trace on the transformed program
3. Collect path constraints for the original program
4. Collect crashing constraint
5. Satisfiable?
   - Yes: Generate Crashing Input in Original Program
   - No: False Positive
T-Fuzz Design: Reproducing Bugs

- Design input to satisfy constraints (in the original program)

(a) Original Program

```c
void main()
{
    int x, y;
    read(0, &x, sizeof(x));
    read(0, &y, sizeof(y));
    if(x == 0xdeadbeef)
    {
        *(int *)y = 0;
    }
}
```

(b) Transformed Program

```c
void main()
{
    int x, y;
    read(0, &x, sizeof(x));
    read(0, &y, sizeof(y));
    if(x != 0xdeadbeef)
    {
        *(int *)y = 0;
    }
}
```

Fig 6: Reproducing Bugs

Negate to get original
T-Fuzz Design: Reproducing Bugs (Cont.)

• When constraints are unsatisfiable

• When constraints cannot be determined, e.g. complex checks like MD5SUM
  ◦ Errs on the side of false negative

```c
void main() {
    char secrets[4] = "DEAD";
    int index;
    read(0, &index, sizeof(index));

    if (index >= 0 && index <= 3)
        output(secrets[index]);
}
```

Transformed Program

Unsatisfiable!
T-Fuzz Limitation: False Crashes

- False-positive crashes (in transformed program) may hinder true bug discovery

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

Fig 7: False Crashes
T-Fuzz Limitation: Transformation Explosion

- Happens if bug hidden in deep code path
  - protected by complex cascaded sanity checks
- Too many transformations required to trigger true bug

Fig 8: Transformation Explosion
T-Fuzz Limitation: Crash Analyzer

- Consecutive checks on the same input
  - May result in conflicting constraints in the transformed program

- Tracing could get stuck in an infinite loop

- High overhead (to remove false-positives)
  - Collecting constraints by tracing
  - Solving constraints by symbolic analysis

- Unable to solve complex constraints (e.g. Hashcheck, MD5SUM)
Implementation

• Language – Python
• Fuzzer – Python interface of AFL
• Transformer
  ◦ angr tracer
  ◦ radare2
• Crash Analyzer
  ◦ Angr
• Github - https://github.com/HexHive/T-Fuzz
Evaluation

• DARPA CGC
  • 248 challenges, 296 binaries

• Lava-M
  • 4 utilities

• 4 real world programs
DARPA CGC

- Catches 61 more bugs than AFL
- Catches 55 more bugs than Driller
- Couldn’t find 10 bugs that Driller could
  - 3 False-positive crashes before true bug and 7 transformation explosion

<table>
<thead>
<tr>
<th>AFL</th>
<th>Driller</th>
<th>T-Fuzz</th>
<th>T-Fuzz - AFL</th>
<th>T-Fuzz - Driller</th>
<th>Driller – T-Fuzz</th>
<th>Driller – AFL</th>
</tr>
</thead>
<tbody>
<tr>
<td>105</td>
<td>121</td>
<td>166</td>
<td>61</td>
<td>55</td>
<td>10</td>
<td>16</td>
</tr>
</tbody>
</table>

Fig 9: T-Fuzz on DARPA CGC
LAVA-M

- Favorable for VUzzer and Steelix
  - Checks input against magic bytes
    - favorable to static analysis
    - well-formatted seed
- T-Fuzz still manages to outperform in 3 programs
  - Better at overcoming hard checks – md5sum
- T-Fuzz loses to Steelix in 1 program
  - Reason - Transformation Explosion

<table>
<thead>
<tr>
<th>Program</th>
<th>No. 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>

Fig 10: T-Fuzz on LAVA-M
Real-world programs

• While AFL gets stuck – T-Fuzz pushes through
• Found 3 new bugs in stable releases

<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>

Fig 11: T-Fuzz on Real-world programs
False Positives

• Much Lower False Positives
  ◦ Static analyses false positive rates around 90%

• 2.8 alerts for every true positive in CGC

• 1.1 alerts for every true positive in LAVA-M

• False positives may actually hint at true bugs
  ◦ Crash analyzer errs in favor of false negative if SATness cannot be determined (e.g. MD5 checksums)
Case Study: CROMU_00030 (from CGC)

```c
int main() {
    int step = 0;
    Packet packet;
    while (1) {
        memset(packet, 0, sizeof(packet));
        if (step >= 9) {
            char name[5];
            // stack buffer overflow BUG
            printf("Well done, stdin, name, 25!");
            return SUCCESS;
        }
        // read a packet from the user
        read(stdin, &packet, sizeof(packet));
        // initial sanity check
        if (strcmp((char *)&packet, "1212") == 0) {
            return FAIL;
        }
        // other trivial checks on the packet omitted
        if (compute_checksum(&packet) != packet.checksum) {
            return FAIL;
        }
        // handle the request from the user, e.g., authentication
        if (handle_packet(&packet) != 0) {
            return FAIL;
        }
        // all tests in this step passed
        step++;
    }
}
```

- **Check1:** Magic Value
- **Check2:** Checksum
- **Check3:** User info

**Inversion**

**Stack Buffer Overflow**

**CSCI 6450 - PRINCIPLES OF PROGRAM ANALYSIS**
Critique

• Oversimplifying “stuck” state?
  ◦ Doesn’t keep track of individual paths
  ◦ No notion of “hardness” of a particular check
    ◦ Could lead to wasted time/erroneous NCCs

• Blind transformation – simple, but is it the best option?
  ◦ Transformation explosion – many paths may not even be satisfiable

• Can we take feedback from crash analysis?
  ◦ False crashes

• Additionally use other tools, such as dataflow analysis (albeit sparingly)?
  ◦ On multiple consecutive input checks – keep constraints consistent
Future Direction

• Smarter selection of NCCs
• Combine with other techniques - dataflow analysis, taint analysis
• Parallel crash analysis and feedback for smarter transformation
• Smarter fuzzing (i.e. the heuristic algorithm) instead of off-the-shelf fuzzer
References


References (Cont.)


Thank You