2018 IEEE Symposium on Security and Privacy

T-Fuzz: fuzzing by program transformation

Hui Peng
Purdue University
peng124@purdue.edu

Yan Shoshitaishvili
Arizona State University
yans@asu.edu

Mathias Payer
Purdue University
mathias.payer@nebelwelt.net

***Abstract*—Fuzzing is a simple yet effective approach to discover software bugs utilizing randomly generated inputs. However, it is limited by coverage and cannot find bugs hidden in deep execution paths of the program because the randomly generated inputs fail complex *sanity check*s, e.g., checks on magic values, checksums, or hashes.**

To improve coverage, existing approaches rely on imprecise heuristics or complex input mutation techniques (e.g., symbolic execution or taint analysis) to bypass sanity checks. Our novel method tackles coverage from a different angle: by removing sanity checks in the target program. T-Fuzz leverages a coverage guided fuzzer to generate inputs. Whenever the fuzzer can no longer trigger new code paths, a light-weight, dynamic tracing based technique detects the input checks that the fuzzer-generated inputs fail. These checks are then removed from the target program. Fuzzing then continues on the transformed program, allowing the code protected by the removed checks to be triggered and potential bugs discovered.

Fuzzing transformed programs to find bugs poses two challenges: (1) removal of checks leads to over-approximation and false positives, and (2) even for true bugs, the crashing input on the transformed program may not trigger the bug in the original program. As an auxiliary post-processing step, T-Fuzz leverages a symbolic execution-based approach to filter out false positives and reproduce true bugs in the original program.

By transforming the program *as well as mutating the input*, T-Fuzz covers more code and finds more true bugs than any existing technique. We have evaluated T-Fuzz on the DARPA Cyber Grand Challenge dataset, LAVA-M dataset and 4 real-world programs (`pngfix`, `tiffinfo`, `magick` and `pdftohtml`). For the CGC dataset, T-Fuzz finds bugs in 166 binaries, Driller in 121, and AFL in 105. In addition, found 3 new bugs in previously-fuzzed programs and libraries.

I. Introduction

Fuzzing is an automated software testing technique that discovers faults by providing randomly-generated inputs to a program. It has been proven to be simple, yet effective [1], [2]. With the reduction of computational costs, fuzzing has become increasingly useful for both hackers and software vendors, who use it to discover new bugs/vulnerabilities in software. As such, fuzzing has become a standard in software development to improve reliability and security [3], [4].

Fuzzers can be roughly divided into two categories based on how inputs are produced: generational fuzzers and mutational fuzzers. Generational fuzzers, such as PROTOS [5], SPIKE [6], and PEACH [7], construct inputs according to some provided format specification. By contrast, mutational fuzzers, including AFL [8], honggfuzz [9], and zzuf [10], create inputs by randomly mutating analyst-provided or randomly-generated seeds. Generational fuzzing requires an input format specification, which imposes significant manual effort to create (especially when attempting to fuzz software on a large scale) or may be infeasible if the format is not available. Thus, most recent work in the field of fuzzing, including this paper, focuses on mutational fuzzing.

Fuzzing is a dynamic technique. To find bugs, it must trigger the code that contains these bugs. Unfortunately, mutational fuzzing is limited by its coverage. Regardless of the mutation strategy, whether it be a purely randomized mutation or coverage-guided mutation, it is highly unlikely for the fuzzer to generate inputs that can bypass complex sanity checks in the target program. This is because, due to their simplicity, mutational fuzzers are ignorant of the actual input format expected by the program. This inherent limitation prevents mutational fuzzers from triggering code paths protected by sanity checks and finding "deep" bugs hidden in such code.

Fuzzers have adopted a number of approaches to better mutate input to satisfy complex checks in a program. AFL [8], considered the state-of-art mutational fuzzer, uses coverage to guide its mutation algorithm, with great success in real programs [11]. To help bypass the sanity checks on magic values in the input files, AFL uses coverage feedback to heuristically infer the values and positions of the magic values in the input. Several recent approaches [12], [13], [14], [15] leverage symbolic analysis or taint analysis to improve coverage by generating inputs to bypass the sanity checks in the target program. However, limitations persist — as we discuss in our evaluation, state-of-the-art techniques such as AFL and Driller find vulnerabilities in less than half of the programs in a popular vulnerability analysis benchmarking dataset (the challenge programs from the DARPA Cyber Grand Challenge).

Recent research into fuzzing techniques focuses on finding new ways to generate and evaluate inputs. However, there is no need to limit mutation to program inputs alone. In fact, the *program itself* can be mutated to assist bug finding in the fuzzing process. Following this intuition, we propose

DOI 10.1109/SP.2018.00056

IEEE computer society

Transformational Fuzzing, a novel fuzzing technique aimed at improving the bug finding ability of a fuzzer by disabling input checks in the program. This technique turns the dynamic code coverage problem on its head: rather than necessitating time-consuming and heavyweight program analysis techniques to generate test cases to bypass complex input checks in the code, we simply detect and disable these sanity checks. Fuzzing the transformed programs allows an exploration of code paths that were previously-protected by these checks, discovering potential bugs in them.

Of course, removing certain sanity checks may break the logic of the original program and the bugs which are found in transformed programs may thus contain *false positives*, potentially overwhelming the analyst. To remove false positives, we develop a post-processing symbolic execution-based analysis. The remaining inputs reproduce true bugs in the original program. Though this method is complex and heavyweight (like test case mutation techniques of related work), it only needs to be done to verify detections after the fact, and (unlike existing test case mutation techniques) does not slow down the actual analysis itself.

To show the usefulness of transformational fuzzing, we developed a prototype named T-Fuzz. At its base, it employs an off-the-shelf coverage-guided fuzzer to explore a program. Whenever the fuzzer can no longer generate new inputs to trigger unexplored code paths, a lightweight dynamic tracing-based approach discovers all input checks that the fuzzer-generated inputs failed to satisfy, and the program is transformed by selectively disabling these checks. Fuzzing then continues on the transformed programs.

In comparison to existing symbolic analysis based approaches, T-Fuzz excels in two aspects: (1) better scalability: by leveraging lightweight dynamic tracing-based techniques during the fuzzing process, and limiting the application of heavyweight symbolic analysis to detected crashes, the scalability of T-Fuzz is not influenced by the need to bypass complex input checks; and (2) the ability to cover code paths protected by "hard" checks.

To determine the relative effectiveness against state-of-the-art approaches, we evaluated T-Fuzz on a dataset of vulnerable programs from the DARPA Cyber Grand Challenge (CGC), the LAVA-M dataset, and four real-world programs relying on popular libraries (`pngfix/libpng`, `tiffinfo/libtiff`, `magick/ImageMagick` and `pdftohtml/libpoppler`). In the CGC dataset, T-Fuzz finds bugs in 166 binaries out of 296, improving over Driller [12] by 45 binaries and over AFL by 61 binaries, and demonstrating the effectiveness of transformational fuzzing. Evaluation of the LAVA-M dataset shows that T-Fuzz significantly outperforms Steelix and VUzzer in the presence of "hard" input checks, such as checksums. The ground truth provided by these two datasets allows us to determine that our tool is able to filter out false positives at the cost of surprisingly few false negatives (6%-30%). Finally, the evaluation of T-Fuzz on real-world applications leads to the discovery of 3 new bugs.

In summary, this paper makes the following contributions:

Header(4)	Keys(95)	Len(4)	Data(Len)	CRC(4)

Fig. 1: Secure compressed file format

1) We show that fuzzing can more effectively find bugs by transforming the target program, instead of resorting to heavy weight program analysis techniques.
2) We present a set of techniques that enable fuzzing to mutate both inputs and the programs, including techniques for (i) automatic detection of sanity checks in the target program, (ii) program transformation to remove the detected sanity checks, (iii) reproducing bugs in the original program by filtering false positives that only crash in the transformed program.
3) We evaluated T-Fuzz on the CGC dataset, LAVA-M dataset and 4 real-world programs. Experimental results show the effectiveness of our technique as compared to other state-of-art fuzzing tools.
4) We found 3 new bugs: two in `magick/ImageMagicK` and one in `pdftohtml/libpoppler`.

II. Motivation

State-of-art mutational fuzzers like AFL (American Fuzzy Lop) [8] and honggfuzz [9] — called coverage guided fuzzers — use coverage as feedback from the target program to guide the mutational algorithm to generate inputs. Specifically, they keep track of a set of interesting inputs that triggered new code paths and focus on mutating the interesting inputs while generating new inputs. However, generation of inputs to pass through complex sanity checks remains a well-known challenge for mutational fuzzers because they are ignorant of the expected input format. When mutational fuzzers fail to generate inputs to bypass the sanity checks, they become "stuck" and continue to generate random inputs without covering new code paths.

As an example, Figure 1 shows a secure compressed file format: the first 4 bytes are a file header which contains hardcoded magic values ("SECO"); the *Keys* field declares 95 unique chars whose ASCII values must be within the range of [32, 126]; the *Len* field specifies the length of the following compressed data; and finally a 4-byte *CRC* field to integrity check the compressed data. The example is based on CGC KPRCA_00064, extended with a CRC check.

Listing 1 is a program that parses and decompresses the compressed file format shown above. It has a "deep" stack buffer overflow bug in *decompress* function in line 31. Before calling *decompress*, the program performs a series of checks on the input:

C1. check on the magic values of the header field in line 8.
C2. check for range and uniqueness on the next 95-byte Keys field in line 13-18.
C3. check on the CRC field for potential data corruption in line 24.

If any of these checks fail, the input is rejected without calling *decompress*, thereby not triggering the bug.

These checks highlight the challenges in mutational fuzzers and related techniques. First of all, it takes a lot of effort for a

```c

#define KEY_SIZE  95
int sc_decompress(int infd, int outfd) {
  unsigned char keys[KEY_SIZE];
  unsigned char data[KEY_SIZE];
  char *header = read_header(infd)
  // C1: check for hardcoded values
  if (strcmp(header, "SECO") != 0)
    return ERROR;
  read(infd, keys, KEY_SIZE);
  memset(data, 0, sizeof(data));
  // C2: range check and duplicate check for keys
  for (int i = 0; i < sizeof(data); ++i) {
    if (keys[i] < 32 || keys[i] > 126)
      return ERROR;
    if (data[keys[i] - 32]++ > 0)
      return ERROR;
  }
  unsigned int in_len = read_len(infd);
  char *in = (char *) malloc(in_len);
  read(infd, in, in_len);
  unsigned int crc = read_checksum(infd);
  // C3: check the crc of the input
  if (crc != compute_crc(in, in_len)) {
    free(in);
    return ERROR;
  }
  char *out;
  unsigned int out_len;
  // Bug: function with stack buffer overflow
  decompress(in, in_len, keys, &out, &out_len);
  write(outfd, out, out_len);
  return SUCCESS;
}
```

Listing 1: An example containing various sanity checks

mutational fuzzer like AFL [8] to bypass C1 without the help of other techniques. As mutational fuzzers are unaware of the file format, they will struggle to bypass C2 or C3. Additionally, although symbolic analysis based approaches like Driller [12] can quickly bypass C1 and C2 in this program, they will fail to generate accurate inputs to bypass C3 as the constraints derived from the checksum algorithm are too complex for modern constraint solvers [16]. It is therefore unlikely that the buggy *decompress* function will be triggered through either mutational fuzzing or symbolic analysis.

Regarding sanity checks in the context of fuzzing, we make the following observations:

1) Sanity checks can be divided into two categories: **NCC** (Non-Critical Check) and **CC** (Critical Check). NCCs are those sanity checks which are present in the program logic to filter some orthogonal data, e.g., the check for a magic value in the decompressor example above. CCs are those which are essential to the functionality of the program, e.g., a length check in a TCP packet parser.
2) NCCs can be removed without triggering spurious bugs as they are not intended to prevent bugs. Removal of NCCs simplifies fuzzing as the code protected by these checks becomes exposed to fuzzer generated inputs. Assume we remove the three checks in the decompressor above, producing a transformed decompressor. All inputs generated by the fuzzer will be accepted by the transformed decompressor and the buggy *decompress* function will be covered and the bug found.
3) Bugs found in the transformed program can be reproduced in the original program. In the decompressor above, as the checks are not intended for preventing the stack buffer overflow bug in the decompress function, bugs found in the transformed decompressor are also present in the original decompressor. Assume that the fuzzer found a bug in the transformed decompressor with a crashing input X, it can be reproduced in the original decompressor by replacing the Header, Keys, and CRC fields with values that satisfy the check conditions in the program.
4) Removing CCs may introduce spurious bugs in the transformed program which may not be reproducible in the original program. These false positive bugs need to be filtered out during a post-processing phase to ensure that only the bugs present in the original program are reported.

NCCs are omnipresent in real-world programs. For example on Unix systems, all common file formats use the first few bytes as magic values to identify the file type. In network programs, checksums are widely used to detect data corruption.

Based on these observations, we designed T-Fuzz to improve fuzzing by detecting and removing NCCs in programs. By removing NCCs in the program, the code paths protected by them will be exposed to the fuzzer generated inputs and potential bugs can be found. T-Fuzz additionally helps fuzzers cover code protected by “hard” sanity checks like C3 in Listing 1.

III. T-Fuzz Intuition

Figure 2 depicts the main components and overall workflow of T-Fuzz. Here we summarize its main components, the details will be covered in the following section.

Fuzzer: T-Fuzz uses an existing coverage guided fuzzer, e.g., AFL [8] or honggfuzz [9], to generate inputs. T-Fuzz depends on the fuzzer to keep track of the paths taken by all the generated inputs and realtime status information regarding whether it is “stuck”. As output, the fuzzer produces all the generated inputs. Any identified crashing inputs are recorded for further analysis.

Program Transformer: When the fuzzer gets “stuck”, T-Fuzz invokes its Program Transformer to generate transformed programs. Using the inputs generated by the fuzzer, the Program Transformer first traces the program under test to detect the NCC candidates and then transforms copies of the program by removing certain detected NCC candidates.

Crash Analyzer: For crashing inputs found against the transformed programs, the Crash Analyzer filters false positives using a symbolic-execution based analysis technique.

Algorithm 1 shows how a program is fuzzed using T-Fuzz. First, T-Fuzz iteratively detects and disables NCC candidates that the fuzzer encounters in the target program. A queue (`programs`) is used to save all the programs to fuzz in each iteration, and initially contains the original program. In

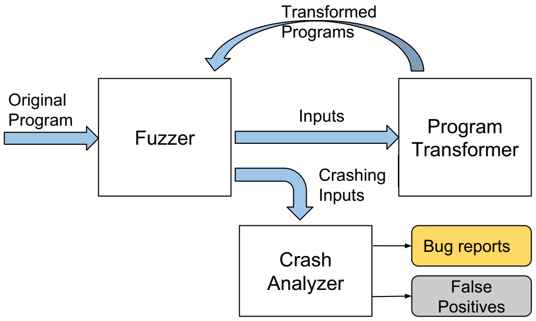


Fig. 2: Overview of T-Fuzz

each iteration, T-Fuzz chooses a program from the queue and launches a fuzzer process (in the algorithm, the invocation of `Fuzzer`) to fuzz it until it is unable to generate inputs that further improve coverage. Using inputs generated by the fuzzer before it gets "stuck", T-Fuzz detects additional NCC candidates and generates multiple transformed programs (by invoking `Program_Transformer`) with different NCCs disabled. The transformed programs are added to the queue for fuzzing in the following iterations. All crashes found by the Fuzzer in each iteration are post-processed by the Crash Analyzer to identify false positives.

Algorithm 1: Fuzzing with T-Fuzz

```
Input: $program$: original program
$programs \leftarrow \{program\}$
while $programs \neq \varnothing$ do
    $p \leftarrow Choose\_Program(programs)$
    $inputs \leftarrow Fuzzer(p)$
    $programs \leftarrow$
      $programs \cup Program\_Tranformer(p, inputs)$
```

IV. T-Fuzz Design

In this section we describe T-Fuzz and discuss the technical details of its key components. T-Fuzz uses an off-the-shelf version AFL for its fuzzer. The design and details of mutational fuzzing are orthogonal to this work and covered in [17] for AFL.

Detecting NCCs: Detecting NCCs is the most important step in T-Fuzz. To this end, different approaches are possible. For performance reasons, in T-Fuzz we over-approximate NCCs to *all the checks that fuzzer-generated inputs could not bypass* and leverage an imprecise, light-weight dynamic tracing based approach to detect them. This gives us a set of *NCC Candidates* (shown in Section IV-A). As the detected NCC Candidates may contain other sanity checks, fuzzing the transformed program may result in *false positives*.

Program Transformation: Program Transformation is the process of removing the detected NCC candidates in the target program. In T-Fuzz, we use binary rewriting to negate detected NCC candidates in the target program. The details are presented in Section IV-C.

Filtering out False Positives and Reproducing true bugs: As mentioned above, the bugs found by fuzzing the transformed program may contain false positives. To help the analyst identify true bugs, as post processing, we provide a symbolic analysis pass that filters false positives (Section IV-D). As it relies on symbolic analysis, it may not work in the presence of "hard" checks. In that case, manual analysis is needed.

A. Detecting NCC Candidates

To detect the NCCs in a program, different options are available with varying precision and overhead. For example, we can use complex data flow and control flow analysis to track dependencies between the sanity checks in the program and input. This approach has good precision, but involves very high overhead (which is extremely detrimental for fuzzing, as fuzzers are heavily optimized for performance), and often needs to be based on fairly brittle techniques (which is detrimental to the applicability of the technique). Considering this, in T-Fuzz, we use a less precise, but lightweight approach that approximates NCCs; we use the set of checks that could not be satisfied by *any* fuzzer-generated inputs when the fuzzer gets stuck.

Sanity checks are compiled into conditional jump instructions in the program, and represented as a source basic block S with 2 outgoing edges in the control flow graph (CFG)[1]. Each outgoing edge corresponds to either True or False of the condition, which are denoted as T and F respectively. Failing to bypass a sanity check means that only the T or F edge is ever taken by any fuzzer-generated input.

In T-Fuzz we use all the boundary edges in the CFG — the edges connecting the nodes that were covered by the fuzzer-generated inputs and those that were not — as *approximation* of NCC candidates. Denote all the nodes in the CFG of the program as N and all the edges as E, and let CN be the union of executed nodes, CE be the union of taken edges by all inputs in I. The boundary edges mentioned above can be formalized as all edges e satisfying the following conditions:

1) e is not in CE;
2) source node of e is in CN;

Algorithmically, T-Fuzz first collects the cumulative node and edge coverage executed by the fuzzer generated inputs. Then, it uses this cumulative coverage information to calculate the NCC candidates. T-Fuzz uses a dynamic tracing based approach to get the cumulative edge and node coverage for a set of inputs. As shown in Algorithm 3, for each input i, it invokes $dynamic\_trace$ to get the sequence of executed edges under input i. The union of edges and nodes in all traces of inputs is returned.

Algorithm 2 shows how NCC candidates are detected based on the cumulative edge and node coverage collected by Algorithm 3. It builds the CFG of the program and then iterates over all the edges, returning those that satisfy the conditions shown above as NCC candidates.

[1] To simplify the discussion we assume that switch statements are compiled to a tree of if conditions, instead of a jump table, although the tool itself makes no such assumption.

Algorithm 2: Detecting NCC candidates

```
Input: $program$: The binary program to be analyzed
Input: $CE$: cumulative edge coverage
Input: $CN$: cumulative node coverage
$cfg \leftarrow CFG(program)$
$NCC \leftarrow \varnothing$
for $e \in cfg.edges$ do
    if $e \notin CE \wedge e.source \in CN$ then
        $NCC \leftarrow NCC \cup \{e\}$
Output: $NCC$: detected NCC candidates
```

Algorithm 3: Calculating cumulative edge and node coverage

```
Input: $inputs$: inputs to collect cumulative coverage
$CE \leftarrow \varnothing$
$CN \leftarrow \varnothing$
for $i \in inputs$ do
    $trace \leftarrow dynamic\_trace(i)$
    for $e \in trace$ do
        $CE \leftarrow CE \cup \{e\}$
        $CN \leftarrow CN \cup \{(e.source, e.destination)\}$
Output: $CE$: cumulative edge coverage
Output: $CN$: cumulative node coverage
```

We use the following example to demonstrate the effect of this algorithm. Listing 2 takes 16 bytes as input and uses the first two bytes as magic values and the third byte to decide whether to use `format1` or `format2` to process the input. Figure 3a shows the CFG of the program.

Assume the fuzzer component has generated a set of inputs {“123...”, “A12..”, “AB_...”, “AB{...”}, and gets “stuck” without being able to cover `format1` and `format2` we are interested in. Running our algorithm we can easily detect the sanity checks that are guarding the invocation of `format1` and `format2`. As “123...” triggers execution path A→H, “A12..” triggers execution path A→B→H, “AB_..” triggers A→B→C→E→G→H, and “AB{..” triggers A→B→C→D→H, the cumulative node and edge coverage are {A, B, C, D, E, G, H} and { A→H, A→B, B→H, B→C, C→D, D→H, C→E, E→G, G→H} (see Figure 3b), and the detected NCC candidates are {C→D, E→H, G→I} (see Figure 3c). Obviously D→F and G→I are the sanity checks that preventing the fuzzer generated inputs to cover `format1` and `format2`.

B. Pruning Undesired NCC Candidates

The NCC candidates detected using the algorithm in Section IV-A is an over-approximation of sanity checks and may contain undesired checks. Before feeding the NCC candidates from Algorithm 2 into the Program Transformer, a filtering step prunes any undesired candidates that we deem unlikely to help bug finding. We list the types of undesired checks we encountered, the consequences of removing these checks, and our approaches to remove undesired checks.

```
void main() {
  char x[16];
  read(stdin, x, 16);

  if (x[0] == 'A' && x[1] == 'B') {
      if (x[2] >= 'a') {
        if (x[2] <= 'z') {
           format1(x);
        } else {
          goto failure;
        }
      } else {
        if (x[2] >= 'A' && x[2] <= 'Z') {
           format2(x);
        } else {
           goto failure;
        }
      }
  }
failure:
  error();
}
```

Listing 2: An example demonstrating the effect of NCC detection algorithm

```
1  void main() {
2    char x[10];
3
4    if (read(0, x, 10) == -1)
5      goto failure;
6    // main logic for processing x
7    ...
8    return;
9  failure:
10   error();
11 }
```

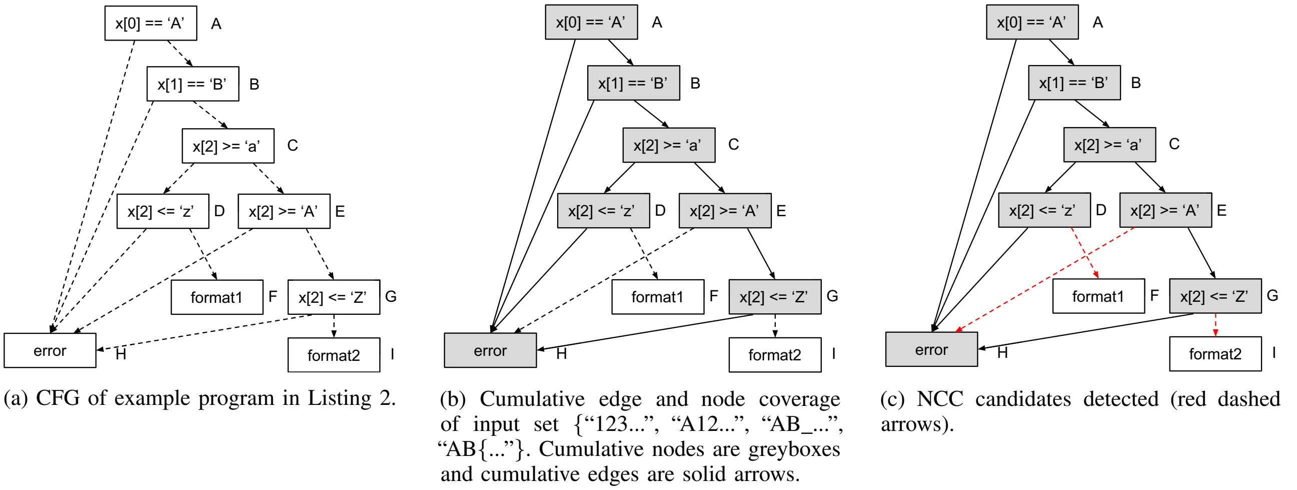
Listing 3: An program showing check for error code

Algorithm 2 in Section IV-A detects NCC candidates in all executed (or not executed) code. When fuzzing, we are often interested in just the program executable or a specific library. In a first step, we therefore prune any candidates that are not in the desired object.

The second source of undesired checks are checks for error codes that immediately terminate the program. For example in the program shown in Listing 3, the return value of the *read* system call is checked for possible errors in line 4. As read errors happen infrequently, the error checking code is not executed and thus detected as NCC candidate.

Treating these checks as NCCs does not result in useful detections from T-Fuzz. Consider the detected check shown in Listing 3. Removing the check results in a program execution where only the error handling code is executed before the program is abnormally terminated. It is unlikely for the fuzzer to find bugs along this path.

Given that the program often terminates with very short code paths after detecting such a severe error, we heuristically use the number of basic blocks following the detected NCC candidate as the length of code paths and define a threshold value to tell an error handling code path. The intuition behind this approach is to focus on NCCs that result in a large amount of increased coverage compared to NCCs that immediately terminate the program under test (due to, e.g., a severe error).



(a) CFG of example program in Listing 2.

(b) Cumulative edge and node coverage of input set {"123...", "A12...", "AB_...", "AB{..."}. Cumulative nodes are greyboxes and cumulative edges are solid arrows.

(c) NCC candidates detected (red dashed arrows).

Fig. 3: How NCC candidates are detected

C. Program Transformation

We considered different options to remove detected NCC candidates, including as dynamic binary instrumentation, static binary rewriting, and simply flipping the condition of the conditional jump instruction. Dynamic binary instrumentation often results in high overhead and static binary translation results in additional complexity due to the changed CFG. On the other hand, flipping conditions for conditional jumps is straight-forward and neutral to the length of the binary, providing the advantages of static rewriting without the need for complex program analysis techniques. This technique maintains the inverted path condition in the program, and the path condition in the original program can be easily recovered.

T-Fuzz transforms programs by replacing the detected NCC candidates with a negated conditional jump. Doing so maintains the structure of the original program while keeping necessary information to recover path conditions in the original program. As the addresses of the basic blocks stay the same in the transformed program, the traces of the transformed program directly map to the original program. Maintaining the trace mapping greatly reduces the complexity of analyzing the difference between the original program and transformed program in the Crash Analyzer.

Algorithm 4 shows the pseudo code of the Program Transformer. The Program Transformer takes a program to transform and NCC candidates to remove as input. As there is at most one jump instruction in a basic block, it simply scans all the instructions in the source block of the NCC candidate and overwrites the first conditional jump instruction with its negated counterpart instruction. To keep track of the modified conditional jump (by invocation of $negate\_conditional\_jump$), the addresses of modified instructions are passed in as argument, and the address of each modified instruction is recorded and returned as part of the output.

Algorithm 4: Transforming program

```
Input: $program$: the binary program to transform
Input: $c\_addrs$: the addresses of conditional jumps
       negated in the input program
Input: $NCC$: NCC candidates to remove
$transformed\_program \leftarrow Copy(program)$
for $e \in NCC$ do
    $basic\_block \leftarrow$
      $BasicBlock(transformed\_program, e.source)$
    for $i \in basic\_block$ do
        if $i$ is a conditional jump instruction and
          $i.addr \notin c\_addrs$ then
            $negate\_conditional\_jump(program, i.addr)$
            $c\_addrs \leftarrow c\_addrs \cup \{i.addr\}$
            $break$
Output: $transformed\_program$: the generated
        program with NCC candidate disabled
Output: $c\_addrs$: the locations modified in the
        transformed program
```

D. Filtering out False Positives and Reproducing Bugs

As the removed NCC candidates might be meaningful guards in the original program (as opposed to, e.g., magic number checks), removing detected NCC edges might introduce new bugs in the transformed program. Consequently, T-Fuzz's Crash Analyzer verifies that each bug in the transformed program is also present in the original program, thus filtering out false positives. For the remaining true positives, an example input that reproduces the bug in the original program is generated.

The Crash Analyzer uses a *transformation-aware* combination of the preconstrained tracing technique leveraged by Driller [12] and the Path Kneading techniques proposed by ShellSwap [18] to collect path constraints of the original program by tracing the program path leading to a crash in

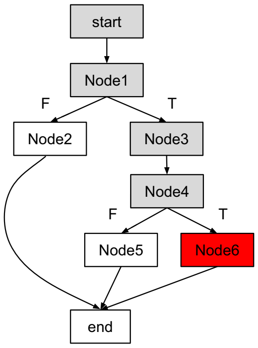


Fig. 4: An example of transformed program

the transformed program. The satisfiability of the collected path constraints indicates whether the crash is a false positive or not. If the path constraints are satisfiable, the Crash Analyzer reproduces the bug in the original program by solving the path constraints.

To illustrate the idea behind the algorithm, we show a transformed program P whose CFG is represented in Figure 4, and a crashing input I. I executes a code path shown as grey nodes in Figure 4, with the crash in Node6 at address CA. Node1 and Node4 contain conditional jumps that are negated by the Program Transformer and because of this, the T edges are taken when executing I. The constraints associated with NCCs in Node1 and Node4 are denoted as C_1 and C_4 respectively.

When the Crash Analyzer traces the transformed program, it maintains two sets of constraints: one for keeping track of the constraints in the transformed program (denoted as CT) the other for keeping track of that in the original program (denoted as CO). Before the tracing starts, I is converted to a preconstraint (denoted as PC) and added to CT, this ensures that the trace will follow the code path shown in Figure 4. While tracing the transformed program, if the basic block contains a negated conditional jump, the inverted path constraint associated with the conditional jump is added to CO, otherwise, the path constraints are added to CO. In this example, $\neg C_1$ and $\neg C_4$ are added to CO. When the tracing reaches the crashing instruction in the program, the cause of the crash (denoted as CC) is encoded to a constraint and also added to CO. For example, if it is an out-of-bound read or write, the operand of the load instruction is used to encode the constraint, if it is a divide by zero crash, the denominator of the div instruction is used to encode the constraint. If the path constraints in CO can be satisfied, it means that it is possible to generate an input that will execute the same program path and trigger the same crash in the original program. Otherwise it is marked as a false positive.

The pseudo code of the Crash Analyzer is shown in Algorithm 5. It takes the transformed program and the addresses of negated conditional jumps, a crashing input and the crash address in the transformed program as input. It traces the transformed program with the crashing input as pre-constraints using $preconstraint\_trace$ instruction by instruction, and collects the path constraints returned by it in TC. In case a negated jump instruction is encountered, the inverted constraint is saved in CO. In the end, the satisfiability of constraints in CO is checked, if it is unsatisfiable, the input is identified as a false positive, otherwise the constraints collected in CO can be used to generate input to reproduce the bug in the original program.

Algorithm 5: Process to filter out false positives

```
Input: $transformed\_program$: the transformed program
Input: $c\_addrs$: addresses of negated conditional jumps
Input: $input$: the crashing input
Input: $CA$: the crashing address
$PC \leftarrow make\_constraint(input)$
$CT \leftarrow PC$
$CO \leftarrow \varnothing$
$TC, addr \leftarrow preconstraint\_trace(transformed\_program, CT, entry)$
while $addr \neq CA$ do
    if $addr \in c\_addrs$ then
        $CO \leftarrow CO \cup \neg TC$
    else
        $CO \leftarrow CO \cup TC$
    $TC, addr \leftarrow preconstraint\_trace(transformed\_program, CT, i)$
$CO \leftarrow CO \cup extract\_crashing\_condition(TC)$
$result \leftarrow SAT(CO)$
Output: $result$: A boolean indicating whether $input$ is a false positive
Output: $CO$: The set of constraints for generating the inputs in the original program
```

Note that, to err on the side of not overwhelming human analysts with false detections, the Crash Analyzer errs on the side of introducing false negatives over allowing false positives. That is, it is possible that detections marked as false positives by the Crash Analyzer, because they could not be directly reproduced in a symbolic trace, do actually represent bugs in the program. This will be discussed in detail in the case study shown in Section V-E. Further improvements to the Crash Analyzer, beyond transformation-aware symbolic tracing, would improve T-Fuzz's effectiveness.

E. Running Examples

To illustrate how the idea of filtering false positives and reproducing bugs in the original program works, we provide several concrete examples.

Listing 4 demonstrates a fuzzer's weakness to bypass NCCs. It reads an integer from the user and checks it against a specific value. Then a second integer is read from the user and used as a pointer to write an integer into the memory space. As the likelihood that the fuzzer can generate the specific value is exceedingly small, it is unlikely to find the bug.

```
1 void main() {
2    int x, y;
3    read(0, &x, sizeof(x));
4    read(0, &y, sizeof(y));
5    if (x == 0xdeadbeef)
6       *(int *)y = 0;
7 }
```

Listing 4: A simple example showing how Crash Analyzer reproduce bugs in the original program

```
1 void main() {
2    char secrets[4] = "DEAD";
3    int index;
4    read(0, &index, sizeof(index));
5
6    if (index >= 0 && index <= 3)
7       output(secrets[index]);
8 }
```

Listing 5: An example showing how Crash Analyzer filters out false positives

Assume that the fuzzer has generated a few inputs that could not bypass the check in line 5 and the NCC candidate detecting algorithm has identified the transition from line 5 to line 6 as a NCC candidate. Then the Program Transformer generates a transformed program, negating the check condition `(x != 0xdeadbeef)`. Then the fuzzer can easily find the bug in the transformed program.

Assume the fuzzer triggered the bug in the transformed program with `x = 0` and `y = 1`. When the Crash Analyzer traces the transformed program, $\neg(x! = 0xdeadbeef)$ will be added to CO when it reaches the negated conditional jumps shown in line 5. And when tracing reaches the crashing instruction in line 6, the destination of the memory write will be deemed as the crashing condition and thus `y == 1` will be added to CO. As $\{\neg(x! = 0xdeadbeef), y == 1\}$ is satisfiable, the crash is a true bug in the original program. And input `(x = 0xdeadbeef, y = 1)` can be used to reproduce the bug in the original program.

Listing 5 is an example demonstrating how the Crash Analyzer filters out false positives. The example program contains a list of secrets in memory and reads an index from the user to retrieve the secret at the position. To prevent out-of-bound read, it checks the index is within the valid range or not at line 6.

Assume the fuzzer generated inputs cannot bypass the checks in line 6 and the algorithm has detected the transition from line 6 to line 7 as a NCC candidate and the Program Transformer has generated a program with the negated condition `(index < 0 || index > 3)` in line 6. Fuzzing the transformed program, we could get a crash input `index = 0x12345678` that incurs an invalid memory reference.

The Crash Analyzer traces the transformed program in the same way shown as above, and when it reaches the condition statement in line 6, the negated constraint `(index >= 0 && index <= 3)` is added to CO. And when tracing reaches the crashing instruction shown in line 7, the operand for memory read will be deemed as the crashing condition, thus `index == 0x12345678` will be added to CO, as {`index >= 0 && index <= 3, index == 0x12345678`} is unsatisfiable, it is reported as a false positive.

V. Implementation and Evaluation

We have implemented our prototype in Python based on a set of open source tools: the Fuzzer component was built on AFL [8], the Program Transformer was implemented using the angr tracer [19] and radare2 [20], and the Crash Analyzer was implemented using angr [21].

To determine T-Fuzz's bug finding effectiveness, we performed a large-scale evaluation on three datasets (the DARPA CGC dataset, the LAVA-M dataset, and a set of 4 real-world programs built on wide-spread libraries, consisting of `pngfix/libpng`, `tiffinfo/libtiff`, `magick/ImageMagick`, and `pdftohtml/libpoppler`) and compared T-Fuzz against a set of state-of-art fuzzing tools.

The experiments were run on a cluster in which each node is running Ubuntu 16.04 LTS and equipped with an Intel i7-6700K processor and 32 GB of memory.

A. DARPA CGC Dataset

The DARPA CGC dataset [22] contains a set of vulnerable programs used in the Cyber Grand Challenge event hosted by DARPA. These programs have a wide range of authors, functionalities, and vulnerabilities. Crucially, they also provide *ground truth* for the (known) vulnerabilities in these programs. The dataset contains a total of 248 challenges with 296 binary programs, as some of the challenges include multiple vulnerable binaries. For each bug in these programs, the dataset contains a set of inputs as a *Proof of Vulnerability*. These inputs are used as ground truth in our evaluation. Programs in this dataset contain a wide range of sanity checks on the input that are representative of real world programs, and thus are used widely as benchmark in related work [12], [13], [14].

In this evaluation, we run the CGC binaries with three different configurations: (1) to evaluate T-Fuzz against heuristic based approaches, we run AFL on the set of CGC binaries ; (2) to evaluate T-Fuzz against symbolic execution-based approaches, we run Driller [12] on the CGC binaries; (3) we run T-Fuzz on the same set of binaries. Each binary is fuzzed for 24 hours with an initial seed of “fuzz”.

AFL. In this experiment, each binary is assigned one CPU core. Before fuzzing starts, we create a dictionary using angr [21]to help AFL figure out the possible magic values used in the program.

Driller. When running the experiment with Driller [23], each binary is assigned one CPU core for fuzzing and one CPU core for dedicated concolic execution (i.e., Driller uses two CPU cores, double the resources of the other experiments). For resource limits, we use the same settings as the original Driller evaluation [12].

T-Fuzz. To evaluate T-Fuzz, we assign the same CPU limits as for AFL and use one CPU core (half the resources of the Driller experiment). When multiple transformed binaries are generated, they are queued and fuzzed in first-in-first-out order.

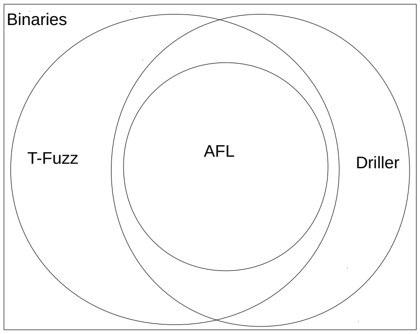


Fig. 5: Venn diagram of bug finding results

TABLE I: Details of experimental results

Method	Number of Binaries
AFL	105
Driller	121
T-Fuzz	166
Driller∖AFL	16
T-Fuzz∖AFL	61
Driller∖T-Fuzz	10
T-Fuzz∖Driller	45

To get an idea of the overall effectiveness of the system, we evaluate the three different configurations and discuss the bugs they found. To validate the T-Fuzz results (i.e., the stability of T-Fuzz's program transformation strategy), we performed the DARPA CGC evaluation three times and verified that the same set of vulnerabilities was found each time. This makes sense, as aside from randomness in the fuzzing process, the T-Fuzz algorithm is fully deterministic.

a) Comparison with AFL and Driller: As the results in Figure 5 and Table I show, T-Fuzz *significantly* outperforms Driller in terms of bug finding. Given the time budget and resource limits mentioned above, T-Fuzz found bugs in 166 binaries (out of 296), compared to Driller which only found bugs in 121 binaries and AFL which found bugs in 105 binaries. All of the bugs found by AFL were discovered by both Driller and T-Fuzz. T-Fuzz found bugs in 45 additional binaries in which Driller did not, while failing to find bugs in 10 that Driller managed to crash. It is important to note that all the bugs found by T-Fuzz mentioned here are *true positives*, verified using the ground truth that the CGC dataset provides. The false positives resulting from T-Fuzz's analysis are discussed later.

Out of these 166 binaries in which T-Fuzz found crashes, 45 contain complex sanity checks that are hard for constraint solvers to generate input for. Failing to get accurate input that is able to bypass the "hard" sanity checks from a symbolic execution engine, the fuzzing engine (AFL) in Driller keeps generating random inputs blindly until it uses up its time budget without making any progress in finding new code paths. This is where the difference between Driller and T-Fuzz comes in. Once the fuzzer gets "stuck", T-Fuzz disables the offending conditional checks, and lets the fuzzer generate inputs to cover code previously protected by them, finding new bug candidates. We use a case study in Section V-E to demonstrate the difference in detail.

Driller found bugs in 10 binaries for which T-Fuzz failed to find (true) bugs. This discrepancy is caused by 2 limitations in the current implementation of T-Fuzz. First, if a crash caused by a false positive stemming from an NCC negation occurs in the code path leading to a true bug, the execution of the transformed program will terminate with a crash without executing the vulnerable code where the true bug resides (L1). T-Fuzz "lost" to Driller in three of the 10 binaries because of this limitation. Secondly, when the true bug is hidden deep in a code path containing many sanity checks, T-Fuzz undergoes a sort of "transformation explosion", needing to fuzz too many different versions of transformed program to trigger the true bug (L2). While this is not very frequent in our experiments, it does happen: T-Fuzz failed to find the bugs in the remaining 7 binaries within the 24-hour time budget. We plan to explore these issues in T-Fuzz in our future work.

These results show that T-Fuzz greatly improves the performance of bug finding via fuzzing. By disabling the sanity checks, T-Fuzz finds bugs in 45 more CGC binaries than Driller. The additional bugs found by T-Fuzz are heavily guarded by hard checks and hidden in deep code paths of programs.

Of course, there is no requirement to use only a single analysis when performing fuzzing in the real world. The union of detections from T-Fuzz and Driller is *176* identified bugs, *significantly* higher than any other reported metric from experimentation done on the CGC dataset.

b) Comparison with other tools: In Steelix [13], 8 binaries of the CGC dataset were evaluated. Steelix only found an additional bug in KPACA_00001. As mentioned in the Steelix paper, the main challenge for fuzzing it is to bypass the check for magic values inside the program. Using a manually-provided seed that reaches the sanity check code in the program, Steelix detects the comparison against magic values in the program and successfully generates input that bypassed the sanity check, finding the bug in less than 10 minutes. T-Fuzz finds the bug in its first transformed program in around 15 minutes – without requiring manual effort to provide a seed that reaches the checking code.

In VUzzer [14], 63 of the CGC binaries were evaluated with a 6-hour time budget, among which VUzzer found bugs in 29. T-Fuzz found bugs in 47 of the 63 binaries, 29 of which were found within 6 hours. In the 29 binaries in which VUzzer found bugs, T-Fuzz found bugs in 23 of them (in 6 hours). T-Fuzz failed to find the bugs within a 24-hour time budget in the remaining 6 for the same reasons mentioned above (2 for L1 and 4 for L2). However, VUzzer is unable to run on the full CGC dataset, making a comprehensive comparison difficult.

B. LAVA-M Dataset

The LAVA dataset contains a set of vulnerable programs created by automatically injecting bugs using the technique proposed in [24]. **LAVA-M** is a subset of the LAVA dataset consisting of 4 utilities from coreutils, each of which contains multiple injected bugs. The authors evaluated a coverage guided fuzzing tool (FUZZER) and a symbolic analysis based tool (SES) for 5 hours [24]. The dataset was also used in VUzzer [14] and Steelix [13] as part of their evaluation. As at the time of this evaluation, Steelix is not available and VUzzer cannot be run (due to dependencies on a closed IDA plugin), we ran T-Fuzz for 5 hours on each of the binaries in LAVA-M dataset to compare our results with those stated by the authors of VUzzer and Steelix in their papers.

The evaluation results are summarized in Table II. The results of T-Fuzz is shown in the last column and results from other work are shown in other columns. It is important to note that the bugs mentioned here are bugs that have been confirmed by Crash Analyzer or manually. We first run Crash Analyzer on the reported bugs and then manually analyzed the ones marked as false positives by Crash Analyzer. From the results we can see that T-Fuzz found almost the same number of bugs as Steelix in `base64` and `uniq`, far more bugs in `md5sum` and less bugs in `who` than Steelix. The reasons are summarized as follows:

- Steelix and VUzzer performed promisingly well on `base64`, `uniq` and `who` because of two important facts. First of all, bugs injected into these programs are all protected by sanity checks on values copied from input against magic bytes **hardcoded** in the program. Thus, the static analysis tools used in Steelix or VUzzer can easily recover the expected values used in the sanity checks that guard the injected bugs. Secondly, the LAVA-M dataset provides well-formated seeds that help the fuzzer reach the code paths containing injected bugs. If either of the conditions fails to hold, Steelix and VUzzer would perform worse. However, with the inability to evaluate Steelix (which is unavailable) and VUzzer (which we could not run), this is hard to verify.
- T-Fuzz can trigger the bugs in code paths protected by "hard" checks which both Steelix and VUzzer can not bypass. In md5sum, T-Fuzz found 15 bugs that were not found by Steelix. These bugs are protected by sanity checks on values computed from the MD5 sum of specified files, instead of being copied from the input directly. As the expected values can no longer be constructed easily using the hardcoded values present in the program, Steelix failed to find these bugs.
- T-Fuzz performed worse than Steelix in `who` due to the limited time budget. As the number of injected bugs is huge and each injected bug is protected by a sanity check, T-Fuzz was limited by the emergent issue of "transformal explosion" and generated 578 transformed programs. Within 5 hours, T-Fuzz only found 63 bugs.

In summary, T-Fuzz performs well in comparison with state-of-art fuzzing tools in terms of bug finding even given conditions favorable for other counterparts. In the presence of "hard" checks on the input, e.g. checksums, T-Fuzz performs better than existing techniques for finding bugs "guarded" by such checks.

C. Real-world Programs

We evaluated T-Fuzz on a set of real-world program/library pairs (`pngfix/libpng`, `tiffinfo/libtiff`, `magick-/libMagickCore`, and `pdftohtml/libpoppler`) and compare it against AFL in terms of crashes found. Each program was fuzzed for 24 hours with random seeds of 32 bytes.

Table III summarizes the number of unique true-positive crashes found by T-Fuzz and AFL. T-Fuzz found 11, 124, 2 and 1 unique crashes in `pngfix`, `tiffinfo`, `magick` and `pdftohtml` respectively. AFL did not trigger any crashes in `pngfix`, `magick` and `pdftohtml` and only found less than half of the crashes T-Fuzz found in `tiffinfo`. As random seeds were provided, AFL got stuck very quickly, failing to bypass the sanity checks on the file type bytes (the file header for PNG files, the "II" bytes for TIF files, etc.). Within 24 hours, although AFL succeeded in generating inputs to bypass these checks, it failed to generate inputs that could bypass further sanity checks in the code, thus being unable to find bugs protected by them. In particular, in `pngfix`, `magick`, and `pdftohtml`, the bugs found by T-Fuzz are hidden in code paths protected by multiple sanity checks, and thus were not found by AFL; in `tiffinfo`, AFL found crashes, failing to find the 71 additional crashes caused by code paths that are guarded by more sanity checks.

These larger real-world programs demonstrate drawbacks of the underlying symbolic execution engine: angr simply does not have the environment support to scale to these programs. While this is not something that we can fix in the prototype without extensive effort by the angr team itself, our observation is that T-Fuzz actually causes surprisingly few false positives in practice. For example, for `pdftohtml`, the true positive bug was the only alert that T-Fuzz generated.

After inspecting the crashes resulting from T-Fuzz, we found 3 new bugs (marked by * in Table III): two in `magick` and one in `pdftohtml`. It is important to note that these bugs are present in the latest stable releases of these programs, which have been intensively tested by developers and security researchers. One of the new bugs found in `magick` has already been fixed in a recent revision, and we have reported the remaining 2 to the developers [25], [26], waiting to be acknowledged and fixed. Importantly, AFL failed to trigger any of these 3 bugs. As these bugs are hidden in code paths protected by several checks, it is very hard for AFL to generate inputs bypassing all of them. In constrast, T-Fuzz successfully found them by disabling checks that prevented the fuzzer-generated input to cover them.

D. False Positive Reduction

T-Fuzz utilizes a Crash Analyzer component to filter out false positive detections stemming from program transformations. This component is designed to avoid the situation, common in related fields such as static analysis, where a vulnerability detection tool overwhelms an analyst with false positives. In this section, we explore the need for this tool, in terms of its impact on the alerts raised by T-Fuzz.

TABLE II: LAVA-M Dataset evaluation results

program	Total # of bugs	FUZZER	SES	VUzzer	Steelix	T-Fuzz
base64	44	7	9	17	43	43
md5sum	57	2	0	1	28	49
uniq	28	7	0	27	24	26
who	2136	0	18	50	194	63

TABLE III: Real-world programs evaluation results, with crashes representing new bugs found by T-Fuzz in `magick` (2 new bugs) and `pdftohtml` (1 new bug) and crashes representing previously-known bugs in `pngfix` and `tiffinfo`.

Program	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*

False-positive-prone static analyses report false positive rates of around 90% for the analysis of binary software [21], [27]. Surprisingly, we have found that even in the presence of program transformations that could introduce unexpected behavior, T-Fuzz produces relatively few false-positive bug detections. We present an analysis of the alerts raised by T-Fuzz on a sampling of the CGC dataset and LAVA-M dataset in Table IV and Table V, along with the reports from our Crash Analyzer and ratio of false negatives.

In the CGC dataset, T-Fuzz provides the Crash Analyzer component with 2.8 alerts for every true positive detection on average, with a median of 2 alerts for every true positive. In the LAVA-M dataset, T-Fuzz only raised Crash Analyzer with 1.1 alerts for each true bug on average. Compared to static techniques, this is a significant advantage — even without the Crash Analyzer component, a human analyst would have to investigate only three alerts to locate an actual bug. Compared with other fuzzing techniques, even with the aggressive false positive reduction performed by the Crash Analyzer (resulting in only actual bug reports as the output of T-Fuzz), T-Fuzz maintains higher performance than other state-of-the-art systems.

As mentioned in Section IV-D, the Crash Analyzer may mark as false positives detections that actually *do* hint at a bug (but are not trivially repairable with the adopted approach), resulting false negative reports. E.g., if there is a "hard" check (e.g., checksum) that was disabled in the code path leading to a crash found by T-Fuzz, applying Crash Analyzer on the crash would involve solving hard constraint sets. As current constraint solvers can not determine the SATness of such hard constraint sets, Crash Analyzer would err on the false negative side and mark it as a false bug. Another example is shown in Section V-E. In the selected sample of CGC dataset shown in Table IV, Crash Analyzer mark detected crash in the first 3 binaries as false alerts. In LAVA-M dataset, Crash Analyzer has an average false negative rate of 15%. It shows a slightly higher false negative rate (30%) in `md5sum` because 15 of the detected crashes are in code paths protected by checks on MD5 checksums.

TABLE IV: A sampling of T-Fuzz bug detections in CGC dataset, along with the amount of false positive alerts that were filtered out by its Crash Analyzer.

Binary	# Alerts	# True Alerts	# Reported Alerts	% FN
CROMU_00002	1	1	0	100%
CROMU_00030	1	1	0	100%
KPRCA_00002	1	1	0	100%
CROMU_00057	2	1	1	0
CROMU_00092	2	1	1	0
KPRCA_00001	2	1	1	0
KPRCA_00042	2	1	1	0
KPRCA_00045	2	1	1	0
CROMU_00073	3	1	1	0
KPRCA_00039	3	1	1	0
CROMU_00083	4	1	1	0
KPRCA_00014	4	1	1	0
KPRCA_00034	4	1	1	0
CROMU_00038	5	1	1	0
KPRCA_00003	6	1	1	0

TABLE V: T-Fuzz bug detections in LAVA-M dataset, along with the amount of false positive alerts that were filtered out by its Crash Analyzer.

Program	# Alerts	# True Alerts	# Reported Alerts	% FN
base64	47	43	40	6%
md5sum	55	49	34	30%
uniq	29	26	23	11%
who	70	63	55	12%

E. Case Study

CROMU_00030[2] contains a stack buffer overflow bug (line 11) in a code path guarded by multi-stage "hard" checks. As shown in Listing 6, to reach the buggy code, the input needs to bypass 10 rounds of checks and each round includes a basic sanity check (line 19), a check on checksum (line 25) and a check on the request (line 30, in *handle_packet*).

When T-Fuzz fuzzes this binary, after roughly 1 hour of regular fuzzing,the fuzzer gets "stuck", failing to pass the check in line 25. T-Fuzz stops the fuzzer and uses the fuzzer-generated inputs to detect NCC candidates, pruning undesired candidates using the algorithm from Section IV-A, returning a set of 9 NCC candidates. Next T-Fuzz transforms the original program and generates 9 different binaries (shown as CROMU_00030_0-CROMU_00030_8 in Figure 6) with one detected NCC candidate removed in each. They are then fuzzed and transformed sequentially in FIFO order in the same way

[2] This program simulates a game over a protocol similar to IEEE802.11 and is representative of network programs.

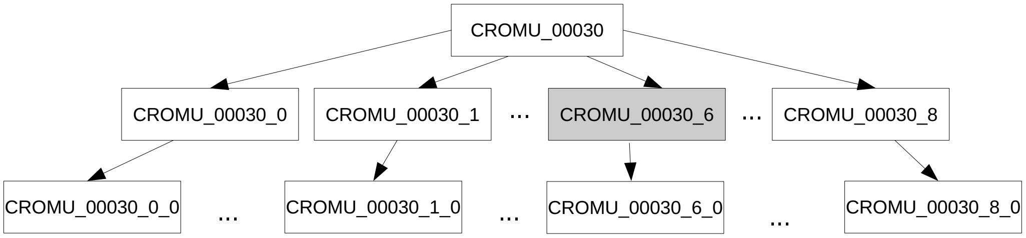


Fig. 6: The transformed binaries T-Fuzz generates and fuzzes

```
1  int main() {
2    int step = 0;
3    Packet packet;
4    while (1) {
5      memset(packet, 0, sizeof(packet));
6      if (step >= 9) {
7         char name[5];
8         // stack buffer overflow BUG
9         int len = read(stdin, name, 25);
10        printf("Well done, %s\n", name);
11        return SUCCESS;
12     }
13     // read a packet from the user
14     read(stdin, &packet, sizeof(packet));
15     // initial sanity check
16     if(strcmp((char *)&packet, "1212") == 0) {
17       return FAIL;
18     }
19     // other trivial checks on the  packet
       omitted
20     if (compute_checksum(&packet) != packet.
       checksum) {
21       return FAIL;
22     }
23     // handle the request from the user, e.g.,
       authentication
24     if (handle_packet(&packet) != 0) {
25       return FAIL;
26     }
27     // all tests in this step passed
28     step ++;
29   }
30 }
```

Listing 6: Code excerpt of CROMU_00030, slightly simplified for readability

as the original. When CROMU_00030_6 (marked as grey in Figure 6), which is the binary with the check in line 8 negated, is fuzzed, a crash is triggered within 2 minutes, which is the true bug in the original binary. The total time it takes T-Fuzz to find the bug is about 4 hours, including the time used for fuzzing the original binary (CROMU_00030) and the time for fuzzing CROMU_00030_0-CROMU_00030_5. After the real bug is found by fuzzing CROMU_00030_6, T-Fuzz continues to fuzz and transform other (transformed) binaries until it uses up its time budget.

It is important to note that T-Fuzz can also find the bug by fuzzing the transformed binary with the sanity checks in line 25 and 30 negated. In that case, all the user provided input “bypasses” these two complex checks and the buggy code in line 11 is executed after looping for 10 iterations. As we fuzz the transformed binaries in FIFO order, this configuration is not reached within the first 24 hours.

In contrast, Driller failed to find the bug in this binary. Driller’s symbolic execution engine cannot produce an accurate input to bypass the check in line 25, as it is too complex. Unable to get inputs to guide execution through the sanity check, the fuzzer blindly mutates the inputs without finding any new paths until it uses up its time budget. Note also that it is highly unlikely for Driller to generate input to bypass the check in line 30 even without the check in line 25 because of the complexity involved in encoding the state of the protocol.

Listing 6 also showcases an example where Crash Analyzer marks a true bug as a false positive. As the *step* variable is not read from user and initialized as 0 in line 2, when the Crash Analyzer reaches the crash in CROMU_00030_6, the accumulated constraints set is $\{step == 0, step >= 9\}$ which is UNSAT, thus it is marked as false positive. This bug was identified by manual analysis.

VI. Related work

A large body of related work has tried to improve the efficiency of fuzzing. For example, Rebert et al. [28] and Woo et al.[29] propose empirical seed selection and fuzzing scheduling algorithms to find more bugs in a given limited computation budget, in AFLFast [30] and AFLGo [31] Böhme et al. model fuzzing as a Markov chain, assigning more fuzzing energy on low-frequency paths. The directions of such work are orthogonal to the fundamental challenge that fuzzer-generated inputs cannot bypass the complex sanity checks in programs. Thus in this section, we focus our discussion on related work that improves fuzzing by bypassing sanity checks in programs.

A. Feedback based Approaches

Feedback based approaches make heuristics of possible magic values and their positions in the input based on feedback from the target program. E.g., AFL [8] and libFuzzer [32] can automatically guess syntax tokens based on the change in coverage and mutate input based on those tokens [33]. Further, AFL-lafintel [34] and improve the feedback by dividing a check on multiple bytes values into multiple nested checks on one byte values and Steelix [13] introduces “comparison progress” of checks to the feedback.

These approaches are based on *hindsight* to extend coverage past a check, i.e., the fuzzer must have already generated/mutated an input that passes some check in the program. Also, these approaches cannot handle checks on values computed on the fly or based on other input values such as checksums. T-Fuzz, on the other hand, is not limited by such restrictions.

B. Symbolic and Concolic Execution based Approaches

Symbolic execution encodes the sanity checks along a program path as a set of constraints (represented as logical formula), reducing the problem of finding a passing input to solving the encoded constraints. Several tools have implemented symbolic execution, e.g., KLEE [35], Veritesting [36], SAGE [37], DART [38], SYMFUZZ [39], CUTE [40], SmartFuzz [41], and Driller [12], covering domains like automatic testcase generation, automatic bug finding and fuzzing.

Among the tools mentioned above, Driller [12] is the closest to T-Fuzz. Driller uses selective concolic execution to generate inputs when the fuzzer gets "stuck".As mentioned in previous sections, symbolic and concolic execution based approaches, including Driller, suffer in scalability and ability to cover code paths protected by "hard" checks, where T-Fuzz excels.

C. Taint Analysis based Approaches

Dynamic taint analysis identifies the dependencies between the program logic and input. Taintscope [15] focuses mutating the security-sensitive parts of the input identified by dynamic taint analysis. Other works apply additional analysis based on the identified dependencies to improve input generation, e.g., VUzzer uses data-flow and control analysis, Dowser [42] and BORG [43] use symbolic analysis.

Dynamic taint analysis is heavy weight, application of other techniques (data and control flow analysis in VUzzer and symbolic analysis in Dowser and BORG) adds up the overhead. In contrast,T-Fuzz only uses lightweight dynamic tracing technique for identifying and disabling sanity checks in the target program.

D. Learning based Approaches

This category of approaches generate inputs by learning from large amount of valid inputs. E.g., Skyfire [44] and Learn&Fuzz [45] generate seeds or inputs for the fuzzer using the learned probabilistic distribution of different values from samples, while GLADE [46] generates inputs based on the synthesized grammar learned from provided seeds.

Learning based approaches has been shown to be effective in generating well structured inputs (e.g., XML files) for fuzzers in Skyfire [44] and Learn&Fuzz [45]. However, it is difficult to learn less structured inputs like images or complex dependencies among different parts of data like checksums without external knowledge. In addition, learning requires a large corpus of valid inputs as training set. In contrast, T-Fuzz does not have such limitations.

E. Program transformation based approaches

Some existing work uses the idea of program transformation to overcome sanity checks in the target program, but requires significant **manual** effort. E.g., Flayer [47] relies on user provided addresses of the sanity checks to perform transformation. TaintScope [15] depends on a on a pair of inputs with one able to bypass a sanity check on checksum and the other not, requiring a significant amount of manual analysis. MutaGen [48] depends on the availability and identification of program code that can generate the proper protocols, a process involving much manual effort. In addition, they use dynamic instrumentation to alter the execution of the target, which typically involves a slowdown of 10x in execution speed.

T-Fuzz is the only program transformation-based technique, known to us, that is able to leverage program transformation, in a completely automated way, to augment the effectiveness of fuzzing techniques.

VII. Conclusions

Mutational fuzzing so far has been limited to producing new program inputs. Unfortunately, hard checks in programs are almost impossible to bypass for a mutational fuzzer (or symbolic execution engine). Our proposed technique, transformational fuzzing, extends the notion of mutational fuzzing to the program as well, mutating both program and input.

In our prototype implementation *T-Fuzz* we detect whenever a baseline mutational fuzzer (AFL in our implementation) gets stuck and no longer produces inputs that extend coverage. Our lightweight tracing engine then infers all checks that could not be passed by the fuzzer and generates mutated programs where the checks are negated. This change allows our fuzzer to produce input that trigger deep program paths and therefore find vulnerabilities hidden deep in the program itself.

We have evaluated our prototype on the CGC dataset, the LAVA-M dataset, and 4 real-world programs (`pngfix`, `tiffinfo`, `magick` and `pdftohtml`). In the CGC dataset, T-Fuzz finds true bugs in 166 binaries, improving the results by 45 binaries compared to Driller and 61 binaries compared to AFL alone. In the LAVA-M dataset, T-Fuzz shows significantly better performance in the presence of "hard" checks in the target program. In addition, we have found 3 new bugs in evaluating real-world programs: two in `magick` and one in `pdftohtml`. T-Fuzz is available at https://github.com/HexHive/T-Fuzz.

Future work includes developing heuristics on combining program transformation and input mutation to better cover the large search space and better approaches to filter the set of candidate crashes.

Acknowledgment

This material is based in part upon work supported by the National Science Foundation under award CNS-1513783, by ONR awards N00014-17-1-2513 and N00014-17-1-2897, and by Intel Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] R. McNally, K. Yiu, D. Grove, and D. Gerhardy, "Fuzzing: the state of the art," DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), Tech. Rep., 2012.

[2] P. Godefroid, “From blackbox fuzzing to whitebox fuzzing towards verification,” in *Presentation at the 2010 International Symposium on Software Testing and Analysis*, 2010.

[3] Microsoft, “Microsoft security development lifecycle,” 2017, [Online; accessed 10-April-2017]. [Online]. Available: https://www.microsoft.com/en-us/sdl/process/verification.aspx

[4] Google, “Oss-fuzz - continuous fuzzing for open source software,” 2016, [Online; accessed 10-April-2017]. [Online]. Available: https://github.com/google/oss-fuzz

[5] J. Röning, M. Lasko, A. Takanen, and R. Kaksonen, “Protos-systematic approach to eliminate software vulnerabilities,” *Invited presentation at Microsoft Research*, 2002.

[6] D. Aitel, “An introduction to spike, the fuzzer creation kit,” *presentation slides), Aug*, vol. 1, 2002.

[7] M. Eddington, “Peach fuzzing platform,” *Peach Fuzzer*, p. 34, 2011.

[8] M. Zalewski, “american fuzzy lop,” 2017, [Online; accessed 1-August-2017]. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[9] Google, “Honggfuzz,” 2017, [Online; accessed 10-April-2017]. [Online]. Available: https://google.github.io/honggfuzz/

[10] caca labs, “zzuf - multi-purpose fuzzer,” 2017, [Online; accessed 10-October-2017]. [Online]. Available: http://caca.zoy.org/wiki/zzuf

[11] M. Zalewski, “The bug-o-rama trophy case,” 2017, [Online; accessed 20-Septempber-2017]. [Online]. Available: http://lcamtuf.coredump.cx/afl/#bugs

[12] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution.” in *NDSS*, vol. 16, 2016, pp. 1–16.

[13] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, “Steelix: program-state based binary fuzzing,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 627–637.

[14] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, “Vuzzer: Application-aware evolutionary fuzzing,” in *NDSS*, 2017.

[15] T. Wang, T. Wei, G. Gu, and W. Zou, “Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection,” in *SP’10*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 497–512. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.37

[16] D. Brumley, P. Poosankam, D. Song, and J. Zheng, “Automatic patch-based exploit generation is possible: Techniques and implications,” in *SP’08*. IEEE, 2008, pp. 143–157.

[17] M. Zalewski, “Technical whitepaper for afl-fuzz,” 2017, [Online; accessed 1-August-2017]. [Online]. Available: http://lcamtuf.coredump.cx/afl/technical_details.txt

[18] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley, “Your exploit is mine: Automatic shellcode transplant for remote exploits,” in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 824–839.

[19] github, “tracer: Utilities for generating dynamic traces,” 2017, [Online; accessed 21-Oct-2017]. [Online]. Available: https://github.com/angr/tracer

[20] radare2 team, “radare2: unix-like reverse engineering framework and commandline tools,” 2017, [Online; accessed 21-Oct-2017]. [Online]. Available: https://github.com/radare/radare2

[21] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “Sok: (state of) the art of war: Offensive techniques in binary analysis,” in *SP’16*, May 2016, pp. 138–157.

[22] L. Lunge Technology, “Cgc corpus,” 2017, [Online; accessed 20-September-2017]. [Online]. Available: http://www.lungetech.com/2017/04/24/cgc-corpus/

[23] S. Group, “Driller: augmenting afl with symbolic execution!” 2017, [Online; accessed 20-Septempber-2017]. [Online]. Available: https://github.com/shellphish/fuzzer

[24] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, “Lava: Large-scale automated vulnerability addition,” in *SP’16)*, May 2016, pp. 110–121.

[25] I. team, “assertion error in relinquishmagickresource,” 2018, [Online; accessed 23-Jan-2018]. [Online]. Available: https://github.com/ImageMagick/ImageMagick/issues/955

[26] bugzilla, “Bug 104798 - endless loop resulting oom,” 2018, [Online; accessed 26-Jan-2018]. [Online]. Available: https://bugs.freedesktop.org/show_bug.cgi?id=104798

[27] D. A. Ramos and D. R. Engler, “Under-constrained symbolic execution: Correctness checking for real code.” in *USENIX Security Symposium*, 2015, pp. 49–64.

[28] A. Rebert, S. K. Cha, T. Avgerinos, J. M. Foote, D. Warren, G. Grieco, and D. Brumley, “Optimizing seed selection for fuzzing.” USENIX, 2014.

[29] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, “Scheduling black-box mutational fuzzing,” in *CCS*. ACM, 2013, pp. 511–522.

[30] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based greybox fuzzing as markov chain,” in *CCS*. ACM, 2016, pp. 1032–1043.

[31] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, “Directed greybox fuzzing,” in *CCS*, 2017, pp. 1–16.

[32] L. team, “libfuzzer – a library for coverage-guided fuzz testing,” 2018, [Online; accessed 28-Jan-2018]. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[33] M. Zalewski, “afl-fuzz: making up grammar with a dictionary in hand,” 2015, [Online; accessed 16-September-2017]. [Online]. Available: https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html

[34] afl lafintel, “Circumventing fuzzing roadblocks with compiler transformations,” 2017, [Online; accessed 16-September-2017]. [Online]. Available: https://lafintel.wordpress.com/

[35] C. Cadar, D. Dunbar, D. R. Engler *et al.*, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.” in *OSDI*, vol. 8, 2008, pp. 209–224.

[36] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, “Enhancing symbolic execution with veritesting,” in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1083–1094.

[37] P. Godefroid, M. Y. Levin, and D. Molnar, “Sage: whitebox fuzzing for security testing,” *Queue*, vol. 10, no. 1, p. 20, 2012.

[38] P. Godefroid, N. Klarlund, and K. Sen, “Dart: directed automated random testing,” in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 213–223.

[39] S. K. Cha, M. Woo, and D. Brumley, “Program-adaptive mutational fuzzing,” in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 725–741.

[40] K. Sen, D. Marinov, and G. Agha, “Cute: A concolic unit testing engine for c,” in *ESEC/FSE-13*. New York, NY, USA: ACM, 2005, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081750

[41] D. Molnar, X. C. Li, and D. Wagner, “Dynamic test generation to find integer bugs in x86 binary linux programs.” in *SEC*, vol. 9, 2009, pp. 67–82.

[42] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, “Dowser: a guided fuzzer to find buffer overflow vulnerabilities,” in *SEC’13*, 2013, pp. 49–64.

[43] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos, “The borg: Nanoprobing binaries for buffer overreads,” in *SP’15*. ACM, 2015, pp. 87–97.

[44] J. Wang, B. Chen, L. Wei, and Y. Liu, “Skyfire: Data-driven seed generation for fuzzing,” 2017.

[45] P. Godefroid, H. Peleg, and R. Singh, “Learn&fuzz: Machine learning for input fuzzing,” *CoRR*, vol. abs/1701.07232, 2017. [Online]. Available: http://arxiv.org/abs/1701.07232

[46] O. Bastani, R. Sharma, A. Aiken, and P. Liang, “Synthesizing program input grammars,” in *PLDI 2017*. New York, NY, USA: ACM, 2017, pp. 95–110. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062349

[47] W. Drewry and T. Ormandy, “Flayer: Exposing application internals.” *WOOT*, vol. 7, pp. 1–9, 2007.

[48] U. Kargén and N. Shahmehri, “Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 782–792.