From Reaching to Triggering: Intelligent Directed Fuzzing for Vulnerability Discovery

Long Wang
Institute for Network Science and Cyberspace (INSC)
Tsinghua University

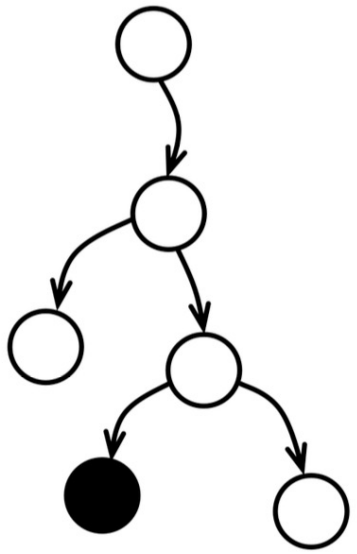
At 90th IFIP WG 10.4 Meeting
June 27, 2026

What Is Fuzzing?

- Fuzzing is an automated testing technique that finds bugs by feeding a program with many generated or mutated inputs.
- How it works
  - Generate or mutate test inputs
  - Run the target program
  - Monitor crashes, hangs, or memory errors
  - Keep inputs that expose new behaviors
- Key message:
  - Fuzzing discovers bugs by exploring how programs react to unexpected inputs

Directed Fuzzing for Vulnerability Discovery

- Directed fuzzing aims to test specific target code sites in programs
  - crash reproduction
  - candidate vulnerability confirmation
  - patch testing



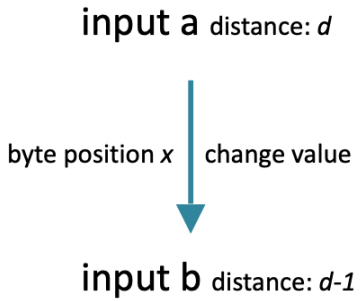
Reach target code more quickly!

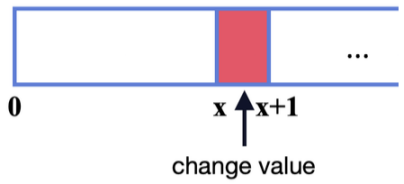
But reaching a target line is not sufficient

- Previous focus:
  - seed scheduling, path pruning
- Remaining gap:
  - Reaching target lines faster by means of identifying critical fields of the input
  - Arriving at the target does not mean the bug condition is satisfied; triggering it is needed

Reaching targets faster by identifying critical input fields

- We can identify ***critical fields*** for mutation based on ***past mutations***





- ***critical field*** of input a: offset *x*, length *1*
- Mutate the critical field directly rather than making blind attempts!!!

Key Observations

1: Learn the high-level pattern through **a neural network model**

2: learn to reach an unexplored branch from **covered sibling branches**

IDFuzz: Intelligent Directed Grey-box Fuzzing

(USENIX Security 25)

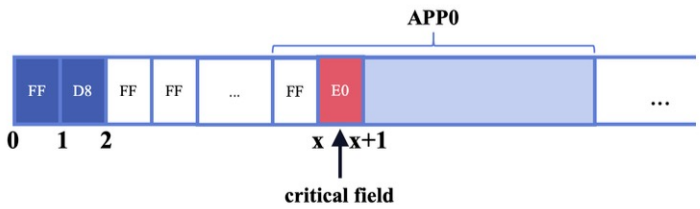
- Key Ideas
  - Model the relationship between input bytes and input distance using an NN
  - Identify the critical bytes/fields by locating the maximum model gradients

Example

```
1  static int SectionsRead;
2  int ReadJpegSections(FILE* infile, ReadMode_t ReadMode) {
3      int a = fgetc(infile);
4      if (a != 0xff || fgetc(infile) != M_SOI) return FALSE;
5      for(;;) {
6          int prev = 0;
7          int marker = 0;
8          for (a=0;;a++) {
9              marker = fgetc(infile);
10             if (marker != 0xff && prev == 0xff) break;
11             if (marker == EOF) {
12                 ErrFatal(''Unexpected end of file'');
13             }
14             prev = marker;
15         }
16         Sections[SectionsRead].Type = marker;
17         SectionsRead += 1;
18         switch(marker) {
19             case M_SOS: ...
20             case M_DQT: ...
21             case M_DHT: ...
22             ...
23             case M_COM:
24                 // target
25             ...
26             case M_SOF15: ...
27             default: ...
28         }
29     }
30     return TRUE;
31 }
```

Listing 1: A motivating example.

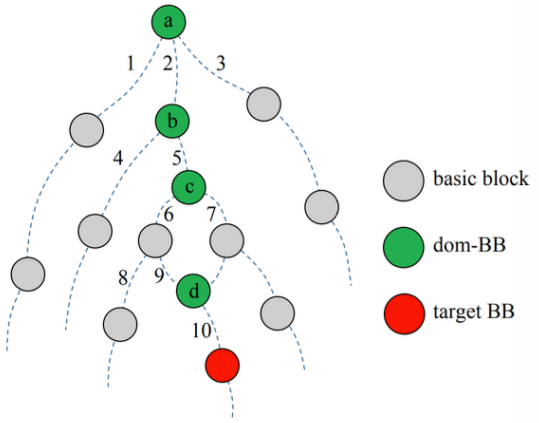
- Part of the *jhead* program for reading a JPEG format input file
- Mutate the **marker** byte to **M_COM**
  - ➢ offset: the 1st non-0xff byte
  - ➢ length: 1



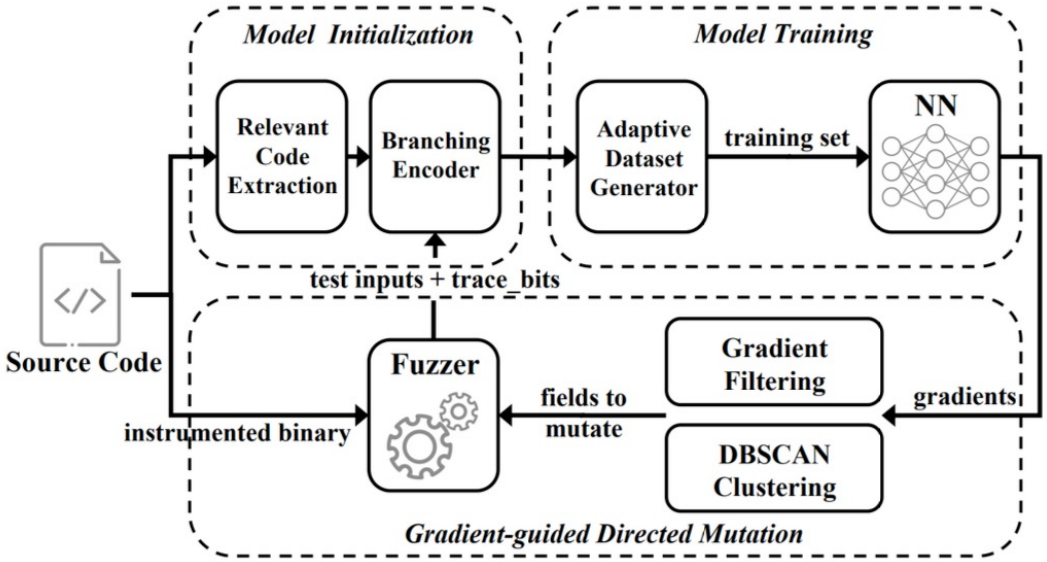
Modeling input byte-target distance Relationship

- Model Input: Input byte values
- Model Output: ???
  - ➢ distance value -> inaccurate **X**
- Relevant Code Extraction
  - ➢ modeling the coverage of only **dominating basic blocks**
- Branching Encoding
  - ➢ 0 for uncovered, 1 for covered
  - ➢ A proper fraction between 0 and 1 for covered sibling branches
- Build a neural network model for capturing the relationships

"1" : < 1 ,0.25, 0 ,0 ,0 >
"2→4" : < 1 , 1 ,0.5, 0 , 0 >
"2→5→6→8" : < 1 , 1 , 1 , 0 , 0 >

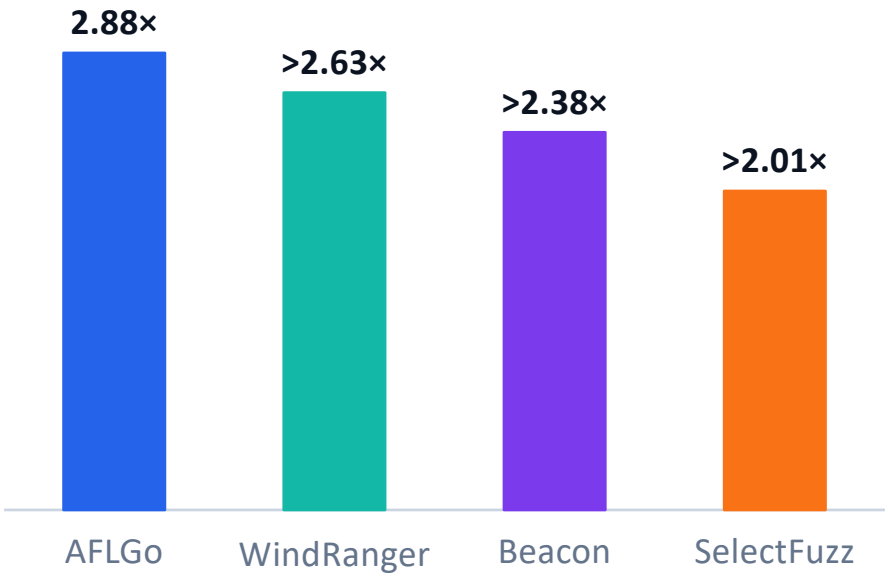


IDFuzz Workflow/Architecture



- **Model Initialization**
- **Model Training**
- **Gradient-guided Directed Mutation**

IDFuzz Results (1)



Speedup for hard-to-reproduce targets

>2.48× average speedup

91.86% fewer ineffective mutations

Only 6% runtime overhead

6 new bugs + 1 incomplete fix

Interpretation: IDFuzz improves the first half of the story — it helps directed fuzzers spend mutations on fields that matter for reaching the target.

IDFuzz Results (2)

- 6 new vulnerabilities + 1 incomplete fix
  - (4 CVEs, 2 high-severity vulnerabilities)

Project	ID	Vuln. Type	Status	CVE ID
tcpreplay	#1	NULL Pointer Dereference	patched	CVE-2023-43279
	#2	Infinite Loop	patched	CVE-2024-22654
gifsicle	#3	Floating Point Exception	patched	CVE-2023-46009 (High Severity)
yasm	#4	Memory Leak	patched	
	#5	NULL Pointer Dereference	patched	CVE-2024-22653
w3m	#6	OOB Read	reported	
	#7	OOB Write	patched	Incomplete fix for CVE-2022-38223 (High Severity)

Fast bug triggering is needed

- Existing tools often degenerate into coverage-guided fuzzing after reaching the target.
- Triggering depends on semantic constraints: values, execution order, and vulnerability type.
- Prior heuristics such as templates or call traces are too coarse for many targets.

TrigFuzz: Triggering Conditions Guided Directed Fuzzing

(IEEE S&P 26)

- Key Ideas
  - Define quantifiable trigger conditions
  - Design a mechanism to use the trigger condition values for guiding the triggering-oriented fuzzing
    - While still preserving the reachability

Motivating Example

```
#include <stdio.h>

void foo() {
  int a, b, c, d;
  int v = sscanf(input.field,
    "%d.%d.%d.%d", &a, &b, &c, &d);
  use(b); // use-of-uninitialized-variable
}
```

Human/LLM inference →

Triggering condition

$v < 2$

- sscanf initializes fields from left to right.
- If fewer than 2 assignments succeed, b may be uninitialized.
- Fuzzing should prefer inputs that reduce v and mutate input.field bytes.

Define Triggering Conditions

TCU = <cond, loc, seq, conj, w>

cond	loc	seq	conj	w
logical predicate	code location	execution order	AND/OR group	runtime weight

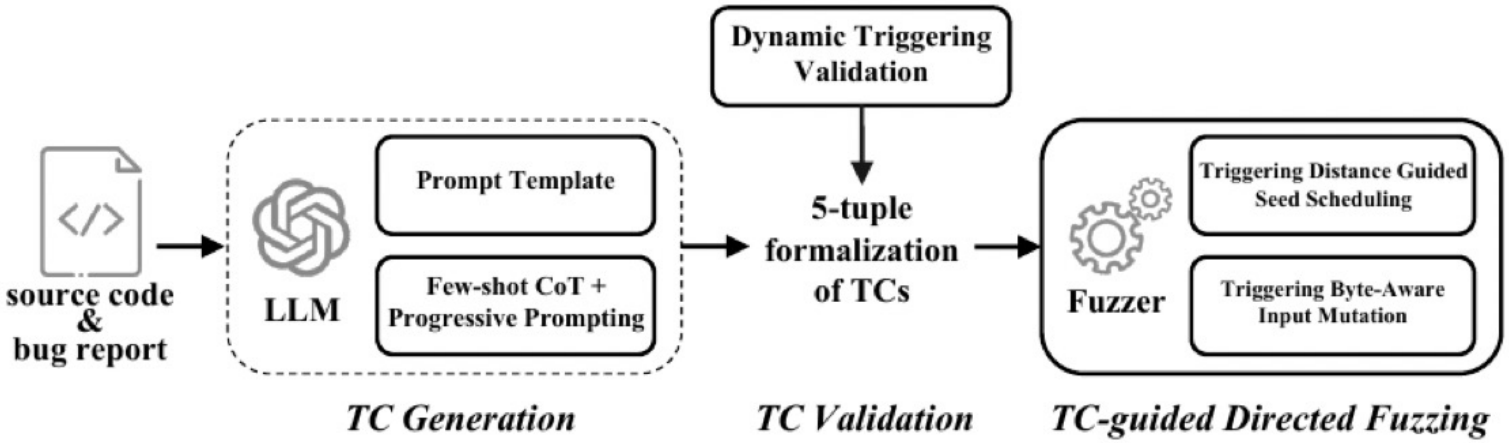
Why this representation works

- Captures value constraints and ordering across multi-point bugs.
- Decomposes complex predicates into DNF-style primitive units.
- Maps each unit into a numeric triggering distance.

Example distance rules

```
a < b  ->  a - b
a == b -> |a - b|
ptr == NULL -> 0 or 1
```

TrigFuzz Workflow/Architecture



1 TC generation

LLM analyzes source code and bug report; outputs formal triggering condition units.

2 TC validation

Online dynamic validation filters unreliable conditions and estimates weights.

3 TC-guided fuzzing

Triggering distance drives seed scheduling and byte-aware mutation.

TrigFuzz Results (1)

30

targets triggered by TRIGFUZZ

>1.72x

avg. speedup over AFLGo

>2.02x

avg. speedup over SDFuzz

- 7 new vulnerabilities in libarchive / gpac
  - 2 CVEs and 5 confirmed by software authors

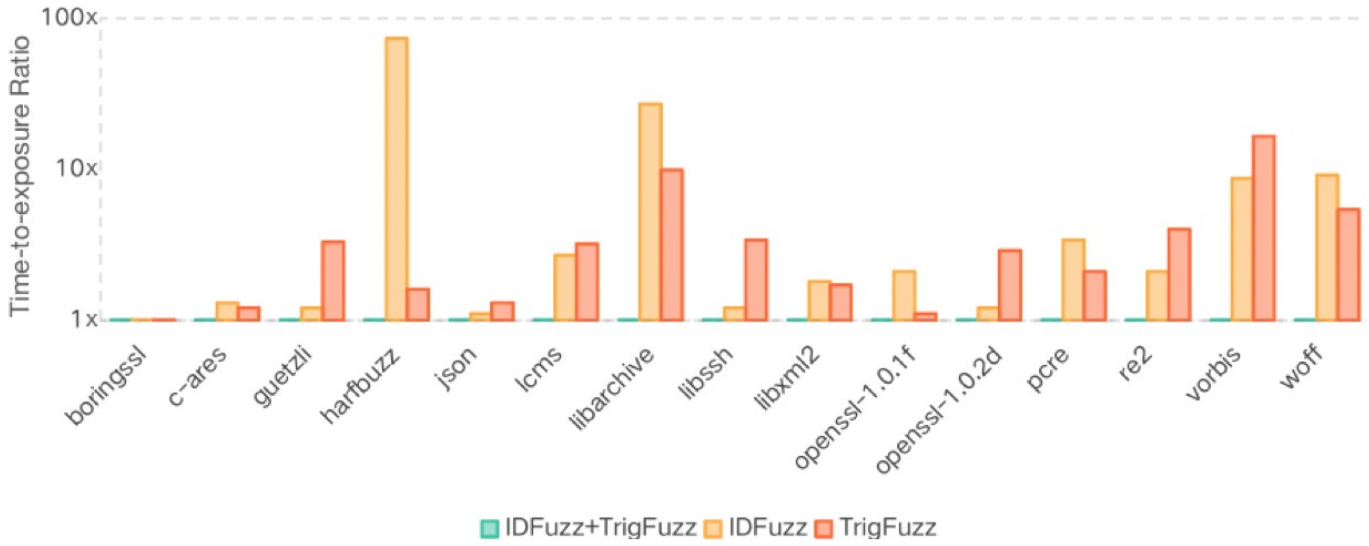
Case Study: libarchive ISO 9660 Joliet heap out-of-bounds write

Root cause: isoent_gen_joliet_identifier() lacks a boundary check when computing the noff offset. As a result, noff can become negative, causing the pointer to write before the beginning of the buffer.
Triggering condition: noff < 0.
Result: TrigFuzz successfully triggered the vulnerability within 24 hours, whereas AFLGo failed to trigger it within 48 hours.

TrigFuzz Results (2)

- Integration of IDFuzz and TrigFuzz

3.14x faster than IDFuzz alone **2.73x** faster than TrigFuzz alone



Conclusions

- From reaching to triggering: a complete view of directed fuzzing
  - IDFuzz accelerates target reachability by identifying critical input fields with neural-network gradients.
  - TrigFuzz accelerates vulnerability triggering by generating, validating, and exploiting formal triggering conditions.
- The two techniques are complementary:
  - IDFuzz guides mutations toward the target, while
  - TrigFuzz guides mutations toward the bug condition
- Together, they move directed fuzzing from simple target-guided reachability to efficiently exploiting real vulnerabilities

Thanks a lot!

Contact: Long Wang
longwang@tsinghua.edu.cn