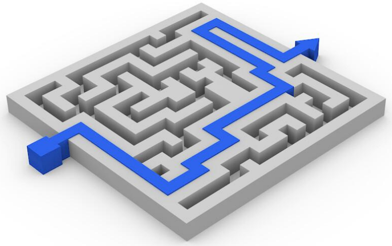
One Fuzz Doesn’t Fit All: Optimizing Directed Fuzzing via Target-tailored Program State Restriction

Prashast Srivastava, Stefan Nagy, Matthew Hicks, Antonio Bianchi, Mathias Payer

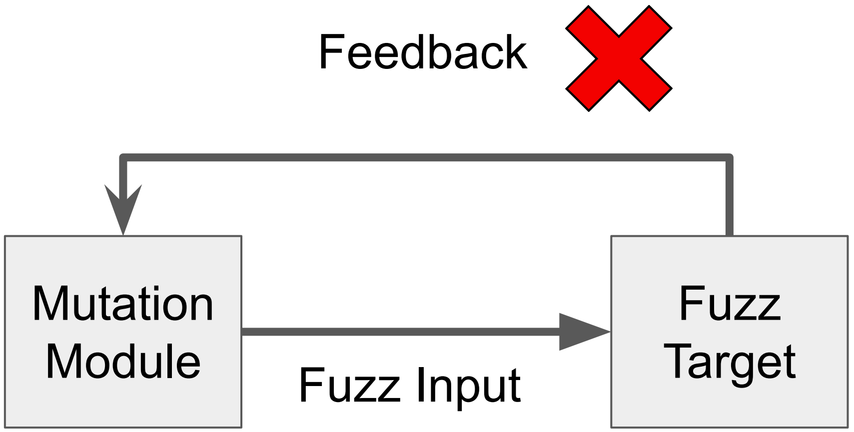
Motivation

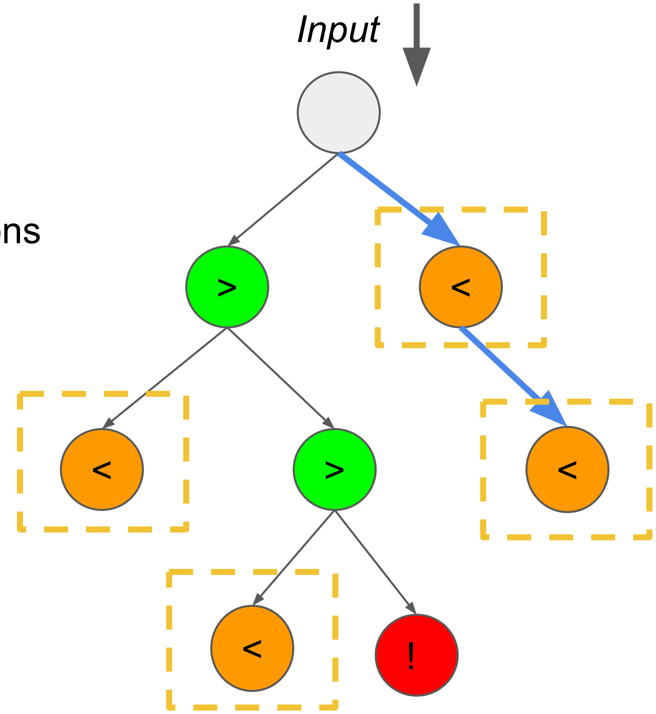
- Fuzzing is a highly effective dynamic testing methodology
- Off-the-shelf fuzzers are unsuitable for targeted testing
- Directed fuzzing designed to test specific code locations
- Existing approaches are wasteful, calling for an effective, lightweight solution



Directed Fuzzing

- Existing directed fuzzers employ distance minimization
- Distance minimization biases fuzzers towards targeted locations
- Wasteful exploration of target-unreachable code regions

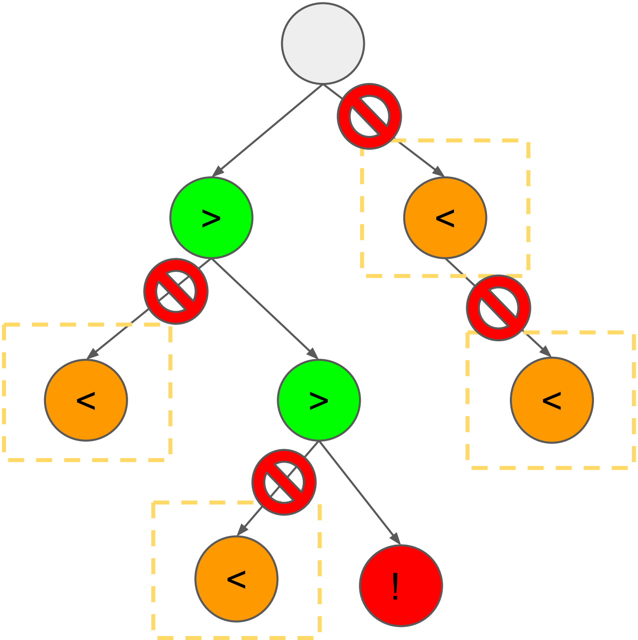




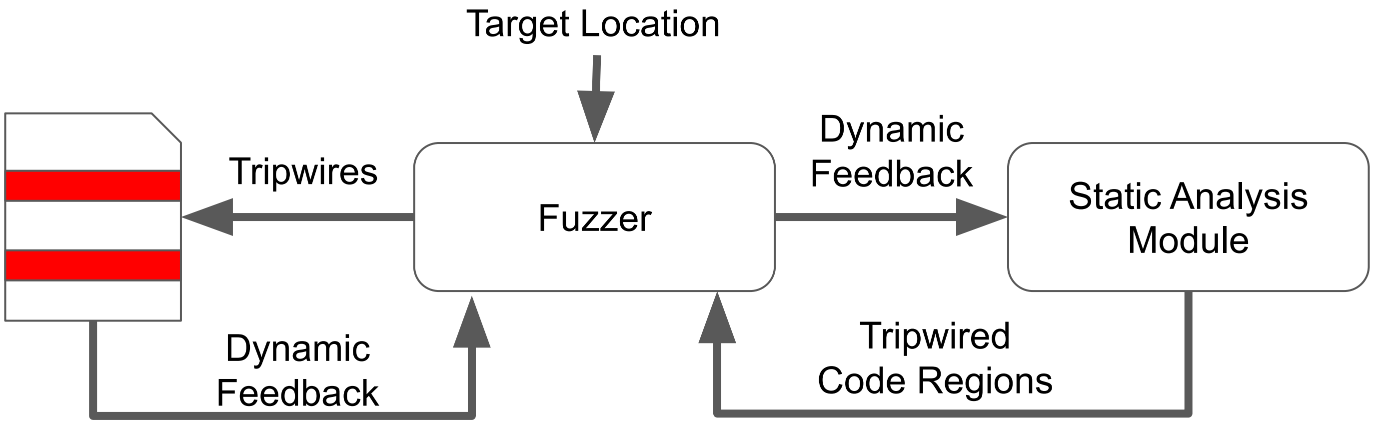
Problem: Existing directed fuzzers needlessly explore code regions that cannot reach the target

Observation: Current exploration schemes are ill-suited for *disjoint* target locations

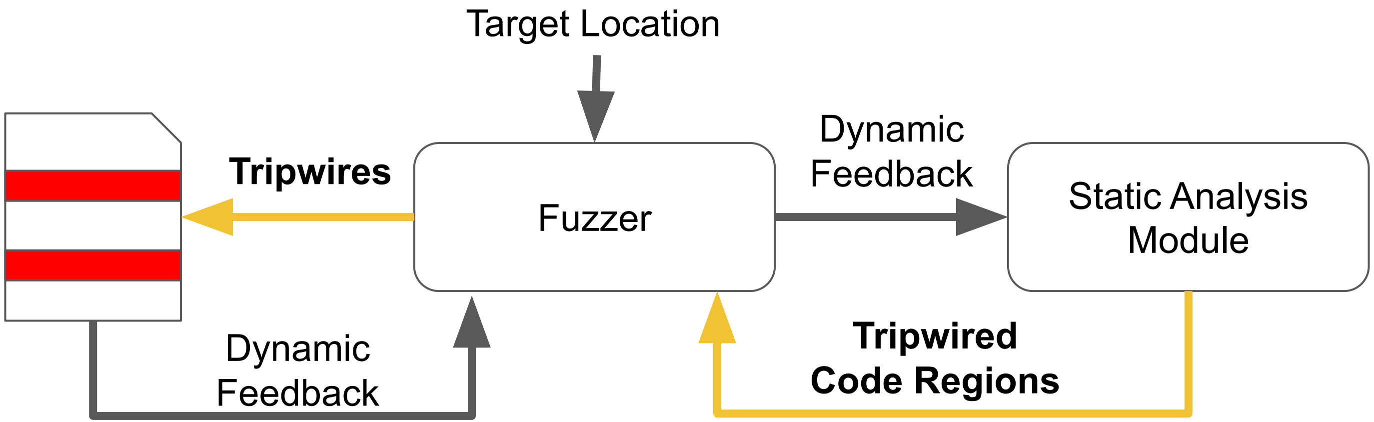
Solution: Terminate execution when target becomes unreachable by ***tripwiring*** dead ends



Tripwiring-directed Fuzzing

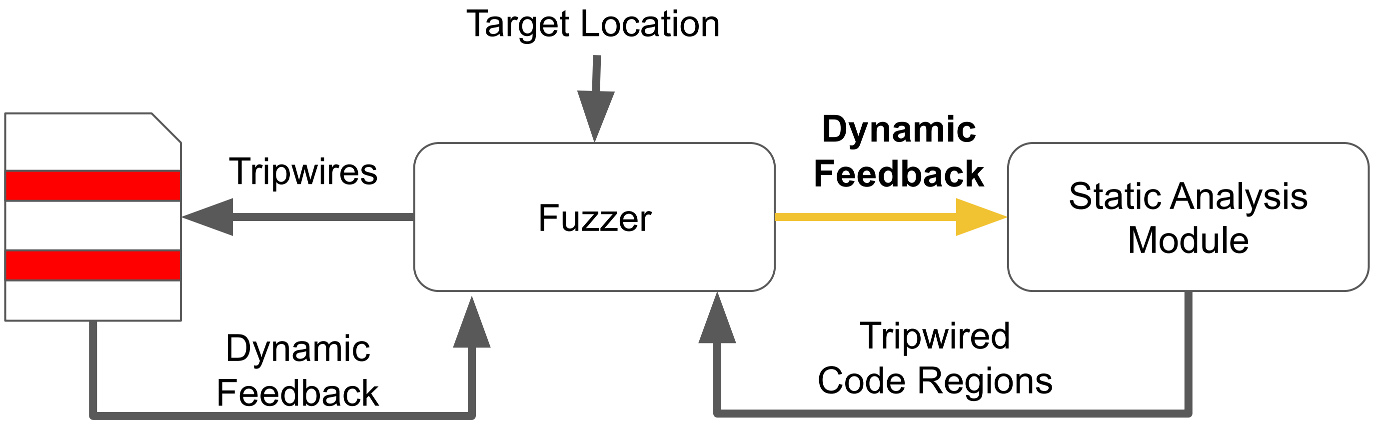


Tripwiring-directed Fuzzing



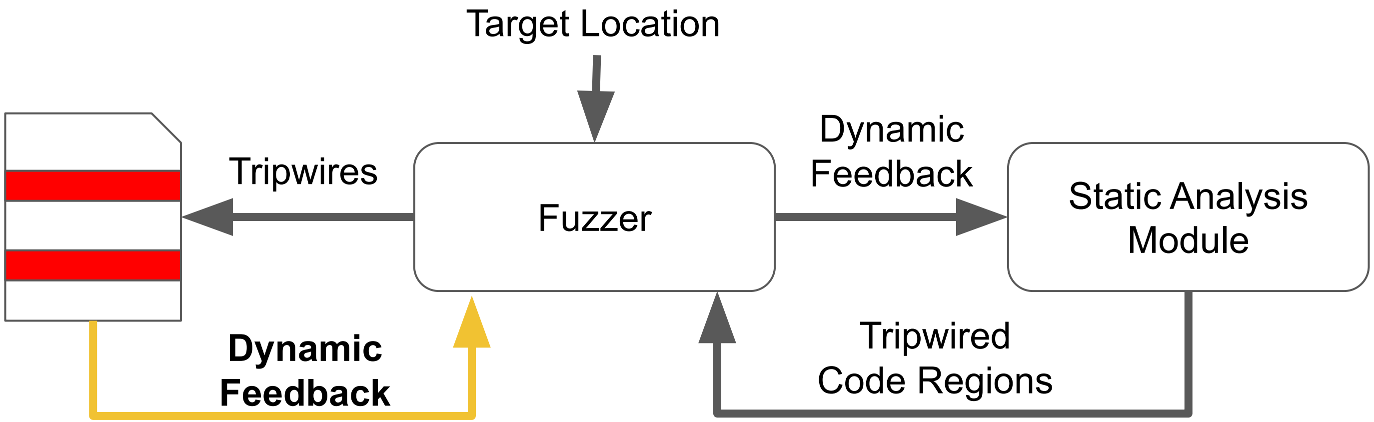
- Lightweight reachability analysis statically identifies target-reachable code regions

Tripwiring-directed Fuzzing



- Lightweight reachability analysis statically identifies target-reachable code regions
- Refine target-reachable code regions with dynamically observed indirect call edges

Tripwiring-directed Fuzzing



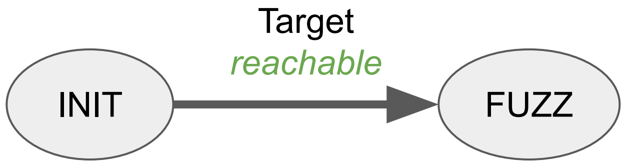
- Lightweight reachability analysis statically identifies target-reachable code regions
- Refine target-reachable code regions with dynamically observed indirect call edges
- Prioritize mutation of test cases with greater coverage of target-relevant code regions

SieveFuzz — Tripwiring-directed fuzzer

- Extends AFL++ and uses SVF for static analysis

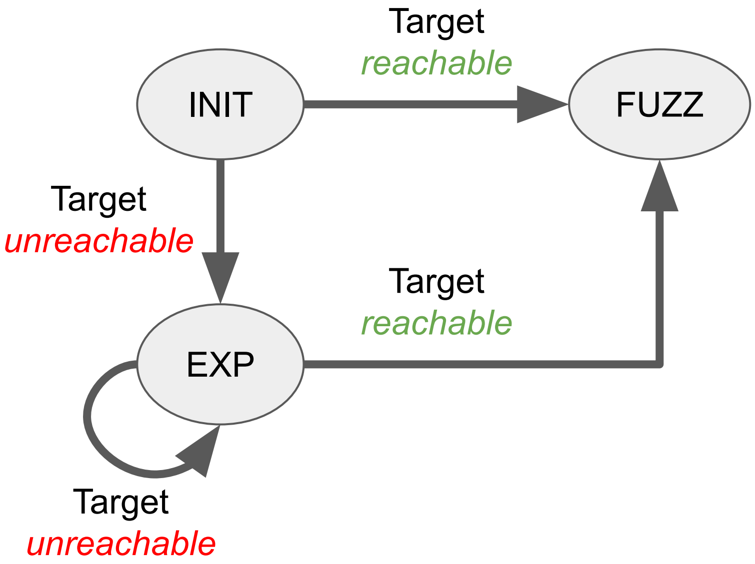
SieveFuzz — Tripwiring-directed fuzzer

- Extends AFL++ and uses SVF for static analysis
- High-level workflow consists of three separate stages:
  - *INIT*: Query whether the target is reachable from the fuzz target entry point
  - *FUZZ*: Identify target-unreachable regions and perform tripwired fuzzing updated with dynamic indirect call information



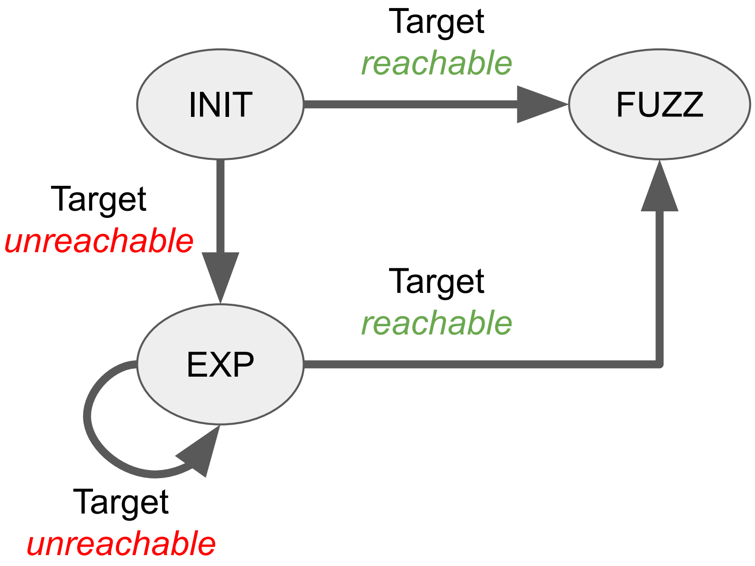
SieveFuzz — Tripwiring-directed fuzzer

- Extends AFL++ and uses SVF for static analysis
- High-level workflow consists of three separate stages:
    - *INIT*: Query whether the target is reachable from the fuzz target entry point
    - *FUZZ*: Identify target-unreachable regions and perform tripwired fuzzing updated with dynamic indirect call information
    - *EXP*: Perform undirected fuzzing to recover indirect edges until target becomes reachable



SieveFuzz — Tripwiring-directed fuzzer

- Extends AFL++ and uses SVF for static analysis
- High-level workflow consists of three separate stages:
  - *INIT*: Query whether the target is reachable from the fuzz target entry point
  - *FUZZ*: Identify target-unreachable regions and perform tripwired fuzzing updated with dynamic indirect call information
  - *EXP*: Perform undirected fuzzing to recover indirect edges until target becomes reachable
- Enforces tripwiring at function-level granularity



SieveFuzz — Lightweight Implementation

- Client-server communication between the fuzzer and static analysis module
- Function activation bitmap allows tripwiring functions dynamically
- Diversity heuristic implemented using input trace length

Evaluation Overview

Benchmarks: 10 security vulnerabilities across 9 varied benchmarks (3 synthetic + 6 real-world)

Experiments: 10x24hr fuzzing campaigns comparing against AFL++, AFLGo, and BEACON

Evaluation Metrics:

- *Tripwiring Efficiency*: Quantify restricted search space and time taken to perform tripwiring
- *Bug-discovery Effectiveness*: Time taken to discover the ground truth bugs

Evaluation: Tripwiring Efficiency

- Quantified the cumulative time taken to perform tripwiring during fuzzing campaigns

Benchmark	Analysis Cost (ms)	Re-runs	Re-run Cost (s)
gif2tga	2	0	0.0
jasper	60	29	1.74
listswf	10	31	0.31
mjs	26	2	0.05
Tidy	91	44	4.00
tiffcp-1	194	29	5.62
tiffcp-2	175	29	5.07

Evaluation: Tripwiring Efficiency

- Quantified the cumulative time taken to perform tripwiring during fuzzing campaigns
- Re-running tripwiring takes **less than 6 seconds** of the total fuzzer runtime

Benchmark	Analysis Cost (ms)	Re-runs	Re-run Cost (s)
gif2tga	2	0	0.0
jasper	60	29	1.74
listswf	10	31	0.31
mjs	26	2	0.05
Tidy	91	44	4.00
tiffcp-1	194	29	5.62
tiffcp-2	175	29	5.07

Evaluation: Tripwiring Effectiveness

- Quantified the amount of code regions removed using tripwiring

Benchmark	Reduction
gif2tga	38%
jasper	8%
listswf	12%
mjs	39%
Tidy	20%
tiffcp-1	18%
tiffcp-2	18%

Evaluation: Tripwiring Effectiveness

- Quantified the amount of code regions removed using tripwiring
- Tripwiring eliminates **29% of code regions** on average as target-irrelevant functionality

Benchmark	Reduction
gif2tga	38%
jasper	8%
listswf	12%
mjs	39%
Tidy	20%
tiffcp-1	18%
tiffcp-2	18%

Evaluation: Bug-discovery Effectiveness

Benchmark	Bug Discovery Effectiveness (# trials)			Mean Exposure Time (#hrs)		
	AFL++	AFLGo	SieveFuzz	AFL++	AFLGo	SieveFuzz
gif2tga	2	0	**4**	9.86	n/a	**6.83**
jasper	4	**8**	**8**	16.85	**6.10**	8.77
listswf	**10**	9	**10**	3.49	5.27	**0.97**
mjs	2	**8**	5	8.16	10.02	**7.20**
Tidy	4	5	**7**	19.10	14.28	**6.20**
tiffcp-1	4	2	**10**	4.20	4.80	**1.36**
tiffcp-2	0	0	**2**	n/a	n/a	**0.32**

Evaluation: Bug-discovery Effectiveness

Benchmark	Bug Discovery Effectiveness (# trials)			Mean Exposure Time (#hrs)		
	AFL++	AFLGo	SieveFuzz	AFL++	AFLGo	SieveFuzz
gif2tga	2	0	**4**	9.86	n/a	**6.83**
jasper	4	**8**	**8**	16.85	**6.10**	8.77
listswf	**10**	9	**10**	3.49	5.27	**0.97**
mjs	2	**8**	5	8.16	10.02	**7.20**
Tidy	4	5	**7**	19.10	14.28	**6.20**
tiffcp-1	4	2	**10**	4.20	4.80	**1.36**
tiffcp-2	0	0	**2**	n/a	n/a	**0.32**

Evaluation: Bug-discovery Effectiveness

Benchmark	Bug Discovery Effectiveness (# trials)		Mean Exposure Time (#hrs)	
	BEACON	SieveFuzz	BEACON	SieveFuzz
gif2tga	10	**10**	2.15	**0.17**
jasper	**10**	6	8.51	**7.8**
listswf	8	**10**	13.36	**0.51**
tiffcp-1	0	**9**	n/a	**0.30**
tiffcp-2	0	**6**	n/a	**6.65**

SieveFuzz is more effective at bug-discovery than existing state-of-the-art undirected fuzzer (AFL++) and directed fuzzers (AFLGo, BEACON)

Conclusion

- Existing directed fuzzers wastefully explore target-irrelevant code regions
- Disjoint target locations cause particular large amounts of wastage
- Tripwiring is an effective directed fuzzing strategy for disjoint targets
- SieveFuzz's tripwiring triggers bugs on average **47% more consistently** and **117% faster** than undirected (AFL++) and directed fuzzers (AFLGo, BEACON)

Code and artifact available at: https://github.com/HexHive/SieveFuzz

Backup Slides

Listing 1 Simplified code snippet to show distance minimization's wastefulness.

```c
int main(void) {
    io_t io;
    program_t p;
    cgc_io_init_fd(&io, STDIN);
    cgc_program_init(&p, &io);
    // Bug-triggering path through cgc_program_parse
    if (cgc_program_parse(&p)) {
        // Irrelevant functionality below not
        // relevant towards triggering the bug
        if (!cgc_program_run(&p, &io)) { ... }
    }
    // Irrelevant functionality below not relevant
    // towards triggering the bug
    else { ... }
}
static int cgc_program_parse(program_t *prog) {
   ...
   stmt_t * tail = NULL;
   while(1) {
     stmt_t *tmp;
     // cgc_parse_statements may return NULL value in `tmp`
     if (!cgc_parse_statements(prog, &tmp)){
         goto fail;
     }
     if (stmt == NULL) {  tail = stmt = tmp; }
     // Possible null dereference below due to missing null check on `tmp`
     else  { tail = tail->next = tmp }
   }
}
```

Listing 1 Simplified code snippet to show distance minimization's wastefulness.

```
int main(void) {
    io_t io;
    program_t p;
    cgc_io_init_fd(&io, STDIN);
    cgc_program_init(&p, &io);
    // Bug-triggering path through cgc_program_parse
    if (cgc_program_parse(&p)) {
        // Irrelevant functionality below not
        // relevant towards triggering the bug
        if (!cgc_program_run(&p, &io)) { ... }
    }
    // Irrelevant functionality below not relevant
    // towards triggering the bug
    else { ... }
}
static int cgc_program_parse(program_t *prog) {
   ...
   stmt_t * tail = NULL;
   while(1) {
     stmt_t *tmp;
     // cgc_parse_statements may return NULL value in `tmp`
     if (!cgc_parse_statements(prog, &tmp)){
         goto fail;
     }
     if (stmt == NULL) {  tail = stmt = tmp; }
     // Possible null dereference below due to missing null check on `tmp`
     else  { tail = tail->next = tmp }
   }
}
```