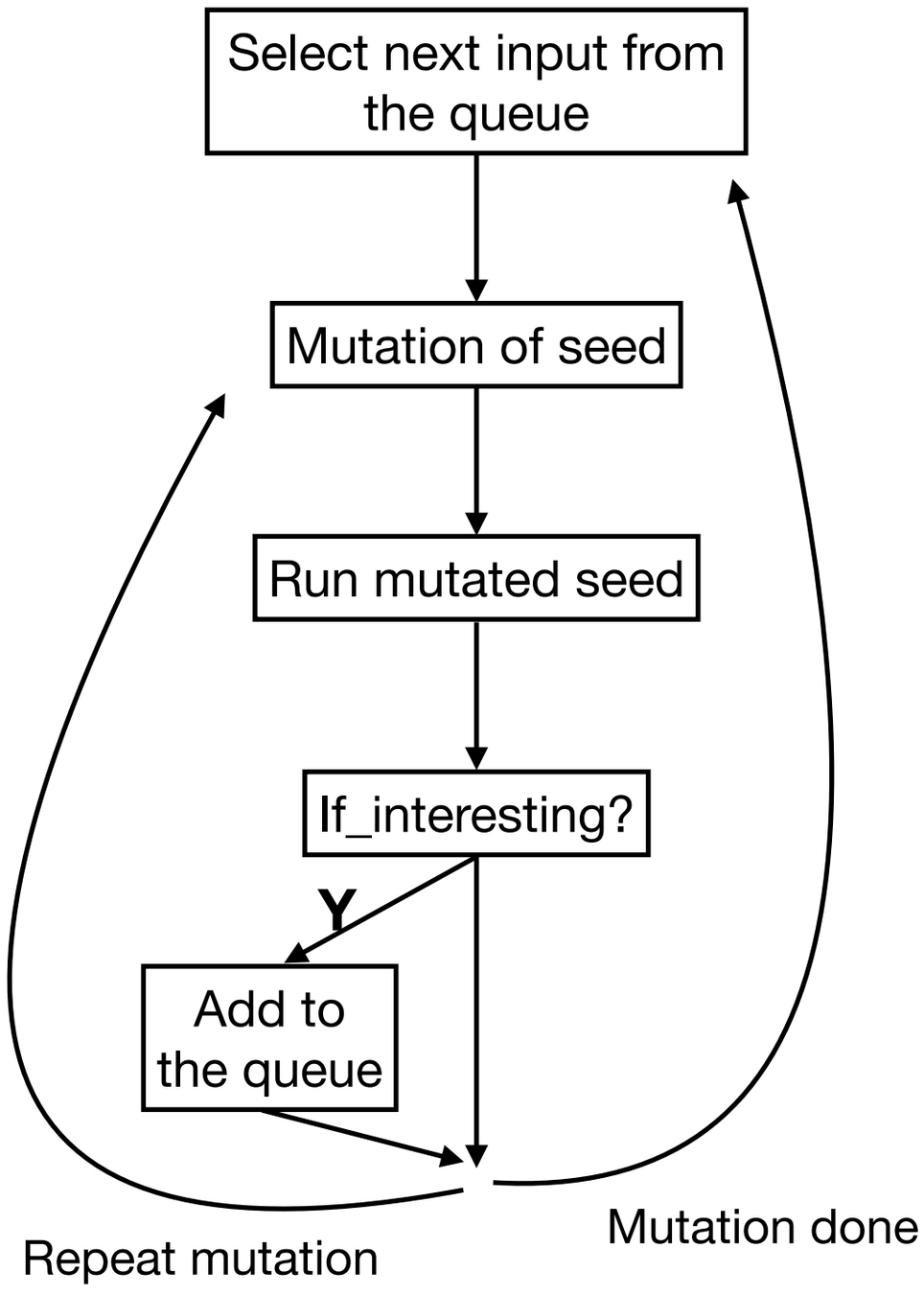
Optimization influence on Fuzzing

TA: Shaohua Li

General Workflow of a Fuzzer

- Given the source code of a program, **compile** and **instrument** it with fuzzer-modified compiler.



1. Select a seed input from the **seed pool**.
2. **Mutate** the input with one of the available mutation operators.
3. Execute the program on the mutated input and collect the **feedback** such as branch coverage from the execution.
4. Add the mutated input to the seed pool if it **increases branch coverage**, otherwise drop it. Then Go to step 1.

Why/How do optimizations affect fuzzers?

```
int foo (int x) {
    int i;
    for (i=0; i < 5; i++) {
        x++;
    }
    return x;
}
```

clang -O0

```
foo:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     DWORD PTR [rbp-4], 0
        jmp     .L2
.L3:
        add     DWORD PTR [rbp-20], 1
        add     DWORD PTR [rbp-4], 1
.L2:
        cmp     DWORD PTR [rbp-4], 4
        jle     .L3
        mov     eax, DWORD PTR [rbp-20]
        leave
        ret
```

clang -O3

```
foo:
        lea     eax, [rdi+5]
        ret
```

General workflow of this project

1. Select fuzzer(s). AFL[1] and AFL++[2] recommended.
2. Select fuzzing targets. They need to be well-studied in the fuzzing community. A set of benchmark programs available oneline[3,4,5].
3. Seed selection. Use official test suite whenever possible. Some benchmarks provide good seeds.
4. Compile your target with optimizations that you are going to study. You may need to modify fuzzer's instrumentation code to do this.
5. Fuzz the target for a fixed length of time (min. 6 hours) and repeat it for a few times. Analyze fuzzers' performance in terms of coverage or bug-finding.
    1) Edge coverage is mandatory metric for evaluation.
    2) Bug-finding is an optional metric. Be sure to use old versions of targets that contain known bugs; otherwise you may end up with no sufficient experimental data. Aslo make sure to enable addresssanitizer and use last/top three stack traces for triage.

[1] https://github.com/google/AFL
[2] https://github.com/AFLplusplus/AFLplusplus
[3] Google Fuzzbench. https://github.com/google/fuzzbench
[4] Magam. https://hexhive.epfl.ch/magma/
[5] UNIFUZZ. https://github.com/unifuzz

Compiler optimizations

- There are typically hundreds of compiler optimization passes in C/C++ compilers such as gcc and clang.
- It is generally infeasible to find the optimal optimization sequences to maximize execution speed or minimize code size.

Optional questions to answer:

- How do default optimization levels (e.g., -O0, -O1, and -O2) affect fuzzers' performance?
- How do the alternating optimization sequences found by other tools affect fuzzers' performance? (Bintuner, google scholar search "phrase ordering")
- Which kind of optimizations have the most impact? (e.g., loop unrooling, function inlining…) It is not practical to study all possible options, just select a subset of them and justify the reason.

An example work flow:

- suppose that you're studying the impact of clang -O0, clang -O3 on the target objdump
- first, compile objdump with them seperately to obstain objdump-O0, objdump-O3
- seond, fuzz-objdump-O0, fuzz-objdump-O3
- third, collect the generated coverage-increasing test input :
  test inputs-O0(saved in the queue/), test inputs-O3(saved in the queue/),
  [the test inputs would be in the crashes/ if you're evaluating bug-finding ability.]
- then, evaluating edge coverage, objdump-O0, run both two sets of test inputs

What should you submit:

- 1) report
- 2) binaries+test inputs -> edge coverage in your report