L14: Fuzzing

Software Analysis

IIT Guwahati · August 2026

Motivating Example

```c
int main(int argc, char **argv) {
    int  granted = 0;
    char pw[8];
    strcpy(pw, argv[1]);
    if (strcmp(pw, "pass") == 0) granted = 1;
    if (granted) printf("Access Granted -> root access\n");
    else         printf("Access Denied!\n");
}
```

```
$ ./buggy pass
Access Granted -> root access
```

Motivating Example

```c
int main(int argc, char **argv) {
    int  granted = 0;
    char pw[8];
    strcpy(pw, argv[1]);
    if (strcmp(pw, "pass") == 0) granted = 1;
    if (granted) printf("Access Granted -> root access\n");
    else         printf("Access Denied!\n");
}
```

```
$ ./buggy pass
Access Granted -> root access
```

```
$ ./buggy abcd
Access Denied!
```

Motivating Example

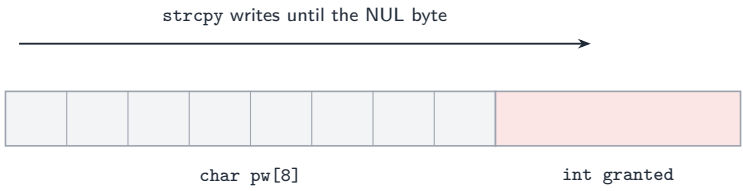
```c
int main(int argc, char **argv) {
    int  granted = 0;
    char pw[8];
    strcpy(pw, argv[1]);
    if (strcmp(pw, "pass") == 0) granted = 1;
    if (granted) printf("Access Granted -> root access\n");
    else         printf("Access Denied!\n");
}
```

```
$ ./buggy pass
Access Granted -> root access
```

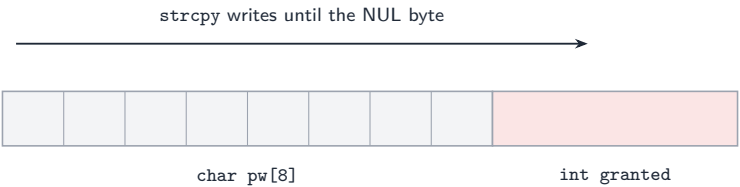
```
$ ./buggy abcd
Access Denied!
```

```
$ ./buggy abcdefghijk
Access Granted -> root access
```

Why the Long Input Is Accepted



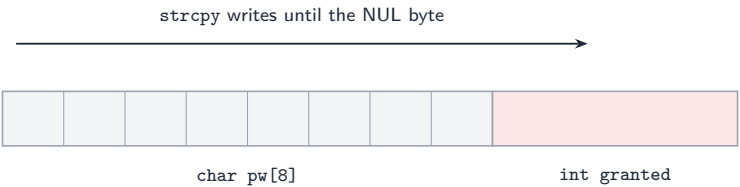
Why the Long Input Is Accepted



A nine-character argument writes nine bytes — eight characters and the terminating NUL — so the ninth byte lands in `granted` and makes it non-zero:

```
len 8   -> Access Denied!          len 9   -> Access Granted -> root access
```

Why the Long Input Is Accepted



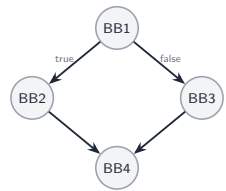
A nine-character argument writes nine bytes — eight characters and the terminating NUL — so the ninth byte lands in `granted` and makes it non-zero:

```
len 8   -> Access Denied!          len 9   -> Access Granted -> root access
```

The program never crashes, so nothing signals a failure: the wrong answer is returned quietly.

Background: the Control-Flow Graph

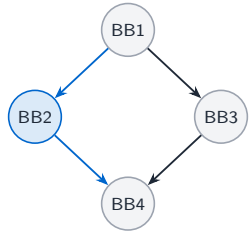
```
int check(int x) {              BB1
    if (x > 3)                  BB1 ends with the branch
        x = x - 1;              BB2
    else
        x = x + 1;              BB3
    return x;                   BB4
}
```



A **basic block** is a straight-line run of instructions with one entry and one exit.

Zeller et al., *The Fuzzing Book*

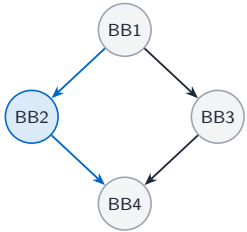
Execution Paths



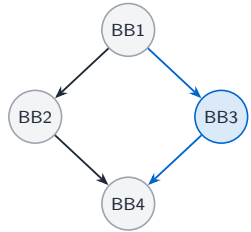
input 4: BB1 → BB2 → BB4

Zeller et al., *The Fuzzing Book*

Execution Paths



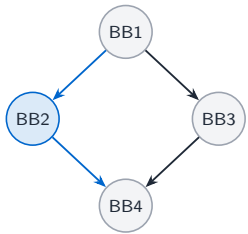
input 4: BB1 → BB2 → BB4



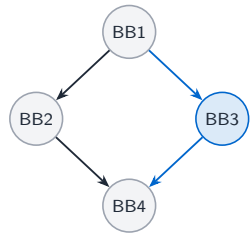
input 3: BB1 → BB3 → BB4

Zeller et al., *The Fuzzing Book*

Execution Paths



input 4: BB1 → BB2 → BB4

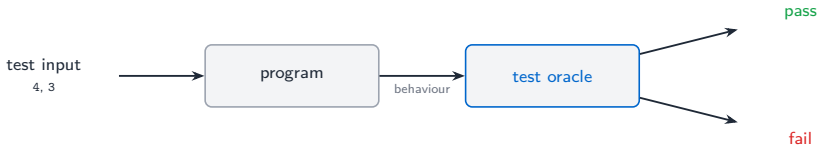


input 3: BB1 → BB3 → BB4

An **execution path** is one possible flow of control through the program. The two inputs $\{4, 3\}$ execute every block and both branch outcomes: **100% statement and branch coverage** with two test inputs.

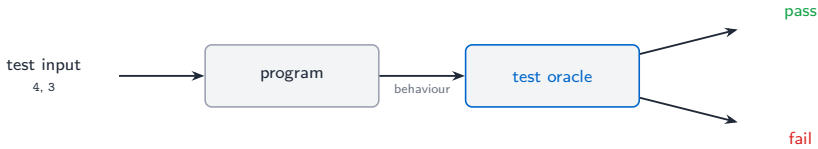
Zeller et al., *The Fuzzing Book*

Testcases



Barr et al., *The Oracle Problem in Software Testing: A Survey*, TSE 41(5), 2015

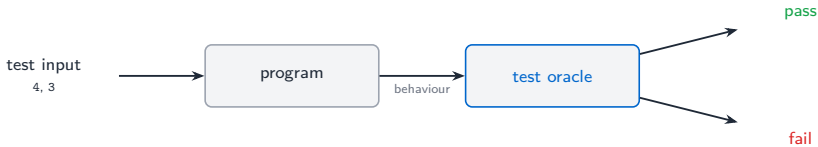
Testcases



test case = test input + test oracle

Barr et al., *The Oracle Problem in Software Testing: A Survey*, TSE 41(5), 2015

Testcases

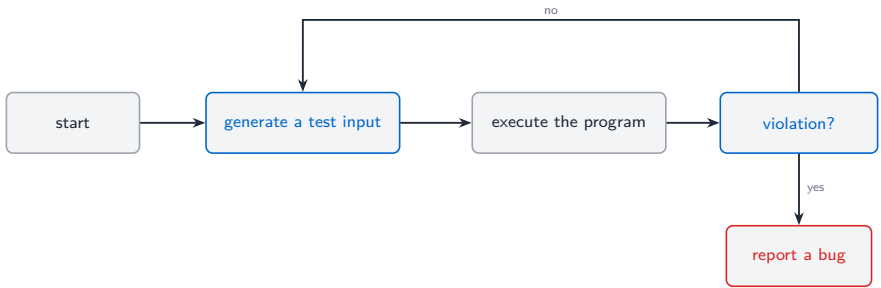


test case = test input + test oracle

An input alone only produces behaviour; the oracle is what turns behaviour into a verdict. Automating the two halves raises two different problems.

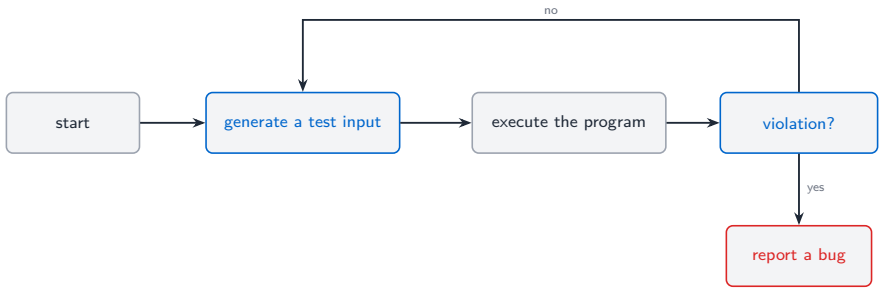
Barr et al., *The Oracle Problem in Software Testing: A Survey*, TSE 41(5), 2015

The Fuzzing Loop



Zeller et al., *The Fuzzing Book*

The Fuzzing Loop



The loop is simple; everything of interest is in the two boxes that are not: how an input is **generated**, and what counts as a **violation**.

Zeller et al., *The Fuzzing Book*

Challenges for a Fuzzer

- **Shallow coverage.** Most generated inputs are rejected early and exercise the same small part of the program.

Technical whitepaper for afl-fuzz

Challenges for a Fuzzer

- **Shallow coverage.** Most generated inputs are rejected early and exercise the same small part of the program.
- **Deep defects are hard to reach.** A defect behind several accepted checks is reached only by an input that passes all of them.

Technical whitepaper for afl-fuzz

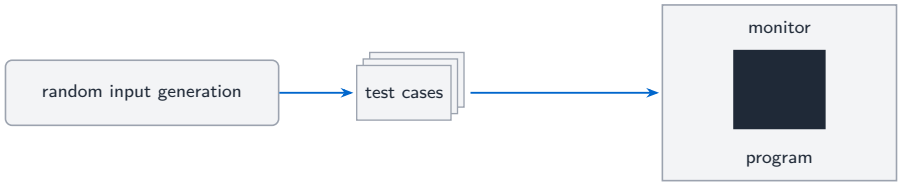
Challenges for a Fuzzer

- **Shallow coverage.** Most generated inputs are rejected early and exercise the same small part of the program.
- **Deep defects are hard to reach.** A defect behind several accepted checks is reached only by an input that passes all of them.
- **Sanity checks block progress.** Magic bytes, checksums, lengths and format tags are exactly the conditions a randomly generated input fails.

Technical whitepaper for afl-fuzz

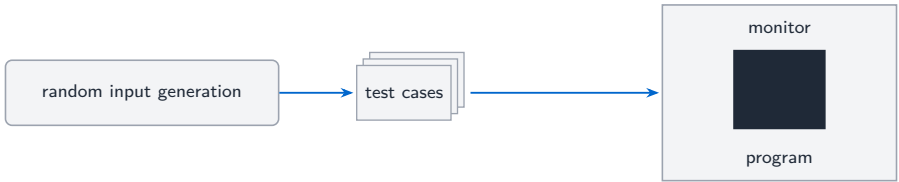
Kinds of Fuzzing

Black-box Fuzzing



Zeller et al., *The Fuzzing Book*

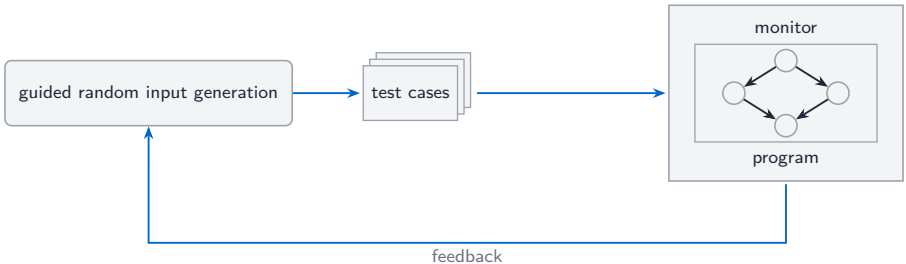
Black-box Fuzzing



The generator sees nothing of the program; the monitor reports only whether a run terminated abnormally. Inputs are produced independently of every run that preceded them.

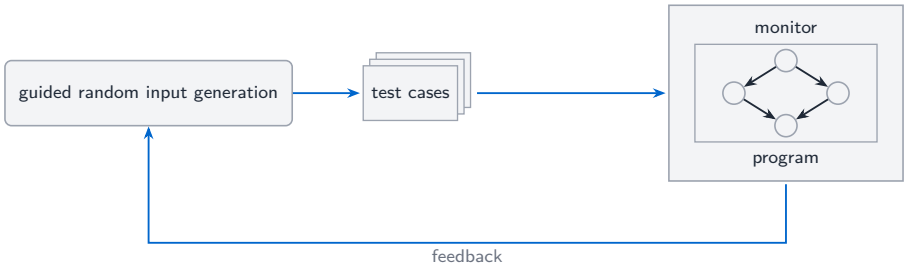
Zeller et al., *The Fuzzing Book*

Grey-box Fuzzing



Zeller et al., *The Fuzzing Book* · *Technical whitepaper for afl-fuzz*

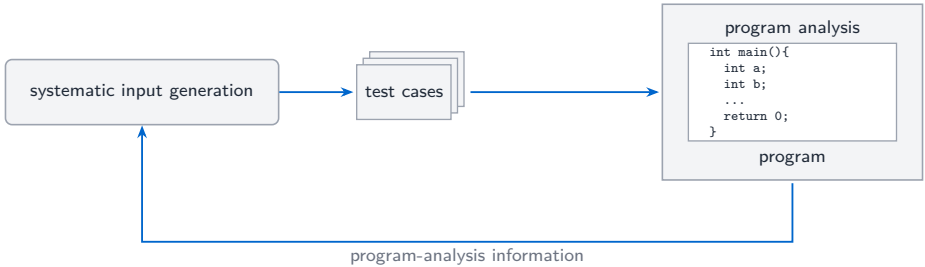
Grey-box Fuzzing



The monitor also reports which parts of the program the run executed, and the generator uses that report when choosing the next input.

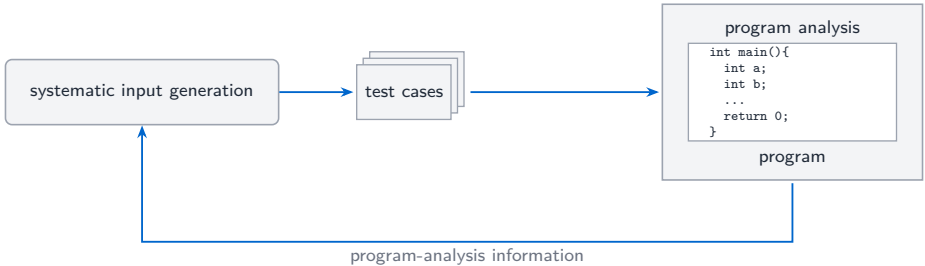
Zeller et al., *The Fuzzing Book* · *Technical whitepaper for afl-fuzz*

White-box Fuzzing



Zeller et al., *The Fuzzing Book*

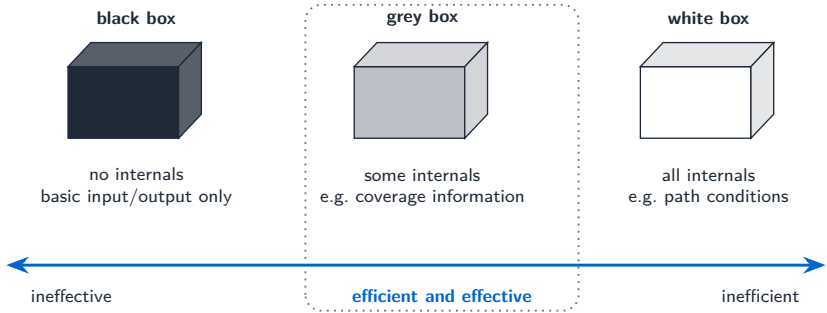
White-box Fuzzing



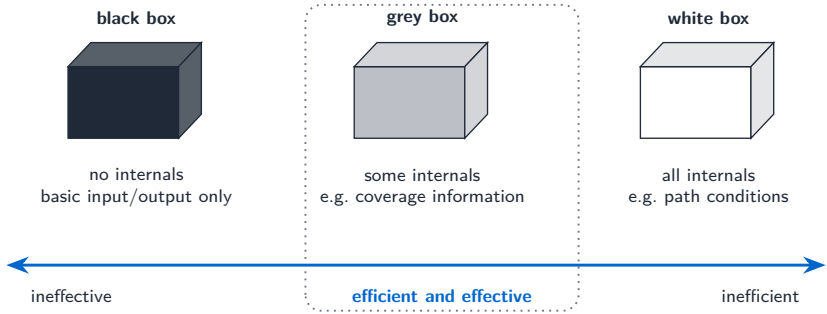
The generator reasons about the program text itself — path conditions, symbolic values — and derives inputs that reach a chosen branch.

Zeller et al., *The Fuzzing Book*

Types of Fuzzer



Types of Fuzzer



Most fuzzers in use combine elements of all three.

Challenges

```
void process(int x) {
    if (x == 12345678)
        vulnerable_function();
}
```

Zeller et al., *The Fuzzing Book*

Challenges

```
void process(int x) {
    if (x == 12345678)
        vulnerable_function();
}
```

A generator drawing 32-bit values uniformly at random reaches the call with probability

$$\frac{1}{2^{32}} \approx 2.3 \times 10^{-10}$$ — about **one in four billion** executions.

Zeller et al., *The Fuzzing Book*

Challenges

```
void process(int x) {
    if (x == 12345678)
        vulnerable_function();
}
```

A generator drawing 32-bit values uniformly at random reaches the call with probability

$$\frac{1}{2^{32}} \approx 2.3 \times 10^{-10}$$ — about **one in four billion** executions.

A white-box technique treats `x` as a symbol, forms the path condition `x == 12345678` and asks a constraint solver for a model, obtaining the input directly rather than by search.

Zeller et al., *The Fuzzing Book*

Challenges

```
void process(int x) {
    if (x == 12345678)
        vulnerable_function();
}
```

A generator drawing 32-bit values uniformly at random reaches the call with probability

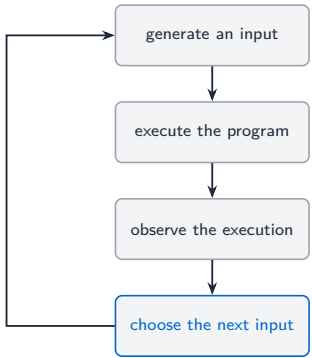
$$\frac{1}{2^{32}} \approx 2.3 \times 10^{-10}$$ — about **one in four billion** executions.

A white-box technique treats `x` as a symbol, forms the path condition `x == 12345678` and asks a constraint solver for a model, obtaining the input directly rather than by search.

Constraint solving is expensive and does not scale to every path–>grey-box techniques.

Zeller et al., *The Fuzzing Book*

The Common Structure: a Feedback Loop

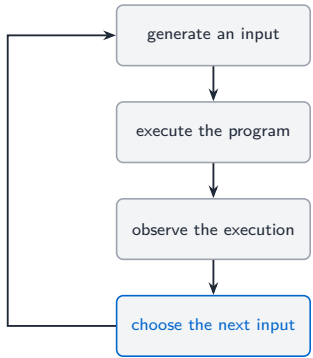


what is observed

black box: crash or no crash

Zeller et al., *The Fuzzing Book* · *Technical whitepaper for afl-fuzz*

The Common Structure: a Feedback Loop



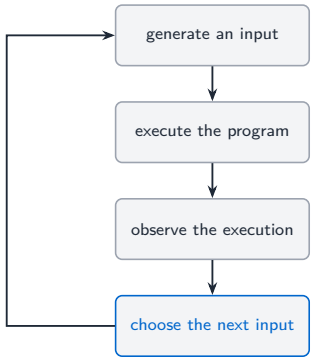
what is observed

black box: crash or no crash

grey box: new coverage or not

Zeller et al., *The Fuzzing Book* · *Technical whitepaper for afl-fuzz*

The Common Structure: a Feedback Loop



what is observed

black box: crash or no crash

grey box: new coverage or not

also: a new path, a new state, progress on a comparison

Zeller et al., *The Fuzzing Book* · *Technical whitepaper for afl-fuzz*