CS412 Software Security

Program Testing – Fuzzing

Mathias Payer

EPFL, Spring 2019

Why testing?

> *Testing is the process of executing a program to find errors.*

An error is a deviation between observed behavior and specified behavior, i.e., a violation of the underlying specification:

- Functional requirements (features a, b, c)
- Operational requirements (performance, usability)
- Security requirements?

Forms of testing

- Manual testing
- Fuzz testing
- Symbolic and concolic testing

Manual testing

Three levels of testing:

- Unit testing (individual modules)
- Integration testing (interaction between modules)
- System testing (full application testing)

Manual testing strategies

- Exhaustive: cover all input; not feasible due to *massive* state space
- Functional: cover all requirements; depends on specification
- Random: automate test generation (but incomplete)
- Structural: cover all code; works for unit testing

Testing example

How do you decide when you have tested a function “enough”?

```
double doFun(double a, double b, double c) {
  if (a == 23.0 && b == 42.0) {
    return a * b / c;
  }
  return a * b * c;
}
```

Testing example

How do you decide when you have tested a function "enough"?

```
double doFun(double a, double b, double c) {
  if (a == 23.0 && b == 42.0) {
    return a * b / c;
  }
  return a * b * c;
}
```

Fails for `a == 23.0 && b == 42.0 && c == 0.0`.

Testing approaches

```
double doFun(double a, double b, double c)
```

- Exhaustive: `2^{64}^3` tests
- Functional: generate test cases for true/false branch, ineffective for errors in specification or coding errors
- Random: probabilistically draw `a`, `b`, `c` from value pool
- Structural: aim for full *code coverage*, generate test cases for all paths

Coverage as completeness metric

Intuition: A software flaw is only detected if the flawed statement is executed. Effectiveness of test suite therefore depends on how many statements are executed.

Is statement coverage enough?

```
int func(int elem, int *inp, int len) {
  int ret = -1;
  for (int i = 0; i <= len; ++i) {
    if (inp[i] == elem) { ret = i; break; }
  }
  return ret;
}
```

Test input: `elem = 2, inp = [1, 2], len = 2`. Full statement coverage.

Is statement coverage enough?

```
int func(int elem, int *inp, int len) {
  int ret = -1;
  for (int i = 0; i <= len; ++i) {
    if (inp[i] == elem) { ret = i; break; }
  }
  return ret;
}
```

Test input: `elem = 2, inp = [1, 2], len = 2`. Full statement coverage.

Loop is never executed to termination, where out of bounds access happens. Statement coverage does not imply *full* coverage. Today's standard is *branch* coverage, which would satisfy the backward edge from `i <= len` to the end of the loop. Full branch coverage implies full statement coverage.

Is branch coverage enough?

```
int arr[5] = { 0, 1, 2, 3, 4};
int func(int a, int b) {
  int idx = 4;
  if (a < 5) idx -= 4; else idx -= 1;
  if (b < 5) idx -= 1; else idx += 1;
  return arr[idx];
}
```

Test inputs: `a = 5, b = 1` and `a = 1, b = 5`. Full branch coverage.

Is branch coverage enough?

```
int arr[5] = { 0, 1, 2, 3, 4};
int func(int a, int b) {
  int idx = 4;
  if (a < 5) idx -= 4; else idx -= 1;
  if (b < 5) idx -= 1; else idx += 1;
  return arr[idx];
}
```

Test inputs: `a = 5, b = 1` and `a = 1, b = 5`. Full branch coverage.

Not all paths through the function are executed: `a = 1, b = 1` results in a bug when both statements are true at the same time. Full path coverage evaluates all possible paths but this can be expensive (path explosion due to each branch) or impossible for loops. Loop coverage (execute each loop 0, 1, n times), combined with branch coverage probabilistically covers state space.

How to measure code coverage?

Several (many) tools exist:

- gcov: `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`
- SanitizerCoverage: `https://clang.llvm.org/docs/SourceBasedCodeCoverage.html`

How to achieve full testing coverage?

Idea: look at data flow.

Track constraints of conditions, generate inputs for all possible constraints.

Testing completeness

We now have a metric for testing completeness.
How can we explore programs to maximize coverage?

Testing completeness

We now have a metric for testing completeness.
How can we explore programs to maximize coverage?
Discuss: is coverage enough to find all bugs?

Fuzzing

Fuzz testing (fuzzing) is an automated software testing technique. The fuzzing engine generates inputs based on some criteria:

- Random mutation
- Leveraging input structure
- Leveraging program structure

The inputs are then run on the test program and, if it crashes, a crash report is generated.

Fuzzing effectiveness

- Fuzzing finds bugs effectively (CVEs)
- Proactive defense, part of testing
- Preparing offense, part of exploit development

Fuzz input generation

Fuzzers generate new input based on generations or mutations.
Generation-based input generation produces new input seeds in each round, independent from each other.
Mutation-based input generation leverages existing inputs and modifies them based on feedback from previous rounds.

Fuzz input structure awareness

Programs accept some form of input/output. Generally, the input/output is structured and follows some form of protocol.
Dumb fuzzing is unaware of the underlying structure.
Smart fuzzing is aware of the protocol and modifies the input accordingly.
Example: a checksum at the end of the input. A dumb fuzzer will likely fail the checksum.

Fuzz program structure awareness

The input is processed by the program, based on the program structure (and from the past executions), input can be adapted to trigger new conditions.

- *White box* fuzzing leverages semantic program analysis to mutate input
- *Grey box* leverages program instrumentation based on previous inputs
- *Black box* fuzzing is unaware of the program structure

Coverage-guided grey box fuzzing

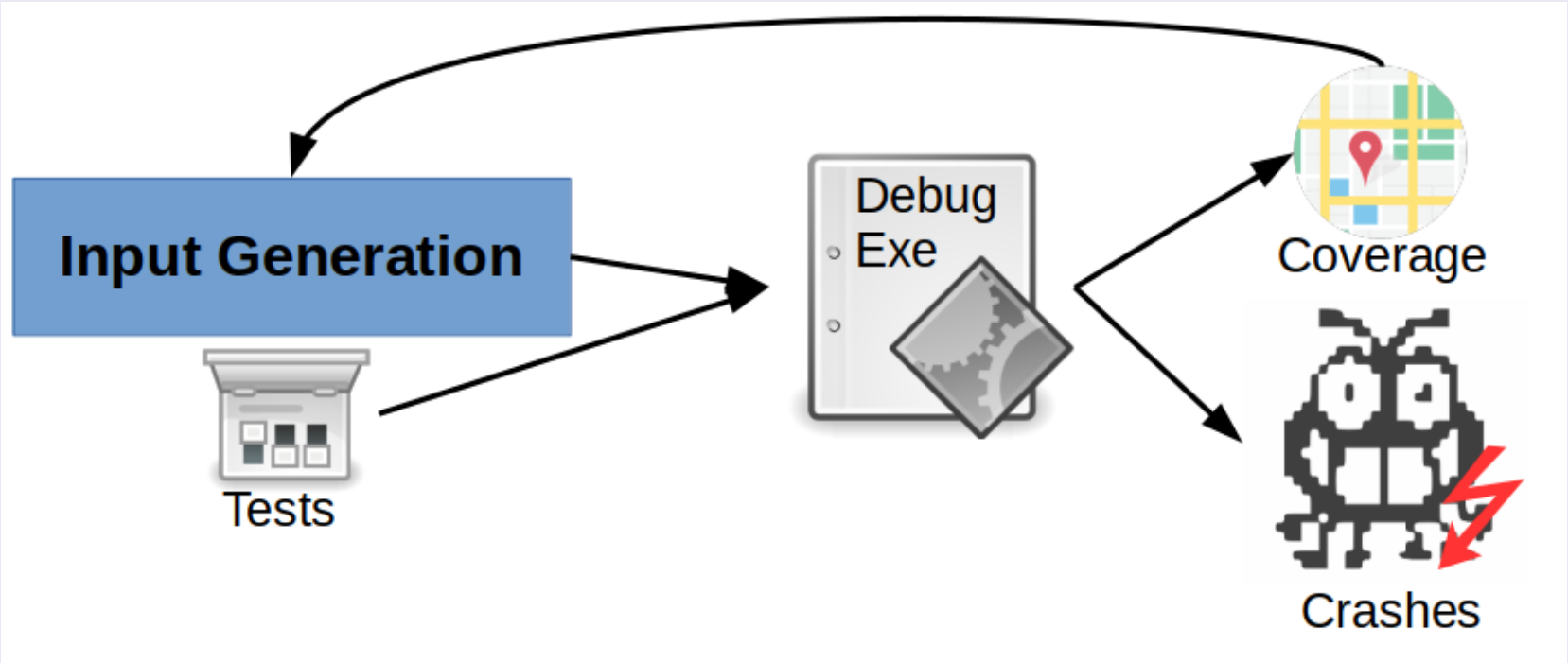

Figure 1:

American Fuzzy Lop

- AFL is the most well-known fuzzer currently
- AFL uses grey-box instrumentation to track branch coverage and mutate fuzzing seeds based on previous branch coverage
- The branch coverage tracks the last two executed basic blocks
- New coverage is detected on the history of the last two branches: `cur XOR prev>>1`
- AFL: `http://lcamtuf.coredump.cx/afl/`

Fuzzer challenges: coverage wall

- After certain iterations the fuzzer no longer makes progress
- Hard to satisfy checks
- Chains of checks
- Leaps in input changes

Fuzzer challenges: coverage wall

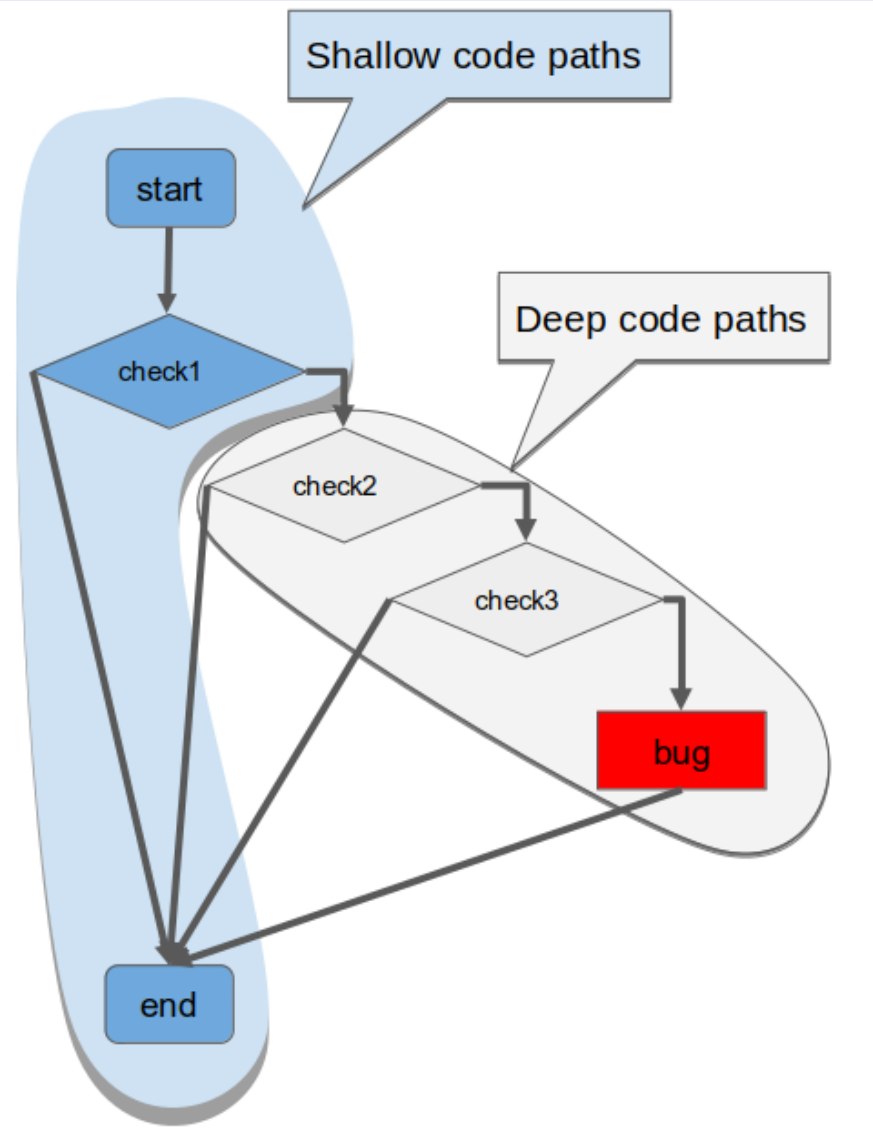

Figure 2:

Symbolic execution

- Reason about program behavior through "execution" with symbolic values
- Concrete values (input) replaced with symbolic values
  - Can have any value (think variable x instead of value `0x15`)
  - Track all possible execution paths at once
- Operations (read, write, arithmetic) become constraint collection
  - Allows *unknown* symbolic variables in evaluation
  - Execution paths that depend on symbolic variables *fork*

Symbolic execution: example

```
void func(int a, int b, int c) {
  int x = 0, y = 0, z = 0;
  if (a) x = -2;
  if (b < 5) {
    if (!a && c) y = 1;
    z = 2;
  }
  assert(x + y + z != 3);
}
```

Symbolic execution: example

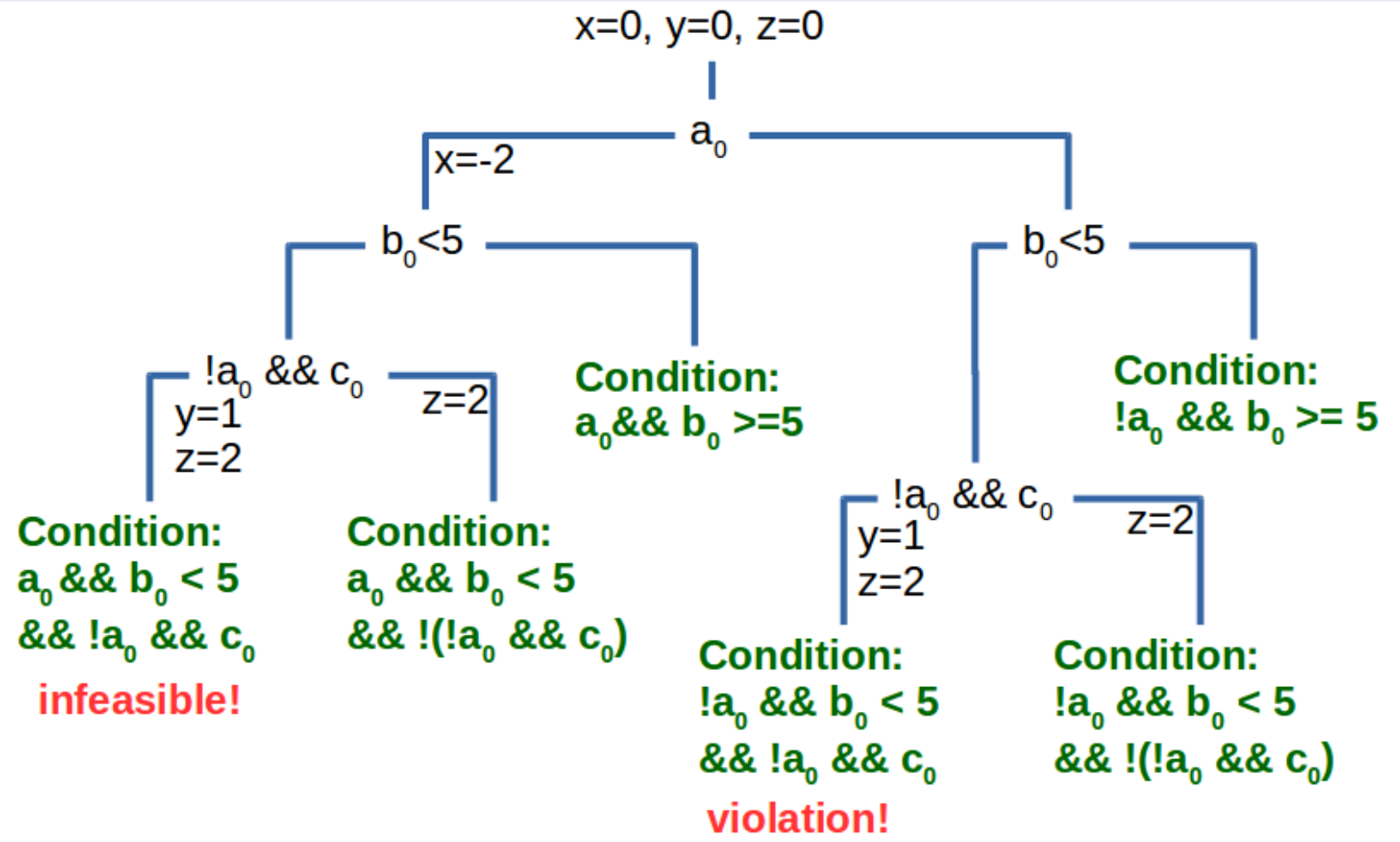

Figure 3:

Symbolic paths

Path condition: quantifier-free formula over symbolic inputs that encodes all branch decisions (so far).
Determine whether the path is feasible: check if path condition is satisfiable. SMT solver provides satisfying assignment, counter example, or timeout.

Challenges for symbolic execution

- Loops and recursion result in infinite execution traces
- Path explosion (each branch doubles the number of paths)
- Environment modeling (system calls are complex)
- Symbolic data (symbolic arrays and symbolic indices)

Concolic testing

Idea: mix concrete and symbolic execution

- Record actual execution
- Symbolically execute *near* recorded trace
- Negate one condition, generate new input, repeat

KLEE

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, Cadar et al., OSDI'08

- Large scale symbolic execution tool
- Leverages LLVM to compile programs
- Abstracts environment
- Many different search strategies

Bounded Model Checking

- Unwind recursion, loops up to certain depth
- Translate program into boolean formula
  - Transform to SSA (static single assignment)
  - Bitblast! (Translate into logical adders)
- Pass to solver

BMC example (1/3)

```
x = x + y;
if (x != 1) {
  x = 2;
  if (z) x++;
}
assert(x<=3);
```

BMC example (2/3)

```
x1 = x0 + y0;
if (x1 != 1) {
  x2 = 2;
  if (z0) x3 = x2 + 1;
}
x4 = (x1!=1) ? x3 : x1;
assert(x4<=3);
```

BMC example (3/3)

```
C:= x1 = x0 + y0 AND
    x2 = ((x1 != 1) ? 2 : x1) AND
    x3 = ((x1 != 1 AND z0) ? x2 + 1 : x2)
P:= x3 <= 3
```

Comparisons

- BMC: translate program to formula, solve all possible paths at once
  - Advantage: pinpoint flaw, overview of all paths
  - Disadvantage: scalability
- Symbolic execution: instead of concrete values, track constraints
  - Advantage: iteratively explore program based on paths
  - Disadvantage: scalability
- Concolic execution: select a concrete execution, track constraints
  - Advantage: explore a path at a time
  - Disadvantage: scalability, incomplete
- Fuzzing: create random input based on feedback
  - Advantage: high scalability
  - Disadvantage: incomplete

All solutions struggle with *depth*, i.e., exploring past a coverage wall.

Summary and conclusion

- Software testing finds bugs before an attacker can exploit them
- Manual testing: write test cases to trigger exceptions
- Sanitizers allow early bug detection, not just on exceptions
- Fuzz testing automates and randomizes testing
- Symbolic and concolic testing allow full coverage analysis (at high overheads)