# CS527 Software Security

## Program Testing

Mathias Payer

Purdue University, Spring 2018

## 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?

## Limitations of testing

A successful test finds a deviation.

> *Testing can only show the presence of bugs, never their absence. (Edsger W. Dijkstra)*

Complete testing of all control-flow/data-flow paths reduces to the halting problem, in practice, testing is hindered due to state explosion.

## 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

```
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

```
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.

## Sanitizer

- Test cases detect bugs through
  - Assertions (`assert(var != 0x23 && "var has illegal value");`) detect violations
  - Segmentation faults
  - Division by zero traps
  - Uncaught exceptions
  - Mitigations triggering termination
- How can you increase the chances of detecting a bug?

## Sanitizer

- Test cases detect bugs through
    - Assertions (`assert(var != 0x23 && "var has illegal value");`) detect violations
    - Segmentation faults
    - Division by zero traps
    - Uncaught exceptions
    - Mitigations triggering termination
- How can you increase the chances of detecting a bug?

Sanitizers enforce some policy, detect bugs earlier and increase effectiveness of testing.

## AddressSanitizer

AddressSanitizer (ASan) detects memory errors. It places red zones around objects and checks those objects on trigger events. The tool can detect the following types of bugs:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return (configurable)
- Use-after-scope (configurable)
- Double-free, invalid free
- Memory leaks (experimental)

Typical slowdown introduced by AddressSanitizer is 2x.

## LeakSanitizer

LeakSanitizer detects run-time memory leaks. It can be combined with AddressSanitizer to get both memory error and leak detection, or used in a stand-alone mode.
LSan adds almost no performance overhead until process termination, when the extra leak detection phase runs.

## MemorySanitizer

MemorySanitizer detects uninitialized reads. Memory allocations are tagged and uninitialized reads are flagged.
Typical slowdown of MemorySanitizer is 3x.
Note: do not confuse MemorySanitizer and AddressSanitizer.

## UndefinedBehaviorSanitizer

UndefinedBehaviorSanitizer (UBSan) detects undefined behavior. It instruments code to trap on typical undefined behavior in C/C++ programs. Detectable errors are:

- Unsigned/misaligned pointers
- Signed integer overflow
- Conversion between floating point types leading to overflow
- Illegal use of NULL pointers
- Illegal pointer arithmetic
- ...

Slowdown depends on the amount and frequency of checks. This is the only sanitizer that can be used in production. For production use, a special minimal runtime library is used with minimal attack surface.

## ThreadSanitizer

ThreadSanitizer detects data races between threads. It instruments writes to global and heap variables and records which thread wrote the value last, allowing detecting of WAW, RAW, WAR data races. Typical slowdown is 5-15x with 5-15x memory overhead.

## HexType

HexType detects type safety violations. It records the true type of allocated objects and makes all type casts explicit.
Typical slowdown is 0.5x.

## Sanitizers

- AddressSanitizer: `https://clang.llvm.org/docs/AddressSanitizer.html`
- LeakSanitizer: `https://clang.llvm.org/docs/LeakSanitizer.html`
- MemorySanitizer: `https://clang.llvm.org/docs/MemorySanitizer.html`
- UndefinedBehaviorSanitizer: `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`
- ThreadSanitizer: `https://clang.llvm.org/docs/ThreadSanitizer.html`
- HexType: `https://github.com/HexHive/HexType`

Use sanitizers to test your code. More sanitizers are in development.

## 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.

## 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

## 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
- AFL: `http://lcamtuf.coredump.cx/afl/`

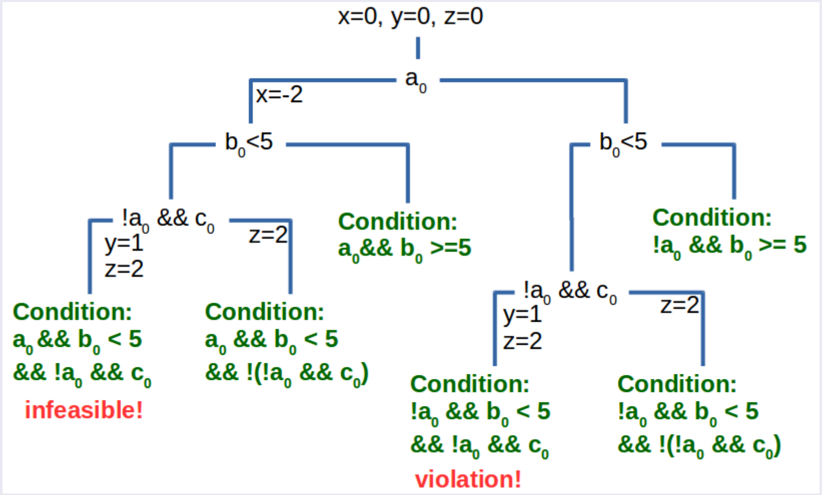
## Symbolic execution

- 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

```c
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



x=0, y=0, z=0
$a_0$
x=-2
$b_0$<5
$b_0$<5
!$a_0$ && $c_0$
y=1
z=2
z=2
Condition: $a_0$&& $b_0$ >=5
Condition: !$a_0$ && $b_0$ >= 5
Condition: $a_0$ && $b_0$ < 5 && !$a_0$ && $c_0$
infeasible!
Condition: $a_0$ && $b_0$ < 5 && !(!$a_0$ && $c_0$)
!$a_0$ && $c_0$
y=1
z=2
z=2
Condition: !$a_0$ && $b_0$ < 5 && !$a_0$ && $c_0$
violation!
Condition: !$a_0$ && $b_0$ < 5 && !(!$a_0$ && $c_0$)

Figure 1:

## 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

## 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)