CS-527 Software Security

Bug finding techniques

Asst. Prof. Mathias Payer

Department of Computer Science
Purdue University

TA: Kyriakos Ispoglou

https://nebelwelt.net/teaching/17-527-SoftSec/

Spring 2017

Table of Contents

Software Testing

- Software testing (e.g., unit testing) uses a set of test cases to decide if the program conforms to a specification.
- Testing tests for the presence of functionality (and not the absence of security bugs).
- Involved security tests may run memory checkers like ASan, type safety checkers, or thread safety checkers.
- ... given specific input. Again, it does not test for generic errors or properties (like memory safety or type safety).
- Testing is a great to detect regressions or missing functionality.
- Newer software comes with large amounts of unit tests and other test cases to test macro and unit functionality.

Bug triangulation

Given a failing test case, how do we detect the location of the bug?

Testing/tracing by printf

```
1 int max = 0;
2 for (p = head; p; p = p->next) {
3   printf("in loop\n");
4   if (p->value > max) {
5     printf("True branch\n");
6     max = p->value;
7   }
8 }
```

Statistical Debugging

- Relies on a large pool of test cases (both failing and passing).
- Dynamic information from failing and passing test cases is aggregated to localize possible faulty statements.
- Output is often a list of ranked statements.

Test case reduction: Delta Debugging

- Minimize test-cases: if you change any thing in the test case the bug is no longer triggered[1].
- Let's use a smaller bug report as running example:
  `<SELECT NAME="priority" MULTIPLE SIZE=7>`
- How can we simplify this input?
- Idea: remove parts of the input and see if the program still crashes (i.e., minimize the test case).
- For the above example assume that we remove characters of the input file and start the program with this new test case.

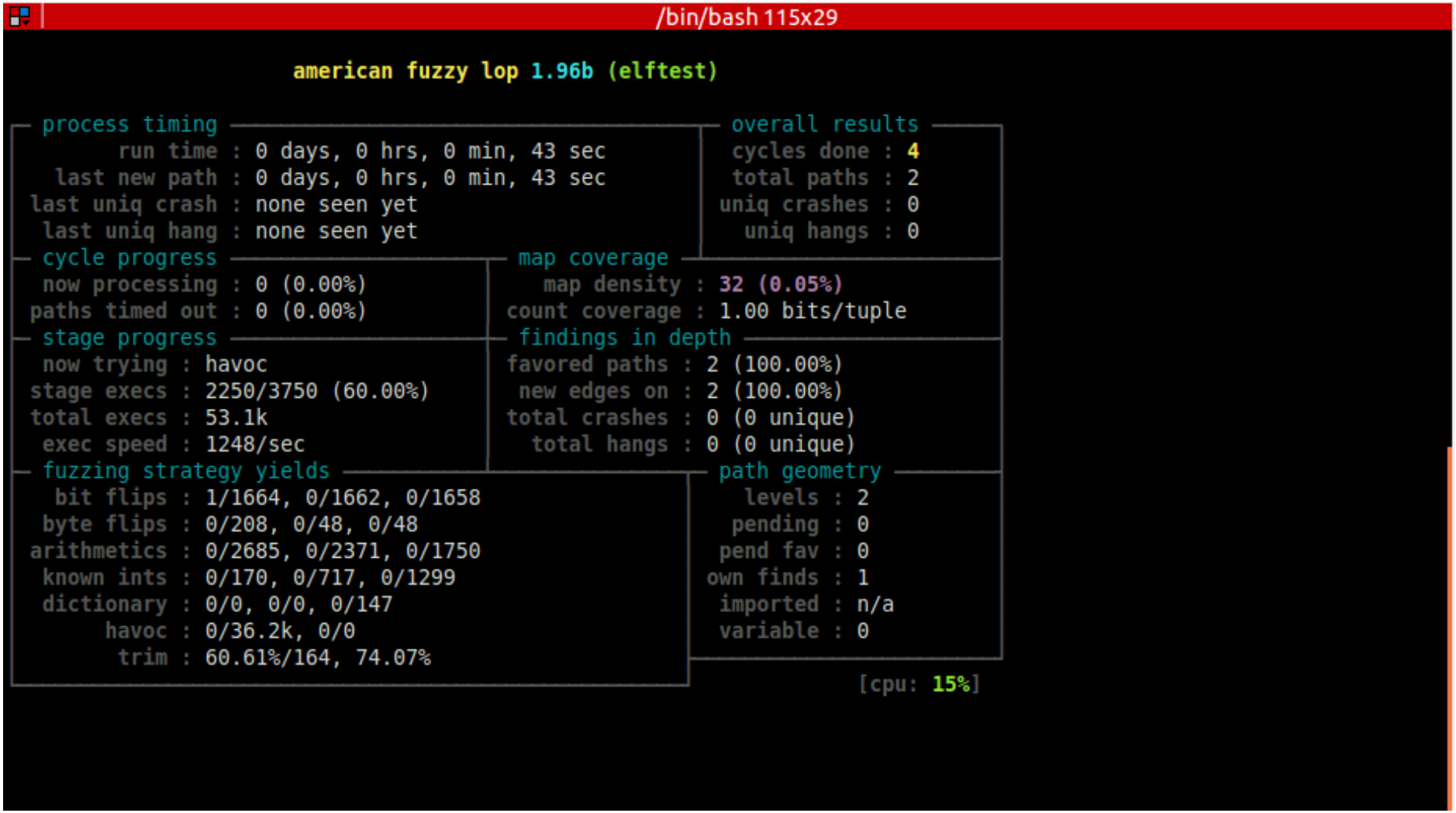
[1] Andreas Zeller and Ralf Hildebrandt, Simplifying and Isolating Failure-Inducing Input, IEEE Trans. SE, 2002.

Table of Contents

Fuzzing

- Fuzzing is an automated form of testing that runs code on (semi) random input.
- Mutation-based fuzzing generates test cases by mutating existing test cases.
- Generation-based fuzzing generates test cases based on a model of the input (i.e., a specification).
- Any inputs that crash the program are recorded.
- Crashes are then sorted, reduced, and bugs are extracted.
- Bugs are then analyzed individually (is it a security vulnerability).

American Fuzzy Lop



American Fuzzy Lop

- American fuzzy lop is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary.
- Low-overhead and low initialization cost (i.e., fast forward to interesting points in binaries before you start fuzzing)
- Synthesizes complex file formats
- Employs different fuzzing strategies, switches them on demand

Homepage: `http://lcamtuf.coredump.cx/afl/`.

Table of Contents

Static Analysis

- Static analysis analyzes a program without executing it.
- Static analysis is widely used in bug finding, vulnerability detection, or property checking.
- "*Easier*" to apply compared to dynamic analysis (as long as you have code): analysis can be transparent to the user.
- Better scalability than to some dynamic analysis (e.g., tracing).
- Large success in recent years: findbug, coverity[2], codesurfer.

[2]Reading material: Al Bessey et al., A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World, CACM'10.

Static Analysis: Syntax/Structure

- Focus on syntax and structure, not semantics.
- Look at CFG, dominator, post-dominator, loop detection
- Application: detect code copies (comparison based on text, AST, CFG)
- Application: Malware analysis
- Recover information about the program, serve as basis for further advanced static/dynamic analysis.
- Limitation: cannot reason about program semantics or state.

Static Analysis: Semantics

- Focus on program semantics.
- Reason about program meaning/logic.
- Evaluate meaning of syntactically legal strings defined by a programming language, reason about involved computation.
- (Illegal strings – according to the language definition – result in non-computation).

Static Analysis: Requirements

- Abstract domain: contains the results we want to compute by static analysis.
- Transfer function: how the abstract values are computed/updated at each relevant instruction.
- (We must consider the instruction semantics for the transfer function!)

Static Analysis: Loops

- When shall we terminate a loop path?
- How many iterations should we consider?
- Is the loop bound? How to infer possible values?
- Observation: we are interested in the aggregation of abstract values along paths.
- If the aggregation stabilizes, we can terminate.
- Assumption: monotonic growth.
- Assumption: abstract domain is finite.

Static Analysis: Use-cases

- Optimization: Global Common Subexpression
- Optimization: Copy Propagation
- Optimization: Dead-Code Elimination
- Optimization: Code Motion
- Optimization: Strength Reduction
- All these optimizations depend on data-flow analysis!

Table of Contents

What is symbolic execution?

- An abstract interpretation of code (values are symbolic, not concrete)
- Agnostic to concrete values (values turn into formulas, constraints make formulas concrete)
- Finds concrete input (triggers "interesting" conditions)

Using symbolic execution

- Define a set of conditions at code locations. Symbolic Execution then determines triggering input.
- Testing: finding bugs in applications
- Step 1: Infer pre/post conditions and add assertions
- Step 2: Use symbolic execution to negate conditions
- Exploit generation: generate PoC input (vulnerability condition is predefined)

SAT Solver

- Find satisfying valuations to a propositional formula.
- Develop a systematic approach to *test* all possible valuations to find a satisfiable valuation.
- SAT solving is NP-complete, so the worst-case complexity will always be exponential.
- ... but good heuristics exist.
- Speed improvements in SAT solving enable Symbolic Execution

Symbolic execution tools

FuzzBALL: Works on binaries, generic SE engine. Used to, e.g., find PoC exploits given a vulnerability condition.

KLEE: Instruments through LLVM-based pass, relies on source code. Used to, e.g., find bugs in programs.

S2E: Selective Symbolic Execution: automatic testing of large source base, combines KLEE with an concolic execution. Used to, e.g., test large source bases (e.g., drivers in kernels) for bugs.

Efficiency of SE tool depends on the search heuristics and search strategy. As search space grows exponentially, a good search strategy is crucial for efficiency and scalability.

Symbolic execution summary

- Symbolic execution is a great tool to find vulnerabilities or to create PoC exploits.
- Symbolic execution is limited in its scalability. An efficient search strategy is crucial.

Want to learn how to use symbolic execution to create PoC exploits: `http://nebelwelt.net/publications/files/1330c3-presentation.pdf` and `https://www.youtube.com/watch?v=Febh70kldP0`.

Table of Contents

Formal verification

Definition: Formal Verification

Formal verification is the act of using formal methods to proving or disproving the correctness of a certain system given its formal specification. Formal verification requires a specification and an abstraction mechanism to show that the formal specification either holds (i.e., its correctness is proven) or fails (i.e., there is a bug).

Verification is carried out by providing a formal proof on the abstracted mathematical model of the system according to the specification. Many different forms of mathematical objects can be used for formal verification like finite state machines or formal semantics of programming languages (e.g., operational semantics or Hoare logic).

Model checking

(Bounded) model checking is an example of formal verification.

- Simplify control flow (remove side effects: `j=i++;` becomes `j = i; i = i + 1;`, make control flow explicit by replacing `continue, break` with `goto`, and rewriting all loops into `while` loops).
- Convert CFG into SSA form.
- Convert IR into equations.
- Unwind loops (bounded).
- Bit-blast.
- Solve with SAT solver.
- (Convert SAT assignment into counter example).

Transforming programs into equations

```
1 x = a;
2 y = x + 1;
3 z = y - 1;
```

```
1 x = a &&
2 y = x + 1 &&
3 z = y - 1
```

SSA is crucial, otherwise there are ambiguities.

Full example: from C to propositional logic

```
1  int main() {
2    int x,y;
3    y = 8;
4    if (x)
5      y--;
6    else
7      y++;
8
9    assert
10     (y == 7 ||
11     y == 9);
12 }
```

```
1  int main() {
2    int x0,y0;
3    y1 = 8;
4    if (x0)
5      y2=y1-1;
6    else
7      y3=y1+1;
8    y4 = x0?y2:y3;
9    assert
10     (y4 == 7 ||
11     y4 == 9);
12 }
```

```
1
2
3    (y1 = 8 &&
4
5      y2=y1-1 &&
6
7      y3=y1+1 &&
8    y4 = x0?y2:y3)
9    ==>
10     (y4 == 7 ||
11     y4 == 9);
```

Pitfalls

- All loops must be unwound, unwinding depth is fixed.
- Pointers are resolved through points-to analysis, dereference operations result in case splits. Array offsets can be problematic.
- Dynamic memory allocation results in state explosion.
- Floating point operations and modulo/division are complex.

Table of Contents

Summary

- Testing is simple but only tests for presence of functionality.
- Fuzzing uses test cases to explore other paths, might run forever.
- Static analysis has limited precision (e.g., aliasing).
- Symbolic execution needs guidance when searching through program.
- Formal verification is precise but arithmetic operations can be difficult.

All mechanisms (except testing) run into state explosion.

Questions?

?