Practical Program Analysis Milestone Discussion

2020-01-10

Prof. Michael Pradel, Daniel Lehmann

Agenda

- Course Organization (5 mins)
- Project Milestone Discussion (60 mins)
  - Task by Task
  - Common questions
  - Concepts recap: What do we need for concolic execution of Wasm?
  - No ready-to-copy solutions, but interactive Q&A with you
- Final Submission Tasks (20 mins)

Course Organization

- Milestone submission: yesterday (I cloned your repos)
- Final submission: February 9 (ca. 4 weeks from now)
- Presentations: week of February 10 to 14
    - 20% of your final grade
    - 15 mins talk + 10 mins Q&A
    - Every team member should present a part
    - Scheduling a date and time slot: via email (more details later)
    - Contents:
        - What were the goals of this project?
        - Presenting your project: concepts that you used, challenges you had
        - Live demo (make sure it works, plan for about ~5 minutes)

Milestone Discussion

A Note on Plagiarism

- Copying of other people’s code is not permitted
- Neither is copying of examples directly from tutorials
- See also https://www.student.uni-stuttgart.de/pruefungsorganisation/document/Leitfaden_Plagiatspraevention_Studierende.pdf (in German)

Task 1: Setup

- Install **WABT**, **Wasabi**, **Z3**, **Node.js**, **NPM** or **Yarn**, and a recent **browser** that can run WebAssembly.
- Install the dependencies, then run the **webserver** in `server/`, then go to http://localhost:8000 to check out the test harness.

- Just to get you started, should have been easy

- Both groups used Z3 from https://rise4fun.com/z3
    - You will need to install it locally for the next step!
    - Did anybody do that? Were there any problems with that?

Task 2: Test Programs

- Write **at least 5 programs** in the WebAssembly text format, that are used later for testing your concolic execution tool.
- Put them in `project/programs/milestone-tests/`.
- They should be **less than 30 instructions** each. (Smaller, focused tests are easier to understand. We can try larger programs later.)
- They should be **non-trivial**. (E.g., if they have zero control-flow constructs, that would be pointless because there is only a single path to explore.)
- Each `.wat` program should contain one **top-level comment** that explains why this test case is interesting. (E.g. "Tests whether our engine can handle paths of length > 1, because this test contains two nested ifs.")

- Written by **hand** ✔️
  - One group started with C programs (cool, but not required here)
- Less than 30 instructions, but not trivial ✔️
- **Top-level comment** why this **test case** is interesting ❌
  - "later used for testing your concolic execution tool"

Task 3: Instruction Coverage

- Ultimately, your concolic testing tool should produce inputs that explore the program "as good as possible". For that we need to measure coverage, given a program and a concrete input.
- Implement a Wasabi analysis that **measures instruction coverage**.
  - Side-question (you don't need to write down an answer, just think about it): What other types of coverage are there except instruction coverage?
  - Subtask: How do you know what 100% coverage is for a given program?
- See the analysis template in `project/analysis/coverage.js`.
- Make sure that when running any program with `project/harness.html`, it reports a coverage in the appropriate part of the page.

- What is **instruction coverage** (vs. **branch** coverage or instruction **counting**)? When is it used?
- **Total instructions**: `Wasabi.module.info.functions[1].instrCount`
- Some engineering required:
  1. **Instrument** binaries in programs/ dir (e.g., by adding call to wasabi to programs/build.sh)
  2. Load generated **wasabi.js** file and **instrumented wasm** file in harness.js
  3. **Run** instrumented binary instead of original one
- Analysis itself is simple: **all hooks**, adding `location.instr` to a `Set()`

Task 4: Z3 Warmup

- Ultimately, your concolic testing tool will generate path constraints (= formulas where the program's inputs are symbolic variables) that are solved by an SMT solver. Here, you get to know **Z3** (an SMT solver) by manually writing some inputs for it (= formulas in the SMT-lib 2.0 format) and letting Z3 solve them.
- Write **5 simple inputs** for Z3 (as `.smt2` files), e.g., formulas using propositional logic and integer arithmetic. Make sure they are valid, i.e., give them to Z3 to solve.
- Put those 5 files in `tasks/milestone/z3-warmup/`.
- See `tasks/milestone/z3-warmup/example.smt2` for an example representation of the formula `x > 0 & y > 0 & x + y < 42`. Solving that formula with Z3 gives:

- Conceptually easy: just think of some **simple logical formulas**
- **Syntax** maybe a bit weird
  - SMTlib boilerplate: declare-const, check-sat, get-model
  - Parentheses + prefix notation
- Write some examples together

```
(declare-const arg0 Int)
(assert (= (+ arg0 0) 0))
(check-sat)
(get-model)
```

Task 5: Manual Path Constraints

- The programs you analyze are in WebAssembly, but Z3 only understands its own format (that we got to know in the previous task). Before implementing this in your *automated* tool, you should practice generating path constraints from a program by hand.
- For the **5 programs in `project/programs/basic/`**, manually write down the **path constraints** as Z3 input files, if the program would have received **`0` for all its inputs**.
  - First step: What path did the program take, given these inputs?
- Put the `.smt2` files in `tasks/milestone/manual-path-constraints/`, with the basename of your `.smt2` files corresponding to the basename of the program.
- See `tasks/milestone/manual-path-constraints/example.smt2` for an example path constraint of the program `basic/if-eqz.wat` if it executed the `else` branch of the `if` (i.e., where the input was *not* zero).

- **Recap**: How is WebAssembly executed? What are path constraints?
- We do **two examples** together

Recap: WebAssembly Execution

- WebAssembly is a **stack machine**
  - There is an implicit “operand stack”
  - All instructions pop their inputs from the stack
  - And push their results onto the stack
- **Locals**: first N locals are the N function arguments, rest are like local variables
- Let’s evaluate the right program **by hand**
  - Draw explicit operand stack
  - Draw locals, indexed by their number

```
(func (export "main") (param i32)
   local.get 0
   i32.eqz
   if
      i32.const 0
      call $print
   else
      i32.const 1
      call $print
   end
)
```

Recap: Path Constraints

- Concolic testing wants to explore **new execution paths** in a program
- Execution path is determined by all **branches taken**
  - For an if: was the condition true or false?
- Express branch condition as a **logical formula** in terms of the program inputs
- Here: program = 1 function,
  inputs = function arguments

```
(func (export "main") (param i32)
   local.get 0
   i32.eqz
   if
       i32.const 0
       call $print
   else
       i32.const 1
       call $print
   end
)
```

Concolic Execution for WebAssembly

- Similar to regular execution
  - Uses also operand stack and locals etc.
- But: logical formulas (“**symbolic state**”) instead of concrete values
  - One “symbolic data structure” for each data structure during concrete execution
    - **Symbolic operand stack**
    - **Symbolic locals array**
    - (Symbolic globals, symbolic memory)
- Add to **path constraint** for **each branch**
- Together on the blackboard

```
(func (export "main") (param i32)
   local.get 0
   i32.eqz
   if
       i32.const 0
       call $print
   else
       i32.const 1
       call $print
   end
)
```

add-eqz.wat

```
(func (export "main") (param i32 i32) (result)
   local.get 0
   local.get 1
   i32.add
   i32.eqz
   if
       i32.const 0
       call $print
   else
       i32.const 1
       call $print
   end
)
```

Step 1: **Which path** is taken when **all inputs** are **zero**?

Step 2: Start with a template SMT2 file.
(See task 4)

Step 3: Go through **instructions step-by-step**.
What does each instruction do with its operands?
When we reach the if: translate into formula.

locals-2.wat

```
(func (export "main") (param i32) (result)
   (local i32)
   local.get 0
   i32.const 42
   i32.sub
   local.set 1
   local.get 0
   local.get 1
   i32.add
   i32.eqz
   if
       i32.const 0
       call $print
   else
       i32.const 1
       call $print
   end
)
```

Task 6: Implement what we just did

- `symbolicStack` array
  - Strings of SMT formulas
  - Initially empty
  - On binary instruction, such as `i32.add`: pop two values, build result formula, push result
- `symbolicLocals` array
  - Strings of SMT formulas
  - First N entries are initialized, e.g., to `"arg0"`
  - `get.local N`: push contents `symbolicLocals[N]` onto `symbolicStack`
  - `set.local N`: pop value from `symbolicStack`, write to `symbolicLocals[N]`
- `pathConstraint` array
  - Whenever `if` is reached: pop value (=formula) from `symbolicStack`, wrap in `(not …)` if condition was false, add to `pathConstraint` array
- After execution: `and`-together all values in `pathConstraint`, add SMT2 boilerplate (declare-const, get-model etc.)

Your Questions

Final Submission Tasks