Lab: Code Verification and Z3 Theorem Prover

(Week 5)

Yulei Sui, Mohamad Barbar, Hanyuan Li, Angela He

School of Computer Science and Engineering

University of New South Wales, Australia

Assignment-2 Spec and Code Released

Remember to `git pull` or `docker pull`!

Quiz-2, Exercise-2 and Assignment-2

- Quiz-2 with 25 questions (5 points), **due date: 23:59 Wednesday, Week 7**
  - Logical formula and predicate logic
  - Z3's knowledge and translation rules
- Lab-Exercise-2 (5 points), **due date: 23:59 Wednesday, Week 7**
  - **Goal:** Manually translate code into z3 formulas/constraints and verify the assertions embedded in the code.
  - **Specification:** `https://github.com/SVF-tools/Software-Security-Analysis/wiki/Lab-Exercise-2`
  - **SVF Z3 APIs:** `https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVF-Z3-API`
- Assignment-2 (20 points) **due date: 23:59 Wednesday, Week 8**
  - **Goal:** automatically perform assertion-based verification for code using static symbolic execution.
  - **Specification:** `https://github.com/SVF-tools/Software-Security-Analysis/wiki/Assignment-2`

Hints for Lab-Exercise-2

Looking at `test0()`:

```
//  assert(x==5);
z3::expr assert_cond =
    (z3Mgr->getZ3Expr("x") == z3Mgr->getZ3Expr(5));

result = z3Mgr->checkNegateAssert(assert_cond);
```

`checkNegateAssert()`

Given expression Q, adds $\neg Q$ to the solver and attempts to solve for satisfiability, thereby returning whether $prog \wedge \neg Q$ is **sat**.

`checkNegateAssert()`

Given expression Q, adds $\neg Q$ to the solver and attempts to solve for satisfiability, thereby returning whether *prog* $\wedge \neg Q$ is **sat**.

- Directly checking to see if `assert`(Q) is true for *all* cases is not a satisfiability problem (which checks for *existence*).

`checkNegateAssert()`

Given expression Q, adds $\neg Q$ to the solver and attempts to solve for satisfiability, thereby returning whether $prog \wedge \neg Q$ is **sat**.

- Directly checking to see if `assert`(Q) is true for *all* cases is not a satisfiability problem (which checks for *existence*).
- Checking **sat** of $prog \wedge \neg Q$ is an equivalent problem - if **sat**, then a counterexample exists, which means the assertion fails.

`checkNegateAssert()`

Given expression Q, adds $\neg Q$ to the solver and attempts to solve for satisfiability, thereby returning whether $prog \wedge \neg Q$ is **sat**.

- Directly checking to see if `assert`(Q) is true for *all* cases is not a satisfiability problem (which checks for *existence*).
- Checking **sat** of $prog \wedge \neg Q$ is an equivalent problem - if **sat**, then a counterexample exists, which means the assertion fails.
- We can turn assertions to satisfiability problems (ones that can be solved efficiently via Z3)!

`checkNegateAssert()`

Given expression Q, adds $\neg Q$ to the solver and attempts to solve for satisfiability, thereby returning whether $prog \wedge \neg Q$ is **sat**.

- Directly checking to see if `assert`(Q) is true for *all* cases is not a satisfiability problem (which checks for *existence*).
- Checking **sat** of $prog \wedge \neg Q$ is an equivalent problem - if **sat**, then a counterexample exists, which means the assertion fails.
- We can turn assertions to satisfiability problems (ones that can be solved efficiently via Z3)!
- If the program is *closed-world* (variables all fixed, nothing external), then we can directly check satisfiability of $prog \wedge Q$.

`addToSolver` and `getEvalExpr`

- `addToSolver(e)` adds `e` as a constraint to the solver.
- `getEvalExpr(e)` *does not add anything* to the solver, but evaluates `e` with all constraints added to the model at that point in time.

Methods to Be Implemented

You need to implement the following six functions in `Assignment-2`:

- `SSE::reachability`
- `SSE::collectAndTranslatePath`
- `SSE::handleCall`
- `SSE::handleRet`
- `SSE::handleNonBranch`
- `SSE::handleBranch`
- The required implementation parts are indicated with TODO comments, and you only need to fill up the code template if a method is partially implemented.

Debugging – Calling Context

The callstack object you will use to keep track of calling context is just a list of ICFG nodes. You can iterate over this and print the nodes.

C++

```
typedef std::vector<const ICFGNode*> CallStack;
CallStack callingCtx;
```

(`std::string Z3SSEMgr::callingCtxToStr(const CallStack& callingCtx).`)

Debugging – Calling Context

The callstack object you will use to keep track of calling context is just a list of ICFG nodes. You can iterate over this and print the nodes.

Python

```
self.callingCtx = []
(callingCtxToStr(self, callingCtx:  list) -> str.)
```

Debugging – Show Evaluation

You can dump an evaluation of the current constraints with `printExprValues()` (**C++**) or `printExprValues(callingCtx)` (**Python**).

```
-----------SVFVar and Value-----------
ObjVar2 (0x7f000002) Value:  NULL
ObjVar3 (0x7f000003) Value:  NULL
ObjVar5 (0x7f000005) Value:  NULL
ObjVar9 (0x7f000009) Value:  NULL
ObjVar12 (0x7f00000c) Value:  NULL
ObjVar15 (0x7f00000f) Value:  NULL
ValVar1 Value:  500
----------------------------------------
```

Debugging – Show Evaluation (unsat!)

I tried `printExprValues()` but my constraints aren't satisfiable!

```
-----------SVFVar and Value-----------
ass2:
/home/mbarbar/git/Software-Security-Analysis/Lab-Exercise-2/CPP/Z3Mgr.cpp:72:
z3::expr SVF::Z3Mgr::getEvalExpr(z3::expr):  Assertion ‘res != z3::unsat &&
"unsatisfied constraints!  Check your contradictory constraints added to
the solver"’ failed.
Aborted
```

Debugging – Show Constraints

Just print the solver!

C++

```
std::cout << getSolver() << std::endl;
```

Python

```
print(self.z3mgr.solver.sexpr())
```

```
(declare-fun |ctx:[ ] ValVar1| () Int)
(assert (= |ctx:[ ] ValVar1| 5000))
(assert (= |ctx:[ ] ValVar1| 3))
```

Assignment-1 Marks Released

Marks are out (please check via https://cgi.cse.unsw.edu.au/ give/Student/sturec.php) and let us go through some Assignment-1 issues!

Assignment-1 Common Errors

- **ICFG reachability**: without a callstack limit (e.g, 5), it is possible to get stuck in an infinite loop in the presence of recursion.
- **Points-to analysis**: Looking at the inappropriate edges of the node popped from the worklist (usually looking at outgoing edges for all types of edges).
  - Notably, when considering STORE edges, the destination node's points-to set changing (i.e., that node being pushed on the worklist) may induce new COPY edges. That means looking at incoming, not outgoing, STORE edges of the node popped from the worklist2.