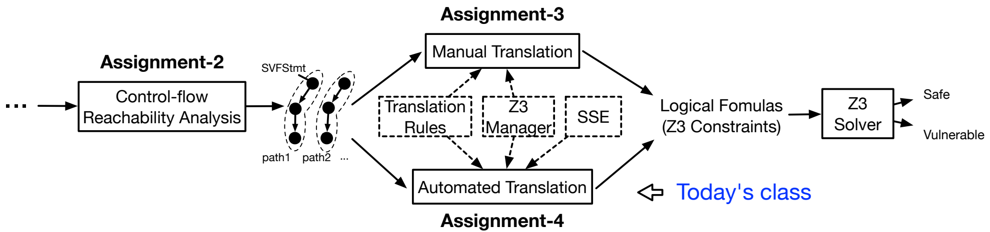
Assertion-based Verification Using Static Symbolic Execution

Yulei Sui

University of Technology Sydney, Australia

Automated Assertion-based Verification



Static Symbolic Execution (SSE)

- An static interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would.
- Automated testing technique that symbolically executes a program.
- Use symbolic execution to explore all program paths to find latent bugs.

Static Symbolic Execution for Assertion-based Verification

- Given a Hoare triple $P\ \{prog\}\ Q$,
  - P represents program inputs,
  - *prog* is the actual source code,
  - Q is the assertion(s) to be verified.
- SSE translates `SVFStmt` of each program path (which ends with an assertion) into a Z3 logical formula.
  - In our project, the path of each loop is bounded once for verification.
- Prove satisfiability of the logic formulas of each program path from the program entry to each assertion on the ICFG.

Recall (What We Have From Assignment 2)

Algorithm 1 Context sensitive control-flow reachability

```
Input : curEdge : ICFGEdge   dst : ICFGNode path : vector⟨ICFGEdge⟩   visited : set⟨ICFGEdge, callstack⟩;
dfs(path, curEdge, dst)
  curItem ← ⟨curEdge, callstack⟩
  visited.insert(curItem)
  path.push_back(curEdge)
  if src == dst then
    printICFGPath(path)
  foreach edge ∈ curEdge.dst.getOutEdges() do
    if edge.dst ∉ visited then
      if edge.isIntraCFGEdge() then
        dfs(path, edge, dst)
      else if edge.isCallCFGEdge() then
        callNode ← getSrcNode(edge)
        callstack.push_back(callNode)
        dfs(path, edge, dst)
      else if edge.isRetCFGEdge() then
        if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
          callstack.pop()
          dfs(path, edge, dst)
        else if callstack == ∅ then
          dfs(path, edge, dst)
  visited.erase(curItem)
  path.pop_back(src)
```

Translate each ICFG path into Z3 formulas

Algorithm 2 translatePath(path)

```
foreach edge ∈ path do
    if intra ← dyn_cast⟨Intra⟩(edge) then
        if handleIntra(intra) == false then
            return false
        else if call ← dyn_cast⟨CallEdge⟩(edge) then
            handleCall(call)
        else if ret ← dyn_cast⟨RetEdge⟩(edge) then
            handleRet(ret)
    return true
```

Algorithm 3 handleIntra(intraEdge)

```
if intraEdge.getCondition() && !handleBranch(intraEdge) then
    return false
else
    handleNonBranch(edge)
```

Algorithm 4 handleCall(callEdge)

```
getSolver().push();
foreach callPE ∈ calledge.getCallPEs() do
    lhs ← getZ3Expr(callPE.getLHSVarID());
    rhs ← getZ3Expr(callPE.getRHSVarID());
    addToSolver(lhs == rhs);
return true;
```

Algorithm 5 handleRet(retEdge)

```
rhs(getCtx());
if retPE ← retEdge.getRetPE() then
    rhs ← getEvalExpr(getZ3Expr(retPE.getRHSVarID()));
getSolver().pop();
if retPE ← retEdge.getRetPE() then
    lhs ← getZ3Expr(retPE.getLHSVarID());
    addToSolver(lhs == rhs);
return true;
```

Handle Intra-procedural CFG Edges (`handleIntra`)

Algorithm 2 `handleIntra(intraEdge)`

```
if intraEdge.getCondition() && !handleBranch(intraEdge) then
    return false
else
    handleNonBranch(edge)
```

`handleBranch(intraEdge)`

```
cond = intraEdge.getCondition()
successorVal = intraEdge.getSuccessorCondValue()
res = getEvalExpr(cond == successorVal)
if res.is_false() then
    addToSolver(cond! = successorVal)
    return false
else if res.is_true() then
    addToSolver(cond == successorVal)
    return true
else
    return true
```

`HandleNonBranch(intraEdge)`

```
dst ← intraEdge.getDstNode(); src ← intraEdge.getSrcNode()
foreach stmt ∈ dst.getSVFStmts() do
    if addr ← dyn_cast⟨AddrStmt⟩(stmt) then
        obj ← getMemObjAddress(addr.getRHSVarID())
        lhs ← getZ3Expr(addr.getLHSVarID())
        addToSolver(obj == lhs)
    else if copy ← dyn_cast⟨CopyStmt⟩(stmt) then
        lhs ← getZ3Expr(copy.getLHSVarID())
        rhs ← getZ3Expr(copy.getRHSVarID())
        addToSolver(rhs == lhs)
    else if load ← dyn_cast⟨LoadStmt⟩(stmt) then
        lhs ← getZ3Expr(load.getLHSVarID())
        rhs ← getZ3Expr(load.getRHSVarID())
        addToSolver(lhs == z3Mgr.loadValue(rhs))
    else if store ← dyn_cast⟨StoreStmt⟩(stmt) then
        lhs ← getZ3Expr(store.getLHSVarID())
        rhs ← getZ3Expr(store.getRHSVarID())
        z3Mgr.storeValue(lhs, rhs)
    else if gep ← dyn_cast⟨GepStmt⟩(stmt) then
        lhs ← getZ3Expr(gep.getLHSVarID())
        rhs ← getZ3Expr(gep.getRHSVarID())
        offset ← z3Mgr.getGepOffset(gep)
        gepAddress ← z3Mgr.getGepObjAddress(rhs, offset)
        addToSolver(lhs == gepAddress)
```

Scalar Example

Comparison between the concrete and symbolic states before the assertion.

```
void foo(unsigned x){
    if(x > 10) {
        y = x + 1;
    }
    else {
        y = 10;
    }
assert(y >= x + 1);
}
```

Scalar Example

Comparison between the concrete and symbolic states before the assertion.

```
void foo(unsigned x){
    if(x > 10) {
        y = x + 1;
    }
    else {
        y = 10;
    }
assert(y >= x + 1);
}
```

Concrete Execution
(Concrete states of x, y)

One execution:
x : 20
y : 21

Another execution:
x : 8
y : 10

Scalar Example

Comparison between the concrete and symbolic states before the assertion.

```
void foo(unsigned x){
    if(x > 10) {
        y = x + 1;
    }
    else {
        y = 10;
    }
assert(y >= x + 1);
}
```

Concrete Execution
(Concrete states of `x`, `y`)

One execution:
`x` : 20
`y` : 21

Another execution:
`x` : 8
`y` : 10

Symbolic Execution
(`getZ3Expr(x)` **represents** `x`'s **symbolic state**)

If branch:
`x` : $\texttt{getZ3Expr(x)} > 10 \land \texttt{getZ3Expr(x)} < \text{UINT\_MAX})$
`y` : $\texttt{getZ3Expr(x)} + 1$

Else branch:
`x` : $\texttt{getZ3Expr(x)} > 0 \land \texttt{getZ3Expr(x)} < 10$
`y` : 10

Scalar Example

Comparison between the concrete and symbolic states before the assertion.

```c
void foo(unsigned x){
    if(x > 10) {
        y = x + 1;
    }
    else {
        y = 10;
    }
assert(y >= x + 1);
}
```

Concrete Execution (Concrete states of x, y)	Symbolic Execution (getZ3Expr(x) **represents** x's **symbolic state**)
One execution:	If branch:
$x : 20$	x : getZ3Expr(x) $> 10 \land$ getZ3Expr(x) $<$ UINT_MAX)
$y : 21$	y : getZ3Expr(x) + 1
Another execution:	Else branch:
$x : 8$	x : getZ3Expr(x) $> 0 \land$ getZ3Expr(x) < 10
$y : 10$	y : 10

- Concrete execution: verify the assertion by **exhaustively** finding concrete states of x and y by exercising all possible inputs.
- Symbolic execution: verify the assertion by feeding the symbolic states (**logical formulas**) of x and y into **SMT Solver**.

Memory Operation Example

```
void foo(unsigned x) {
  int* p;
  int y;

  p = malloc(..);
  *p = x + 5;
  y = *p;
  assert(y>5);
}
```

Memory Operation Example

```
void foo(unsigned x) {
 int* p;
 int y;

 p = malloc(..);
 *p = x + 5;
 y = *p;
 assert(y>5);
}
```

Concrete Execution
(Concrete states)

One execution:

x	:	10
p	:	0x1234
0x1234	:	15
y	:	15

Another execution:

x	:	0
p	:	0x1234
0x1234	:	5
y	:	5

Memory Operation Example

```
void foo(unsigned x) {
 int* p;
 int y;

 p = malloc(..);
 *p = x + 5;
 y = *p;
 assert(y>5);
}
```

Concrete Execution
(Concrete states)

One execution:

x	:	10
p	:	0x1234
0x1234	:	15
y	:	15

Another execution:

x	:	0
p	:	0x1234
0x1234	:	5
y	:	5

Symbolic Execution
(Symbolic states)

x	:	getZ3Expr(x)
p	:	0x7f000001 virtual address from getMemObjAddress("malloc")
0x7f000001	:	getZ3Expr(x) + 5
y	:	getZ3Expr(x) + 5

Field Access for Struct and Array Example

```
struct st{
    int a;
    int b;
}
void foo(unsigned x) {
 struct st* p = malloc(..);
 q = &(p->b);
 *q = x;
 assert(*(&p->b) == x);
}
```

Field Access for Struct and Array Example

```c
struct st{
    int a;
    int b;
}
void foo(unsigned x) {
 struct st* p = malloc(..);
 q = &(p->b);
 *q = x;
 assert(*(&p->b) == x);
}
```

Concrete Execution
(Concrete states)
One execution:

x	:	10
p	:	0x1234
$\&(p \rightarrow b)$	:	0x1238
q	:	0x1238
0x1238	:	10

Another execution:

x	:	20
p	:	0x1234
$\&(p \rightarrow b)$	:	0x1238
q	:	0x1238
0x1238	:	20

Field Access for Struct and Array Example

```c
struct st{
    int a;
    int b;
}
void foo(unsigned x) {
 struct st* p = malloc(..);
 q = &(p->b);
 *q = x;
 assert(*(&p->b) == x);
}
```

Concrete Execution
(Concrete states)

One execution:

x	:	10
p	:	0x1234
$\&(p \rightarrow b)$	:	0x1238
q	:	0x1238
0x1238	:	10

Another execution:

x	:	20
p	:	0x1234
$\&(p \rightarrow b)$	:	0x1238
q	:	0x1238
0x1238	:	20

Symbolic Execution
(Symbolic states)

x	:	`getZ3Expr(x)`
p	:	0x7f000001 `virtual address from getMemObjAddress("malloc")`
$\&(p \rightarrow b)$	:	0x7f000002
q	:	0x7f000002 `field virtual address from getGepObjAddress(base, offset)`
0x7f000002	:	`getZ3Expr(x)`

The virtual address for modeling a field is based on the index of the field offset from the base pointer of a struct (nested struct will be flattened to allow each field to have a unique index)

Call and Return Example

```
1  int foo(int z) {
2      k = z;
3      return k;
4  }
5  int main(unsigned z) {
6    int x;
7    int y;
8    x = foo(3);
9    y = foo(z);
10   assert(x == 3);
11 }
```

Concrete Execution
(Concrete states)

One execution:

`z` : 10
stack push (calling foo at line 8)
`k` : 3
stack pop (returning from foo at line 4)
`x` : 3
stack push (calling foo at line 9)
`k` : 10
stack pop (returning from foo line 4)
`y` : 10

Symbolic Execution
(Symbolic states)

One execution:

`z` : `getZ3Expr(z)`
stack push (calling foo at line 8)
`k` : 3
stack pop (returning from foo at line 4)
`x` : 3
stack push (calling foo at line 9)
`k` : `getZ3Expr(z)`
stack pop (returning from foo line 4)
`y` : `getZ3Expr(z)`

What's next?

- (1) Understand SSE algorithms in the slides
- (2) Read through Assignment-4.pdf on Canvas to understand some examples for automated code verification.
- (3) Finish the quizzes of Assignment 4 on Canvas
- (4) Implement a automated translation from code to Z3 formulas using `SSE` and `Z3Mgr` i.e., coding task in Assignment 4