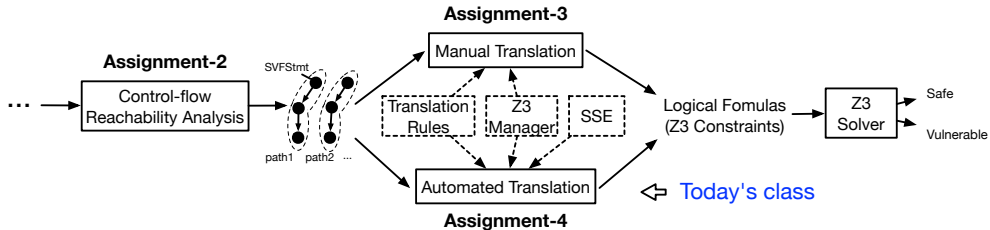


Assertion-based Verification Using Static Symbolic Execution

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 SVF_{stmt} 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.

Driver Program of SSE (What We Have From Assignment 2)

Algorithm 1 Context sensitive control-flow reachability

Input : src : ICFGNode dst : ICFGNode

path : vector<ICFGNode> visited : set<ICFGNode>;

```
1 dfs(path, src, dst)
2   visited.insert(src)
3   path.push_back(src)
4   if src == dst then
5     | print path
6   foreach edge ∈ src.getOutEdges() do
7     if edge.dst ∉ visited then
8       if edge.isIntraCFGEdge() then
9         | if handleIntra(edge) then
10          | | dfs(path, edge.dst, dst)
11       else if edge.isCallCFGEdge() then
12         | if handleCall(edge) then
13         | | dfs(path, edge.dst, dst)
14       else if edge.isRetCFGEdge() then
15         | if handleRet(edge) then
16         | | dfs(path, edge.dst, dst)
17   visited.erase(src)
18   path.pop_back(src)
```

Algorithm 2 handleIntra(intraEdge) (Override in SSE)

```
1   return true
```

Algorithm 3 handleCall(callEdge) (Override in SSE)

```
1   callNode ← getSrcNode(callEdge)
2   callstack.push_back(callNode)
3   return true
```

Algorithm 4 handleRet(retEdge) (Override in SSE)

```
1   retNode ← getDstNode(retEdge)
2   if callstack ≠ ∅ then
3     if callstack.back() == getCallICFGNode(retNode) then
4       | callstack.pop()
5       | return true
6     else
7       | return false
8   return true
```

Driver Program of SSE (What We Have From Assignment 2)

Algorithm 1 Context sensitive control-flow reachability

Input : src : ICFGNode dst : ICFGNode

path : vector<ICFGNode> visited : set<ICFGNode>;

```
1 dfs(path, src, dst)
2   visited.insert(src)
3   path.push_back(src)
4   if src == dst then
5     | print path
6   foreach edge ∈ src.getOutEdges() do
7     if edge.dst ∉ visited then
8       if edge.isIntraCFGEdge() then
9         | if handleIntra(edge) then
10          | | dfs(path, edge.dst, dst)
11       else if edge.isCallCFGEdge() then
12         | if handleCall(edge) then
13         | | dfs(path, edge.dst, dst)
14       else if edge.isRetCFGEdge() then
15         | if handleRet(edge) then
16         | | dfs(path, edge.dst, dst)
17   visited.erase(src)
18   path.pop_back(src)
```

Algorithm 2 `handleIntra(intraEdge)` (Override in SSE)

```
1   return true
```

Algorithm 3 `handleCall(callEdge)` (Override in SSE)

```
1   callNode ← getSrcNode(callEdge)
2   callstack.push_back(callNode)
3   return true
```

Algorithm 4 `handleRet(retEdge)` (Override in SSE)

```
1   retNode ← getDstNode(retEdge)
2   if callstack ≠ ∅ then
3     if callstack.back() == getCallICFGNode(retNode) then
4       | callstack.pop()
5       | return true
6     else
7       | return false
8   return true
```

Override the above three methods in SSE implementation!

Handle Intra-procedural CFG Edges (handleIntra)

Algorithm 2 `handleIntra(intraEdge)`

```
1 if intraEdge.getCondition() && !handleBranch(intraEdge)
  then
2   | return false
3 else
4   | handleNonBranch(edge)
```

`handleBranch(intraEdge)`

```
1 cond = intraEdge.getCondition()
2 successorVal = intraEdge.getSuccessorCondValue()
3 res = getEvalExpr(cond == successorVal)
4 if res.is_false() then
5   | addToSolver(cond != successorVal)
6   | return false
7 else if res.is_true() then
8   | addToSolver(cond == successorVal)
9   | return true
10 else
11   | return true
```

`HandleNonBranch(intraEdge)`

```
1 dst ← intraEdge.getDstNode(); src ← intraEdge.getSrcNode()
2 foreach stmt ∈ dst.getSVFStmts() do
3   if addr ← dyn_cast<AddrStmt>(stmt) then
4     | obj ← getMemObjAddress(addr.getRHSVarID())
5     | lhs ← getZ3Expr(addr.getLHSVarID())
6     | addToSolver(obj == lhs)
7   else if copy ← dyn_cast<CopyStmt>(stmt) then
8     | lhs ← getZ3Expr(copy.getLHSVarID())
9     | rhs ← getZ3Expr(copy.getRHSVarID())
10    | addToSolver(rhs == lhs)
11   else if load ← dyn_cast<LoadStmt>(stmt) then
12     | lhs ← getZ3Expr(load.getLHSVarID())
13     | rhs ← getZ3Expr(load.getRHSVarID())
14     | addToSolver(lhs == z3Mgr.loadValue(rhs))
15   else if store ← dyn_cast<StoreStmt>(stmt) then
16     | lhs ← getZ3Expr(store.getLHSVarID())
17     | rhs ← getZ3Expr(store.getRHSVarID())
18     | z3Mgr.storeValue(lhs, rhs)
19   else if gep ← dyn_cast<GepStmt>(stmt) then
20     | lhs ← getZ3Expr(gep.getLHSVarID())
21     | rhs ← getZ3Expr(gep.getRHSVarID())
22     | offset ← z3Mgr.getGepOffset(gep)
23     | gepAddress ← z3Mgr.getGepObjAddress(rhs, offset)
24     | addToSolver(lhs == gepAddress)
```

Handle Call (handleCall) and Return (handleRet) CFG Edges

Algorithm 3 `handleCall(callEdge)`

```
1  callNode ← callEdge.getSrcNode();
2  FunEntryNode ← callEdge.getDstNode();
3  callstack.push_back(callNode);
4  getSolver().push();
5  foreach callPE ∈ callEdge.getCallPEs() do
6    lhs ← getZ3Expr(callPE.getLHSVarID());
7    rhs ← getZ3Expr(callPE.getRHSVarID());
8    addToSolver(lhs == rhs);
9  return true;
```

Algorithm 4 `handleRet(retEdge)`

```
1  retNode ← retEdge.getDstNode();
2  rhs(getCtx());
3  lhs(getCtx());
4  if retPE ← retEdge.getRetPE() then
5    rhs ← getEvalExpr(getZ3Expr(retPE.getRHSVarID()));
6    lhs ← getZ3Expr(retPE.getLHSVarID());
7    if callstack ≠ ∅ then
8      if callstack.back() == getCallICFGNode(retNode) then
9        callstack.pop_back();
10       getSolver().pop();
11     else
12       return false;
13  if retEdge.getRetPE() then
14    addToSolver(lhs == rhs);
15  return true;
```

Scalar Example

Comparison between the concrete and symbolic states before the assertion.

```
1 void foo(unsigned x){  
2     if(x > 10) {  
3         y = x + 1;  
4     }  
5     else {  
6         y = 10;  
7     }  
8     assert(y >= x + 1);  
9 }
```

Scalar Example

Comparison between the concrete and symbolic states before the assertion.

Concrete Execution
(Concrete states of x, y)

```
1 void foo(unsigned x){  
2     if(x > 10) {  
3         y = x + 1;  
4     }  
5     else {  
6         y = 10;  
7     }  
8     assert(y >= x + 1);  
9 }
```

One execution:

x : 20

y : 21

Another execution:

x : 8

y : 9

Scalar Example

Comparison between the concrete and symbolic states before the assertion.

```
1 void foo(unsigned x){  
2     if(x > 10) {  
3         y = x + 1;  
4     }  
5     else {  
6         y = 10;  
7     }  
8     assert(y >= x + 1);  
9 }
```

Concrete Execution
(Concrete states of x, y)

One execution:

x : 20

y : 21

Another execution:

x : 8

y : 9

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

If branch:

x : $\text{getZ3Expr}(x) > 10 \wedge \text{getZ3Expr}(x) < \text{UINT_MAX}$

y : $\text{getZ3Expr}(x) + 1$

Else branch:

x : $\text{getZ3Expr}(x) > 0 \wedge \text{getZ3Expr}(x) < 10$

y : 10

Memory Operation Example

```
1 void foo(unsigned x) {  
2     int* p;  
3     int y;  
4  
5     p = malloc(..);  
6     *p = x + 5;  
7     y = *p;  
8     assert(y>5);  
9 }
```

Memory Operation Example

Concrete Execution
(Concrete states)

One execution:

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

Another execution:

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

```
1 void foo(unsigned x) {  
2   int* p;  
3   int y;  
4  
5   p = malloc(..);  
6   *p = x + 5;  
7   y = *p;  
8   assert(y>5);  
9 }
```

Memory Operation Example

Concrete Execution
(Concrete states)

One execution:

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

Another execution:

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

```
1 void foo(unsigned x) {
2   int* p;
3   int y;
4
5   p = malloc(..);
6   *p = x + 5;
7   y = *p;
8   assert(y>5);
9 }
```

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

```
1 struct st{  
2     int a;  
3     int b;  
4 }  
5 void foo(unsigned x) {  
6     struct st* p = malloc(..);  
7     q = &(p->b);  
8     *q = x;  
9     assert(*(&p->b) == x);  
10 }
```

Field Access for Struct and Array Example

Concrete Execution

(Concrete states)

One execution:

x	:	10
p	:	0x1234
&(p→b)	:	0x1238
q	:	0x1238
0x1238	:	10

Another execution:

x	:	20
p	:	0x1234
&(p→b)	:	0x1238
q	:	0x1238
0x1238	:	20

```
1 struct st{  
2     int a;  
3     int b;  
4 }  
5 void foo(unsigned x) {  
6     struct st* p = malloc(..);  
7     q = &(p->b);  
8     *q = x;  
9     assert(*(&p->b) == x);  
10 }
```


Field Access for Struct and Array Example

```
1 struct st{  
2     int a;  
3     int b;  
4 }  
5 void foo(unsigned x) {  
6     struct st* p = malloc(..);  
7     q = &(p->b);  
8     *q = x;  
9     assert(*(&p->b) == x);  
10 }
```

Concrete Execution
(Concrete states)

One execution:

x : 10
p : 0x1234
&(p→b) : 0x1238
q : 0x1238
0x1238 : 10

Another execution:

x : 20
p : 0x1234
&(p→b) : 0x1238
q : 0x1238
0x1238 : 20

Symbolic Execution
(Symbolic states)

x : getZ3Expr(x)
p : 0x7f000001
virtual address from
getMemObjAddress("malloc")
&(p→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

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)
```

```
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 }
```

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 3