Code Verification Using Symbolic Execution

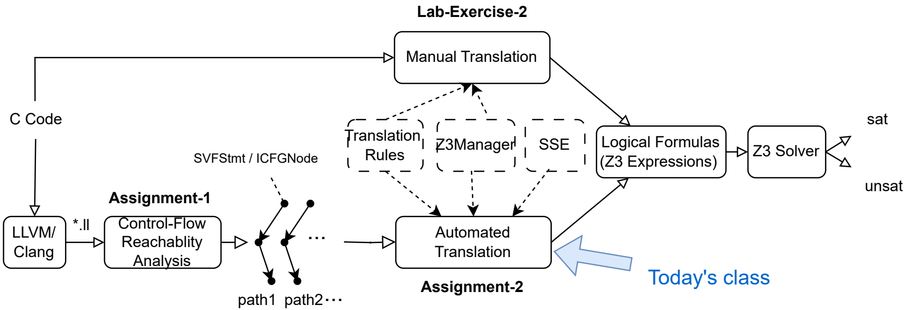
(Week 5)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Code Verification Using Static Symbolic Execution



Static Symbolic Execution (SSE)

- Automated analysis and testing technique that symbolically analyzes a program without runtime execution.
- Use symbolic execution to explore all program paths to find bugs and assertion validations.
- A static interpreter follows the program, assuming symbolic values for variables and inputs rather than obtaining actual inputs as normal program execution would.
- International Competition on Software Verification (SV-COMP): `https://sv-comp.sosy-lab.org/`

SSE for Assertion-based Verification

- Given a Hoare triple $P \; \{prog\} \; Q$,
  - P represents pre-condition,
  - *prog* is the program,
  - Q is the post-condition i.e., assertion(s) specifications.

```
// no-precondition
assume(true);        // P
    if(x > 10) {
        y = x + 1;
    }
    else {
        y = 10;
    }
assert(y >= x + 1);  //Q
```

$$\xrightarrow{\text{translate}} \underbrace{\exists \mathtt{x} \, \exists \mathtt{y} \;\; P \wedge S_{prog}(\mathtt{x}, \mathtt{y})) \wedge \neg Q(\mathtt{x}, \mathtt{y})}_{\text{logical formula } \psi} \xrightarrow{\text{feed into}} \text{SMT Solver}$$

SSE for Assertion-based Verification

- Given a Hoare triple P $\{prog\}$ Q,
  - P represents pre-condition,
  - $prog$ is the program,
  - Q is the post-condition i.e., assertion(s) specifications.

```
// no-precondition
assume(true);        // P
    if(x > 10) {
        y = x + 1;
    }
    else {
        y = 10;
    }
assert(y >= x + 1);  //Q
```

$$\xrightarrow{\text{translate}} \underbrace{\exists \mathtt{x}\ \exists \mathtt{y}\ \ P \wedge S_{prog}(\mathtt{x},\mathtt{y})) \wedge \neg Q(\mathtt{x},\mathtt{y})}_{\text{logical formula } \psi} \xrightarrow{\text{feed into}} \text{SMT Solver}$$

Check **unsat** of ψ (non-existence of counterexamples)!

SSE for Assertion-based Verification

- Translate each $\forall path \in prog$ consisting of a sequence of ICFGNodes $path = [N_1, N_2, \ldots, N_i, Q]$, from the entry node N_1 to an assertion Q on ICFG.
  - In `Assignment-2`, the node on each path appears at most once for verification.

SSE for Assertion-based Verification

- Translate each $\forall path \in prog$ consisting of a sequence of ICFGNodes $path = [N_1, N_2, \ldots, N_i, Q]$, from the entry node N_1 to an assertion Q on ICFG.
  - In `Assignment-2`, the node on each path appears at most once for verification.
- SSE translates `SVFStmts` of each ICFGNode (except the last one) on each *path* into Z3 expressions and validate whether they conform to the assertion Q by proving non-existence of counterexamples (Week 4).
  - $\forall path \in prog : \psi_{path} = \psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$
  - Checking **unsat** of each ψ_{path}. A **sat** of ψ_{path} indicates that there exists at least one counterexample from the **model** from the z3 solver.

SSE for Assertion-based Verification

- Translate each $\forall path \in prog$ consisting of a sequence of ICFGNodes $path = [N_1, N_2, \ldots, N_i, Q]$, from the entry node N_1 to an assertion Q on ICFG.
  - In `Assignment-2`, the node on each path appears at most once for verification.
- SSE translates `SVFStmts` of each ICFGNode (except the last one) on each *path* into Z3 expressions and validate whether they conform to the assertion Q by proving non-existence of counterexamples (Week 4).
  - $\forall path \in prog : \psi_{path} = \psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$
  - Checking **unsat** of each ψ_{path}. A **sat** of ψ_{path} indicates that there exists at least one counterexample from the **model** from the z3 solver.

```
void main(int x){
  assume(true);
    if(x > 10)
        y = x + 1;
    else
        y = 10;
  assert(y >= x + 1);
}
```

ψ_{path_1}: $\exists \mathtt{x}\ true \wedge ((\mathtt{x} > 10) \wedge (\mathtt{y} \equiv \mathtt{x} + 1)) \wedge \neg(\mathtt{y} \geq \mathtt{x} + 1)$ (`if` branch)
unsat (no counterexample found!)

ψ_{path_2}: $\exists \mathtt{x}\ true \wedge ((\mathtt{x} \leq 10) \wedge (\mathtt{y} \equiv 10)) \wedge \neg(\mathtt{y} \geq \mathtt{x} + 1)$ (`else` branch)
sat (a counterexample $x = 10$ found!)

Closed-World Programs and Assertion Checking

- If the program operates in a **closed-world** (value initializations are fixed and there are no inputs from externals and always has a single execution path), there is no need to find the existence of invalid inputs or counterexamples.
- For closed-world programs, only **logical errors** are verified against assertions, rather than finding the **counterexamples**. Simply checking satisfiability is **the same** as checking the non-existence of counterexamples.
  - Checking **unsat** of the $\psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$.
  - Checking **sat** of the $\psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \psi(Q)$.

```
void main(int x){
  x = 5;
    if(x > 10)
        y = x + 1;
    else
        y = 10;
  assert(y >= x + 1);
}
```

ψ_{path_1}: (`if` branch)
checking **unsat** of $x \equiv 5 \wedge ((\mathrm{x} > 10) \wedge (\mathrm{y} \equiv \mathrm{x} + 1)) \wedge \neg(\mathrm{y} \geq \mathrm{x} + 1)$
checking **sat** of x $x \equiv 5 \wedge ((\mathrm{x} > 10) \wedge (\mathrm{y} \equiv \mathrm{x} + 1)) \wedge (\mathrm{y} \geq \mathrm{x} + 1)$

ψ_{path_2}: (`else` branch)
checking **unsat** of $x \equiv 5 \wedge ((\mathrm{x} \leq 10) \wedge (\mathrm{y} \equiv 10)) \wedge \neg(\mathrm{y} \geq \mathrm{x} + 1)$
checking **sat** of $x \equiv 5 \wedge ((\mathrm{x} \leq 10) \wedge (\mathrm{y} \equiv 10)) \wedge (\mathrm{y} \geq \mathrm{x} + 1)$

Reachability Paths (Recall Assignment-1)

Algorithm 1: Context sensitive control-flow reachability

```
Input : curNode : ICFGNode   snk : ICFGNode   path : vector⟨ICFGNode⟩   callstack : vector⟨SVFInstruction⟩
        visited : set⟨ICFGNode, callstack⟩;
dfs(curNode, snk)                    // Argument curNode becomes to curEdge in Assignment-2
  pair = ⟨curNode, callstack⟩;
  if pair ∈ visited then
      return;
  visited.insert(pair);
  path.push_back(curNode);
  if src == snk then
    collectICFGPath(path);           // collectAndTranslatePath in Assignment-2
  foreach edge ∈ curNode.getOutEdges() do
    if edge.isIntraCFGEdge() then
        dfs(edge.dst, snk);
    else if edge.isCallCFGEdge() then
        callstack.push_back(edge.getCallSite());
        dfs(edge.dst, snk);
        callstack.pop_back();
    else if edge.isRetCFGEdge() then
        if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
            callstack.pop_back();
            dfs(edge.dst, snk);
            callstack.push_back(edge.getCallSite());
        else if callstack == ∅ then
            dfs(edge.dst, snk);
  visited.erase(pair);
  path.pop_back();
```

Overview of SSE Algorithms: Translate Paths 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)

```
expr_vector preCtxExprs(getCtx());
callPEs ← calledge → getCallPEs();
foreach callPE ∈ callPEs do
    preCtxExprs.push_back(rhs);
pushCallingCtx(calledge → getCallSite());
for i = 0; i < callPEs.size(); + + i do
    lhs ← getZ3Expr(callPEs[i] → getLHSVarID());
    addToSolver(lhs == preCtxExprs[i]);
return true;
```

Algorithm 5: handleRet(retEdge)

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

Handle Intra-procedural CFG Edges (`handleIntra`)

Algorithm 6: `handleIntra(intraEdge)`

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

Algorithm 7: `handleBranch(intraEdge)`

```
cond = intraEdge.getCondition();
succ = intraEdge.getSuccessorCondValue();
getSolver().push();
addToSolver(cond == succ);
res = getSolver().check();
getSolver().pop();
if res == unsat then
    return false;
else
    addToSolver(cond == succ);
    return true;
```

Algorithm 8: `HandleNonBranch(intraEdge)`

```
dst ← intraEdge.getDstNode();
src ← intraEdge.getSrcNode();
foreach stmt ∈ dst.getSVFStmts() do
    if addr ← dyn_cast⟨AddrStmt⟩(stmt) then
        // handle AddrStmt
    else if copy ← dyn_cast⟨CopyStmt⟩(stmt) then
        // handle CopyStmt
    else if load ← dyn_cast⟨LoadStmt⟩(stmt) then
        // handle LoadStmt
    else if store ← dyn_cast⟨StoreStmt⟩(stmt) then
        // handle StoreStmt
    else if gep ← dyn_cast⟨GepStmt⟩(stmt) then
        // handle GepStmt
    else if binary ← dyn_cast⟨BinaryStmt⟩(stmt) then
        // handle BinaryStmt
    else if cmp ← dyn_cast⟨CmpStmt⟩(stmt) then
        // handle CmpStmt
    else if phi ← dyn_cast⟨PhiStmt⟩(stmt) then
        // handle PhiStmt
    else if select ← dyn_cast⟨SelectStmt⟩(stmt) then
        // handle SelectStmt
    ...
```

Example 1: CmpStmt and BinaryOPStmt

```
void foo(int x) {
 int y, z, b;
 y = x;
 // C-like CmpStmt
 b = (x == y);
 // C-like BinaryOPStmt
 z = x + y;
 assert(z == 2 * x)
}
```

Example 1: CmpStmt and BinaryOPStmt

```
void foo(int x) {
 int y, z, b;
 y = x;
 // C-like CmpStmt
 b = (x == y);
 // C-like BinaryOPStmt
 z = x + y;
 assert(z == 2 * x)
}
```

One Concrete Execution
(Concrete states)

x : 5
y : 5
b : 1
z : 10

Another execution:

x : 10
y : 10
b : 1
z : 20

Example 1: CmpStmt and BinaryOPStmt

```
void foo(int x) {
 int y, z, b;
 y = x;
 // C-like CmpStmt
 b = (x == y);
 // C-like BinaryOPStmt
 z = x + y;
 assert(z == 2 * x)
}
```

One Concrete Execution
(Concrete states)

x : 5
y : 5
b : 1
z : 10

Another execution:

x : 10
y : 10
b : 1
z : 20

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

x : `getZ3Expr(x)`
y : `getZ3Expr(x)`
b : `ite(getZ3Expr(x)≡getZ3Expr(y), 1, 0)`
z : `getZ3Expr(x) + getZ3Expr(y)`

Checking satisfiability using "`getSolver().check()`".

Checking non-existence of counterexamples: $\psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$	Satisfiability
$\mathtt{y} \equiv \mathtt{x} \wedge \mathtt{b} \equiv \mathtt{ite}(\mathtt{x} \equiv \mathtt{y}, 1, 0) \wedge \mathtt{z} \equiv \mathtt{x} + \mathtt{y} \wedge \mathtt{z} \not\equiv 2 * \mathtt{x}$	unsat

Example 1: CMPSTMT and BINARYOPSTMT

```
void foo(int x) {
 int y, z, b;
 y = x;
 // C-like CmpStmt
 b = (x == y);
 // C-like BinaryOPStmt
 z = x + y;
 assert(z == 2 * x)
}
```

One Concrete Execution
(Concrete states)

x : 5
y : 5
b : 1
z : 10

Another execution:

x : 10
y : 10
b : 1
z : 20

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

x : `getZ3Expr(x)`
y : `getZ3Expr(x)`
b : `ite(getZ3Expr(x)≡getZ3Expr(y), 1, 0)`
z : `getZ3Expr(x) + getZ3Expr(y)`

In `Assignment-2`, we **only handle signed integers** including both positive and negative numbers and the assume that the program is **integer-overflow-free** in this assignment.

Pseudo-Code for Handling CmpStmt

Algorithm 9: Handle CmpStmt

```
op0 ← getZ3Expr(cmp.getOpVarID(0));
op1 ← getZ3Expr(cmp.getOpVarID(1));
res ← getZ3Expr(cmp.getResID());
switch cmp.getPredicate() do
    case CmpInst :: ICMP_EQ do
        addToSolver(res == ite(op0 == op1,
            getCtx().int_val(1), getCtx().int_val(0)));
    case CmpInst :: ICMP_NE do
        addToSolver(res == ite(op0! = op1,
            getCtx().int_val(1), getCtx().int_val(0)));
    case CmpInst :: ICMP_UGT do
        addToSolver(res == ite(op0 > op1,
            getCtx().int_val(1), getCtx().int_val(0)));
    case CmpInst :: ICMP_SGT do
        addToSolver(res == ite(op0 > op1,
            getCtx().int_val(1), getCtx().int_val(0)));
    case CmpInst :: ICMP_UGE do
        addToSolver(res == ite(op0 >= op1,
            getCtx().int_val(1), getCtx().int_val(0)));
    ...
```

Algorithm 10: Pseudo-Code for Handling CmpStmt

```
case CmpInst :: ICMP_SGE do
    addToSolver(res == ite(op0 >= op1,
        getCtx().int_val(1), getCtx().int_val(0)));
case CmpInst :: ICMP_ULT do
    addToSolver(res == ite(op0 < op1,
        getCtx().int_val(1), getCtx().int_val(0)));
case CmpInst :: ICMP_SLT do
    addToSolver(res == ite(op0 < op1,
        getCtx().int_val(1), getCtx().int_val(0)));
case CmpInst :: ICMP_ULE do
    addToSolver(res == ite(op0 <= op1,
        getCtx().int_val(1), getCtx().int_val(0)));
case CmpInst :: ICMP_SLE do
    addToSolver(res == ite(op0 <= op1,
        getCtx().int_val(1), getCtx().int_val(0)));
```

Handle BinaryOPStmt

Algorithm 10: Handle BinaryOpStmt

```
op0 ← getZ3Expr(binary.getOpVarID(0));
op1 ← getZ3Expr(binary.getOpVarID(1));
res ← getZ3Expr(binary.getResID());
switch binary.getOpcode() do
    case BinaryOperator :: Add do
        addToSolver(res == op0 + op1);
    case BinaryOperator :: Sub do
        addToSolver(res == op0 − op1);
    case BinaryOperator :: Mul do
        addToSolver(res == op0 × op1);
    case BinaryOperator :: SDiv do
        addToSolver(res == op0/op1);
    case BinaryOperator :: SRem do
        addToSolver(res == op0%op1);
    case BinaryOperator :: Xor do
        addToSolver(int2bv(32, res) ==
            (int2bv(32, op0) ⊕ int2bv(32, op1)));
    case BinaryOperator :: And do
        addToSolver(int2bv(32, res) ==
            (int2bv(32, op0)&int2bv(32, op1)));
    ...
```

Algorithm 10: Handle BinaryOpStmt

```
case BinaryOperator :: Or do
    addToSolver(int2bv(32, res) ==
        (int2bv(32, op0)|int2bv(32, op1)));
case BinaryOperator :: AShr do
    addToSolver(int2bv(32, res) ==
        ashr(int2bv(32, op0), int2bv(32, op1)));
case BinaryOperator :: Shl do
    addToSolver(int2bv(32, res) ==
        shl(int2bv(32, op0), int2bv(32, op1)));
```

Example 2: Memory Operation

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

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

Example 2: Memory Operation

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

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

Example 2: Memory Operation

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

 p = malloc(..);
 *p = x + 5;
 y = *p;
 assert(y==x+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(ObjVarID)
0x7f000001	:	getZ3Expr(x) + 5
y	:	getZ3Expr(x) + 5

Checking non-existence of counterexamples:

$\psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$	Satisfiability
$\mathtt{p} \equiv \mathtt{0x7f000001} \wedge \mathtt{y} \equiv \mathtt{x} + 5 \wedge \mathtt{y} \neq \mathtt{x} + 5$	unsat

Pseudo-Code for Handling Memory Operation

Algorithm 11: Handle AddrStmt

```
obj ← getMemObjAddress(addr.getRHSVarID());
lhs ← getZ3Expr(addr.getLHSVarID());
addToSolver(obj == lhs);
```

Algorithm 12: Handle LoadStmt

```
lhs ← getZ3Expr(load.getLHSVarID());
rhs ← getZ3Expr(load.getRHSVarID());
addToSolver(lhs == z3Mgr.loadValue(rhs));
```

Algorithm 13: Handle StoreStmt

```
lhs ← getZ3Expr(store.getLHSVarID());
rhs ← getZ3Expr(store.getRHSVarID());
addToSolver(lhs == z3Mgr.storeValue(rhs));
```

Example 3: Field Access for Struct and Array

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

Example 3: Field Access for Struct and Array

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

Concrete Execution
(Concrete states)

One execution:

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

Another execution:

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

Example 3: Field Access for Struct and Array

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

Concrete Execution
(Concrete states)
One execution:

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

Another execution:

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

Symbolic Execution
(Symbolic states)

x	:	getZ3Expr(x)
p	:	0x7f000001 virtual address from getMemObjAddress(ObjVarID)
&(p→b)	:	0x7f000002
q	:	0x7f000002 field virtual address from getGepObjAddress(base, offset)
0x7f000002	:	getZ3Expr(x)
k	:	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)

Example 3: Field Access for Struct and Array

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

Concrete Execution
(Concrete states)

One execution:

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

Another execution:

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

Symbolic Execution
(Symbolic states)

x	:	`getZ3Expr(x)`

p	:	0x7f000001
		`virtual address from`
		`getMemObjAddress(ObjVarID)`

$\&(p \rightarrow b)$	:	0x7f000002

q	:	0x7f000002
		`field virtual address from`
		`getGepObjAddress(base, offset)`

0x7f000002	:	`getZ3Expr(x)`
k	:	`getZ3Expr(x)`

Checking non-existence of counterexamples:

$\psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$	Satisfiability
$p \equiv 0x7f000001 \wedge q \equiv 0x7f000002 \wedge k \equiv x \wedge k \neq x$	unsat

Pseudo-Code for Handling Field Access (GEPSTMT)

Algorithm 13: Handle GEPSTMT

```
lhs ← getZ3Expr(gep.getLHSVarID());
rhs ← getZ3Expr(gep.getRHSVarID());
offset ← z3Mgr.getGepOffset(gep);
gepAddress ← z3Mgr.getGepObjAddress(rhs, offset);
addToSolver(lhs == gepAddress);
```

Method `getGepObjAddress` supports both `struct` and `array` accesses using a base pointer and element index.

In `Assignment-2`, **we don't consider object byte sizes** and low-level incompatible type casting in `Assignment-2`.

```
z3::expr Z3SSEMgr::getGepObjAddress(z3::expr pointer, u32_t offset) {
    NodeID obj = getInternalID(z3Expr2NumValue(pointer));
    // Find the baseObj and return the field object.
    // The indices of sub-elements of a nested aggregate object has been flattened
    NodeID gepObj = svfir->getGepObjVar(obj, offset);
    if (obj == gepObj)
        return getZ3Expr(obj);
    else
        return createExprForObjVar(SVFUtil::cast<GepObjVar>(svfir->getGNode(gepObj)));
}
```

Example 4: Branches

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

Example 4: Branches

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

One Concrete Execution
(concrete states)

One execution:
x : 20
y : 21

Another execution:
x : 8
y : 10

Example 4: Branches

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

One Concrete Execution (concrete states)	Symbolic Execution (Symbolic states)
One execution:	If branch:
$\mathtt{x}$: 20	$\mathtt{x}$: `getZ3Expr(x)` > 10
$\mathtt{y}$: 21	$\mathtt{y}$: `getZ3Expr(x)` + 1
Another execution:	Else branch:
$\mathtt{x}$: 8	$\mathtt{x}$: `getZ3Expr(x)` ≤ 10
$\mathtt{y}$: 10	$\mathtt{y}$: 10

Checking non-existence of counterexamples:

Path	$\psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$	Satisfiability	Counterexample
$\ell_1 \rightarrow \ell_2 \rightarrow \ell_3 \rightarrow \ell_8$ (if.then branch)	$\mathtt{x} > 10 \wedge \mathtt{y} \equiv \mathtt{x} + 1 \wedge \mathtt{y} < \mathtt{x} + 1$	unsat	$\varnothing$
$\ell_1 \rightarrow \ell_2 \rightarrow \ell_6 \rightarrow \ell_8$ (if.else branch)	$\mathtt{x} \leq 10 \wedge \mathtt{y} \equiv 10 \wedge \mathtt{y} < \mathtt{x} + 1$	sat	$\{\mathtt{x} : 10, \mathtt{y} : 10\}$

Getting the potential counterexample via "`getSolver().get_model()`" after "`getSolver().check()`".

- For the if branch, the assertion at ℓ_8 is valid, i.e., no counterexamples exist.
- For the else branch, the assertion at ℓ_8 is invalid. A counterexample is "$\mathtt{x} : 10, \mathtt{y} : 10$".

Example 4: Branches

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

Source code

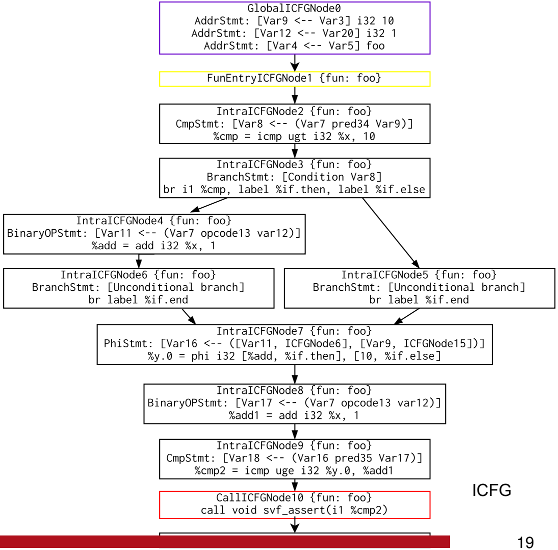
```llvm
define void @foo(i32 %x) #0 {
entry:
  %cmp = icmp ugt i32 %x, 10
  br i1 %cmp, label %if.then, label %if.else

if.then:
  %add = add i32 %x, 1
  br label %if.end

if.else:
  br label %if.end

if.end: = %if.else, %if.then
  %y.0 = phi i32 [%add, %if.then], [10, %if.else]
  %add1 = add i32 %x, 1
  %cmp2 = icmp uge i32 %y.0, %add1
  call void @svf_assert(i1 zeroext %cmp2)
  ret void
}
```

LLVM IR

Example 4: Branches

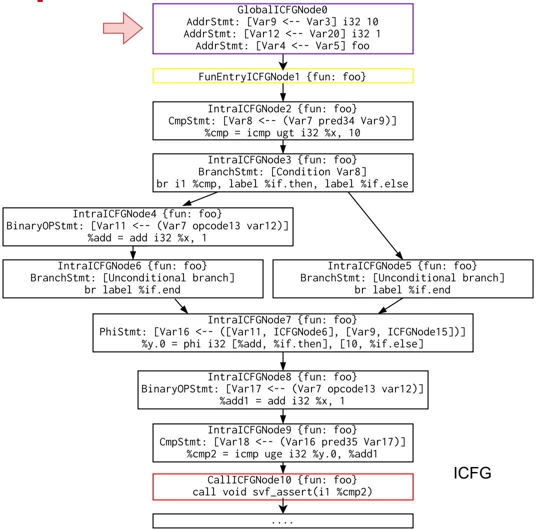
```
define void @foo(i32 %x) #0 {
entry:
  %cmp = icmp ugt i32 %x, 10
  br i1 %cmp, label %if.then, label %if.else

if.then:
  %add = add i32 %x, 1
  br label %if.end

if.else:
  br label %if.end

if.end: = %if.else, %if.then
  %y.0 = phi i32 [%add, %if.then], [10, %if.else]
  %add1 = add i32 %x, 1
  %cmp2 = icmp uge i32 %y.0, %add1
  call void @svf_assert(i1 zeroext %cmp2)
  ret void
}
```

LLVM IR



ICFG

Example 4: Branches



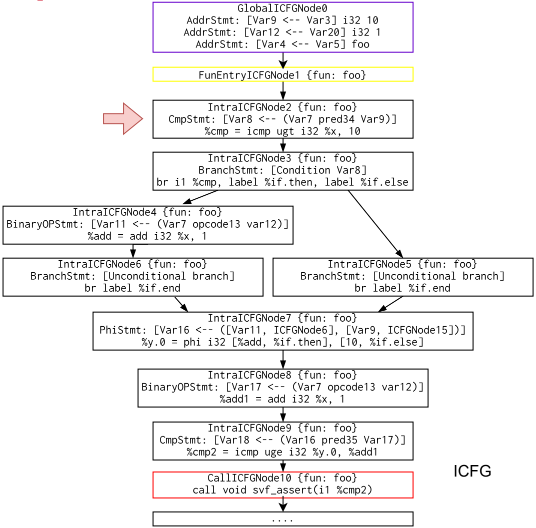
ICFG

Verifying path: 0 → 1 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → *svf_assert* (if.then branch)

```
-------------SVFVar and Value-------------
ObjVar5 (0x7f000005)    Value: NULL
ValVar4                 Value: 0x7f000005
ValVar9                 Value: 10
ValVar12                Value: 1
                 ...
------------------------------------------
```

The values of Z3 expressions for each SVFVar after analyzing `GlobalICFGNode0`

Example 4: Branches



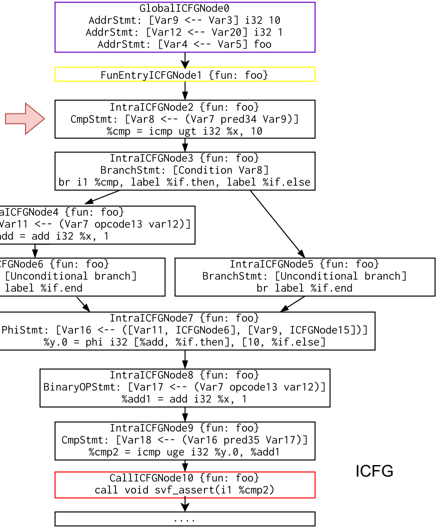
ICFG

Verifying path: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow$ *svf_assert* (if.then branch)

```
## Analyzing IntraICFGNode2 {fun: foo}
CmpStmt: [Var8 <-- (Var7 predicate34 Var9)]
   %cmp = icmp ugt i32 %x, 10
==> (not (<= ValVar7 ValVar9))
==> (= ValVar8 1)
                    ...
```

Code for handling `CmpStmt` has been implemented in the HandleNonBranch() function.

Example 4: Branches



ICFG

Verifying path: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow$ *svf_assert* (if.then branch)

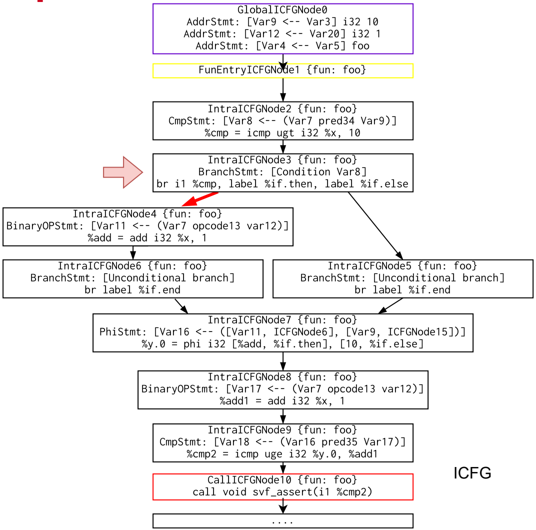
```
-------------SVFVar and Value-------------
ObjVar5 (0x7f000005)     Value: NULL
ValVar4                  Value: 0x7f000005
ValVar9                  Value: 10
ValVar12                 Value: 1
ValVar7                  Value: 11
+ValVar8                 Value: 1
                   ...
------------------------------------------
```

Analyzing IntraICFGNode2 {fun: foo}

CmpStmt: [Var8 ← (Var7 pred34 Var9)]

`%cmp = icmp ugt i32 %x, 10`

Example 4: Branches



ICFG

Algorithm 14: 3 handleIntra(intraEdge)

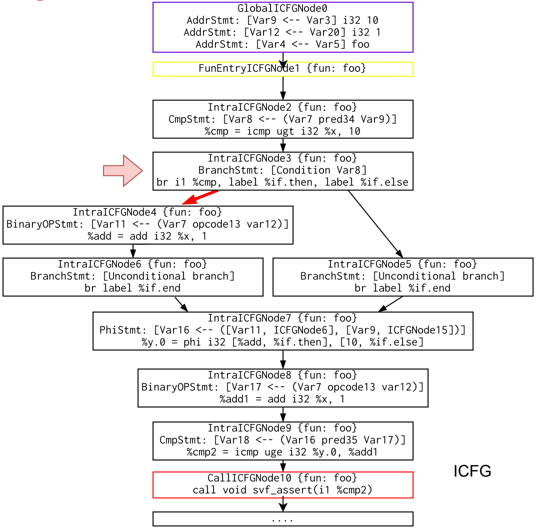
```
if intraEdge.getCondition() &&
  !handleBranch(intraEdge) then
    return false;
else
    handleNonBranch(edge);
```

Algorithm 15: handleBranch(intraEdge)

```
cond = intraEdge.getCondition();
succ = intraEdge.getSuccessorCondValue();
getSolver().push();
addToSolver(cond == succ);
res = getSolver().check();
getSolver().pop();
if res == unsat then
    return false;
else
    addToSolver(cond == succ);
    return true;
```

Note: getSolver().push() creates a new stack frame for maintaining the newly added Z3 constraints.

Example 4: Branches



ICFG

Algorithm 16: 3 `handleIntra(intraEdge)`

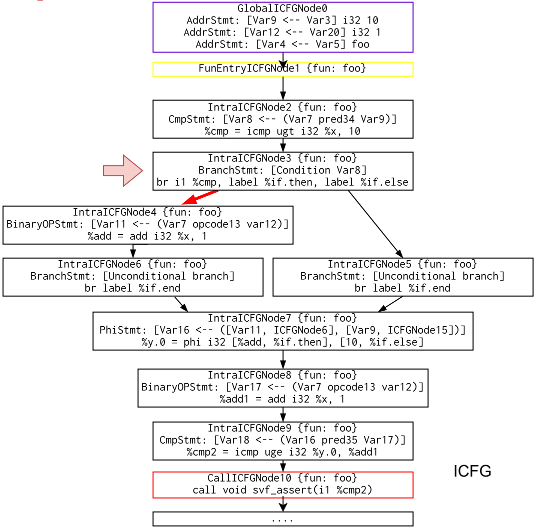
```
if intraEdge.getCondition() &&
  !handleBranch(intraEdge) then
    return false;
else
    handleNonBranch(edge);
```

Algorithm 17: `handleBranch(intraEdge)`

```
cond = intraEdge.getCondition();
succ = intraEdge.getSuccessorCondValue();
getSolver().push();
addToSolver(cond == succ);
res = getSolver().check();
getSolver().pop();
if res == unsat then
    return false;
else
    addToSolver(cond == succ);
    return true;
```

Note: `getSolver().pop()` pops out the top stack frame, which contains the Z3 constraint `cond == succ`.

Example 4: Branches



ICFG

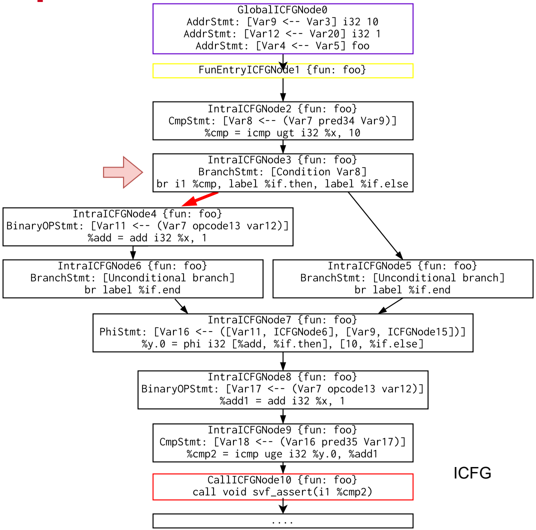
Algorithm 18: 3 handleIntra(intraEdge)

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

Algorithm 19: handleBranch(intraEdge)

```
cond = intraEdge.getCondition();
succ = intraEdge.getSuccessorCondValue();
getSolver().push();
addToSolver(cond == succ);
res = getSolver().check();
getSolver().pop();
if res == unsat then
    return false;
else
    addToSolver(cond == succ);
    return true;
```

Example 4: Branches



ICFG

Verifying path: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow$ *svf_assert* (if.then branch)

```
-------------SVFVar and Value-------------
ObjVar5 (0x7f000005)     Value: NULL
ValVar4                  Value: 0x7f000005
ValVar9                  Value: 10
ValVar12                 Value: 1
ValVar7                  Value: 11
ValVar8                  Value: 1
                   ...
------------------------------------------
```

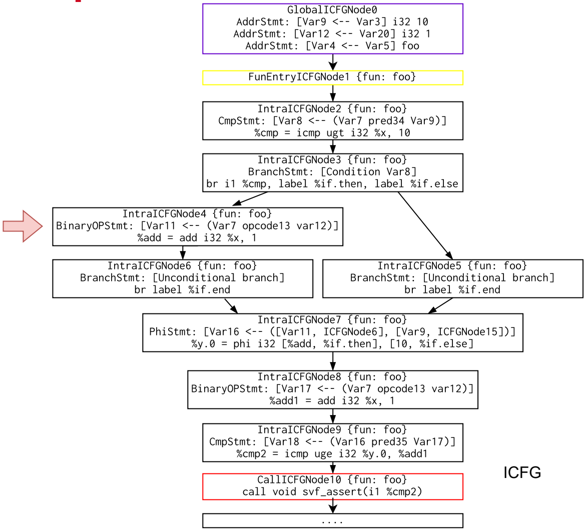
Branch IntraCFGEdge: [`ICFGNode4` $\leftarrow$ `ICFGNode3`]

branchCondition: `%cmp` $=$ `icmp ugt i32 %x, 10`

($=$ `ValVar8` 1)

This conditional ICFGEdge is **feasible**!!

Example 4: Branches



ICFG

Verifying path: 0 → 1 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → *svf_assert* (if.then branch)

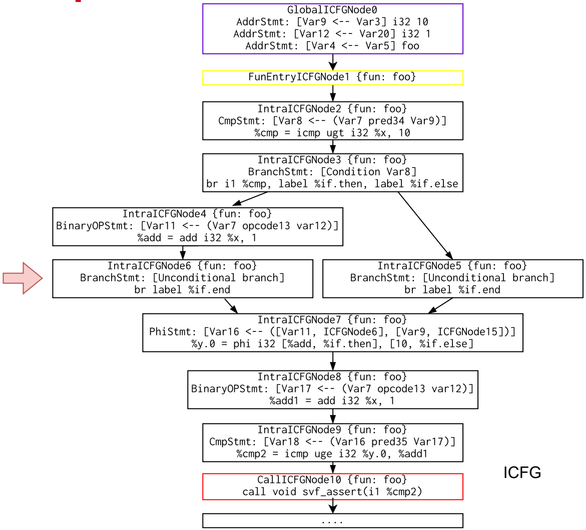
```
-------------SVFVar and Value-------------
ObjVar5 (0x7f000005)    Value: NULL
ValVar4                 Value: 0x7f000005
ValVar9                 Value: 10
ValVar12                Value: 1
ValVar7                 Value: 11
ValVar8                 Value: 1
+ValVar11               Value: 12
                   ...
------------------------------------------
```

Analyzing IntraICFGNode4 {fun: foo}

BinaryOPStmt: [Var11 ← (Var7 opcode13 var12)]

`%add = add i32 %x, 1`

Example 4: Branches

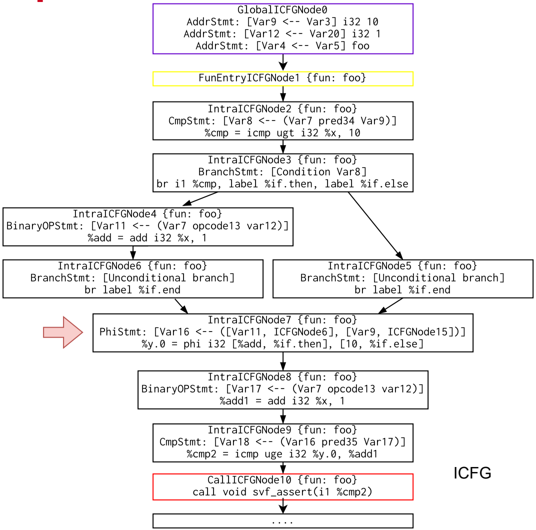


ICFG

Algorithm 20: 3 handleIntra(intraEdge)

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

Example 4: Branches

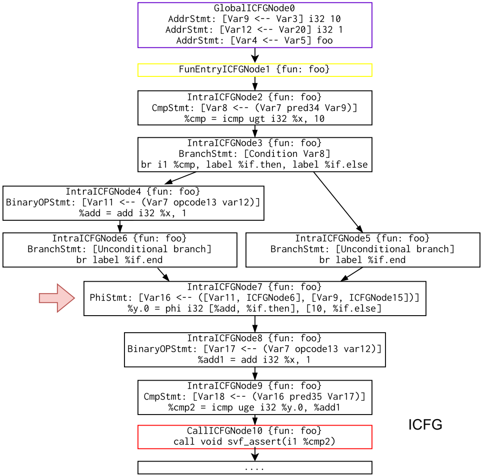


ICFG

Algorithm 21: 3 handlePhi(intraEdge)

```
res ← getZ3Expr(phi.getResID());
opINodeFound ← false;
for i ← 0 to phi.getOpVarNum() − 1 do
    if srcNode.getFun().postDominate(srcNode.getBB(),
    phi.getOpICFGNode(i).getBB()) then
        ope ← getZ3Expr(phi.getOpVar(i).getId());
        addToSolver(res == ope);
        opINodeFound ← true;
```

Example 4: Branches



ICFG

Verifying path: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow$ *svf_assert* (if.then branch)

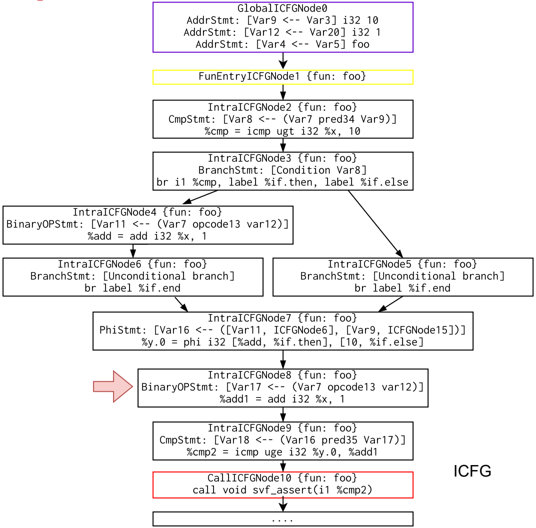
```
-------------SVFVar and Value-------------
ObjVar5 (0x7f000005)     Value: NULL
ValVar4                  Value: 0x7f000005
ValVar9                  Value: 10
ValVar12                 Value: 1
ValVar7                  Value: 11
ValVar8                  Value: 1
ValVar11                 Value: 12
+ValVar16                Value: 12
                    ...
------------------------------------------
```

\## Analyzing IntraICFGNode7 {fun: foo}

PhiStmt: [Var16 ← ([Var11, ICFGNode6], [Var9, ICFGNode15])]

`%y.0 = phi i32 [%add, %if.then], [10, %if.else]`

Example 4: Branches



ICFG

Verifying path: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow$ *svf_assert* (if.then branch)

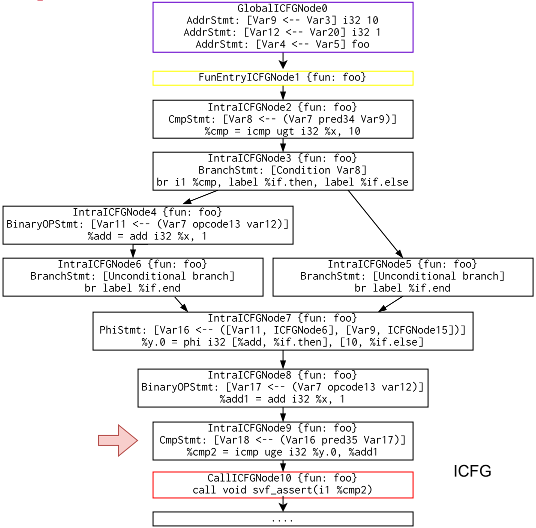
```
-------------SVFVar and Value-------------
ObjVar5 (0x7f000005)     Value: NULL
ValVar4                  Value: 0x7f000005
ValVar9                  Value: 10
ValVar12                 Value: 1
ValVar7                  Value: 11
ValVar8                  Value: 1
ValVar11                 Value: 12
ValVar16                 Value: 12
+ValVar17                Value: 12
                  ...
------------------------------------------
```

Analyzing IntraICFGNode8 {fun: foo}

BinaryOPStmt: [Var17 ← (Var7 opcode13 var12)]

`%add1 = add i32 %x, 1`

Example 4: Branches



ICFG

Verifying path: 0 → 1 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → *svf_assert* (if.then branch)

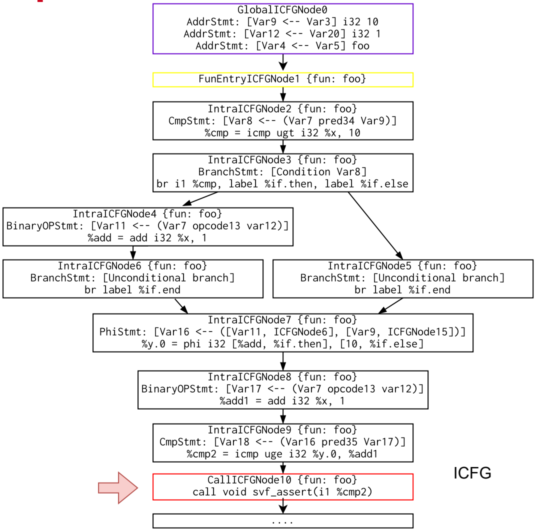
```
-------------SVFVar and Value-------------
ObjVar5 (0x7f000005)     Value: NULL
ValVar4                  Value: 0x7f000005
ValVar9                  Value: 10
ValVar12                 Value: 1
ValVar7                  Value: 11
ValVar8                  Value: 1
ValVar11                 Value: 12
ValVar16                 Value: 12
ValVar17                 Value: 12
+ValVar18                Value: 1
                    ...
------------------------------------------
```

Analyzing IntraICFGNode9 {fun: foo}

CmpStmt: [Var18 ← (Var16 pred35 Var17)]

`%cmp2 = icmp uge i32 %y.0, %add1`

Example 4: Branches



ICFG

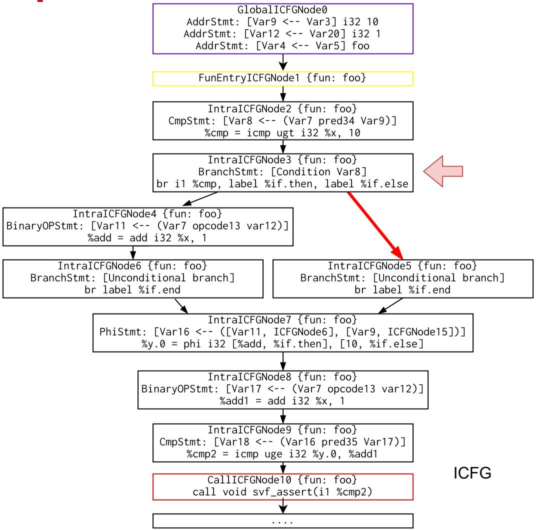
Verifying path: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow$ *svf_assert* (if.then branch)

```
-------------SVFVar and Value-------------
ObjVar5 (0x7f000005)     Value: NULL
ValVar4                  Value: 0x7f000005
ValVar9                  Value: 10
ValVar12                 Value: 1
ValVar7                  Value: 11
ValVar8                  Value: 1
ValVar11                 Value: 12
ValVar16                 Value: 12
ValVar17                 Value: 12
ValVar18                 Value: 1
                         ...
------------------------------------------
```

The assertion is successfully verified!!

START: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow$ *svf_assert*

Example 4: Branches



ICFG

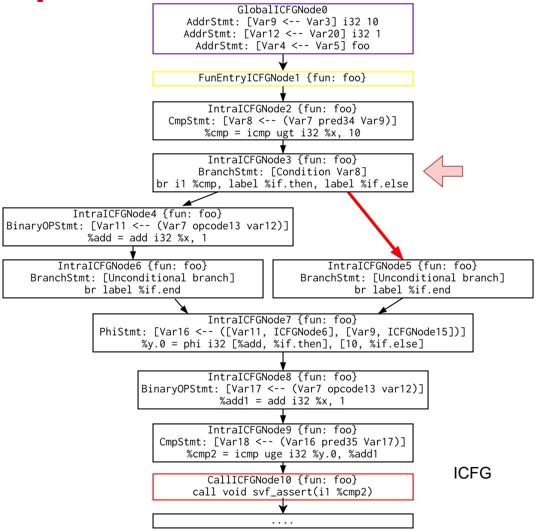
Algorithm 22: 3 `handleIntra(intraEdge)`

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

Algorithm 23: `handleBranch(intraEdge)`

```
cond = intraEdge.getCondition();
succ = intraEdge.getSuccessorCondValue();
getSolver().push();
addToSolver(cond == succ);
res = getSolver().check();
getSolver().pop();
if res == unsat then
    return false;
else
    addToSolver(cond == succ);
    return true;
```

Example 4: Branches



ICFG

Verifying path: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow$ *svf_assert* (if.else branch)

```
-------------SVFVar and Value-------------
ObjVar5 (0x7f000005)     Value: NULL
ValVar4                  Value: 0x7f000005
ValVar9                  Value: 10
ValVar12                 Value: 1
ValVar7                  Value: 10
ValVar8                  Value: 0
                 ...
------------------------------------------
```

Branch IntraCFGEdge: [`ICFGNode4` $\leftarrow$ `ICFGNode3`]

branchCondition: `%cmp` $=$ `icmp ugt i32 %x, 10`

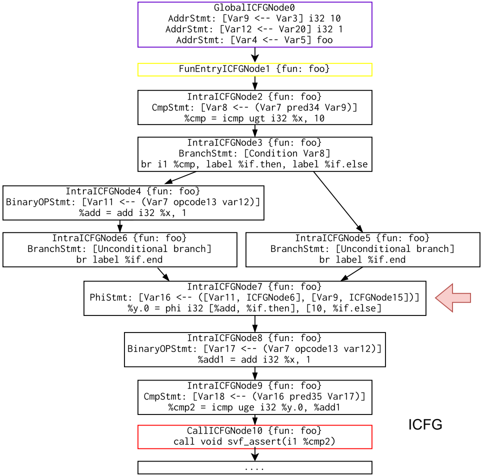
($=$ `ValVar8` 0)

This conditional ICFGEdge is **feasible**!!

In this path, `ValVar8`'s value is chaged to 0,

therefore, `ValVar7`'s value is chaged to 10.

Example 4: Branches



ICFG

Verifying path: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow$ *svf_assert* (if.else branch)

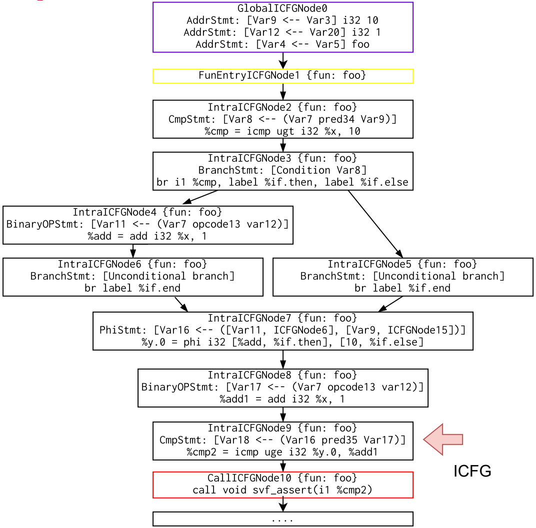
```
-------------SVFVar and Value-------------
ObjVar5 (0x7f000005)     Value: NULL
ValVar4                  Value: 0x7f000005
ValVar9                  Value: 10
ValVar12                 Value: 1
ValVar7                  Value: 10
ValVar8                  Value: 1
+ValVar16                Value: 10
                   ...
------------------------------------------
```

Analyzing IntraICFGNode7 {fun: foo}

PhiStmt: [Var16 $\leftarrow$ ([Var11, ICFGNode6], [Var9, ICFGNode15])]

`%y.0 = phi i32 [%add, %if.then], [10, %if.else]`

Example 4: Branches



ICFG

Verifying path: 0 → 1 → 2 → 3 → 5 → 7 → 8 → 9 → *svf_assert* (if.else branch)

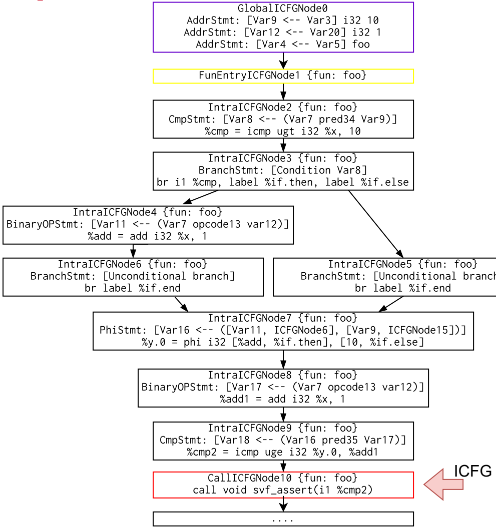
```
-------------SVFVar and Value-------------
ObjVar5 (0x7f000005)     Value: NULL
ValVar4                  Value: 0x7f000005
ValVar9                  Value: 10
ValVar12                 Value: 1
ValVar7                  Value: 11
ValVar8                  Value: 1
ValVar16                 Value: 10
ValVar17                 Value: 11
+ValVar18                Value: 0
                  ...
------------------------------------------
```

Analyzing IntraICFGNode9 {fun: foo}

CmpStmt: [Var18 ← (Var16 pred35 Var17)]

`%cmp2 = icmp uge i32 %y.0, %add1`

Example 4: Branches



Verifying path: 0 → 1 → 2 → 3 → 5 → 7 → 8 → 9 → *svf_assert* (if.else branch)

```
-------------SVFVar and Value-------------
ObjVar5 (0x7f000005)     Value: NULL
ValVar4                  Value: 0x7f000005
ValVar9                  Value: 10
ValVar12                 Value: 1
ValVar7                  Value: 11
ValVar8                  Value: 1
ValVar16                 Value: 10
ValVar17                 Value: 11
ValVar18                 Value: 0
                    ...
------------------------------------------
```

The assertion fails!!

START: 0 → 1 → 2 → 3 → 5 → 7 → 8 → 9 → *svf_assert*

Example 5: Interprocedural

```
1  int foo(int p) {
2      return p;
3  }
4  int main(int argc) {
5    int x;
6    int y;
7    x = foo(3); // ctx_7
8    y = foo(argc); // ctx_8
9    assert(y == argc);
10 }
```

`ctx_7`: context calling `foo` at ℓ_7

`ctx_8`: context calling `foo` at ℓ_8

Example 5: Interprocedural

```c
1  int foo(int p) {
2      return p;
3  }
4  int main(int argc) {
5    int x;
6    int y;
7    x = foo(3); // ctx_7
8    y = foo(argc); // ctx_8
9    assert(y == argc);
10 }
```

`ctx_7`: context calling `foo` at ℓ_7

`ctx_8`: context calling `foo` at ℓ_8

One Concrete Execution
(Concrete states)

One execution:

`argc` : 0

push calling context (calling foo at ℓ_7)

`p` : 3

calling context pop (returning from foo at ℓ_2)

`x` : 3

push calling context (calling foo at ℓ_8)

`p` : 0

pop calling context (returning from foo ℓ_2)

`y` : 0

Example 5: Interprocedural

```
1  int foo(int p) {
2      return p;
3  }
4  int main(int argc) {
5    int x;
6    int y;
7    x = foo(3); // ctx_7
8    y = foo(argc); // ctx_8
9    assert(y == argc);
10 }
```

`ctx_7`: context calling `foo` at ℓ_7

`ctx_8`: context calling `foo` at ℓ_8

One Concrete Execution (Concrete states)

One execution:

`argc` : 0

push calling context (calling foo at ℓ_7)

`p` : 3

calling context pop (returning from foo at ℓ_2)

`x` : 3

push calling context (calling foo at ℓ_8)

`p` : 0

pop calling context (returning from foo ℓ_2)

`y` : 0

Symbolic Execution (Symbolic states)

One execution:

`argc` : `getZ3Expr(argc)`

push calling context (calling foo at ℓ_7)

`ctx_7_p` : 3

pop calling context (returning from foo at ℓ_2)

`x` : `getZ3Expr(p, ctx_7)`

push calling context (calling foo at ℓ_8)

`ctx_8_p` : `getZ3Expr(argc)`

pop calling context (returning from foo ℓ_2)

`y` : `getZ3Expr(p, ctx_8)`

Example 5: Interprocedural

```
1  int foo(int p) {
2      return p;
3  }
4  int main(int argc) {
5    int x;
6    int y;
7    x = foo(3); // ctx_7
8    y = foo(argc); // ctx_8
9    assert(y == argc);
10 }
```

ctx_7: context calling foo at ℓ_7

ctx_8: context calling foo at ℓ_8

One Concrete Execution (Concrete states)

One execution:

argc : 0
push calling context (calling foo at ℓ_7)
p : 3
calling context pop (returning from foo at ℓ_2)
x : 3
push calling context (calling foo at ℓ_8)
p : 0
pop calling context (returning from foo ℓ_2)
y : 0

Symbolic Execution (Symbolic states)

One execution:

argc : getZ3Expr(argc)
push calling context (calling foo at ℓ_7)
ctx_7_p : 3
pop calling context (returning from foo at ℓ_2)
x : getZ3Expr(p, ctx_7)
push calling context (calling foo at ℓ_8)
ctx_8_p : getZ3Expr(argc)
pop calling context (returning from foo ℓ_2)
y : getZ3Expr(p, ctx_8)

Checking non-existence of counterexamples:

$\psi(N_1) \wedge \psi(N_2) \wedge \ldots \psi(N_i) \wedge \neg\psi(Q)$	Satisfiability	Counterexample
$\texttt{ctx\_7\_p} \equiv 3 \wedge \texttt{x} \equiv \texttt{ctx\_7\_p} \wedge \texttt{ctx\_8\_p} \equiv \texttt{argc} \wedge \texttt{y} \equiv \texttt{ctx\_7\_p} \wedge \texttt{y} \neq \texttt{argc}$	unsat	$\varnothing$

foo's argument p needs to be **differentiated and renamed** as ctx_7_p and ctx_8_p due to two calling contexts, ctx_7 and ctx_8 to mimic the runtime call stack which holds the local variable p.

What's next?

- (1) Understand SSE algorithms and examples in the slides
- (2) Finish the Quiz-2 and Lab-2 on WebCMS
- (3) Start implementing the automated translation from code to Z3 formulas using `SSE` and `Z3SSEMgr` in Assignment 2