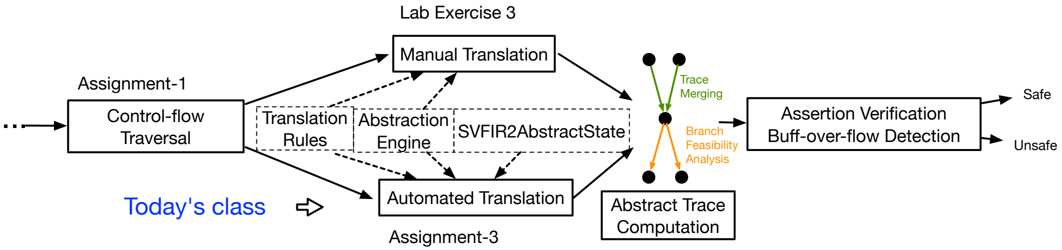
Abstract Interpretation and its Applications

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Today’s class



Abstract Execution on Pointer-Free SVFIR

- For simplicity, let's first consider abstract execution on a pointer-free language.
- This means there are no operations for memory allocation (like $\mathtt{p} = \mathtt{alloc}_o$) or for indirect memory accesses (such as `p = *q` or `*p = q`).
- Here are the pointer-free SVFSTMTs and their C-like forms:

SVFSTMT	**C-Like form**
CONSSTMT	$\ell : \mathtt{p} = \mathtt{c}$
COPYSTMT	$\ell : \mathtt{p} = \mathtt{q}$
BINARYSTMT	$\ell : \mathtt{r} = \mathtt{p} \otimes \mathtt{q}$
PHISTMT	$\ell : \mathtt{r} = \mathtt{phi}(\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_n)$
SEQUENCE	$\ell_1 ; \ell_2$
BRANCHSTMT	ℓ_1 : if$(x < c)$ then ℓ_2 else ℓ_3

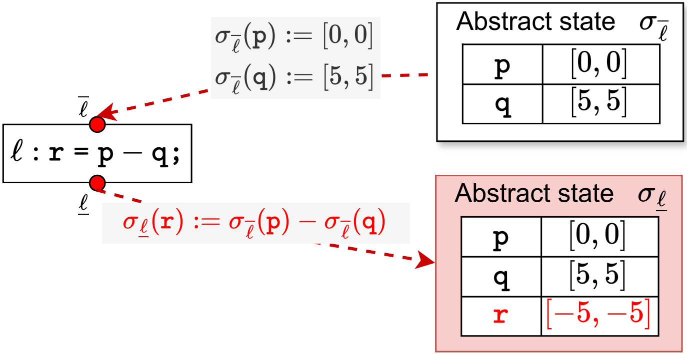
Abstract Execution Rules on Pointer-Free SVFIR

Let's use the *Interval* abstract domain to update σ based on the following rules for different SVFSTMT:

SVFSTMT	C-Like form	Abstract Execution Rule
CONSSTMT	$\ell : \mathtt{p} = \mathtt{c}$	$\sigma_{\underline{\ell}}(\mathtt{p}) := [\mathtt{c}, \mathtt{c}]$
COPYSTMT	$\ell : \mathtt{p} = \mathtt{q}$	$\sigma_{\underline{\ell}}(\mathtt{p}) := \sigma_{\overline{\ell}}(\mathtt{q})$
BINARYSTMT	$\ell : \mathtt{r} = \mathtt{p} \otimes \mathtt{q}$	$\sigma_{\underline{\ell}}(r) := \sigma_{\overline{\ell}}(p) \hat{\otimes} \sigma_{\overline{\ell}}(q)$
PHISTMT	$\ell : \mathtt{r} = \mathtt{phi}(\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_n)$	$\sigma_{\underline{\ell}}(r) := \bigsqcup_{i=1}^{n} \sigma_{\overline{\ell}}(p_i)$
SEQUENCE	$\ell_1; \ell_2$	$\forall v \in \mathcal{V}, \sigma_{\overline{\ell_2}}(v) \sqsupseteq \sigma_{\underline{\ell_1}}(v)$
BRANCHSTMT	ℓ_1 : if$(x < c)$ then ℓ_2 else ℓ_3	$\sigma_{\overline{\ell_2}}(x) := \sigma_{\underline{\ell_1}}(x) \sqcap [-\infty, c-1]$, if $\sigma_{\underline{\ell_1}}(x) \sqcap [-\infty, c-1] \neq \perp$ $\sigma_{\overline{\ell_3}}(x) := \sigma_{\underline{\ell_1}}(x) \sqcap [c, +\infty]$, if $\sigma_{\underline{\ell_1}}(x) \sqcap [c, +\infty] \neq \perp$

An Example: Abstract Execution on BINARYSTMT

SVFSTMT	C-Like form	Abstract Execution Rule
BINARYSTMT	$\ell : \mathtt{r} = \mathtt{p} \otimes \mathtt{q}$	$\sigma_{\underline{\ell}}(r) := \sigma_{\overline{\ell}}(p)\hat{\otimes}\sigma_{\overline{\ell}}(q)$



Abstract Execution on SVFIR in the Presence of Pointers

- SVFIR in the presence of pointers contain pointer-related statements including AddrStmt, GepStmt, LoadStmt and StoreStmt.

SVFStmt	**C-Like form**
ConsStmt	$\ell : \mathtt{p} = \mathtt{c}$
CopyStmt	$\ell : \mathtt{p} = \mathtt{q}$
BinaryStmt	$\ell : \mathtt{r} = \mathtt{p} \otimes \mathtt{q}$
PhiStmt	$\ell : \mathtt{r} = \mathtt{phi}(\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_n)$
Sequence	$\ell_1 ; \ell_2$
BranchStmt	ℓ_1 : if$(x < c)$ then ℓ_2 else ℓ_3
AddrStmt	$\ell : \mathtt{p} = \mathtt{alloc}$
GepStmt	$\ell : \mathtt{p} = \&(\mathtt{q} \rightarrow \mathtt{i})$ or $\mathtt{p} = \&\mathtt{q}[\mathtt{i}]$
LoadStmt	$\ell : \mathtt{p} = *\mathtt{q}$
StoreStmt	$\ell : *\mathtt{p} = \mathtt{q}$

Abstract Execution on SVFIR in the Presence of Pointers

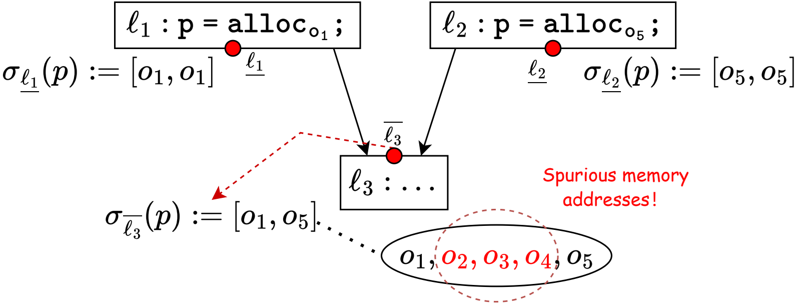
An Example

Let's try analyzing this kind of SVFIR using the same way as we did for pointer-free SVFIR based on a single interval domain.

Abstract Execution on SVFIR in the Presence of Pointers

An Example

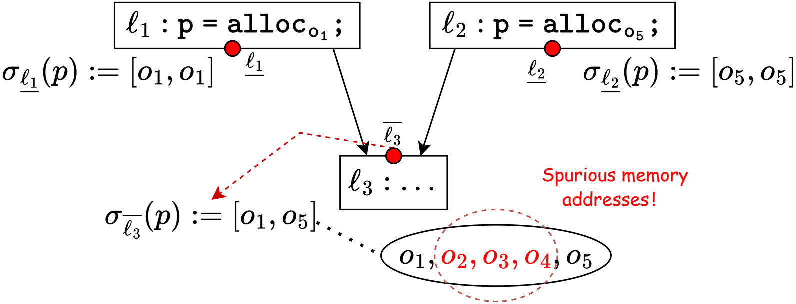
Let's try analyzing this kind of SVFIR using the same way as we did for pointer-free SVFIR based on a single interval domain.



Abstract Execution on SVFIR in the Presence of Pointers

An Example

Let's try analyzing this kind of SVFIR using the same way as we did for pointer-free SVFIR based on a single interval domain.



✗ Using intervals to represent discrete memory address value is imprecise.

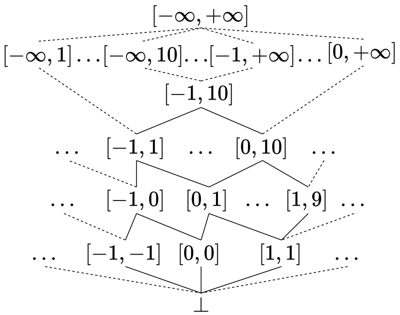
Abstract Execution on SVFIR in the Presence of Pointers

✔ We require **a combination of memory address and interval domains** to precisely and efficiently perform abstract execution on SVFIR in the presence of pointers.

Abstract Execution over Memory Address and Interval Domains

Interval and Memory Address Domains

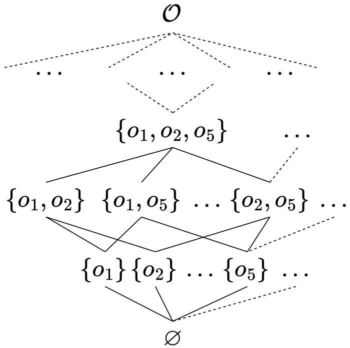
Interval abstraction (*Interval* domain) for scalar variables.



Abstract Execution over Memory Address and Interval Domains

Interval and Memory Address Domains

Discrete values (*MemAddress* domain) for memory addresses.



Abstract Trace for Memory Address and Interval Domains

- The abstract trace for memory address and interval domains is defined as:

	Notation	Domain	Implementation
Abstract trace	σ	$\mathbb{L} \times \mathcal{V} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$	*preAbstractTrace*, *postAbstractTrace*
Abstract state	σ_L	$\mathcal{V} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$	*AbstractState.varToAbsVal*
Abstract value	$\sigma_L(p)$	$\mathit{Interval} \times \mathit{MemAddress}$	*AbstractValue*

Abstract Trace for Memory Address and Interval Domains

- The abstract trace for memory address and interval domains is defined as:

	Notation	**Domain**	**Implementation**
Abstract trace	σ	$\mathbb{L} \times \mathcal{V} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$	*preAbstractTrace*, *postAbstractTrace*
Abstract state	σ_L	$\mathcal{V} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$	*AbstractState.varToAbsVal*
Abstract value	$\sigma_L(p)$	$\mathit{Interval} \times \mathit{MemAddress}$	*AbstractValue*

- *Interval* is used for tracking the interval value of **scalar variables.**

Abstract Trace for Memory Address and Interval Domains

- The abstract trace for memory address and interval domains is defined as:

	Notation	Domain	Implementation
Abstract trace	σ	$\mathbb{L} \times \mathcal{V} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$	*preAbstractTrace*, *postAbstractTrace*
Abstract state	σ_L	$\mathcal{V} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$	*AbstractState.varToAbsVal*
Abstract value	$\sigma_L(p)$	$\mathit{Interval} \times \mathit{MemAddress}$	*AbstractValue*

- *Interval* is used for tracking the interval value of **scalar variables.**
- *MemAddress* is used for tracking the memory addresses of **memory address variables.**

Abstract Trace for Memory Address and Interval Domains

Cross-Domain Interaction

- During abstract execution, the memory address domain and the interval domain interact with each other.

Abstract Trace for Memory Address and Interval Domains

Cross-Domain Interaction

- During abstract execution, the memory address domain and the interval domain interact with each other.
- To track the value to value correlation at each program point, we define: $\delta \in \mathbb{L} \times \mathit{MemAddress} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$.

Abstract Trace for Memory Address and Interval Domains

Cross-Domain Interaction

- During abstract execution, the memory address domain and the interval domain interact with each other.
- To track the value to value correlation at each program point, we define: $\delta \in \mathbb{L} \times \textit{MemAddress} \rightarrow \textit{Interval} \times \textit{MemAddress}$.
- For top-level variables, we still use $\sigma \in \mathbb{L} \times \mathcal{P} \rightarrow \textit{Interval} \times \textit{MemAddress}$ to track the memory addresses or interval values of these variables.

Abstract Trace for Memory Address and Interval Domains

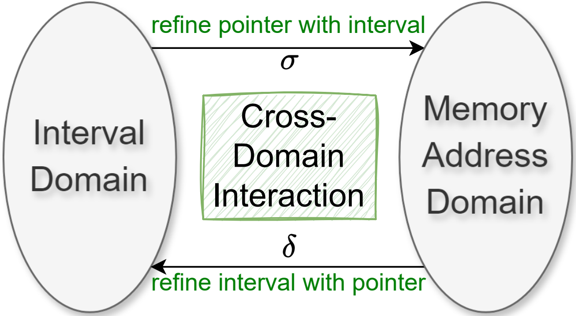
Cross-Domain Interaction

- During abstract execution, the memory address domain and the interval domain interact with each other.
- To track the value to value correlation at each program point, we define: $\delta \in \mathbb{L} \times \mathit{MemAddress} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$.
- For top-level variables, we still use $\sigma \in \mathbb{L} \times \mathcal{P} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$ to track the memory addresses or interval values of these variables.

	Notation	**Domain**	**Implementation**
Abstract trace	σ	$\mathbb{L} \times \mathcal{P} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$	*preAbstractTrace*, *postAbstractTrace*
	δ	$\mathbb{L} \times \mathit{MemAddress} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$	
Abstract state	σ_L	$\mathcal{P} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$	*AbstractState.varToAbsVal*
	δ_L	$\mathit{MemAddress} \rightarrow \mathit{Interval} \times \mathit{MemAddress}$	*AbstractState.locToAbsVal*
Abstract value	$\sigma_L(p)$	$\mathit{Interval} \times \mathit{MemAddress}$	*AbstractValue*
	$\delta_L(o)$		

Abstract Trace for Memory Address and Interval Domains

Cross-Domain Interaction



Abstract Execution Rules on SVFIR in the Presence of Pointers

Now let's use the *Interval* $\times$ *MemAddress* abstract domain to update σ and δ based on the following rules for different SVFSTMT:

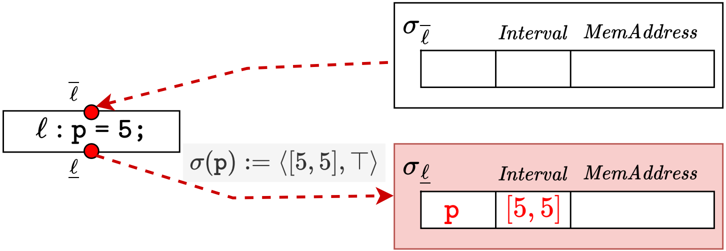
SVFSTMT	**C-Like form**	**Abstract Execution Rule**
CONSSTMT	$\ell : \mathtt{p} = \mathtt{c}$	$\sigma_{\underline{\ell}}(\mathtt{p}) := \langle[\mathtt{c}, \mathtt{c}], \top\rangle$
COPYSTMT	$\ell : \mathtt{p} = \mathtt{q}$	$\sigma_{\underline{\ell}}(\mathtt{p}) := \sigma_{\overline{\ell}}(\mathtt{q})$
BINARYSTMT	$\ell : \mathtt{r} = \mathtt{p} \otimes \mathtt{q}$	$\sigma_{\underline{\ell}}(r) := \sigma_{\overline{\ell}}(p) \hat{\otimes} \sigma_{\overline{\ell}}(q)$
PHISTMT	$\ell : \mathtt{r} = \mathtt{phi}(\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_n)$	$\sigma_{\underline{\ell}}(r) := \bigsqcup_{i=1}^{n} \sigma_{\overline{\ell}}(p_i)$
BRANCHSTMT	ℓ_1 : if$(x < c)$ then ℓ_2 else ℓ_3	$\sigma_{\overline{\ell_2}}(x) := \sigma_{\underline{\ell_1}}(x) \sqcap [-\infty, c-1]$, if $\sigma_{\underline{\ell_1}}(x) \sqcap [-\infty, c-1] \neq \bot$ $\sigma_{\overline{\ell_3}}(x) := \sigma_{\underline{\ell_1}}(x) \sqcap [c, +\infty]$, if $\sigma_{\underline{\ell_1}}(x) \sqcap [c, +\infty] \neq \bot$
SEQUENCE	$\ell_1; \ell_2$	$\delta_{\overline{\ell_2}} \sqsupseteq \delta_{\underline{\ell_1}}, \sigma_{\overline{\ell_2}} \sqsupseteq \sigma_{\underline{\ell_1}}$
ADDRSTMT	$\ell : \mathtt{p} = \mathtt{alloc}_{\mathtt{o_i}}$	$\sigma_{\underline{\ell}}(\mathtt{p}) := \langle \top, \{o_i\}\rangle$
GEPSTMT	$\ell : \mathtt{p} = \&(\mathtt{q} \rightarrow \mathtt{i})$ or $\mathtt{p} = \&\mathtt{q}[\mathtt{i}]$	$\sigma_{\underline{\ell}}(\mathtt{p}) := \bigsqcup_{\mathtt{o} \in \gamma(\sigma_{\overline{\ell}}(\mathtt{q}))} \bigsqcup_{j \in \gamma(\sigma_{\overline{\ell}}(\mathtt{i}))} \langle \top, \{\mathtt{o.fld}_j\}\rangle$
LOADSTMT	$\ell : \mathtt{p} = *\mathtt{q}$	$\sigma_{\underline{\ell}}(\mathtt{p}) := \bigsqcup_{o \in \{o \mid o \in \sigma_{\overline{\ell}}(q)\}} \delta_{\overline{\ell}}(o)$
STORESTMT	$\ell : *\mathtt{p} = \mathtt{q}$	$\delta_{\underline{\ell}} := (\{o \mapsto \sigma_{\overline{\ell}}(\mathtt{q}) \mid o \in \gamma(\sigma_{\overline{\ell}}(\mathtt{p}))\} \sqcup \delta_{\underline{\ell}})$

Implementation of Abstract State and Abstract Trace

- For a program point L, the abstract state AS is an instance of the class named *AbstractState*, consisting of:
  - $varToAbsVal : \sigma_L \in \mathcal{P} \rightarrow Interval \times MemAddress$
  - $locToAbsVal : \delta_L \in MemAddress \rightarrow Interval \times MemAddress$
- The abstract trace is divided into two maps, *preAbstractTrace* and *postAbstractTrace*, which record the abstract states before and after each control flow point respectively.
  - For example, for a control flow node ℓ, $preAbstractTrace(\ell)$ includes $\sigma_{\overline{\ell}}$ and $\delta_{\overline{\ell}}$, and $postAbstractTrace(\ell)$ represents $\sigma_{\underline{\ell}}$ and $\delta_{\underline{\ell}}$.

An Example: Abstract Execution on ConsStmt

SVFStmt	C-Like form	Abstract Execution Rule
ConsStmt	$\ell : \mathrm{p} = \mathrm{c}$	$\sigma_{\underline{\ell}}(\mathrm{p}) := \langle[\mathrm{c}, \mathrm{c}], \top\rangle$

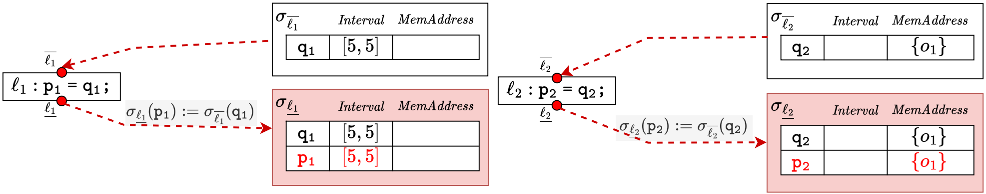


Algorithm 1: Abstract Execution Rule for ConsStmt

Function *updateStateOnAddr(addr)*:
- *node* $=$ *addr* $\rightarrow$ *getICFGNode*()
- *as* $=$ *getAbsState*(*node*)
- *initSVFVar*(*as*, *addr* $\rightarrow$ *getRHSVarID*())
- *as*[*addr* $\rightarrow$ *getLHSVarID*()] $=$ *as*[*addr* $\rightarrow$ *getRHSVarID*()]

An Example: Abstract Execution on CopyStmt

SVFStmt	C-Like form	Abstract Execution Rule
CopyStmt	$\ell : p = q$	$\sigma_{\underline{\ell}}(p) := \sigma_{\overline{\ell}}(q)$

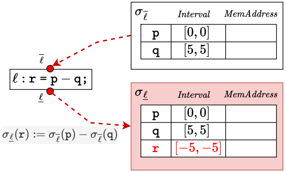


Algorithm 2: Abstract Execution Rule for CopyStmt

Function *updateStateOnCopy(copy)*:
    node = *copy* → *getICFGNode*()
    as = *getAbsState*(*node*)
    lhs = *copy* → *getLHSVarID*()
    rhs = *copy* → *getRHSVarID*()
    if *svfir* → *isBlkPtr*(*lhs*) **then**
        as[*lhs*] = *IntervalValue*::*top*()
    else
        if *as.inVarToValTable*(*rhs*)||*as.inVarToAddrsTable*(*rhs*) **then**
            as[*lhs*] = *as*[*rhs*]

An Example: Abstract Execution on BINARYSTMT

SVFSTMT	C-Like form	Abstract Execution Rule
BINARYSTMT	$\ell : \texttt{r} = \texttt{p} \otimes \texttt{q}$	$\sigma_{\underline{\ell}}(r) := \sigma_{\overline{\ell}}(p) \hat{\otimes} \sigma_{\overline{\ell}}(q)$



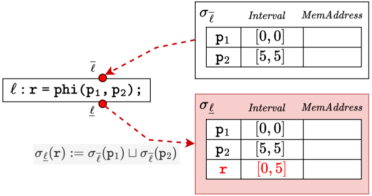
Algorithm 3: Abstract Execution Rule for BINARYSTMT

Function *updateStateOnBinary(binary)*:

- *node* = *binary* → *getICFGNode*()
- *as* = *getAbsState*(*node*)
- *op0* = *binary* → *getOpVarID*(*0*)
- *op1* = *binary* → *getOpVarID*(*1*)
- *res* = *binary* → *getResID*()
- **if** !*as.inVarToValTable*(*op0*) **then**
  - *as*[*op0*] = *IntervalValue* :: *top*()
- **if** !*as.inVarToValTable*(*op1*) **then**
  - *as*[*op1*] = *IntervalValue* :: *top*()
- **if** *as.inVarToValTable*(*op0*)&&*as.inVarToValTable*(*op1*) **then**
  - *as*[*res*] = *as*[*op0*] $\hat{\otimes}$ *as*[*op1*]

An Example: Abstract Execution on PHISTMT

SVFSTMT	C-Like form	Abstract Execution Rule
PHISTMT	$\ell : \mathtt{r} = \mathtt{phi}(\mathtt{p_1}, \mathtt{p_2}, \ldots, \mathtt{p_n})$	$\sigma_{\underline{\ell}}(r) := \bigsqcup_{i=1}^{n} \sigma_{\overline{\ell}}(p_i)$

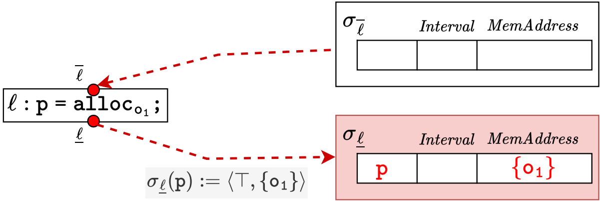


Algorithm 4: Abstract Execution Rule for PHISTMT

```
Function updateStateOnPhi(phi):
    node = phi→getICFGNode()
    as = getAbsState(node)
    res = phi→getResID()
    rhs = AbstractValue :: UnknownType
    for i = 0; i < phi→getOpVarNum(); i++ do
        curId = phi→getOpVarID(i)
        if as.inVarToValTable(curId)||as.inVarToAddrsTable(curId)) then
            rhs.join_with(as[curId])
    if !rhs.isUnknown() then
        as[res] = rhs
```

An Example: Abstract Execution on ADDRSTMT

SVFSTMT	C-Like form	Abstract Execution Rule
ADDRSTMT	$\ell : \texttt{p} = \texttt{alloc}_{\texttt{o}_i}$	$\sigma_{\underline{\ell}}(\texttt{p}) := \langle \top, \{o_i\} \rangle$

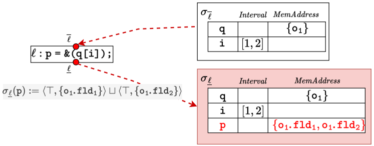


Algorithm 5: Abstract Execution Rule for ADDRSTMT

```
Function updateStateOnAddr(addr):
    node = addr → getICFGNode()
    as = getAbsState(node)
    initSVFVar(as, addr → getRHSVarID())
    as[addr → getLHSVarID()] = as[addr → getRHSVarID()]
```

An Example: Abstract Execution on GEPSTMT

SVFSTMT	C-Like form	Abstract Execution Rule
GEPSTMT	$\ell: \texttt{p} = \&(\texttt{q} \rightarrow \texttt{i})$ or $\texttt{p} = \&\texttt{q}[\texttt{i}]$	$\sigma_{\underline{\ell}}(\texttt{p}) := \bigsqcup_{o \in \gamma(\sigma_{\overline{\ell}}(\texttt{q}))} \bigsqcup_{j \in \gamma(\sigma_{\overline{\ell}}(\texttt{i}))} \langle \top, \{o.\texttt{fld}_j\} \rangle$



Algorithm 6: Abstract Execution Rule for GEPSTMT

```
Function updateStateOnGep(gep):
    node = gep→getICFGNode()
    as = getAbsState(node)
    rhs = gep→getRHSVarID()
    lhs = gep→getLHSVarID()
    if !as.inVarToAddrsTable(rhs) then
        return
    rhsVal = as[rhs]
    offsetPair = getElementIndex(as, gep)
    if !AbstractState :: isVirtualMemAddress(*rhsVal.getAddrs().begin()) then
        return
    else
        gepAddrs = AbstractValue :: UnknownType
        lb = offsetPair.lb().getIntNumeral() < Options :: MaxFieldLimit()?offsetPair.lb().getIntNumeral() : Options :: MaxFieldLimit()
        ub = offsetPair.ub().getIntNumeral() < Options :: MaxFieldLimit()?offsetPair.ub().getIntNumeral() : Options :: MaxFieldLimit()
        for i = lb; i ≤ ub; i + + do
            gepAddrs.join_with(getGepObjAddress(as, rhs, i))
        if !rhsVal.isUnknown() then
            as[lhs] = gepAddrs
```

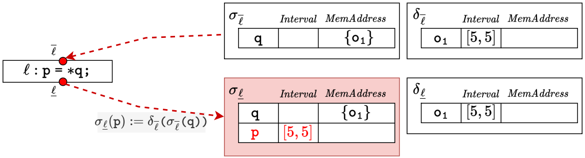
An Example: Buffer Overflow Detection on GEPSTMT

Algorithm 7: Abstract Execution Rule for GEPSTMT

```
as = getAbsState(gep→getICFGNode())
byteOffset = getByteOffset(as, gep)
gepRhsVal := as[gep→getRHSVarID()]
for addr ∈ gepRhsVal.getAddrs() do
    baseObj = AbstractState :: getInternalID(addr);
    valid_sz = obj2size[baseObj];
    new_valid_sz = valid_sz − byteOffset;// interval minus
    if new_valid_sz.lb().getIntNumeral() ≤ 0 then
        Assign3Exception bug(gep→getICFGNode()→toString());
        addBugToReporter(bug, gep → getICFGNode());
    offsetPair = getElementIndex(as, gep)
    lb = offsetPair.lb().getIntNumeral() < Options :: MaxFieldLimit()?offsetPair.lb().getIntNumeral() : Options :: MaxFieldLimit()
    ub = offsetPair.ub().getIntNumeral() < Options :: MaxFieldLimit()?offsetPair.ub().getIntNumeral() : Options :: MaxFieldLimit()
    for i = lb; i ≤ ub; i + + do
        gepObj = svfir→getGepObjVar(baseObj, i)
        obj2size[gepObj] = valid_sz − byteOffset
```

An Example: Abstract Execution on LOADSTMT

SVFSTMT	C-Like form	Abstract Execution Rule
LOADSTMT	$\ell : p = *q$	$\sigma_{\underline{\ell}}(p) := \bigsqcup_{o \in \{o \mid o \in \sigma_{\overline{\ell}}(q)\}} \delta_{\overline{\ell}}(o)$

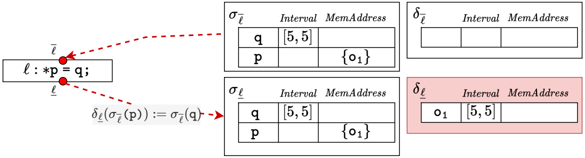


Algorithm 8: Abstract Execution Rule for LOADSTMT

```
Function updateStateOnLoad(load):
    node = load → getICFGNode()
    as = getAbsState(node)
    rhs = load → getRHSVarID()
    lhs = load → getLHSVarID()
    if as.inVarToAddrsTable(rhs) then
        addrs = as.getAddrs(rhs)
        rhsVal = AbstractValue :: UnknownType
        for addr : addrs.getAddrs() do
            objId = AbstractState :: getInternalID(addr)
            if as.inLocToValTable(objId)||as.inLocToAddrsTable(objId) then
                rhsVal.join_with(as.load(addr))
        if !rhsVal.isUnknown() then
            as[lhs] = rhsVal
```

An Example: Abstract Execution on STORESTMT

SVFSTMT	C-Like form	Abstract Execution Rule
STORESTMT	$\ell : *\mathrm{p} = \mathrm{q}$	$\delta_{\underline{\ell}} := (\{o \mapsto \sigma_{\overline{\ell}}(\mathrm{q}) \mid o \in \gamma(\sigma_{\overline{\ell}}(\mathrm{p}))\} \sqcup \delta_{\underline{\ell}})$



Algorithm 9: Abstract Execution Rule for STORESTMT

```
Function updateStateOnStore(store):
    node = store→getICFGNode()
    as = getAbsState(node)
    rhs = store→getRHSVarID()
    lhs = store→getLHSVarID()
    if as.inVarToAddrsTable(lhs) then
        addrs = as[lhs]
        if as.inVarToValTable(rhs)||as.inVarToAddrsTable(rhs) then
            for addr : addrs.getAddrs() do
                val = as.load(addr)
                if val.isInterval() then
                    if val.isTop() then
                        as.store(addr, as[rhs])
                    else
                        val.join_with(as[rhs])
                        as.store(addr, val)
                else
                    val.join_with(as[rhs]) as.store(addr, val)
```

Abstract Execution Pseudo Code

Algorithm 10: Handle cycle

```
Function handleCycle(cycle):
    h := head(cycle)
    INC := true
    iter := 0
    while true do
        iter++
        handleICFGNode(h)
        if iter < widen_delay then
            tmpAS := postAbsTrace[h]
        else
            if INC then
                postAbsTrace[h] := tmpAS.widen(postAbsTrace[h])
                if postAbsTrace[h] ≤ tmpAS then
                    INC := false
                    tmpAS := postAbsTrace[h]
                    continue
                tmpAS := postAbsTrace[h]
            else
                postAbsTrace[h] := tmpAS.narrow(postAbsTrace[h])
                if postAbsTrace[h] ≥ tmpAS then
                    break
                tmpAS := postAbsTrace[h]
        handleCycleBody(cycle)
```

Algorithm 11: Handle cycle body

```
Function handleCycleBody(cycle):
    for it = cycle→begin(); it! = cycle→end(); ++it do
        cur = *it
        if vertex = SVFUtil :: dyn_cast⟨ICFGWTONode⟩(cur) then
            handleWTONode(vertex→node())
        else if cycle2 = SVFUtil :: dyn_cast⟨ICFGWTOCycle⟩(cur)
         then
            handleCycle(cycle2)
        else
            assert(false&&"unknownWTOtype!")
Function bufOverflowDetection(stmt):
    if !SVFUtil :: isa⟨CallICFGNode⟩(stmt→getICFGNode()) then
        if addr = SVFUtil :: dyn_cast⟨AddrStmt⟩(stmt) then
            as = getAbsState(addr→getICFGNode())
            alloc_sz = getAllocaInstByteSize(as, addr)
            memid = SVFUtil :: dyn_cast⟨ObjVar⟩(
                svfir→getGNode(addr→getRHSVarID()))→getId()
            obj2size[memid] = IntervalValue(alloc_sz, alloc_sz)
        else if gep = SVFUtil :: dyn_cast⟨GepStmt⟩(stmt) then
            // Buffer Overflow Detection on GEPSTMT
```

A Running Example: Abstract Execution

```c
extern void assert(int);

int main(){
    int a = 0;
    while(a < 10) {
        a++;
    }
    assert(a == 10);
    return 0;
}
```

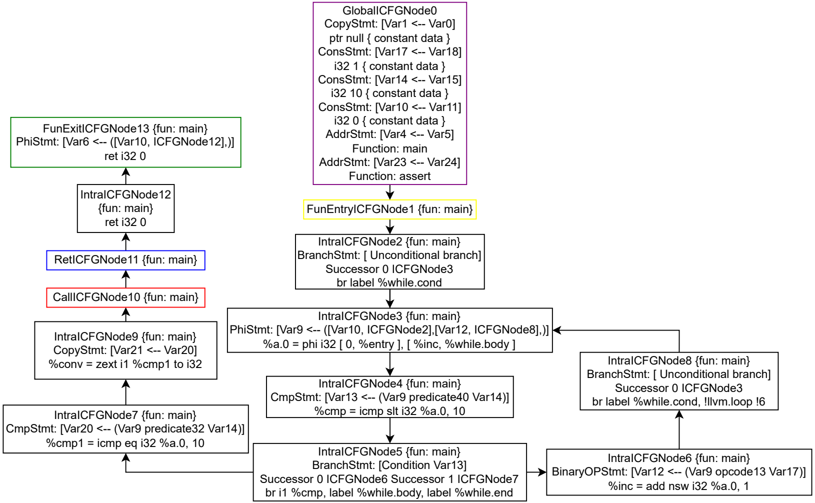
Source Code

Compile to LLVM IR ⇨

```llvm
define dso_local i32 @main() {
entry:
  br label %while.cond
while.cond:
  %a.0 = phi i32 [ 0, %entry ], [ %inc, %while.body ]
  %cmp = icmp slt i32 %a.0, 10
  br i1 %cmp, label %while.body, label %while.end
while.body:
  %inc = add nsw i32 %a.0, 1
  br label %while.cond,
while.end:
  %cmp1 = icmp eq i32 %a.0, 10
  %conv = zext i1 %cmp1 to i32
  call void @assert(i32 noundef %conv)
  ret i32 0
}
```

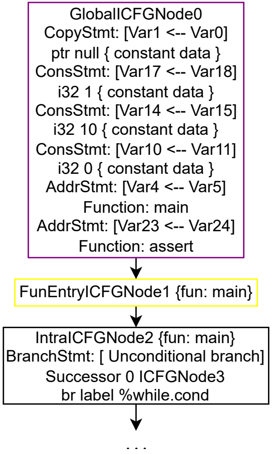
LLVM IR

A Running Example: Abstract Execution



ICFG

A Running Example: Abstract Execution



ICFG

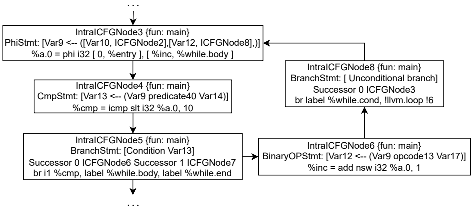
Algorithm 12: Abstract execution guided by WTO

Function `handleStatement(ℓ)`:
- *tmpAS* := *preAbsTrace*[ℓ]
- **if** ℓ *is* ConsStmt *or* AddrStmt **then**
  - `initSVFVar(ℓ.rhs)`
  - *tmpAS*[ℓ.*lhs*] := *tmpAS*[ℓ.*rhs*]
- **else if** ℓ *is* CopyStmt **then**
  - *tmpAS*[ℓ.*lhs*] := *tmpAS*[ℓ.*rhs*]
- . . .

$postAbsTrace[\ell_0].varToAbsVal$:

Variable	*Interval*	*MemAddress*
Var0	$\top$	{0x7f00}
Var1	$\top$	{0x7f00}
Var18	[1,1]	$\top$
Var17	[1,1]	$\top$
Var14	[10,10]	$\top$
Var15	[10,10]	$\top$
Var10	[0,0]	$\top$
Var11	[0,0]	$\top$
. . .		

A Running Example: Abstract Execution



ICFG

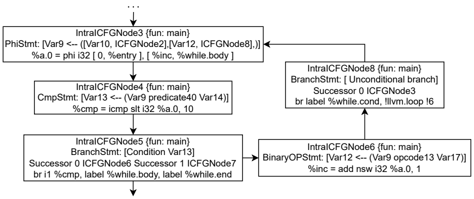
$postAbsTrace[\ell_3].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	. . .	
Var10	[0,0]	$\top$
Var9	[0,0]	$\top$
	. . .	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycle(cycle):
    h := head(cycle)
    INC := true
    iter := 0
    while true do
        iter++
        handleICFGNode(h)
        if iter < widen_delay then
            tmpAS := postAbsTrace[h]// iter ≡ 1
        else
            if INC then
                postAbsTrace[h] := tmpAS.widen(postAbsTrace[h])
                if postAbsTrace[h] ≤ tmpAS then
                    INC := false
                    tmpAS := postAbsTrace[h]
                    continue
                tmpAS := postAbsTrace[h]
            else
                postAbsTrace[h] := tmpAS.narrow(postAbsTrace[h])
                if postAbsTrace[h] ≥ tmpAS then
                    break
                tmpAS := postAbsTrace[h]
        handleCycleBody(cycle)
```

A Running Example: Abstract Execution



ICFG

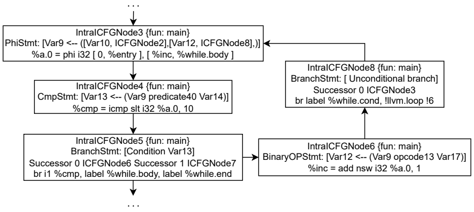
$postAbsTrace[\ell_8].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	...	
Var10	[0,0]	$\top$
Var9	[0,0]	$\top$
Var12	[1,1]	$\top$
	...	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycle(cycle):
    h := head(cycle)
    INC := true
    iter := 0
    while true do
        iter++
        handleICFGNode(h)
        if iter < widen_delay then
            tmpAS := postAbsTrace[h]
        else
            if INC then
                postAbsTrace[h] := tmpAS.widen(postAbsTrace[h])
                if postAbsTrace[h] ≤ tmpAS then
                    INC := false
                    tmpAS := postAbsTrace[h]
                    continue
                tmpAS := postAbsTrace[h]
            else
                postAbsTrace[h] := tmpAS.narrow(postAbsTrace[h])
                if postAbsTrace[h] ≥ tmpAS then
                    break
                tmpAS := postAbsTrace[h]
        handleCycleBody(cycle)// iter ≡ 1
```

A Running Example: Abstract Execution



ICFG

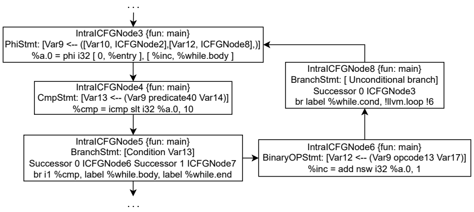
$postAbsTrace[\ell_3].varToAbsVal$:

Variable	Interval	MemAddress
	...	
Var9	[0,1]	$\top$
Var12	[1,1]	$\top$
	...	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycle(cycle):
    h := head(cycle)
    INC := true
    iter := 0
    while true do
        iter++
        handleICFGNode(h)
        if iter < widen_delay then
            tmpAS := postAbsTrace[h]// iter ≡ 2
        else
            if INC then
                postAbsTrace[h] := tmpAS.widen(postAbsTrace[h])
                if postAbsTrace[h] ≤ tmpAS then
                    INC := false
                    tmpAS := postAbsTrace[h]
                    continue
                tmpAS := postAbsTrace[h]
            else
                postAbsTrace[h] := tmpAS.narrow(postAbsTrace[h])
                if postAbsTrace[h] ≥ tmpAS then
                    break
                tmpAS := postAbsTrace[h]
        handleCycleBody(cycle)
```

A Running Example: Abstract Execution



ICFG

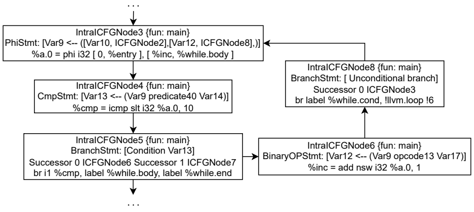
$postAbsTrace[\ell_8].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	...	
Var9	[0,1]	$\top$
Var12	[1,2]	$\top$
	...	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycle(cycle):
    h := head(cycle)
    INC := true
    iter := 0
    while true do
        iter++
        handleICFGNode(h)
        if iter < widen_delay then
            tmpAS := postAbsTrace[h]
        else
            if INC then
                postAbsTrace[h] := tmpAS.widen(postAbsTrace[h])
                if postAbsTrace[h] ≤ tmpAS then
                    INC := false
                    tmpAS := postAbsTrace[h]
                    continue
                tmpAS := postAbsTrace[h]
            else
                postAbsTrace[h] := tmpAS.narrow(postAbsTrace[h])
                if postAbsTrace[h] ≥ tmpAS then
                    break
                tmpAS := postAbsTrace[h]
        handleCycleBody(cycle)// iter ≡ 2
```

A Running Example: Abstract Execution



ICFG

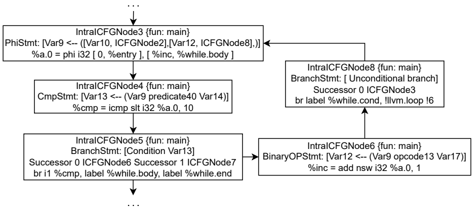
$postAbsTrace[\ell_3].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	. . .	
Var9	$[0, +\infty]$	$\top$
Var12	[1,2]	$\top$
	. . .	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycle(cycle):
    h := head(cycle)
    INC := true
    iter := 0
    while true do
        iter++
        handleICFGNode(h)
        if iter < widen_delay then
            tmpAS := postAbsTrace[h]
        else
            if INC then
                postAbsTrace[h] := tmpAS.widen(postAbsTrace[h]) // iter ≡ 3
                if postAbsTrace[h] ≤ tmpAS then
                    INC := false
                    tmpAS := postAbsTrace[h]
                    continue
                tmpAS := postAbsTrace[h]
            else
                postAbsTrace[h] := tmpAS.narrow(postAbsTrace[h])
                if postAbsTrace[h] ≥ tmpAS then
                    break
                tmpAS := postAbsTrace[h]
        handleCycleBody(cycle)
```

A Running Example: Abstract Execution



ICFG

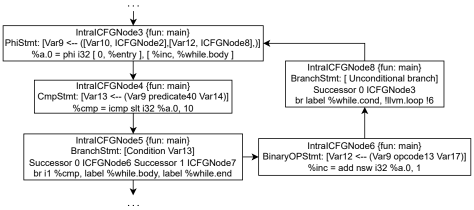
$postAbsTrace[\ell_8].varToAbsVal$:

Variable	*Interval*	*MemAddress*
...		
Var9	[0,9]	$\top$
Var12	[1,10]	$\top$
...		

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycle(cycle):
    h := head(cycle)
    INC := true
    iter := 0
    while true do
        iter++
        handleICFGNode(h)
        if iter < widen_delay then
            tmpAS := postAbsTrace[h]
        else
            if INC then
                postAbsTrace[h] := tmpAS.widen(postAbsTrace[h])
                if postAbsTrace[h] ≤ tmpAS then
                    INC := false
                    tmpAS := postAbsTrace[h]
                    continue
                tmpAS := postAbsTrace[h]
            else
                postAbsTrace[h] := tmpAS.narrow(postAbsTrace[h])
                if postAbsTrace[h] ≥ tmpAS then
                    break
                tmpAS := postAbsTrace[h]
        handleCycleBody(cycle)// iter ≡ 3
```

A Running Example: Abstract Execution



ICFG

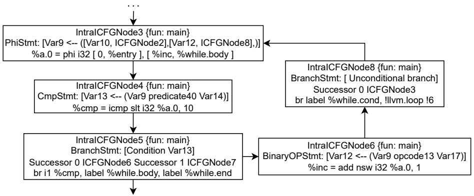
$postAbsTrace[\ell_3].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	...	
Var9	$[0, +\infty]$	$\top$
Var12	[1,10]	$\top$
	...	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycle(cycle):
    h := head(cycle)
    INC := true
    iter := 0
    while true do
        iter++
        handleICFGNode(h)
        if iter < widen_delay then
            tmpAS := postAbsTrace[h]
        else
            if INC then
                postAbsTrace[h] := tmpAS.widen(postAbsTrace[h]) // iter ≡ 4
                if postAbsTrace[h] ≤ tmpAS then
                    INC := false
                    tmpAS := postAbsTrace[h]
                    continue
                tmpAS := postAbsTrace[h]
            else
                postAbsTrace[h] := tmpAS.narrow(postAbsTrace[h])
                if postAbsTrace[h] ≥ tmpAS then
                    break
                tmpAS := postAbsTrace[h]
        handleCycleBody(cycle)
```

A Running Example: Abstract Execution



ICFG

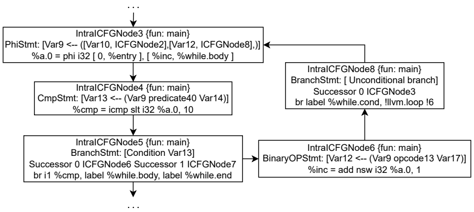
$postAbsTrace[\ell_3].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	...	
Var9	[0, 10]	$\top$
Var12	[1,10]	$\top$
	...	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycle(cycle):
    h := head(cycle)
    INC := true
    iter := 0
    while true do
        iter++
        handleICFGNode(h)
        if iter < widen_delay then
            tmpAS := postAbsTrace[h]
        else
            if INC then
                postAbsTrace[h] := tmpAS.widen(postAbsTrace[h])
                if postAbsTrace[h] ≤ tmpAS then
                    INC := false
                    tmpAS := postAbsTrace[h]
                    continue
                tmpAS := postAbsTrace[h]
            else
                postAbsTrace[h] := tmpAS.narrow(postAbsTrace[h]) // iter ≡ 5
                if postAbsTrace[h] ≥ tmpAS then
                    break
                tmpAS := postAbsTrace[h]
        handleCycleBody(cycle)
```

A Running Example: Abstract Execution



ICFG

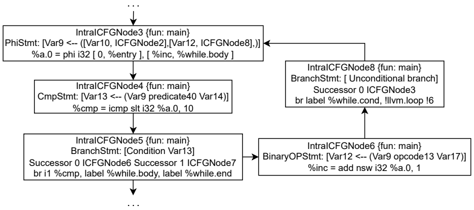
$postAbsTrace[\ell_8].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	...	
Var9	[0,9]	$\top$
Var12	[1,10]	$\top$
	...	

Algorithm 12: Abstract execution guided by WTO (part 2)

```
Function handleCycle(cycle):
    h := head(cycle)
    INC := true
    iter := 0
    while true do
        iter++
        handleICFGNode(h)
        if iter < widen_delay then
            tmpAS := postAbsTrace[h]
        else
            if INC then
                postAbsTrace[h] := tmpAS.widen(postAbsTrace[h])
                if postAbsTrace[h] ≤ tmpAS then
                    INC := false
                    tmpAS := postAbsTrace[h]
                    continue
                tmpAS := postAbsTrace[h]
            else
                postAbsTrace[h] := tmpAS.narrow(postAbsTrace[h])
                if postAbsTrace[h] ≥ tmpAS then
                    break
                tmpAS := postAbsTrace[h]
        handleCycleBody(cycle)// iter ≡ 5
```

A Running Example: Abstract Execution



ICFG

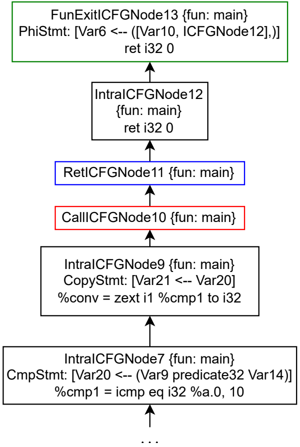
$postAbsTrace[\ell_3].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	. . .	
Var9	[0, 10]	$\top$
Var12	[1,10]	$\top$
	. . .	

Algorithm 12: Abstract execution guided by WTO (part 2)

Function `handleCycle`(*cycle*):
- h := head(cycle)
- INC := true
- iter := 0
- **while** *true* **do**
  - iter++
  - `handleICFGNode`(*h*)
  - **if** $iter < widen\_delay$ **then**
    - $tmpAS := postAbsTrace[h]$
  - **else**
    - **if** *INC* **then**
      - $postAbsTrace[h] := tmpAS.widen(postAbsTrace[h])$
      - **if** $postAbsTrace[h] \leq tmpAS$ **then**
        - INC := false
        - $tmpAS := postAbsTrace[h]$
        - continue
      - $tmpAS := postAbsTrace[h]$
    - **else**
      - $postAbsTrace[h] := tmpAS.narrow(postAbsTrace[h])$ // $iter \equiv 6$
      - **if** $postAbsTrace[h] \geq tmpAS$ **then**
        - break
      - $tmpAS := postAbsTrace[h]$
  - `handleCycleBody`(*cycle*)

A Running Example: Abstract Execution



ICFG

Algorithm 13: Abstract execution guided by WTO

Function `handleStatement(ℓ)`:
- $tmpAS := preAbsTrace[\ell]$
- **if** ℓ *is* BINARYSTMT **then**
  - $postAbsTrace[\ell][\ell.res] := preAbsTrace[\ell][\ell.op1] \hat{\otimes} preAbsTrace[\ell][\ell.op2]$
- ...

$postAbsTrace[\ell_7].varToAbsVal$:

Variable	*Interval*	*MemAddress*
	...	
Var9	[10,10]	$\top$
Var20	[1,1]	$\top$
	...	