

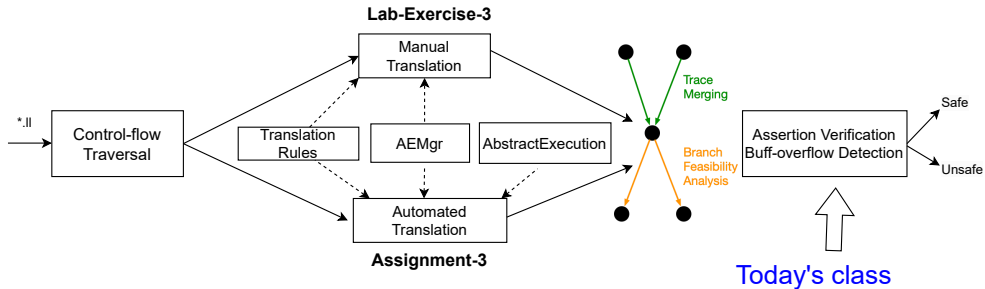
# Buffer Overflow Detection using Abstract Interpretation

## (Week 10)

Yulei Sui

School of Computer Science and Engineering  
University of New South Wales, Australia

# Today's class



# Buffer Overflows

## Definition (Buffer Overflow)

Given a buffer `buf` of `sz` bytes allocated in memory, an overflow occurs if an access offset `off` is used to access `buf` at or beyond its boundary, i.e.,  $off \geq sz$ .

- A buffer overflow vulnerability occurs when a program exceeds the capacity of a fixed-length memory block (buffer) by reading from or writing more data to it than it was designed to hold.
- Excess (overflowed) data can disrupt nearby memory, causing system errors or unauthorised code execution if manipulated by malicious attackers.

# Top ( $\top$ ) and Bottom ( $\perp$ ) and Narrowing Without Loop Bounds

- The default value of an `AbstractValue` is  $\langle \perp, \perp \rangle$ , consisting of an empty interval and an empty address set (if a variable is not found in maps  $\sigma$  or  $\delta$ ).
- The `AbstractValue` of a variable will be set or **initialized as**  $\langle \top, \top \rangle$  if this variable is **a program input** (e.g., arguments of the main function), representing all possible values.
- For a while loop without an explicit bound (e.g., `while(true){...}`), narrowing cannot be performed effectively; it remains a widening over-approximation.
- As in Assignment-2, there is NO need to handle external APIs (e.g., `stdlib's` API without function bodies) or LLVM's intrinsic APIs (e.g., `llvm.memcpy`) in Assignment-3.

# Example 1: Struct and Array

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define NFT_LEN 16
4  typedef struct {
5      char buffer[8];
6  } nft_set_elem;
7  void nft_set_elem_init(nft_set_elem *elem,
8                          int len) {
9      // Some initialization code is omitted here
10     elem->buffer[len - 1] = '\0';
11 }
12 int main() {
13     // Call the initialization function
14     nft_set_elem elem;
15     nft_set_elem_init(&elem, NFT_LEN);
16     return 0;
17 }
```

- Do we have a buffer overflow?

# Example 1: Struct and Array

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define NFT_LEN 16
4  typedef struct {
5      char buffer[8];
6  } nft_set_elem;
7  void nft_set_elem_init(nft_set_elem *elem,
8                          int len) {
9      // Some initialization code is omitted here
10     elem->buffer[len - 1] = '\0';
11 }
12 int main() {
13     // Call the initialization function
14     nft_set_elem elem;
15     nft_set_elem_init(&elem, NFT_LEN);
16     return 0;
17 }
```

- Do we have a buffer overflow?
- Yes, at Line 10.
- The value of `len - 1` is 15, which is out of bounds for the buffer `elem → buffer` which has a size of 8.

## Example 2: Struct and Array

```
1  #include <stdio.h>
2  #include <string.h>
3  #define NFT_LEN 16
4  typedef struct {
5      char buffer[8];
6  } nft_set_elem;
7  void nft_set_elem_init(nft_set_elem *elem,
8                          int len) {
9      // Ensure we do not overflow the buffer
10     if (len > sizeof(elem->buffer))
11         elem->buffer[sizeof(elem->buffer)-1] = '\0';
12     else
13         elem->buffer[len - 1] = '\0';
14 }
15 int main() {
16     // Call the initialization function
17     nft_set_elem elem;
18     nft_set_elem_init(&elem, NFT_LEN);
19     return 0;
20 }
```

- Do we have a buffer overflow?

## Example 2: Struct and Array

```
1  #include <stdio.h>
2  #include <string.h>
3  #define NFT_LEN 16
4  typedef struct {
5      char buffer[8];
6  } nft_set_elem;
7  void nft_set_elem_init(nft_set_elem *elem,
8                          int len) {
9      // Ensure we do not overflow the buffer
10     if (len > sizeof(elem->buffer))
11         elem->buffer[sizeof(elem->buffer)-1] = '\0';
12     else
13         elem->buffer[len - 1] = '\0';
14 }
15 int main() {
16     // Call the initialization function
17     nft_set_elem elem;
18     nft_set_elem_init(&elem, NFT_LEN);
19     return 0;
20 }
```

- Do we have a buffer overflow?
- No
- Line 12 ensures that the buffer is safely accessed. The buffer is not exceeded, and the string ends with a null character.



## Example 3: Struct and Array

```
1  #include <stdio.h>
2  struct Data {
3      int value;
4      char name[5];
5  };
6  void process_data_array(struct Data *data_array,
7                          int size) {
8      for (int i = 0; i < size; i++) {
9          for (int j = 0; j < size; j++) {
10             data_array[i].name[j] = 'A';
11         }
12         data_array[i].name[size-1] = '\0';
13     }
14 }
15 int main() {
16     struct Data data_array[10];
17     process_data_array(data_array, 10);
18     return 0;
19 }
```

- Do we have a buffer overflow?

## Example 3: Struct and Array

```
1  #include <stdio.h>
2  struct Data {
3      int value;
4      char name[5];
5  };
6  void process_data_array(struct Data *data_array,
7                          int size) {
8      for (int i = 0; i < size; i++) {
9          for (int j = 0; j < size; j++) {
10             data_array[i].name[j] = 'A';
11         }
12         data_array[i].name[size-1] = '\0';
13     }
14 }
15 int main() {
16     struct Data data_array[10];
17     process_data_array(data_array, 10);
18     return 0;
19 }
```

- Do we have a buffer overflow?
- Yes, at Line 10 and Line 12
- The loop for (int j = 0; j < size; j++) writes past the end of the name array, as size is larger than the size of name array.

## Example 4: Loop

```
1  #include <stdio.h>
2  #define BUF_LEN 20
3  void handle_buffer(char *input) {
4      char buffer[BUF_LEN];
5      for(int i = 0; i < 30; i++) {
6          buffer[i] = input[i];
7          if (input[i] != '\0')
8              break;
9      }
10     buffer[BUF_LEN-1] = '\0';
11     printf("Buffer content: %s\n", buffer);
12 }
13 int main() {
14     char input[30] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ123";
15     handle_buffer(input);
16     return 0;
17 }
```

- Do we have a buffer overflow?

## Example 4: Loop

```
1  #include <stdio.h>
2  #define BUF_LEN 20
3  void handle_buffer(char *input) {
4      char buffer[BUF_LEN];
5      for(int i = 0; i < 30; i++) {
6          buffer[i] = input[i];
7          if (input[i] != '\0')
8              break;
9      }
10     buffer[BUF_LEN-1] = '\0';
11     printf("Buffer content: %s\n", buffer);
12 }
13 int main() {
14     char input[30] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ123";
15     handle_buffer(input);
16     return 0;
17 }
```

- Do we have a buffer overflow?
- Yes, at Line 6.
- The size of the source buffer input is larger than the destination buffer when performing an element-wise copying.

## Example 5: Loop

```
1  void process_input(char input[5][10]) {
2      char buffer[50];
3      int i, j, k = 0;
4      for (i = 0; i < 5; i++) {
5          for (j = 0; j <= 10; j++) {
6              buffer[k++] = input[i][j];
7          }
8      }
9      buffer[49] = '\0';
10 }
11 int main() {
12     char input[5][10] = {
13         "1234567890",
14         "abcdefghij",
15         "ABCDEFGHIJ",
16         "0987654321",
17         "ZYXWVUTSRQ" };
18     process_input(input);
19     return 0;
20 }
```

- Do we have a buffer overflow?

## Example 5: Loop

```
1  void process_input(char input[5][10]) {
2      char buffer[50];
3      int i, j, k = 0;
4      for (i = 0; i < 5; i++) {
5          for (j = 0; j <= 10; j++) {
6              buffer[k++] = input[i][j];
7          }
8      }
9      buffer[49] = '\0';
10 }
11 int main() {
12     char input[5][10] = {
13         "1234567890",
14         "abcdefghij",
15         "ABCDEFGHIJ",
16         "0987654321",
17         "ZYXWVUTSRQ" };
18     process_input(input);
19     return 0;
20 }
```

- Do we have a buffer overflow?
- Yes, at Line 6.
- The loop for (j = 0; j <= 10; j++) writes past the end of the input[i] array, as the inner loop bound can equal to 10.

## Example 6: Loop

```
1  #define BUF_LEN 20
2  bool continue_copying = true;
3  void copy_data(char *input) {
4      char buffer[BUF_LEN];
5      int i = 0;
6      while (continue_copying) {
7          buffer[i] = input[i];
8          i++;
9          if (input[i] == '\0') {
10             continue_copying = false;
11         }
12     }
13     buffer[BUF_LEN-1] = '\0';
14     printf("Buffer content: %s\n", buffer);
15 }
16 int main() {
17     char input[30] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ123";
18     copy_data(input);
19     return 0;
20 }
```

- Do we have a buffer overflow?

## Example 6: Loop

```
1  #define BUF_LEN 20
2  bool continue_copying = true;
3  void copy_data(char *input) {
4      char buffer[BUF_LEN];
5      int i = 0;
6      while (continue_copying) {
7          buffer[i] = input[i];
8          i++;
9          if (input[i] == '\0') {
10             continue_copying = false;
11         }
12     }
13     buffer[BUF_LEN-1] = '\0';
14     printf("Buffer content: %s\n", buffer);
15 }
16 int main() {
17     char input[30] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ123";
18     copy_data(input);
19     return 0;
20 }
```

- Do we have a buffer overflow?
- Yes, at Line 7.
- The condition while (continue\_copying) does not check the buffer size. If the input string is longer than the buffer, it will write past the end of the buffer.
- **Narrowing will not work effectively, as the bound of the loop is not explicit.**



## Example 7: Interprocedural

```
1  #define BUFFER_SIZE 10
2  void handle_client_request(char *input,
3                             int index) {
4      int buffer[BUFFER_SIZE] = { 0 };
5      if (index >= 0)
6          buffer[index] = input[index];
7      else
8          printf("ERR: Array index is negative\n");
9  }
10 void process_socket_data(char *input,
11                           int index) {
12     handle_client_request(input, index);
13 }
14 int main(int index) {
15     char inputBuffer[BUFFER_SIZE] = {0};
16     process_socket_data(inputBuffer, index);
17     return 0;
18 }
```

- Do we have a buffer overflow?

## Example 7: Interprocedural

```
1  #define BUFFER_SIZE 10
2  void handle_client_request(char *input,
3                             int index) {
4      int buffer[BUFFER_SIZE] = { 0 };
5      if (index >= 0)
6          buffer[index] = input[index];
7      else
8          printf("ERR: Array index is negative\n");
9  }
10 void process_socket_data(char *input,
11                           int index) {
12     handle_client_request(input, index);
13 }
14 int main(int index) {
15     char inputBuffer[BUFFER_SIZE] = {0};
16     process_socket_data(inputBuffer, index);
17     return 0;
18 }
```

- Do we have a buffer overflow?
- Yes, at Line 6.
- The code does not check if index is less than BUFFER\_SIZE in handle\_client\_request. This can lead to a buffer overflow if index is 10 or greater.

## Example 8: Interprocedural

```
1  #define BUFFER_SIZE 10
2  void handle_client_request(char *input,
3                             int index) {
4      int buffer[BUFFER_SIZE] = { 0 };
5      if (index >= 0 && index < BUFFER_SIZE)
6          buffer[index] = input[index];
7      else
8          printf("ERR: Array index is out of bounds\n");
9  }
10 void process_socket_data(char *input,
11                           int index) {
12     handle_client_request(input, index);
13 }
14 int main(int index) {
15     char inputBuffer[BUFFER_SIZE] = {0};
16     process_socket_data(inputBuffer, index);
17     return 0;
18 }
```

- Do we have a buffer overflow?

## Example 8: Interprocedural

```
1  #define BUFFER_SIZE 10
2  void handle_client_request(char *input,
3                             int index) {
4      int buffer[BUFFER_SIZE] = { 0 };
5      if (index >= 0 && index < BUFFER_SIZE)
6          buffer[index] = input[index];
7      else
8          printf("ERR: Array index is out of bounds\n");
9  }
10 void process_socket_data(char *input,
11                           int index) {
12     handle_client_request(input, index);
13 }
14 int main(int index) {
15     char inputBuffer[BUFFER_SIZE] = {0};
16     process_socket_data(inputBuffer, index);
17     return 0;
18 }
```

- Do we have a buffer overflow?
- No
- The code now checks if index is within the valid range (0 to BUFFER\_SIZE - 1) in handle\_client\_request, preventing buffer overflows.

## Example 9: Branch

```
1  #include "stdbool.h"
2  int main(int argc) {
3      int buf[10];
4      int *loc = malloc(sizeof(int));
5      int i = argc % 10;
6      if (argc > 0) {
7          *loc = i;
8      } else {
9          *loc = ++i;
10     }
11     int idx = *loc;
12     buf[idx] = 1;
13 }
```

- Do we have a buffer overflow?

## Example 9: Branch

```
1  #include "stdbool.h"
2  int main(int argc) {
3      int buf[10];
4      int *loc = malloc(sizeof(int));
5      int i = argc % 10;
6      if (argc > 0) {
7          *loc = i;
8      } else {
9          *loc = ++i;
10     }
11     int idx = *loc;
12     buf[idx] = 1;
13 }
```

- Do we have a buffer overflow?
- Yes, at Line 12.
- The value of the index variable `idx` can be 10, which exceeds the size 10 of the buffer `buf`.

## Example 10 : Branch

```
1  #include "stdbool.h"
2  #include <stdlib.h>
3  int main(int argc) {
4      int buf[10];
5      int *loc = malloc(sizeof(int));
6      int i = argc % 10;
7      if (argc > 0) {
8          *loc = i;
9      } else {
10         *loc = ++i;
11     }
12     int idx = *loc;
13     if (idx >= 0 && idx < 10) {
14         buf[idx] = 1;
15     }
16     free(loc);
17     return 0;
18 }
```

- Do we have a buffer overflow?

## Example 10 : Branch

```
1  #include "stdbool.h"
2  #include <stdlib.h>
3  int main(int argc) {
4      int buf[10];
5      int *loc = malloc(sizeof(int));
6      int i = argc % 10;
7      if (argc > 0) {
8          *loc = i;
9      } else {
10         *loc = ++i;
11     }
12     int idx = *loc;
13     if (idx >= 0 && idx < 10) {
14         buf[idx] = 1;
15     }
16     free(loc);
17     return 0;
18 }
```

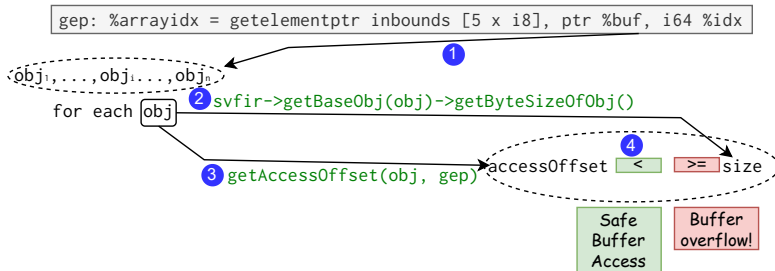
- Do we have a buffer overflow?
- No
- The index variable `idx` is checked to ensure it is within the valid range `[0, 9]` before accessing `buf`.



# How to Detect Buffer Overflow?

Given a buffer access  $r = \text{buf}[\text{idx}]$ , let's check whether there is a buffer overflow:

- ① We find the memory objects (addresses) pointed by `buf`.
  - For each object `obj`:
    - ② Find the byte size of `obj`, denoted as `size = bytesize(obj)`.
    - ③ Calculate access byte offset of `obj` considering both `idx` and its nested offset if `obj` is a sub-object of a memory allocation, via `accessOffset = accessByteOffset(obj, idx)`.
    - ④ Check `accessOffset < size`. If not hold, report a potential buffer overflow. **Note that abstract interpretation is an over-approximation technique and can produce false alarms.**



# Algorithm for Buffer Overflow Detection on SVFIR

---

## Algorithm 1: Buffer Overflow Detection for GEPSTMT

---

```
1 Function bufOverflowDetection(gep):  
2   as = getAbsStateFromTrace(gep → getICFGNode());  
3   lhs = gep → getLHSVarID();  
4   rhs = gep → getRHSVarID();  
5   updateGepObjOffsetFromBase(as[lhs].getAddrs(), as[rhs].getAddrs(), as.getByteOffset(gep))  
6   objAddrs = as[rhs].getAddrs(); ❶  
7   for objAddr ∈ objAddrs do  
8     obj = AESTate :: getInternalID(objAddr);  
9     size = svfir → getBaseObj(obj) → getByteSizeOfObj(); ❷  
10    accessOffset = getAccessOffset(obj, gep); ❸  
11    if accessOffset.ub().getIntNumeral() ≥ size ❹ then  
12      | reportBufOverflow(gep → getICFGNode());
```

---

# Important APIs for Assignment 3

Class	API	Description
AbstractExecution	getAbsStateFromTrace(node)	Returns the abstract state immediately after a given ICFGNode
AState	as.getInternalID(addr)	Returns the internal SVFVar ID of a given address
	as.loadValue(varId)	Loads the abstract value of the given variable ID
	as.storeValue(varId, val)	Stores the abstract value at the given variable ID
	as.getByteOffset(gep)	Returns the byte offset of the GEP statement
	as.getElementIndex(gep)	Returns the element index of the GEP statement
	as.widening(as')	Return a state after widening two given states
AbstractValue	as.narrowing(as')	Return a state after narrowing two given states
	getAddrs()	Returns the address values in the abstract value
IntervalValue	getInterval()	Returns the interval values in the abstract value
	lb()	Returns the lower bound of the interval
Options	ub()	Returns the upper bound of the interval
	WidenDelay()	Returns the value of the widen delay option

\*<https://github.com/SVF-tools/Software-Security-Analysis/wiki/AE-APIs#assignment-3>

# Handling LOADSTMT, STORESTMT and GEPSTMT

---

## Algorithm 2: Abstract Execution Algorithm for LOADSTMT

---

```
1 Function updateStateOnLoad(load):  
2   node = load → getICFGNode();  
3   as = getAbsStateFromTrace(node);  
4   rhs = load → getRHSVarID();  
5   lhs = load → getLHSVarID();  
6   as[lhs] = as.loadValue(rhs);  
7 Function AESTate :: loadValue(varId):  
8   AbstractValue res;  
9   for addr : (*this)[varId].getAddrs() do  
10    | res.join_with(load(addr)); //join values of all objects  
11   return res;
```

---

---

## Algorithm 3: Abstract Execution Algorithm for STORESTMT

---

```
1 Function updateStateOnStore(store):  
2   node = store → getICFGNode();  
3   as = getAbsStateFromTrace(node);  
4   rhs = store → getRHSVarID();  
5   lhs = store → getLHSVarID();  
6   as.storeValue(lhs, as[rhs]);  
7 Function AESTate :: storeValue(varId, val):  
8   for addr : (*this)[varId].getAddrs() do  
9   | store(addr, val);
```

---

---

## Algorithm 4: Abstract Execution Algorithm for GEPSTMT

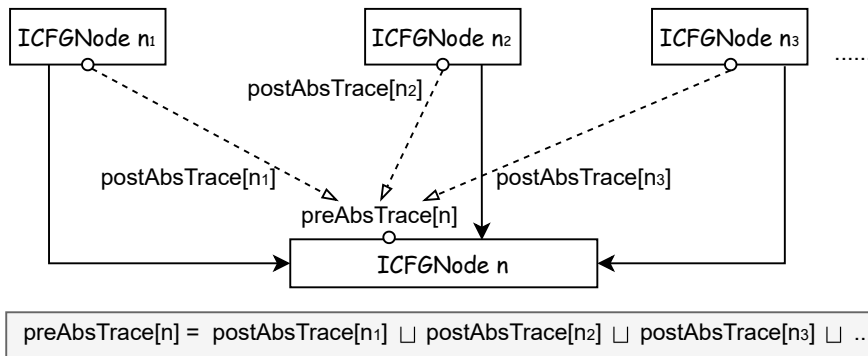
---

```
1 Function updateStateOnGep(gep):  
2   node = gep → getICFGNode();  
3   as = getAbsStateFromTrace(node);  
4   rhs = gep → getRHSVarID();  
5   lhs = gep → getLHSVarID();  
6   as[lhs] = as.getGepObjAddrs(rhs, as.getElementIndex(gep));
```

---

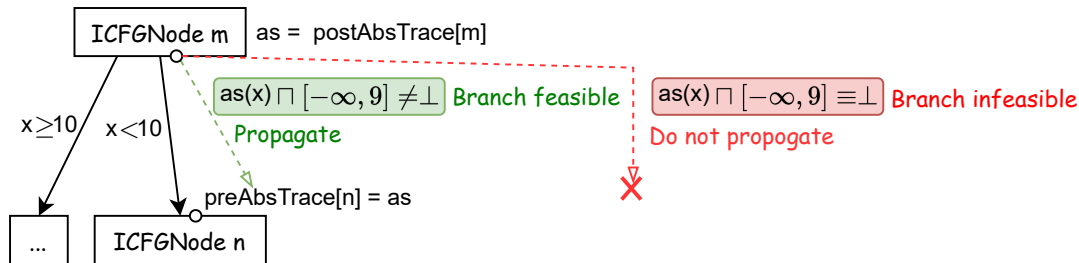
# Merge Abstract State From Predecessors

## Unconditional Branch



# Merge Abstract State From Predecessors

## Conditional Branch



# Step-by-Step: A Branch Example

```
1  #include "stdbool.h"
2  int main(int argc) {
3      int buf[10];
4      int *loc = malloc();
5      int i = argc % 10;
6      if (argc > 0) {
7          *loc = i;
8      } else {
9          *loc = ++i;
10     }
11     int idx = *loc;
12     buf[idx] = 1;
13 }
```

```
define dso_local i32 @main(i32 noundef %argc) #0 {
entry:
    %retval = alloca i32, align 4
    %argc.addr = alloca i32, align 4
    %buf = alloca [10 x i32], align 4
    %loc = alloca ptr, align 8
    %i = alloca i32, align 4
    %idx = alloca i32, align 4
    store i32 0, ptr %retval, align 4
    store i32 %argc, ptr %argc.addr, align 4
    %call = call @malloc(i64 noundef 4) #2
    store ptr %call, ptr %loc, align 8
    %0 = load i32, ptr %argc.addr, align 4
    %rem = srem i32 %0, 10
    store i32 %rem, ptr %i, align 4
    %1 = load i32, ptr %argc.addr, align 4
    %cmp = icmp sgt i32 %1, 0
    br i1 %cmp, label %if.then, label %if.else
}
```

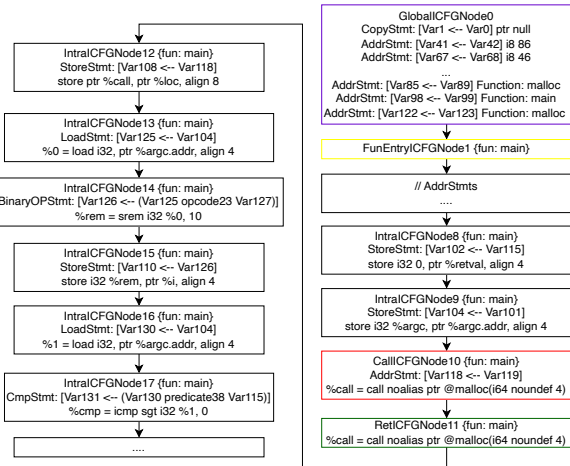
```
if.then:                                ; preds = %entry
    %2 = load i32, ptr %i, align 4
    %3 = load ptr, ptr %loc, align 8
    store i32 %2, ptr %3, align 4
    br label %if.end
```

```
if.else:                                ; preds = %entry
    %4 = load i32, ptr %i, align 4
    %inc = add nsw i32 %4, 1
    store i32 %inc, ptr %i, align 4
    %5 = load ptr, ptr %loc, align 8
    store i32 %inc, ptr %5, align 4
    br label %if.end
```

```
if.end:                                  ; preds = %if.else, %if.then
    %6 = load ptr, ptr %loc, align 8
    %7 = load i32, ptr %6, align 4
    store i32 %7, ptr %idx, align 4
    %8 = load i32, ptr %idx, align 4
    %idxprom = sext i32 %8 to i64
    %arrayidx = getelementptr inbounds [10 x i32], ptr %buf, i64 0, i64 %idxprom
    store i32 1, ptr %arrayidx, align 4
    %9 = load i32, ptr %retval, align 4
    ret i32 %9
}
```

LLVM IR

# Step-by-Step: A Branch Example



Algorithm 5: Abstract execution guided by WTO

```

1 Function handleStatement( $\ell$ ):
2    $tmpAS := preAbsTrace[\ell]$ ;
3   if  $\ell$  is CONSTSTMT or ADDRSTMT then
4      $updateStateOnAddr(\ell)$ ;
5   else if  $\ell$  is COPYSTMT then
6      $updateStateOnCopy(\ell)$ ;
7   ...;

```

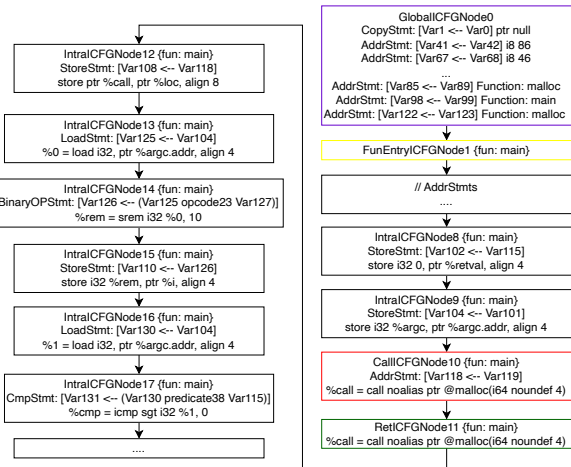
$postAbsTrace[ICFGNode17].varToAbsVal$  :

SVFVar	AbstractValue
Var0	{0x7f00}
Var1	{0x7f00}
Var104	0x7f000069
Var101	$[-\infty, +\infty]$
Var125	$[-\infty, +\infty]$
Var126	$[-9, +9]$
Var130	$[-\infty, +\infty]$

Program input argument Var101 is set to be  $\top$ .  
Both Var125 and Var130 are argc loaded from memory.  
Var126 is variable i, which is  $[-9, 9]$  as  $i = \text{argc} \bmod 10$ .



# Step-by-Step: A Branch Example



**Algorithm 6:** Abstract execution guided by WTO

```

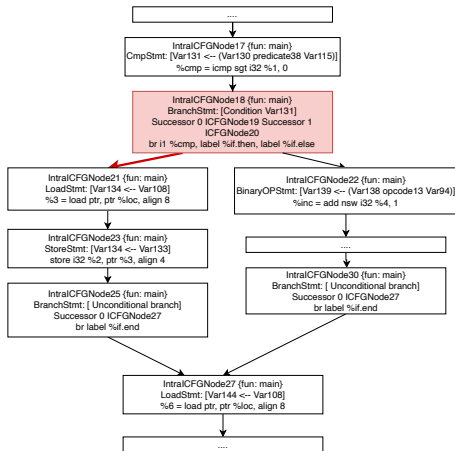
1 Function handleStatement( $\ell$ ):
2    $tmpAS := preAbsTrace[\ell]$ ;
3   if  $\ell$  is CONSTSTMT or ADDRSTMT then
4     updateStateOnAddr( $\ell$ );
5   else if  $\ell$  is COPYSTMT then
6     updateStateOnCopy( $\ell$ );
7   ...;
    
```

$postAbsTrace[ICFGNode17].varToAbsVal$  :

SVFVar	AbstractValue
Var0	{0x7f00}
Var1	{0x7f00}
Var104	0x7f000069
Var101	$[-\infty, +\infty]$
Var125	$[-\infty, +\infty]$
Var126	$[-9, +9]$
Var130	$[-\infty, +\infty]$
Var131	$[-\infty, +\infty]$

Var131 is the boolean branch condition.

# Step-by-Step: A Branch Example



## Algorithm 7: Whether Branch is Feasible

```

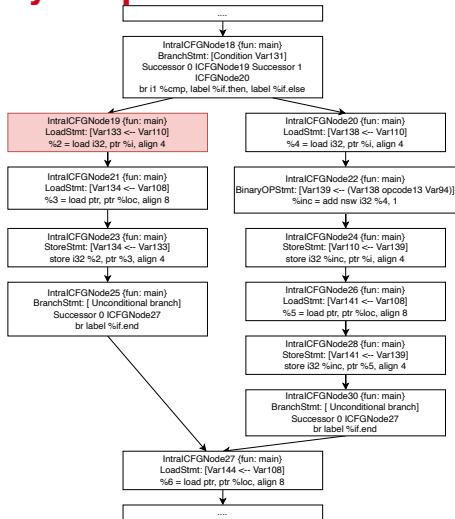
1 Function isBranchFeasible(intraEdge, as):
2   cond = intraEdge → getCondition();
3   cmpID = svfir → getValueNode(cond);
4   cmpVar = svfir → getGNode(cmpID);
5   if cmpVar → getInEdges().empty() then
6     return isSwitchBranchFeasible(cmpVar, intraEdge → getSuccessorCondValue(), as)
7   else
8     cmpVarInStmt = *cmpVar → getInEdges().begin();
9     if cmpStmt = SVFUtil :: dyn.cast < CmpStmt > (cmpVarInStmt) then
10       return isCmpBranchFeasible(cmpStmt, intraEdge → getSuccessorCondValue(), as)
11     else
12       return isSwitchBranchFeasible(cmpVar, intraEdge → getSuccessorCondValue(), as)
  
```

*preAbsTrace[ICFGNode19].varToAbsVal :*

SVFVar	AbstractValue
...	
Var130	$[1, +\infty]$
0x7f000069	$[1, +\infty]$
...	

The abstract state of Var130 (argc) in the if branch is updated to  $[-\infty, +\infty] \sqcap [1, +\infty]$

# Step-by-Step: A Branch Example



## Algorithm 8: Abstract Execution Algorithm for LOADSTMT

```

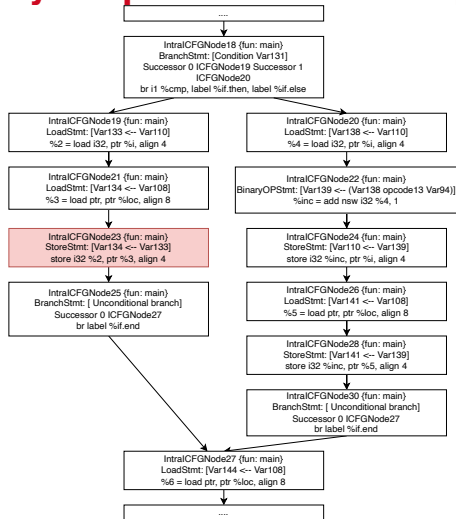
1 Function updateStateOnLoad(load):
2   node = load → getICFGNode();
3   as = getAbsStateFromTrace(node);
4   rhs = load → getRHSVarID();
5   lhs = load → getLHSVarID();
6   as[lhs] = as.loadValue(rhs)
7 Function AESTate :: loadValue(varId):
8   AbstractValue res;
9   for addr : (*this)[varId].getAddrs() do
10     res.join_with(load(addr));
11   return res;
  
```

*postAbsTrace[ICFGNode19].varToAbsVal :*

SVFVar	AbstractValue
...	
Var110	{0x7f00006f}
0x7f00006f	[−9, 9]
Var133	[−9, 9]
...	

Var133 is variable i

# Step-by-Step: A Branch Example



## Algorithm 9: Abstract Execution Algorithm for STORESTMT

```

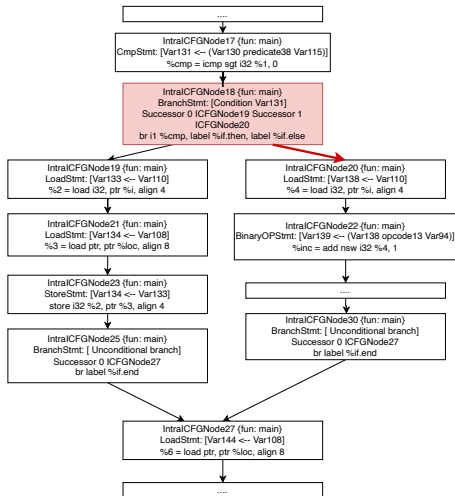
1 Function updateStateOnStore(store):
2   node = store → getICFGNode();
3   as = getAbsStateFromTrace(node);
4   rhs = store → getRHSVarID();
5   lhs = store → getLHSVarID();
6   as.storeValue(lhs, as[rhs])
7 Function AESTate :: storeValue(varId, val):
8   for addr : (*this)[varId].getAddr() do
9     store(addr, val);
  
```

*postAbsTrace[ICFGNode23].varToAbsVal :*

SVFVar	AbstractValue
...	
Var133	$[-9, 9]$
Var134	$\{0x7f000077\}$
0x7f000077	$[-9, 9]$
...	

Var133 is variable i  
 Var134 is pointer loc, which points to  
 address 0x7f000077

# Step-by-Step: A Branch Example



## Algorithm 10: Whether Branch is Feasible

```

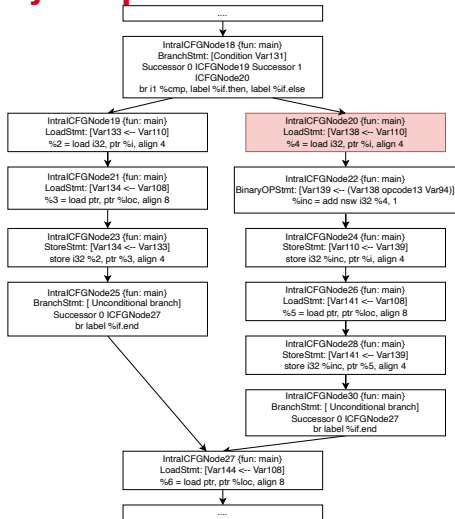
1 Function isBranchFeasible(intraEdge, as):
2   cond = intraEdge → getCondition();
3   cmpID = svfir → getValueNode(cond);
4   cmpVar = svfir → getNode(cmpID);
5   if cmpVar → getInEdges().empty() then
6     return isSwitchBranchFeasible(cmpVar, intraEdge → getSuccessorCondValue(), as)
7   else
8     cmpVarInStmnt = *cmpVar → getInEdges().begin();
9     if cmpStmnt = SVFUtil :: dyn_cast < CmpStmnt > (cmpVarInStmnt) then
10       return isCmpBranchFeasible(cmpStmnt, intraEdge → getSuccessorCondValue(), as)
11     else
12       return isSwitchBranchFeasible(cmpVar, intraEdge → getSuccessorCondValue(), as)
  
```

*preAbsTrace*[ICFGNode20].varToAbsVal :

SVFVar	AbstractValue
...	
Var130	$[\infty, 0]$
0x7f000069	$[-\infty, 0]$
...	

The abstract state of Var130 (argc) in the if.else branch is updated to  $[-\infty, +\infty] \sqcap [-\infty, 0]$   
0x7f000069 is the address of argc

# Step-by-Step: A Branch Example



## Algorithm 11: Abstract Execution Algorithm for LOADSTMT

```

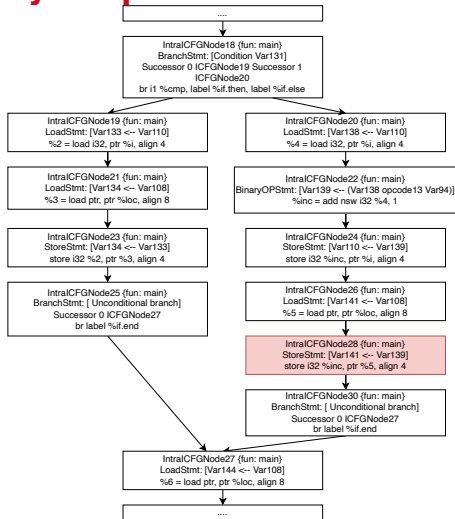
1 Function updateStateOnLoad(load):
2   node = load → getICFGNode();
3   as = getAbsStateFromTrace(node);
4   rhs = load → getRHSVarID();
5   lhs = load → getLHSVarID();
6   as[lhs] = as.loadValue(rhs)
7 Function AESTate :: loadValue(varId):
8   AbstractValue res;
9   for addr : (*this)[varId].getAddrs() do
10     res.join_with(load(addr));
11   return res;
  
```

*postAbsTrace[ICFGNode20].varToAbsVal :*

SVFVar	AbstractValue
	...
Var110	{0x7f00006f}
0x7f00006f	[−9, 9]
Var138	[−9, 9]
	...

Var138 is variable i before increment

# Step-by-Step: A Branch Example



## Algorithm 12: Abstract Execution Algorithm for STORESTMT

```

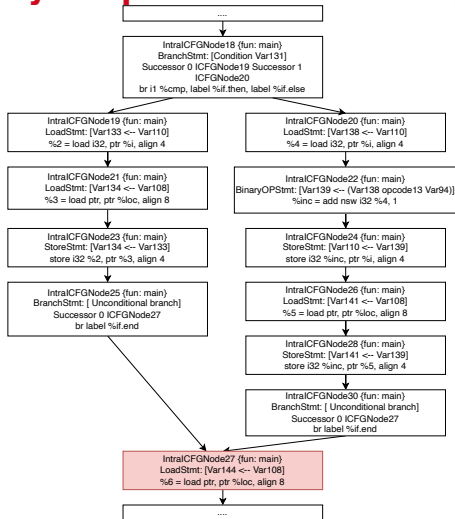
1 Function updateStateOnStore(store):
2   node = store → getICFGNode();
3   as = getAbsStateFromTrace(node);
4   rhs = store → getRHSVarID();
5   lhs = store → getLHSVarID();
6   as.storeValue(lhs, as[rhs])
7 Function AESTate :: storeValue(varId, val):
8   for addr : (*this)[varId].getAddr() do
9     store(addr, val);
  
```

*postAbsTrace[ICFGNode28].varToAbsVal :*

SVFVar	AbstractValue
...	
Var139	$[-8, 10]$
Var141	$\{0x7f000077\}$
0x7f000077	$[-8, 10]$
...	

Var139 is variable *i* after increment  
 Var141 is pointer *loc*, which points to  
 address 0x7f000077

# Step-by-Step: A Branch Example



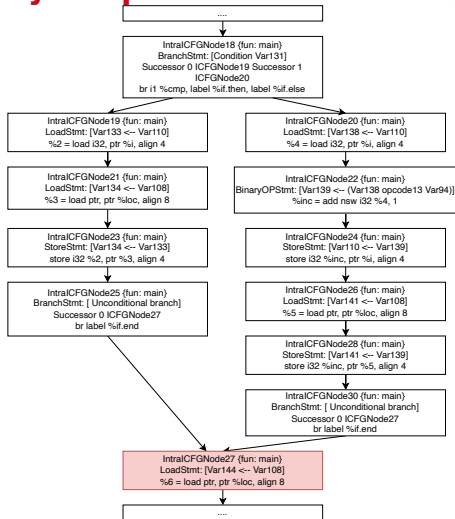
*preAbsTrace[ICFGNode27].varToAbsVal :*

SVFVar	AbstractValue
...	
Var108	{0x7f00006d}
0x7f00006d	{0x7f000077}
0x7f000077	[−9, 10]
...	

Address 0x7f000077 is pointed by pointer loc, its abstract value is [−9,10] formed by joining/merging [−9,9] (from ICFGNode 25) and [−8,10] (from ICFGNode 30)



# Step-by-Step: A Branch Example



## Algorithm 13: Abstract Execution Algorithm for LOADSTMT

```

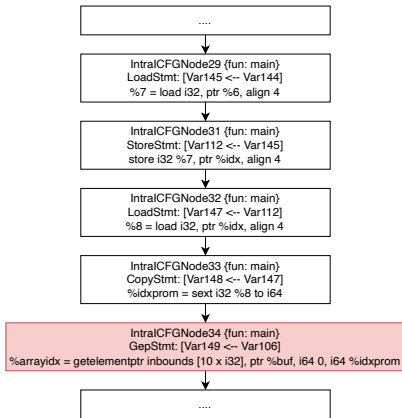
1 Function updateStateOnLoad(load):
2   node = load → getICFGNode();
3   as = getAbsStateFromTrace(node);
4   rhs = load → getRHSVarID();
5   lhs = load → getLHSVarID();
6   as[lhs] = as.loadValue(rhs)
7 Function AESTate :: loadValue(varId):
8   AbstractValue res;
9   for addr : (*this)[varId].getAddrs() do
10    res.join_with(load(addr));
11  returnres;
  
```

*postAbsTrace[ICFGNode27].varToAbsVal :*

SVFVar	AbstractValue
...	
Var108	{0x7f00006d}
0x7f00006d	{0x7f000077}
Var144	{0x7f000077}
0x7f000077	[−9, 10]
...	

Var144 is the value of \*loc, which will be used as an index idx to access array buf

# Step-by-Step: A Branch Example



**Algorithm 14:** Abstract Execution Algorithm for GEPSTMT

```
1 Function updateStateOnGep(gep):  
2   node = gep → getICFGNode();  
3   as = getAbsStateFromTrace(node);  
4   rhs = gep → getRHSVarID();  
5   lhs = gep → getLHSVarID();  
6   as[lhs] = as.getGepObjAddr(rhs, as.getElementIndex(gep));
```

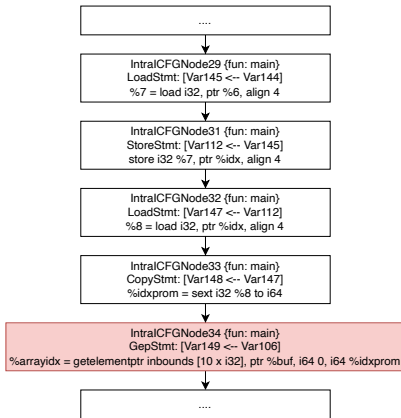
*postAbsTrace*[*ICFGNode27*].*varToAbsVal* :

SVFVar	AbstractValue
...	
Var106	{0x7f00006b}
Var149	{0x7f0000ea}
...	

Var106 is the base memory address  
of array buf

Var149 is the gep address of  
&buf[idx]

# Step-by-Step: A Branch Example



**Algorithm 15:** Buffer Overflow Detection for GEPSTMT

```
1 Function bufOverflowDetection(gep):
2   as = getAbsStateFromTrace(gep → getICFGNode());
3   lhs = gep → getLHSVarID();
4   rhs = gep → getRHSVarID();
5   updateGepObjOffsetFromBase(as[lhs].getAddrs(), as[rhs].getAddrs(), as.getByteOffset(gep))
6   objAddrs = as[rhs].getAddrs(); ①
7   for objAddr ∈ objAddrs do
8     obj = AESTate :: getInternalID(objAddr);
9     size = svfir → getBaseObj(obj) → getByteSizeOfObj(); ②
10    accessOffset = getAccessOffset(obj, gep); ③
11    if accessOffset.ub().getIntNumeral() >= size ④ then
12      | reportBufOverflow(gep → getICFGNode());
```

**Algorithm steps**

Step	Values	Explanation
①	$objAddrs = \{0x7f00006b\}$	from Var106
②	$size = [10, 10]$	from Var106
③	$accessOffset = [-9, 10]$	stored in 0x7f000077
④	True	overflow detected

# Handling Call Site

**Algorithm 16:** Abstract Execution for Function Call

```
1 Function handleCallSite(callNode):
2   as = getAbsStateFromTrace(callNode);
3   callee = SVFUtil :: getCallee(callNode → getCallSite());
4   if callee ∈ recursiveFuns then
5     | return; // we don't handle recursive functions
6   else
7     | callSiteStack.push_back(callNode);
8     | wto = funcToWTO[callee];
9     | handleWTOComponents(wto → getWTOComponents());
10    | callSiteStack.pop_back();
```

**Algorithm 17:** Abstract Execution Algorithm for WTOCOMPONENTS

```
1 Function handleWTOComponents (wtoComps):
2   for wtoNode ∈ wtoComps do
3     | if node = SVFUtil :: dyn_cast<ICFGSingletonWTO>(wtoNode) then
4       | handleSingletonWTO(node)
5     | else if cycle = SVFUtil :: dyn_cast<ICFGCycleWTO>(wtoNode)
6       | then
7         | handleCycleWTO(cycle)
8     | else
9       | assert(false&&"unknownWTOtype!")
```

# Step-by-Step: An Interprocedural Example

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #define CIRC_BUF_SIZE 20
5  #define ERR_MSG "Error: negative index!\n"
6
7  int getIdx(int index) {
8      return index % CIRC_BUF_SIZE;
9  }
10 int main(int input) {
11     char circBuf[CIRC_BUF_SIZE] = {0};
12     if(input < 0) {
13         printf(ERR_MSG);
14         return 1;
15     }
16     int circIdx = getIdx(input);
17     circBuf[circIdx] = 'A';
18     return 0;
19 }
```

```
define dso_local i32 @getIdx(i32 noundef %index) #0 {
entry:
    %index.addr = alloca i32, align 4
    store i32 %index, ptr %index.addr, align 4
    %0 = load i32, ptr %index.addr, align 4
    %rem = srem i32 %0, 20
    ret i32 %rem
}

define dso_local i32 @main(i32 noundef %input) #0 {
entry:
    %retval = alloca i32, align 4
    %input.addr = alloca i32, align 4
    %circularBuffer = alloca [20 x i8], align 1
    %circularIndex = alloca i32, align 4
    store i32 0, ptr %retval, align 4
    store i32 %input, ptr %input.addr, align 4
    call void @llvm.memset.p0.i64(ptr align 1 %circularBuffer,
i8 0, i64 20, i1 false)
    %0 = load i32, ptr %input.addr, align 4
    %cmp = icmp slt i32 %0, 0
    br i1 %cmp, label %if.then, label %if.end
}
```

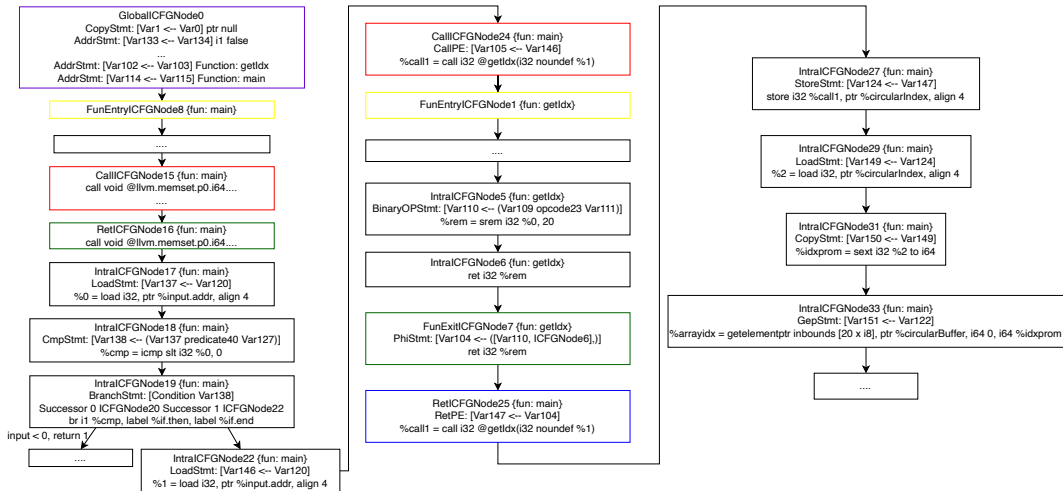
```
if.then:                                ; preds = %entry
    %call = call i32 @printf(ptr noundef @.str)
    store i32 1, ptr %retval, align 4
    br label %return

if.end:                                  ; preds = %entry
    %1 = load i32, ptr %input.addr, align 4
    %call1 = call i32 @getIdx(i32 noundef %1)
    store i32 %call1, ptr %circularIndex, align 4
    %2 = load i32, ptr %circularIndex, align 4
    %idxprom = sext i32 %2 to i64
    %arrayidx = getelementptr inbounds [20 x i8], ptr
%circularBuffer, i64 0, i64 %idxprom
    store i8 65, ptr %arrayidx, align 1
    store i32 0, ptr %retval, align 4
    br label %return

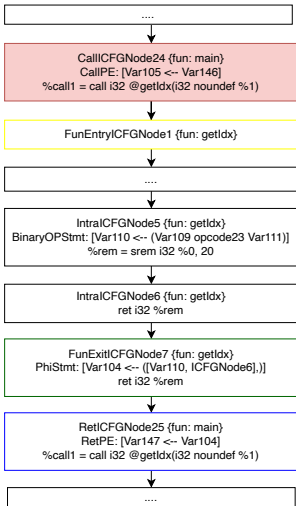
return:                                  ; preds = %if.end, %if.then
    %3 = load i32, ptr %retval, align 4
    ret i32 %3
}
```

LLVM IR

# Step-by-Step: An Interprocedural Example



## Step-by-Step: An Interprocedural Example



---

**Algorithm 18:** Abstract Execution for Function Call

```

1 Function handleCallSite(callNode):
2   as = getAbsStateFromTrace(callNode);
3   callee = SVFUtil :: getCallee(callNode → getCallSite());
4   if callee ∈ recursiveFuns then
5     | return;
6   else
7     callSiteStack.push_back(callNode);
8     wto = funcToWTO[callee];
9     handleWTOComponents(wto → getWTOComponents());
10    callSiteStack.pop_back();

```

```
postAbsTrace[ICFGNode24].varToAbsVal :
```

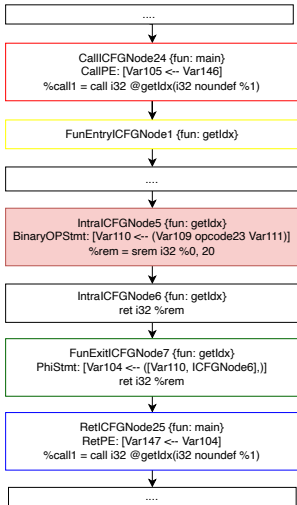
SVFVar	AbstractValue
...	
Var146	$[0, \infty]$
Var105	$[0, \infty]$
...	

*callSiteStack* :

[CallCFGNode24,]

The AbstractExecution in Assignment-3 is **context-insensitive** and callSiteStack is only used to maintain call stack information for bug reporting.

# Step-by-Step: An Interprocedural Example



**Algorithm 19:** Abstract Execution Algorithm for WTOCOMPONENTS

```

1 Function handleWTOComponents (wtoComps):
2   for wtoNode ∈ wtoComps do
3     if node = SVFUtil :: dyn.cast(ICFGSingletonWTO)(wtoNode) then
4       | handleSingletonWTO(node)
5     else if cycle = SVFUtil :: dyn.cast(ICFGCycleWTO)(wtoNode) then
6       | handleCycleWTO(cycle)
7     else
8       | assert(false&&"unknownWTOtype!")
  
```

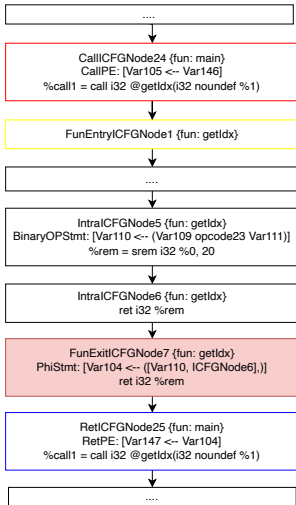
*postAbsTrace*[ICFGNode5].*varToAbsVal* :

SVFVar	AbstractValue
	...
Var109	$[0, +\infty]$
Var110	$[0, 19]$
	...

Var109 is variable index, which is  $[0, +\infty]$ .  
 Var110 is the return value of function *getIdx*,  
 which is  $[0, 19]$  as *Var110* = *index* mod 20.



# Step-by-Step: An Interprocedural Example



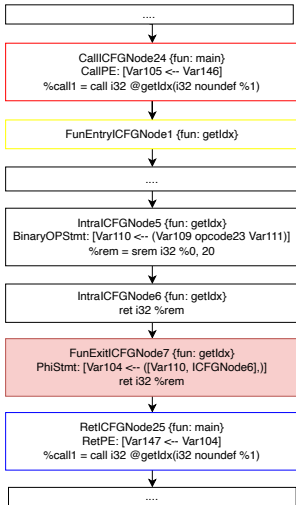
## Algorithm 20: Abstract Execution Algorithm for WTOCOMPONENTS

```
1 Function handleWTOComponents (wtoComps):
2   for wtoNode ∈ wtoComps do
3     if node = SVFUtil :: dyn.cast(ICFSSingletonWTO)(wtoNode) then
4       | handleSingletonWTO(node)
5     else if cycle = SVFUtil :: dyn.cast(ICFGCycleWTO)(wtoNode) then
6       | handleCycleWTO(cycle)
7     else
8       | assert(false&&"unknownWTOtype!")
```

*postAbsTrace*[ICFGNode7].varToAbsVal :

SVFVar	AbstractValue
...	
Var110	[0, 19]
Var104	[0, 19]
...	

# Step-by-Step: An Interprocedural Example



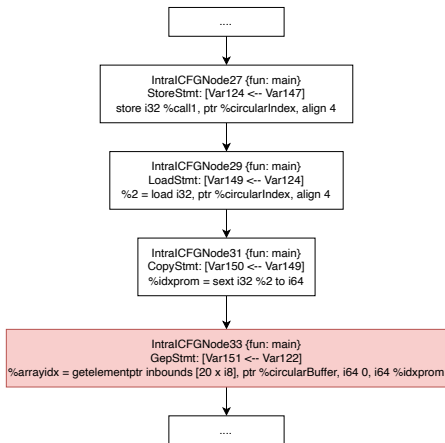
## Algorithm 21: Abstract Execution for Function Call

```
1 Function handleCallSite(callNode):
2   as = getAbsStateFromTrace(callNode);
3   callee = SVFUtil :: getCallee(callNode → getCallSite());
4   if callee ∈ recursiveFuns then
5     return;
6   else
7     callSiteStack.push_back(callNode);
8     wto = funcToWTO[callee];
9     handleWTOComponents(wto → getWTOComponents());
10    callSiteStack.pop_back();
```

*callSiteStack* :

[ ]

# Step-by-Step: An Interprocedural Example



## Algorithm 22: Buffer Overflow Detection for GEPSTMT

```
1 Function bufOverflowDetection(gep):  
2   as = getAbsStateFromTrace(gep → getICFGNode());  
3   lhs = gep → getLHSVarID();  
4   rhs = gep → getRHSVarID();  
5   updateGepObjOffsetFromBase(as[lhs].getAddrs(), as[rhs].getAddrs(), as.getByteOffset(gep))  
6   objAddrs = as[rhs].getAddrs(); ❶  
7   for objAddr ∈ objAddrs do  
8     obj = AESTate :: getInternalID(objAddr);  
9     size = svfir → getBaseObj(obj) → getByteSizeOfObj(); ❷  
10    accessOffset = getAccessOffset(obj, gep); ❸  
11    if accessOffset.ub().getIntNumeral() >= size ❹ then  
12      reportBufOverflow(gep → getICFGNode());
```

## Algorithm behavior

Step	Behavior
❶	$objAddrs = \{0x7f00007b\}$
❷	$size = [20, 20]$
❸	$accessOffset = [0, 19]$
❹	False, the buffer access is safe!

# Final Week and How to Make the Most of This Course

- You are now able to build your own code checkers and verifiers (including information flow tracking, symbolic execution, and abstract interpretation)
- Join and contribute to SVF code analysis framework?
  - <https://github.com/SVF-tools/SVF>
- Participate in software verification competitions (SVC)
  - <https://sv-comp.sosy-lab.org/>
  - <https://docs.google.com/document/d/1bgkx5lnugrwlNzQ2MPRSd47MAkZGJfR9v2jo7oRskd0/edit>
- An honours thesis project or a research degree (MPhil or PhD)?
- Tutor and lab demonstrator next year?