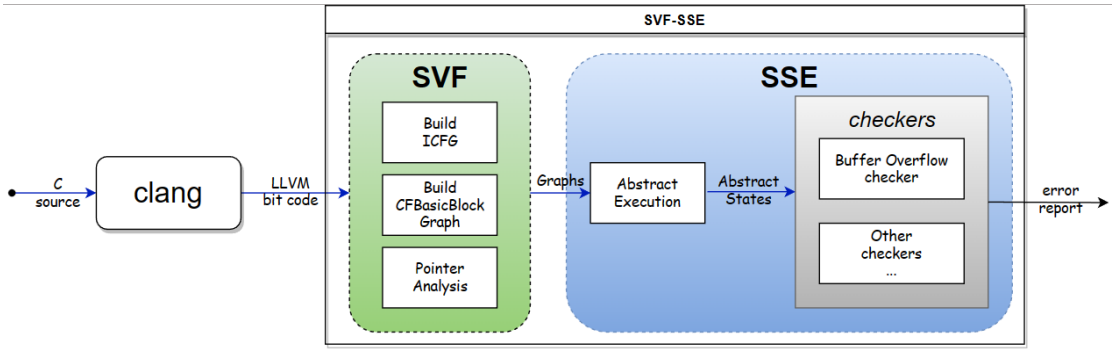
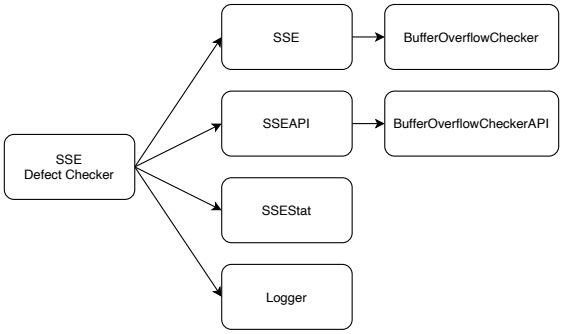
SSE简介

SSE (SVF Symbolic Execution)是一个程序静态分析器，能够输入LLVM比特流文件，并输出比特流文件的缺陷检测报告。其主要可以分为三个模块：Abstract Execution模块和Defect Checker模块。逻辑关系图如下：



Abstract Execution模块主要是通过抽象解释的原理，构建出关于输出程序中变量的可取范围推断；Defect Checker模块则是利用所构建的变量可取范围推断，进一步推断出程序中可能出现的错误，如缓冲区溢出错误等。

SSE项目类继承图

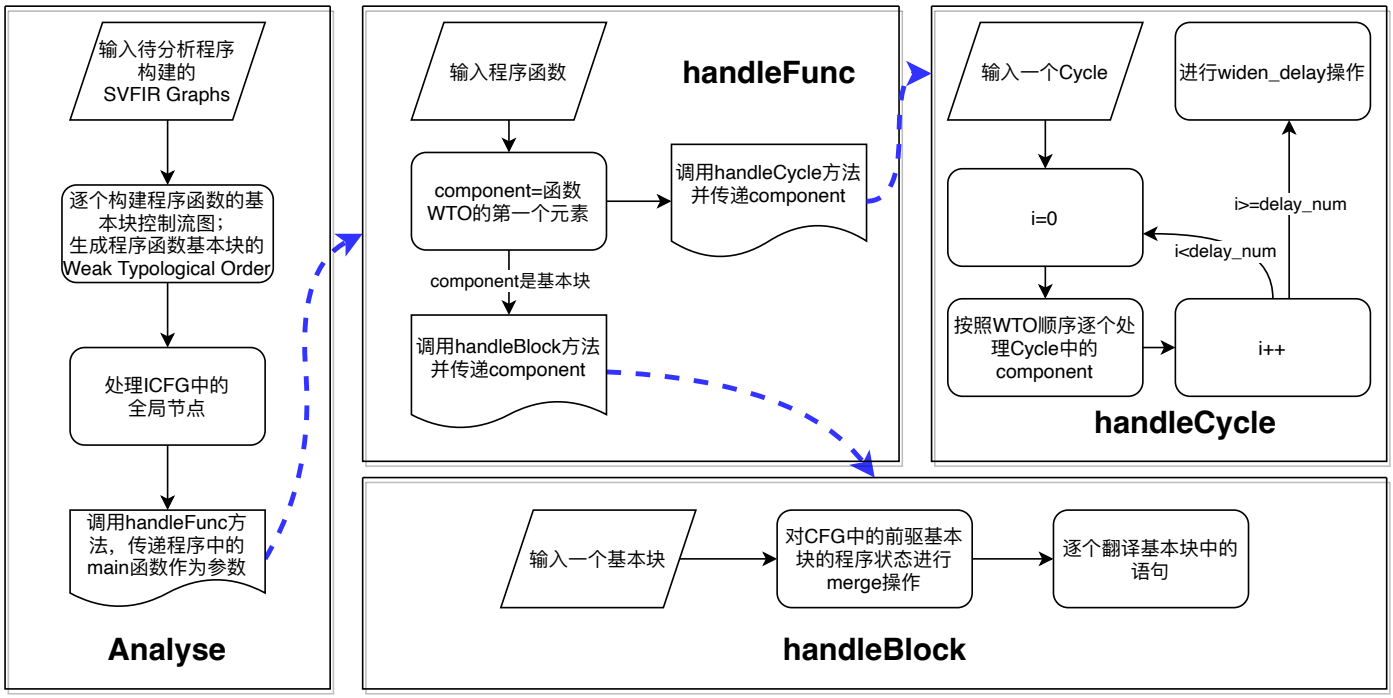


各模块对应的SVF/SSE项目文件

模块	项目	代码文件
Abstract Execution	SSE Project	sse.h/cpp
Defect Checker	SSE Project	bufferoverflow.h/cpp sseapi.h/cpp bufferoverflowapi.h/cpp
SVF相关API简介	SVF Project	svf/include/AbstractExecution/* svf/lib/AbstractExection/* svf/include/Graphs/CFBasicBlockG.h svf/lib/Graphs/CFBasicBlockG.cpp

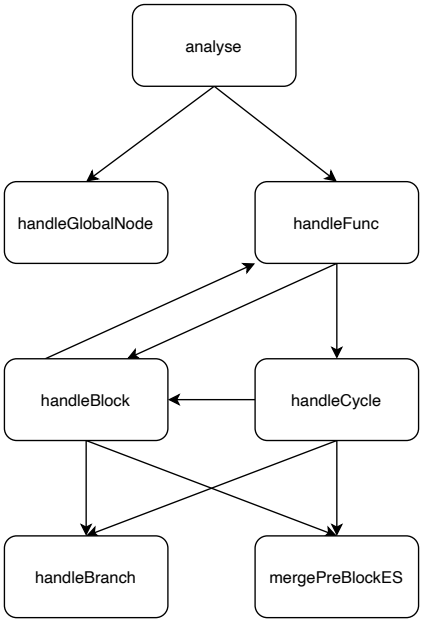
Abstract Execution模块

流程图



主要方法调用关系图

下图表示进行Abstract Execution的主要子过程之间的调用关系：



sse.h/cpp

`SSE`类型是进行静态符号执行的主要模块，以该类型为父类，可以定制各种缺陷探测的client，`BufferOverflowChecker`就是其中一种，用于检测缓冲区溢出缺陷。

variables

Type	Name	Description
ICFG *	icfg	表示被分析程序的过程间控制流图
Map<const CFBasicBlockEdge*, IntervalExeState>	esMap	表示`CFBasicBlockEdge`与执行完该边入节点基本块后一刻的`IntervalExeState`之间的对应关系
Map<const CFBasicBlockNode*, IntervalExeState>	entryNodeToES	表示函数入口基本块与其执行前一刻的执行状态之间的对应关系，由callsite指令信息生成
PTACallGraph *	callgraph	

Type	Name	Description
CFBasicBlockGraph *	_curGraph	表示当前正在处理的函数所对应的CFBasicBlockGraph
Map<const CFBasicBlockGWTOCycle *, IntervalExeState>	wtocycleMap	
Map<const ICFGNode *, IntervalExeState>	assertMap	
Map<const SVFFunction *, CFBasicBlockGWTO *>	funcWto	表示`SVFFunction`与`CFBasicBlockGWTO`之间的对应关系
Map<const CFBasicBlockGWTOComp*, const SVFFunction *>	wtoFunc	表示`CFGBasicBlockGWTO`中的每个`CFBasicBlockGWTOComp`节点和该序列所属的函数之间的对应关系
Map<const SVFFunction, *CFBasicBlockGraph*>	funcToCFBG	表示`SVFFunction`与`CFBasicBlockGraph`之间的对应关系
std::vector<const SVFFunction *>	callstack	表示进行符号执行过程中的调用路径信息
ICFG *	icfg	表示被分析程序的过程间控制流图

`SSE::analyze()`

伪代码

```
# Psudo code: SVF Abstract Execution
# entry of the whole analysis
def analyze(program):
    for func in program:
        # build basic block control flow graph for given func
        func_basic_block_graph = build_basic_block_CFG(func)
        # generate weak typological order of the built basic block CFG of func
        func_wto = generate_weak_typological_order(func_basic_block_graph)

    # pre-analyze recursions and stores recursive SVFFunctions in a set
    analyzeRecursive()

    # handle the global node in func's control flow graph
```

```
    handleGlobalNode()

    # starting from 'main' function to analyze the whole program
    handleFunc(main_func)
```

说明

`analyse`方法是整个抽象运行状态构建与缺陷分析过程的入口，在`main`函数中，该方法继于`initialize`方法之后被调用。该方法首先遍历`SVFIR`中的所有函数，并为每个函数构建一个`CFBasicBlockGraph`。然后，该方法为每个`CFBasicBlockGraph`构建各个函数控制流图的**弱拓扑序**，用于稍后进行的递归遍历。该弱拓扑序用`CFBasicBlockGWTO`类型表示，该类型为容器类型，提供了对`CFBasicBlockGWTOComp`的iterator。

上述准备工作完成之后，`analyse`方法将进行抽象运行状态的构建与缺陷分析。在该过程中，将首先调用`handleGlobalNode`方法处理`ICFG`中的全局节点，然后识别出`SVFIR`中以`main`命名的函数，将之作为第一个参数调用`handleFunc`方法。`handleFunc`方法将以递归的方式，完成整个`SVFIR`的抽象运行状态构建以及缺陷分析的过程。

`SSE::analyzeRecursive()`

伪代码

```
def analyzeRecursive():
    for node in PTACallGraph Nodes:
        # 1) starting from node, do Deep First Search to find loops (recursion)
        # 2) add node->getFunction to a set
```

`SSE::handleFunc()`

伪代码

```
def handleFunc(func, execution state&):
  for component in func_CFBasicBlockGraph_weak_typological_order:
      if component is cycle: # component is a cycle
          handleCycle(component)
      else: # component is a single basic block
          # handle the corresponding CFBasicBlockGraph Node
          handleBlock(component->getCFBasicBlockNode())
```

说明

`handleFunc`方法以某个被分析`SVFIR`中的函数为第一个参数输入，并以整个`SVFIR`的`IntervalExeState`为第二个参数输入。该方法将依次对输入函数中的`CFBasicBlockGWTOComp`进行处理，所依照的顺序是输入函数控制流图的弱拓扑序（已预先在该方法调用前被构建）。处理分为以下两种情况进行：

1. 若`CFBasicBlockGWTOComp`类型的对象实际上归属于`CFBasicBlockGWTONode`子类，那么调用`handleBasicBlock`方法并将其`CFBasicBlockNode`类型成员作为参数传递进行处理；

2. 若对象实际上归属于`CFBasicBlockGWTOCycle`子类，那么调用`handleCycle`方法并将其本身作为参数传递进行处理。

值得注意的是，在`handleFunc`开始处理函数前，会遍历`callstack`，检测当前处理路径中是否已经包含该函数，然后将包含该函数的情形视为递归，直接跳过处理。

`SSE::handleCycle()`

伪代码

```
def handleCycle(cycle):
    # before widen_delay, do normal executions
    for i in (0, widen_delay_num):
        # handle cycle's head basic block node
        handleBlock(cycle_head)

        # handle other blocks in cycle
        for component in cycle:
            if component is cycle: # is cycle
                handleCycle(component)
            else: # is basic block
                handleBlock(basic_block)

    # after looping steps >= widen_delay_num
    do_widen_delay_stuffs()
```

说明

`handleCycle`方法被`handleFunc`方法调用，用于对所输入`CFBasicBlockCycle`进行处理。在对循环中所包含的语句进行符号执行之前，首先调用`mergePreBlockES`方法，获得循环执行前一刻的符号执行状态，用`pre_es`变量保存。接下来循环执行以下的步骤：

1. 调用`handleBlock`方法处理循环的head。
2. 比较循环变量`i`和所设置的`widen_delay`参数大小，如果`i<widen_delay`，则更新pre_es为执行完head之后的ES值；否则进行widen delay操作（该操作的详细步骤省略）。
3. 依此对输入`CFBasicBlockGWTOCycle`的循环列表中所包含的`CFBasicBlockGWTONode`和`CFBasicBlockGWTOCycle`对象调用`handleBlock`和`handleCycle`方法进行处理。

上述循环的退出条件为：`pre_es`经过narrowing操作之后的`new_pre_es`之间满足`new_pre_es >= pre_es`。

`SSE::handleBlock()`

伪代码

```
def handleBlock(basic_block): # note: basic_block a single CFBasicBlockNode
    # do join operation on exeucution states of all predecessors of basic_block
    getInEdgesES(basic_block)

    # handle each statement in the basis block
    for icfg_node in basic_block:
        for svf_stmt in icfg_node:
            handleSVFStatement(statement)
        if icfg_node is CallICFGNode: # handle CallICFGNode
            handleCallSite(icfg_node)
```

`SSE::handleRecursiveCall()`

伪代码

```
def handleRecursiveCall(callICFGNode):
    retNode = get_return_node(callICFGNode)
    set_execution_state_top(retNode) # set return node execution state to top

    # calls handleRecursiveFunc to handle function
    handleRecursiveFunc(callICFGNode_callee)
```

说明

本方法和`handleRecursiveFunc()`均用于处理递归函数。对递归函数的处理是：1）将ret的execution state设置为top；2）将函数中StoreStmt的操作对象的execution state设置为top。

`SSE::handleRecursiveFunc()`

伪代码

```
def handleRecursiveFunc(SVFFunction):
    # get func statements' WTO order
    stmts_WTO = get_func_stmts_WTO(SVFFunction)

    # go through every statements, find StoreStmt and make top value
    for stmt in stmts_WTO:
        if stmt is StoreStmt:
            addresses = stmt_rhs_points_to_set  # get the points to set of rhs of store
stmt
            for address in addresses:
                # set address's execution state to top value
                set_execution_state_top(address)
```

SSE::handleCallsite()

伪代码

```
def handleCallSite(icfg_node):
    callee = get_callee_of_callICFGNode(icfg_node)
    if callee is NOT nullptr: # meaning is not calling by function pointer
        if func has CFBasicBlockGraph:  # meaning this func's implementation is
available
            handleFunc(callee, current_execution_state)
        elif get_func_name(func) is "svf_assert()":
            # add icfg_node to checkpoint
        else: # calling external function's that doesn't has implementation, e.g.,
memcpy()
            handle_external_function(func)
     else: # meaning is calling by function pointer
        # get function pointer's points-to set
        pts_functions = get_pts(function_pointer)
        handleFunc(pts_functions[0]) # pick out the first function and handle it
```

SSE::handleSVFStatement()

伪代码

```
def handleSVFStatement(SVFStmt):
    # translate SVFStmt, using SVFIR2ExeState's translateXX methods
    # 1) case1: address stmt, call translateAddr(SVFStmt)
    # 2) other cases: similar
```

说明

该方法的作用是对`SVFStatement`依此进行处理，根据指令的语义对execution state进行更新。

SSE::getBranchES()

伪代码

```
def getBranchES(in_icfg_edge):
    stmt = get_assignment_SVFStmt_of_edge_condition_SVFVar(in_icfg_edge)
    if(stmt is CmpStmt):  # is a cmp stmt (condition = cmp xxx)
        # note: path_success_cond == condition means the path is feasible
        getCmpBranchES(stmt, path_success_cond)
    else:  # other conditions, i.e., condition is switch(case)
        getSwitchBranchES()
```

`SSE::getCmpBranchES()`

伪代码

```
def getCmpBranchES(cmp_stmt, succ):
    # Note: path condition res is generated by statement: res = cmp op1 op2;
    #       succ is path successful value.

    # take care of condition that op1 = copy othervalue; and op1 = load ptr.
    load_stmt = select_load_stmt(getInedges(op1))

    if(res.meet_with(condition) is bottom): # e.g. res->{true}, path_condition->{false}
        # this path is not feasible
        return false
    else: # e.g. res->{true, false}, path_condition->{false}
        # modify op1 and op2's execution states according to cmp predicate and path
succ value
        if exec_state(op1) is NOT const and exec_state(op2) is const:
            # case1 op1 eq op2
            #   1) modify op1's exec_state to [const, const]
            #   2) also modify ptr's points-to objects exec_state to [const, const]
            # other cases: ne, gt, ge, lt, le are similar
            return true
        else:
            return true
```

说明

此函数的主要作用就是充分利用分支边上的branch条件，提升execution state的精度。以下面的例子说明：

```
int x, arr[5];
scanf("%d", &x);
x = x % 6;
if(x < 5){
  arr[x];
}else{
  // do sth.
}
```

对于if造成的分支，本方法会首先根据路径条件`condition`获取到生成该条件的`cmp`语句`condition = cmpLT x, 5`，然后对从前继节点`x = x % 6`获取的execution state`X->[0,5]`进行更新，使其变成`x->[0,4]`，提升了分析的精度。本函数目前只能够处理比较对象之一是常量的情形，二者都是变量的情形尚未进行处理。

`SSE::getSwitchBranchES()`

伪代码

```
def getSwitchBranchES(SVFVar *var, succ):
    # Note: var is switch condition varaible
    #       succ is case successful value.

    inEdges = getSVFVarInEdges(var)   # inEdges are SVFStmts that assigns to the var

    # first update the execution state of var to [const, const]
    set_var_execution_state([const, const])

    # then set the execution states of related variables
    for stmt in inEdges:
        if stmt is CopyStmt:  # e.g. var = copy rhs
            set_stmt_rhs_execution_state([const, const])
        elif stmt is LoadStmt:  # e.g. var = load ptr
            for address in stmt_rhs_points_to_set:  # address is points to set of ptr
                set_address_execution_state([const, const])
```

说明

设置此函数的作用与`getCmpBranchStmt()`的作用类似，区别在于本函数用于处理分支条件变量并非由cmp指令直接设置的情形。

`SSE::getInEdgesES()`

伪代码

```
def getInEdgesES(basic_block):
    es = new execution state  # new execution state
    if basic_block is function entry node:
        # set es to the pre-configured function entry execution state
    else:
        for in_edge in basic_block_in_edges:
            in_icfg_edge = cfbg_edge2icfg_edge[in_dege] # get corresponding icfg edge
            if in_icfg_edge is a branch: #
                # take care of *branch condition* and get the execution state from
in_edge
                getBranchES(in_icfg_edge)
                # join es with in_edge's execution state
                es.joinwith(in_icfg_edge_execution state)
            else:
                # join es directly with in_edge's src node's execution state
                es.joinwith(src_node_execution_state)
    # set *current execution state* to be es
```

说明

该方法的作用就是merge前继节点的execution state作为当前节点的execution state，并将其设置为`SVFIR2ExeState`对象的当前ES值。具体而言，有以下根据当前节点是不是函数的Entry Node分为两种情况：

1. 如果输入的`CFBasicBlockNode`是某函数的Entry Node，那么由于在处理该函数的callsite时已经将当时的`ExeState`传递给了`handleFunc`方法，并且通过`entryNodeToES`存储了与该函数的Entry Node之间的对应关系，因此可以直接采用该ES作为处理这个输入`CFBasicBlockNode`时的当前ES。
2. 如果输入的`CFBasicBlockNode`是某函数的中间Node，那么直接由该Node的入边为Key，就可以通过`esMap`获取到其所有前继节点的处理完成时的ES，将这些ES进行join操作，即可作为输入Node的当前ES值。

值得注意的是，之所以Entry Node需要做这样的特殊处理，是因为SSE所采用的分析是context sensitive的，而对于函数的Entry Node而言，其入边可能来自不同的callsite，因此必须通过handleFunc进行传递，才能将当前call context的ES值传递给Entry Node。以下面的例子说明：

```
void foo(int x){
    //foo entry
}

int main(){
    foo(1);  // callsite1
    foo(2);  // callsite2
}
```

在处理callsite1时，ES中x的值为1，而在处理callsite2时，ES中x的值则为2。那么在处理到foo entry时，entry node的入边有两条，分别来自callsite1和callsite2，从而仅仅根据entry node的入边将无法分辨应当采用哪个callsite的ES。而当前的做法则可以解决这一问题，在处理foo函数时，将当前ES通过`handleFunc`方法传递，即可实现context sensitive的特性。

Defect Checker模块

bufferoverflowchecker.h/cpp

`BufferOverflowChecker`是SSE类型的子类，重写了`handleSVFStatement`方法和`handleCallSite`方法，并新增了`handleDynMemAlloc`方法。

`handleSVFStatement()`

`handleSVFStatement`方法总体上与父类中的该方法保持一致，不同之处在于：

1. 在处理`GepStmt`的时候，在完成该指令的符号执行以后，将额外调用`BufferOverFlowCheckerAPI`中的`checkOffsetValid`方法，并将`GepStmt`所对应的`SVFInstruction`作为第一个参数传递，对该指令所期望获取的内存对象的地址进行溢出检测。
2. 在处理`CallICFGNode`的时候，检测其中的外部函数调用，如`memcpy`等与内存操作有关的外部函数，并且调用`BufferOverFlowCheckerAPI`中的`handleExtAPI`方法来对外部函数进行溢出检测。

`handleCallSite()`

伪代码

```
def handleCallSite(icfg_node):
    callee = get_callee_of_callICFGNode(icfg_node)
    if callee is NOT nullptr: # meaning is not calling by function pointer
        if func has CFBasicBlockGraph:  # meaning this func's implementation is
available
            if func is recursive_function:
                handleRecursiveCall(icfg_node)  # call handleRecursiveCall to handle
icfg_node
            handleFunc(callee, current_execution_state)
        elif get_func_name(func) is "svf_assert()":
            # add icfg_node to checkpoint
        else: # calling external function's that doesn't has implementation, e.g.,
memcpy()
            try:
                handle_external_function(func)
            except:
                # report bug
     else: # meaning is calling by function pointer
        # get function pointer's points-to set
        pts_functions = get_pts(function_pointer)
        handleFunc(pts_functions[0]) # pick out the first function and handle it
```

说明

总体上与父类保持一致，不同之处在于：

1. 增加了对递归函数的特殊处理
2. 在调用`handle_external_function()`处理外部函数时，增加了exception的处理逻辑。

`handleDynMemAlloc(const AddrStmt *addr): void`

说明

`handleDynMemAlloc`方法用于在动态分配内存空间时，为`MemObj`对象设置其数组元素个数。例如，在处理语句`%1 = alloca i32, %2`时，由于`%2`的值不能从指令中立即获取，因此需要在进行符号执行得到`%2`的可能值后，再另行设置`MemObj`对象的元素个数，通过调用此方法即可完成上述操作。

sseapi.cpp/h

`SSEAPI`类型为SSE中进行符号执行提供公共的utils函数。说明：

getArrayByteSize(u32_t baseExpr): std::vector<u32_t>

`getArrayByteSize`方法的参数是`SVFVarId`，其作用是获取ES中，该`SVFVar`可能指向的`ObjVar`，最终获取该ObjVar的字节大小。例如，对于`int a[10]`以及`int a = (int *)malloc(sizeof(int) * 10)`语句声明的栈数组和堆数组，获取到的数组字节大小是 $4 \times 10 = 40$ Bytes。

Note: 需要注意的是，目前该方法仅适用于获取简单数据结构的列表元素个数，对于双重数组、自定义数据结构等获取到的数组字节大小不精确。

getArrayElemByteSize(u32_t base_addr): std::vector<u32_t>

`getArrayElemByteSize`方法的参数是`SVFVar`ID，其作用是获取ES中，该`SVFVar`可能指向的`ObjVar`，最终获取`ObjVar`所表示的数组元素的字节大小。例如，对于`int a[10]`以及`int a = (int *)malloc(sizeof(int) * 10)`语句声明的栈数组和堆数组，获取到的元素字节大小是4 Bytes。

Note: 需要注意的是，目前该方法仅适用于获取简单数据结构的列表元素个数，对于双重数组、自定义数据结构等获取到的元素字节大小不精确。

getArrayIndexSize(u32_t base_addr): std::vector<u32_t>

`getArrayIndexSize`方法的参数是`SVFVar`ID，其作用是获取ES中，该`SVFVar`可能指向的`ObjVar`，最终获取`ObjVar`所表示的数组的元素个数。例如，对于`int a[10]`以及`int a = (int *)malloc(sizeof(int) * 10)`语句声明的栈数组和堆数组，获取到的元素个数是10个。

Note: 需要注意的是，目前该方法仅适用于获取简单数据结构的列表元素个数，对于双重数组、自定义数据结构等获取到的元素个数不精确。

initFuncMap(): void

初始化SSE可以处理的外部函数`_func_map`，每个函数名对应一个用来处理该外部函数的SSE函数。

handleExtAPI(const CallICFGNode *call): void

通过`CallStmt`中所调用外部函数的函数名，在`_func_map`中查找处理该函数的SSE函数。若查找成功，则调用该SSE函数来处理该外部函数。

bufferoverflowcheckerapi.h/cpp

`BufferOverflowCheckerAPI`是`SSEAPI`的子类型，为BufferOverflowChecker中进行缓冲区溢出检测提供公共的utils函数。其中，`getAllocaBytes`方法的作用是获取alloc指令或者malloc分配的总字节大小；`getVarValidByteSize`方法的作用是获取某个SVFVar的剩余大小（总分配字节-偏移字节）；`getGepByteOffset`方法的作用是获取gep指令的字节偏移大小。使用上述的三个方法，即可对每一处访存进行检查。

`getAllocaBytes()`

伪代码

```
def getAllocaBytes(svf_instruction):
    if svf_instruction is alloca instruction: # meaning alloca instruction on stack
        type_size = get_llvm_type_size(svf_instruction) # e.g. alloc i32, %1 returns 4
bytes
        alloc_num = get_execution_state(svf_instruction) # e.g. alloc i32, %1 returns
upper bound of %1
        return alloc_num * type_size
    else: # meaning malloc on heap
        malloc_size = get_execution_state(svf_instruction)
        return malloc_size
```

说明

如果分配空间大小的execution state 并非const，会将其设置为默认大小（512）并返回这个大小。

`getVarValidByteSize()`

伪代码

```
def getVarValidByteSize(svf_value):
    worklist = []
    total_bytes = [0, 0]
    while(worklist is NOT empty):
        if svf_value is SVFInstruction:
            if get_icfg_node(svf_value) is CallICFGNode: # e.g. svf_value = call func
                worklist.add(phi_rhs_values) # phi rhs values,e.g., svf_value = phi
[value1, value2, value3]
            else:  # other situations e.g. copy, load, alloca and gep
                if svf_value is CopyInstruction:
                    worklist.add(copy_rhs)
                elif svf_value is LoadInstruction:
                    address_set = get_points_to_set(load_rhs)
                    for address in address_set:
                        worklist.add(getSVFValue(address))
                elif svf_value is GepInstruction:
                    worklist.add(gep_rhs) # add gep source to work_list
                    total_bytes -= getGepByteOffset(gep_stmt) # minus the offset on
total_bytes
                elif svf_value is AddrInstruction: #
                    alloc_bytes = getAllocByteSize(alloc_inst) # get the total
allocated byte size
                    total_bytes += alloc_bytes # plus the allocated byte size
                    return total_bytes
        elif svf_value is SVFGlobalValue:
            total_bytes += global_value_total_bytes
            return total_bytes
        elif svf_value is SVFArgument: # e.g. call parameter passing
```

```
            work_list.add(call_pe_rhs)
```

说明

该方法的作用是获取给定的SVFVar的剩余大小（总分配字节-偏移字节）。其原理基于以下的事实：某个内存对象从定义处（如alloca）到最终的使用处（如作为load指令的第一个操作数，或者`memcpy`调用的第一个参数），其ptr会经过若干次的`GepStmt`取偏移、`CallPE`，`RetPE`参数传递、以及`load`，`store`操作等等。而为了计算剩余大小，需要估算出三个信息：内存空间分配的总字节大小，经过若干次`GepStmt`所产生的**累计偏移大小**。该方法的算法原理就是从SVFVar开始，借助指针分析的信息，以及利用SSA的优势，逐步追溯整个指针的来源链条，最后到达所指向对象的分配处（Gloabl或者alloc指令），在这个过程中计算偏移大小。

`getGepByteOffset(const GepStmt* gep): IntervalValue`

输入`GepStmt`，获取并返回该`GepStmt`的字节偏移增量。例如，对于指令`%3 = getelementptr inbounds [10 x i32], [10 x i32]* %1, i32 0, i32 %2`，可以通过程序在该点处的`InvervalExeState`，获取`%2`的`IntervalValue`（假设为 $[1.0, 2.0]$ ），从而进一步推知该指令的字节偏移量为：

$$0 \times sizeof([10 \times i32]) + [1.0, 2.0] \times sizeof(i32) = [4.0, 8.0]$$

`checkOffsetValid(const SVFValue* value, const IntervalValue& len): bool`

说明

该方法与`getVarValidByteSize`大部分一致，区别在于，该方法将计算得到的objects剩余大小与传递进来的偏移访问大小进行比较，判定是否出现溢出访问。该方法的作用是：检查对指针所指的内存对象进行大小为`len`的内存访问是否超出边界。其第二个参数`len`表示以该`GepStmt`所取的内存空间为访问地址，所访问的内存空间的字节大小，如`*(T *)ptr`所访问的内存大小就是`sizeof(T)`。

判断内存访问是否溢出，需要估算出三个信息：内存空间分配的总字节大小，经过若干次`GepStmt`所产生的**累计偏移大小**，以及准备访问的字节大小。准备访问的字节大小可以直接从访问现场的指令获得，例如访问现场`getelementptr inbounds [10 x i32], [10 x i32]* %1, i32 0, i32 1`的字节偏移大小的计算式为 $0 \times sizeof([10 \times i32]) + 1 \times sizeof(i32) = 4$。而内存空间分配的总字节大小以及经过`GepStmt`所产生的**累计偏移大小**则需要通过更加复杂的计算过程获取。以前述的Gep指令为例，其中指针`%1`可能的来源如下：

LLVM IR	说明
`call void foo(%3)`	parameter passing
`%1 = getelementptr inbounds [10 x [10 x i32]], [10 x [10 x i32]]* %i, i32 0, i32 2`	multiple offsets by gep
`%1 = load i32*, i32** %3`	pointer type load stores
`%1 = bitcast i8* %3 to i32*`	type cast

上述算法记录的`vardefs`信息能够被`checkOffsetValid`方法利用，从而计算出内存空间分配的总字节大小，以及经过若干次`GepStmt`所产生的**累计偏移大小**。计算算法在`checkOffsetValid`方法中实现，其伪代码如下：

```
# Psudo code: checking algorithm
def checkOffsetValid(pointer, offset):
    worklist = []
    total_bytes = [0, 0]
    while(worklist is NOT empty):
        if svf_value is SVFInstruction:
            if get_icfg_node(svf_value) is CallICFGNode: # e.g. svf_value = call func
                worklist.add(phi_rhs_values) # phi rhs values,e.g., svf_value = phi
[value1, value2, value3]
            else:  # other situations e.g. copy, load, alloca and gep
                if svf_value is CopyInstruction:
                    worklist.add(copy_rhs)
                elif svf_value is LoadInstruction:
                    address_set = get_points_to_set(load_rhs)
                    for address in address_set:
                        worklist.add(getSVFValue(address))
                elif svf_value is GepInstruction:
                    worklist.add(gep_rhs) # add gep source to work_list
                    total_bytes -= getGepByteOffset(gep_stmt) # minus the offset on
total_bytes
                elif svf_value is AddrInstruction: #
                    alloc_bytes = getAllocByteSize(alloc_inst) # get the total
allocated byte size
                    total_bytes += alloc_bytes # plus the allocated byte size


        elif svf_value is SVFGlobalValue:
            total_bytes += global_value_total_bytes
            return total_bytes
        elif svf_value is SVFArgument: # e.g. call parameter passing
            work_list.add(call_pe_rhs)
```

Example：以下面的程序为例，解释`checkOffsetValid`方法的详细过程：

```
void foo(char *charArr){
    charArr[10] = 'x';  // 第二处
    charArr[40] = 'x';
}

int main(){
    int a[10];
    foo((char *)(a+1));
}
```

该程序翻译为LLVM IR为（clang -O1）：

```
define dso_local void @_Z3fooPc(i8* nocapture) local_unnamed_addr #0 {
  %2 = getelementptr inbounds i8, i8* %0, i32 10
  store i8 120, i8* %2, align 1, !tbaa !4
  %3 = getelementptr inbounds i8, i8* %0, i32 40
  store i8 120, i8* %3, align 1, !tbaa !4
  ret void
}

define dso_local i32 @main() local_unnamed_addr #1 {
  %1 = alloca [10 x i32], align 4
  %2 = bitcast [10 x i32]* %1 to i8*
  call void @llvm.lifetime.start.p0i8(i64 40, i8* nonnull %2) #3
  %3 = getelementptr inbounds [10 x i32], [10 x i32]* %1, i32 0, i32 1
  %4 = bitcast i32* %3 to i8*
  call void @_Z3fooPc(i8* nonnull %4)
  call void @llvm.lifetime.end.p0i8(i64 40, i8* nonnull %2) #3
  ret i32 0
}
```

`%1`所指向的，即分配的`int a[10]`的内存空间首地址，该内存空间大小为40 Bytes。SSE对`main`和`foo`函数中三处`getelementptr`指令进行缓冲区溢出检测。

现以第二处`gep`指令为例，说明`checkOffsetValid`方法进行检测的具体步骤：

1. 调用`checkOffsetValid`，并传递`GepStmt`和`IntervalVal`(1,1)，进入到checkOffsetValid方法，total_offset初始化为(1, 1)。
2. 开始处理语句`%2 = getelementptr inbounds i8, i8* %0, i32 10`，获取`%0`的定义处，得到`CallPE`语句，从而直接跳转到`CallPE`的RHS即`%4`。
3. 获取`%4`的定义处，得到`%4 = bitcast i32* %3 to i8*`，直接跳转到`%3`。
4. 获取`%3`的定义处，得到`%3 = getelementptr inbounds [10 x i32], [10 x i32]* %1, i32 0, i32 1`，跳转到%1，并且累加字节偏移量4到total_offset。
5. 获取`%1`的定义处，得到`%2 = bitcast [10 x i32]* %1 to i8*`，直接跳转到`%1`。
6. 获取`%1`的定义出，得到`%1 = alloca [10 x i32], align 4`，该指令为内存对象的定义指令，因此可以获取到内存对象的字节大小，即40 Bytes。然后调用`getGepByteOffset`指令，获取字节偏移量`%2 = getelementptr inbounds i8, i8* %0, i32 10`，为(10, 10)，累加到total_offset上，最终得到total_offset为(15, 15)。通过`15>40`为假可以判断，此处不存在缓冲区溢出。

`bitcaseChecker()`

说明

该方法对bitcase指令进行check，用一下的例子说明：

```
bitcast %1, struct A
```

该方法首先使用`getVarValidByteSize`获取`%1`所指向对象的剩余空间，然后直接从指令中获取`struct A`的大小信息，如果大小超过了剩余空间，则报告溢出错误。

`initFuncMap(): void`

初始化BufferOverflowChecker可以处理的外部函数_func_map，每个函数名对应一个用来处理该外部函数的BufferOverflowChecker函数。

`handle_ioapi()`

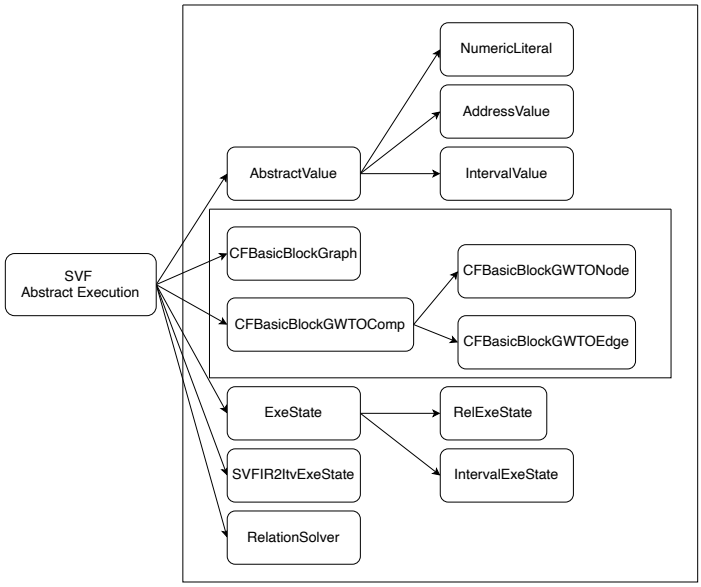
处理诸如`rand()`函数。（未启用）

`initUserInput()`

使用户输入变成top值。（未启用）

SVF相关API简介

Note: 本文档内容关于`SVF`项目中`Abstract Execution`模块和基本块`WTO`顺序构建模块的API及其使用。模块的头文件在`SVF/svf/include/AbstractExecution`与`SVF/svf/include/Graph`目录下，模块的源文件在`SVF/svf/lib/AbstractExecution`与`SVF/svf/lib/Graph`目录下。类型及继承关系如图：



各部分对应SVF lib文件

部分	SVF lib 文件
Part 1	AbstractExection目录：AbstractValue.h/cpp, AddressValue.h/cpp
Part 2	AbstractExection目录：IntervalExeState.h/cpp, ExeState.h/cpp
Part 3	AbstractExection目录：WTO.h/cpp
Part 4	AbstractExection目录：SCFIR2ItvExeState.h/cpp
Part 5	Graphs目录：CFBasicBlockG.h/cpp

Part 1

AbstractValue

`AbstractValue`类型表示符号执行过程中，某变量`SVFVar`的可行域。可行域可以为实数域、区间域，也可以为虚拟地址，分别对应于`AbstractValue`的子类型`NumericLiteral`，`IntervalValue`和`AddressValue`。

NumericLiteral

`NumericLiteral`类型表示实数域，在其内部使用双精度浮点数表示实数。值得注意的是，在SSE中，并不直接使用`NumericLiteral`来表示`SVFVar`/`ObjVar`，而使用一个`IntervalValue`表示SVF变量的状态。

方法声明	返回类型	功能
getNumeral() const	double	获取该对象表示的实数值

AddressValue

`AddressValue`类型表示指针指向的地址集合。由于一个指针可能指向多个地址，因此该类型为可迭代类型，提供iterator对集合内的地址值进行访问。

方法声明	返回类型	功能
getInternalID(u32_t idx)	u32_t	输入AddressValue中的一个虚拟地址值，返回该虚拟地址在SVFIR中对应的`ObjVar`Id

IntervalValue

`IntervalValue`类型表示区间域，如 $[1.0, 2.0]$，其中用两个`NumericLiteral`成员分别表示区间的下界和上界。在SSE中，使用一个`IntervalValue`表示SVF变量的状态，例如在执行完`x = 1`语句后，x所对应的变量可以使用一个上、下界均为 1.0 的`IntervalValue`类型对象表示，即表示 $x \in [1.0, 1.0]$。

方法声明	返回类型	功能
lb() const	NumericLiteral &	获取区间下界
ub() const	NumericLiteral &	获取区间上界

Part 2

ExeState

`ExeState`类型表示程序在某个程序点处的符号执行状态，如`SVFVar`的`Vaddrs`等信息。

方法声明	返回类型	功能
&operator=(const ExeState &rhs)	ExeState	判断两个执行状态是否相同
joinWith(const ExeState &other)	void	
meetWith(const ExeState &other)	void	
getVAddrs(u32_t id)	VAddrs &	输入SVFVar ID，返回该SVFVar的虚拟地址值

IntervalExeState

`InvtervalExeState`是`ExeState`类型的子类型，表示程序某变量在某程序点处的间隔符号执行状态，比如`SVFVar`或者`ObjVar`的`IntervalValue`等信息。例如，在执行完语句`x = 1`之后，会将变量x所对应`ObjVar`的执行状态变成$[1.0, 1.0]$，其对应关系保存在`IntervalExeState`的`Map _locToItvVal`中。

方法声明	返回类型	功能
operator>=(const IntervalExeState &rhs)	bool	判断两个IntervalExeState之间的关系
bottom()	IntervalExeState	
top()	IntervalExeState	
has_bottom()	bool	

Part 3

CFBasicBlockGWTO

`CFBasicBlockGWTO`类型表示`CFBasicBlockGraph`（即各个函数的控制流基本块图）的**弱拓扑序**。该类型为线性结构的容器类型，容器元素的类型为`CFBasicBlockWTOComp`，`CFBasicBlockGWTO`提供itorator以实现对容器内`CFBasicBlockGWTOComp`的访问。

CFBasicBlockGWTOComp

`CFBasicBlockGWTOComp`类型是`CFBasicBlockGWTONode`和`CFBasicBlockGWTOEdge`类型的基类型，用以表示`CFBasicBlockGWTO`容器元素。其中，comp是component的缩写。

CFBasicBlockGWTONode

`CFBasicBlockGWTONode`类型表示控制流基本块图中的普通节点（即不在循环中的节点），每个该类型的对象对应一个`CFBasicBlockNode`类型的对象。

方法声明	返回类型	功能
node() const	CFBasicBlockNode *	获取该对象对应的一个`CFBasicBlockNode`对象

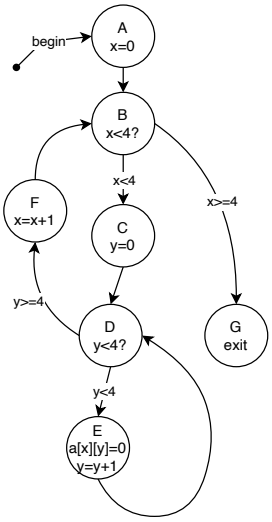
CFBasicBlockGWTOCycle

`CFBasicBlockGWTOCycle`类型表示控制流基本块图中的循环，每个该类型对象对应于图中的一个自然循环，因此包含一个`CFBasicBlockWTOComp`类型的列表成员，以对循环做线性化表示。此外，该类型还包含一个`CFBasicBlockNode *`类型成员指向循环的head（区别于前述列表的第一个元素），通过该类型的`head`方法可获取该成员。`CFBasicBlockGWTOCycle`为容器类型，提供iterator以实现对循环线性化列表中`CFBasicBlockWTOComp`元素的顺序访问。

以一个双重循环为例，说明该类型如何表示循环，程序代码如下：

```
int main() {
    int a[3][4];
    for(int x = 0; x < 4; x++) {
        for(int y = 0; y < 4; y++) {
            a[x][y] = 0; //TP, x index too large
        }
    }
}
```

上述程序可以表示为下图中的控制流基本块图：



若将上述程序中的外层循环和内层循环分别表示为`CFBasicBlockGWTOCycle`类型的cycle1和cycle2。则对于cycle1而言，表示基本块B的`CFBasicBlockGWTONode`对象即为该循环的head。cycle1的基本块表内容为：$(C, cycle2, F)$。在基本块B中，往往包含一个分支语句，用于控制循环终止条件。

Part 4

SVFIR2ExeState

`SVFIR2ExeState`类型用于维护符号执行状态，其中包含两个符号执行状态，分别为`RelExeState`和`IntervalExeState`类型。该类型提供`translatexx`方法用于输入一条语句，并根据规则更新符号执行状态。

方法声明	返回类型	功能
setEs(const IntervalExeState &es)	void	将某个es设置为当前需要处理的es，后续所有的translatexx操作都将基于该es进行
getEs()	IntervalExeState &	获取当前的es值
translatexx(SVFStatement *)	void	基于当前的es内容，执行该SVFStatement，并更新es
getInternalID(u32_t idx)	u32_t	输入AddressValue中的一个虚拟地址值，返回该虚拟地址在SVFIR中对应的`ObjVar`Id

Part 5

Note: 本部分内容的头文件和源码分别位于`SVF/svf/include/Graph`和`SVF/svf/lib/Graph`目录下。

CFBasicBlockGraph

`CFBasicBlockGraph`类型表示某个函数的控制流基本块图，根据该函数的控制流图（CFG，即ICFG中表示该函数的子图）所构建。控制流基本块图的节点用`CFBasicBlockNode`类型表示，边用`CFBasicBlockEdge`类型表示。

CFBasicBlockNode

表示控制流基本块图中的一个基本块，例如在上述的例子中，基本块E就对应一个`CFBasicBlockNode`，该`CFBasicBlockNode`包含两个`ICFGNode`。`CFBasicBlockNode`类型为容器类型，提供iterator对基本块中包含的`ICFGNode`进行顺序访问。

CFBasicBlockEdge

表示控制流基本块图中基本块之间的边。