Data Flow Analysis of Executable Files

M.Tech. Seminar Report

by

Harshada A. Gune

Roll No: 07305904

under the guidance of

Prof. Uday Khedker

Dept. of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Contents

Abstract

In recent few years, there has been increasing need for tools to find bugs and security vulnerabilities. But most of the efforts focused on analysis of source code rathar than analyzing excutables. In the security context, this is particulary unfortunate, because performing source code analysis may not explore certain vulnerabilities dur to the WYSINWYX phenomenon:"**W**hat **Y**ou **S**ee **I**s **N**ot **W**hat **Y**ou e**X**ecute". That is, there may be some mismatch between source code of a program and actual execution that takes place.

The report aims at presenting an algorithm that analyzes the executables. The main obstacle in analyzing the executables is lack of variable like entities. The precision of an analysis is dependent on granularity of recovered variable-like entities. Techniques to find out such variable-like entities are presented which is then followed by value set analysis to analyze the executables.

1 Introduction

There is an increasing need for tools to help programmers and security analysts understand executables. In the past few years, there has been a considerable amount of research activity to develop analysis tools to find bugs and security vulnerabilities. However, most of the effort had been on analysis of source code, and the issue of analyzing executables has largely been ignored. In the security context, this is particularly unfortunate, because performing analysis on the source code can fail to detect certain vulnerabilities because of the WYSINWYX phenomenon: [3] "What You See Is Not What You eXecute". That is, there can be a mismatch between what a programmer intends and what is actually executed on the processor. The following source-code fragment [3], taken from a login program, is an example of such a mismatch:

```
memset(password, '\0', len);
free(password);
```

The login program temporarily stores the user's password in clear text in a dynamically allocated buffer pointed to by the pointer variable password. To minimize the lifetime of the password, which is sensitive information, the code fragment shown above zeroes-out the buffer pointed to by password before returning it to the heap pointed to by password before returning it to the heap. Unfortunately, a compiler that performs useless-code elimination may reason that the program never uses the values written by the call on memset and therefore the call on memset can be removed, thereby leaving sensitive information exposed in the heap. This vulnerability is invisible in the source code; it can only be detected by examining the low-level code emitted by the optimizing compiler.

The remainder of the report is organized as follows: section 1.1 mentions the advantages of analyzing excutables. Section 1.2 describes the obstacles in analyzing the executables. Section 2 describes the abstract memory model for executables and concept of a-locs. Section 3 descirbes value-sets and VSA algorithm for intraprocedural analysis as well as for interprocedural analysis. Section 4 provides an overview of ASI algorithm.

1.1 Advantages of analyzing executables

The example presented earlier showed that an optimizer can cause there to be a mismatch between what a programmer intends and what is actually executed by the processor. In many case a substantial amount of such information is hidden from analyses that are based on source code, which can cause bugs, security vulnerabilities, and malicious behavior to be invisible to such analyses. There are a number of reasons [5] why analyses based on source code do not provide the right level of detail for checking certain kinds of properties:

- Programs typically make extensive use of libraries, including dynamically linked libraries (DLLs), which may not be available in source-code form. Typically, source-level analyses are performed using code stubs that model the effects of library calls. These stubs are likely to miss something which may produce some errors in analyses.

- Programs are sometimes modified subsequent to compilation, e.g., to perform optimizations.(They may also be modified to insert malicious code.) Such modifications are not visible to tools that analyze source.

- Source-level tools are only applicable when source is available, which limits their usefulness in security applications (e.g., to analyzing code from open-source projects).

- The source code may have been written in more than one language. This complicates the life of designers of tools that analyze source code because multiple languages must be supported by the tools.

1.2 Challenges in analyzing executables

To solve the IR-recovery problem, there are numerous obstacles that must be overcome, many of which stem from the fact that a program's data objects are not easily identifiable.

- No Debugging/Symbol Table Information : Debugging or Symbol table information gives us information about contents of data items at program points. This information can be used to find out data dependencies in program. For many kinds of potentially malicious programs (including most COTS products, viruses,and worms), debugging information is entirely absent; for such situations, an alternative source of information about variable-like entities is needed.

- Lack of Variable like entities : When performing source-code analysis, programmer-defined variables provide us with a convenient handle for specifying how a program manipulates its data. For example, a data dependence from statement a to statement b -which represents the fact that a defines some variable x, b uses x, and there is an x-def-free path from a to b. This is what finally required in any analysis. In other words, to perform any kind of analysis of a program we need to now about how data is manipulated. However in executables memory is accessed by specifying absolute addresses directly or indirectly through address expressions of the form [base + index * scale +offset]. That is, in executables there is no concept of variables as such. Data is just laid on continuous memory space and it is referred using absolute addresses only. Because, executables do not have intrinsic entities that are similar to variables that can be used in analysis of program, first crucial step is to identify variable like entities.

- Lack of Information about Structure of Heap Allocated Data : When performing source-code analysis, the structure of heap-allocated objects can be determined to an extent by looking at the types of pointers that point to the block of memory allocated in the heap. However, in executables, unless we have symbol-table or debugging information, only the size of the allocated block is known. Without knowing the structure of the heap-allocated block, little useful information can be obtained about the heap. Therefore, it is desirable to recover information about the structure of heap-allocated data.

- Lack of Information about Register Indirect Jumps : Control flow information is missed due to lack of information about the contents of register. That is targets of register indirect jump instructions can't be determined.

Hence we need some abstraction of data in which memory locations are referred symbolically. Therefore building abstract memory model is necessary before analyzing the executables. This abstract memory model is described in next section.

2 An Abstract Memory Model

In executables, memory is accessed either directly-by specifying an absolute address-or indirectly-through an address expression of the form [base + index * scale + offset], where base and index are registers, and scale and offset are integer constants. It is not clear from such expressions what the natural compartments are that should be used for analysis. Because executables do not have intrinsic entities that can be used for analysis (analogous to source-level variables), a crucial step in the analysis of executables is to identify variable-like entities. In this section, an abstract memory model is presented for analyzing executables [3].

2.1 Memory Regions

Although in concrete semantics the activation records for procedures, the heap and the memory area for global data are all part of *one address space*, for the purpose of analysis, address space is separated into a set of disjoint areas, which are referred as memory regions. Each memory-region represents a group of locations that have similar runtime properties: in particular, the runtime locations that belong to the ARs of a given procedure belong to one memory region. Each (abstract) byte in a memory-region represents a set of concrete memory locations. For a given program, there are three kinds of regions:

1. *global*-regions, for memory locations that hold initialized and uninitialized global data
2. *AR*-regions, each of which contains the locations of the ARs of a particular procedure
3. *malloc*-regions, each of which contains the locations allocated at a particular malloc site

Nothing is assumed about the relative positions of these memory-regions. (See Figure 1)

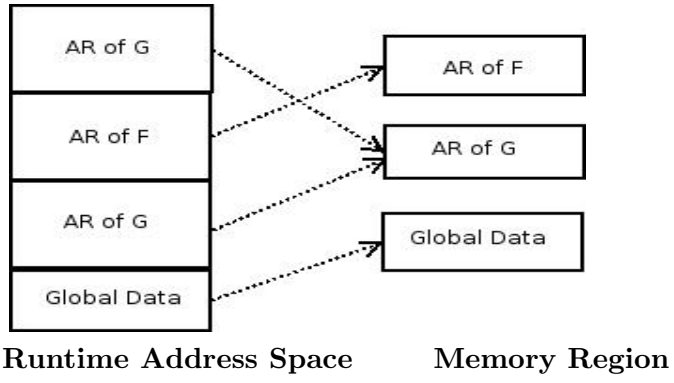


Figure 1: Abstract Memory Model [3]

The analysis treats all data objects, whether local, global, or in the heap, in a fashion similar to the way compilers arrange to access variables in local ARs, namely, via an offset. That is, while doing analysis an abstract memory address is represented by a pair: (memory-region, offset). By convention, esp is the stack pointer in the x86 architecture. On entry to a procedure P, esp points to the top of the stack, where the new activation record for P is

```
int global1 = 1;
int global2 = 2;
void main()
{
  int x,y;
  x = global1;
  y = global2;
  return ;
}
```

```
  proc main
1. mov ebp,esp
2. sub esp,8
3. mov [ebp-8],[4]
4. mov eax,ebp
5. mov [eax-4],[8]
6.add esp,8
7 return
```

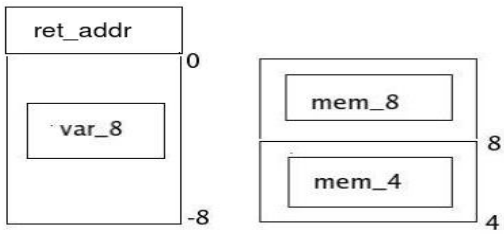


Figure 2: Program 1 [2] and its memory regions

```
typedef struct {
   int x,y;
}Point;

int a = 1, b = 2;
int main()
{
  int i,*py,
  Point pts[5], *p;
  py = &pts[0].y;
  p = &pts[0];
  for(i = 0; i <  5; ++i)
  {
    p->x = a;
    p->y = b;
    p += 8;
  }
  return *py;
}
```

```
     proc main
1    sub esp,44          .. Allocate locals
2    lea eax, [esp+8]    .. t1 = &pts[0].y
3    mov [esp+0], eax    .. py =  t1
4    mov ebx,  [4]       .. ebx = a
5    mov ecx,  [8]       .. ecx = b
6    mov edx, 0          ..i = 0
7    lea eax,[esp+4]     ..p = &pts[0]
L1:  mov [eax],ebx       ..p->x = a
8    mov [eax+4],ecx     ..p->y = b
9    add eax,8           ..p += 8
10   inc edx             ..i++
11   cmp edx, 5
12   jl L1               ..(i < 5)?L1:exit loop
13   mov edi, [esp+0]    ..t2 = py
14   mov eax, [edi]      ..set return value (*t2)
15   add esp, 44         ..Deallocate locals
16   retn
```

Figure 3: Program 2 [3]

created. Therefore, in this abstract memory model, esp holds abstract address $(AR\_P, 0)$ on entry to procedure P, where AR_P is the activation-record region associated with procedure P. Similarly, because malloc returns the starting address of an allocated block, the return value for malloc (if allocation is successful) is the abstract address $(Malloc\_n, 0)$, where Malloc_n is the memory-region associated with the call-site on malloc.

Example 1 Figure 2 shows the memory region for Program 1. There is a single procedure, hence two regions: one for global data and one for the AR of main. Furthermore, the abstract address of global variable global1 is (Global, 4) because it is at offset 4 in the global region.

Example 2 Figure 4 shows the memory region for Program 2. Here also a single procedure, hence two regions: one for global data and one for the AR of main. Furthermore, the abstract address of local variable py is (AR_main, -44) because it is at offset -44 with respect to AR's creation point.

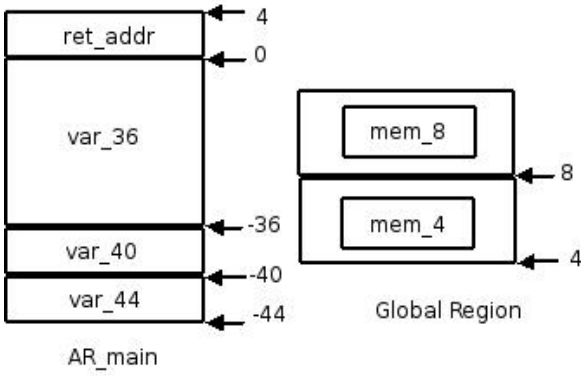


Figure 4: Memory Regions for Program 2

2.2 Abstract Locations

As pointed out earlier, executables do not have intrinsic entities like source-code variables that can be used for analysis; therefore, the next step is to recover variable-like entities from the executable. Such variable-like entities are referred as a-locs (for "abstract locations"). IDAPro, a commercial disassembly toolkit is used for the purpose of recovering variable-like entities. IDAPro's algorithm is based on the observation that the data layout of the program is established before generating the executable; therefore, accesses to global variables appear as [absolute-address], and accesses to local variables appear as [esp + offset] or [ebp - offset] in the executable. IDAPro identifies such statically-known absolute addresses, esp-based offsets, and ebp-based offsets in the program, and treats the set of locations in between two such absolute addresses or offsets to be one a-loc. This algorithm is referred as the Semi-Naïve algorithm [3]. The intuition behind the algorithm is that wheen a variable of size n bytes is accessed, it is accessed on a boundry of n-bytes and can not be accessed across the n-byte boundry.
Example 1 Let's see the a-locs for the Program 1. They are shown in figure 2.

- Local a-locs: Local a-locs are determined on per procedure basis. In the assembly code of a program, there is only one ebp-based offset in instruction $mov[ebp-8],[4]$. This corresponds to offset -8 of activation record. Hence only one a-loc is identified here which is var_8 and its size is 8 bytes.

- Global a-locs: Instructions $mov[ebp-8],[4]$ and $mov[eax-4],[8]$ have direct memory operands, namely, [4] and [8]. IDAPro identifies these statically-known absolute addresses as the starting addresses of global a-locs and treats the locations between these addresses as one a-loc. Consequently, IDAPro identifies addresses 4:7 as one a-loc, and the addresses 8:11 as another a-loc. Therefore, we have two a-locs: mem_4 (for addresses 4:7) and mem_8 (for addresses 8:11).

- Heap a-locs: In addition to globals and locals, we have one a-loc per heap-region. There are no heap a-locs for Program 1 because it does not access the heap.

- Registers: In addition to the global, heap, and local a-locs, registers are also considered to be a-locs.

Example 2 a-locs for the Program 2 are shown in figure 4.

- Local a-locs: Instructions *lea eax*, $[esp+8]$, $mov[esp+0], eax$ and *lea eax*, $[esp+4]$, *mov edi*, $[esp+0]$ have esp based indirect operands. Value of esp at the start of procedure is

0 and it is -44 at each of the above instructions. Therefore, the memory locations that are refered by these instructions have offsets -36, -44 and -40 in AR_main. This gives rise to three sets of locations between these three offsets and hence three local a-locs: var_44, var_40, and var_36.

- Global a-locs: Instructions $mov[ebx], [4]$ and $mov[ecx], [8]$ have direct memory operands, namely, [4] and [8]. Therefore, we have two a-locs: mem_4 (for addresses 4:7) and mem_8 (for addresses 8:11)
- Heap a-locs: There are no heap a-locs for Program 1 because it does not access the heap.
- Registers: Registers are also considered to be a-locs.

Once the a-locs are identified, a mapping from a-locs to (region,offset,size)triples is maintained. Here region refers to the memory-region to which the a-loc belongs, off is the starting offset of the a-loc in rgn, and size is the size of the a-loc. This mapping is used to interpret memory-dereferencing operations as described in next chapter.

3 Value Set Analysis

As described already, one of the significant obstacles in analyzing executables is that it is very difficult to obtain useful information about how the program manipulates data in memory due to lack of variable like entities. This chapter describes the value-set analysis (VSA) [3] algorithm, which provides useful information about memory accesses in an executable. VSA is a combined numeric-analysis and pointer-analysis algorithm that determines a safe approximation of the set of numeric values or addresses that each register and a-loc holds at each program point. VSA determines an over-approximation of the set of addresses that a register or a memory location holds at each program point. A key feature of VSA is that it tracks integer-valued and address-valued quantities simultaneously. This is crucial for analyzing executables because numeric values and addresses are indistinguishable at runtime. VSA is based on abstract interpretation, where the aim is to determine the possible states that a program reaches during execution, but without actually running the program on specific inputs. Abstract-interpretation techniques explore the program's behaviour for all possible inputs and all possible states that the program can reach. To make this feasible, the program is run in the aggregate, i.e., on descriptors that represent collections of memory configurations. VSA is a flow-sensitive, context-sensitive, interprocedural algorithm that associates each a-loc (including registers) in the executable with an (abstract) set of memory addresses and numeric values.

3.1 Value Set

A value-set [3] represents a set of memory addresses and numeric values that an a-loc can hold. Value set is an r-tuple where r is the number of memory regions. That is for each a-loc there is some r-tuple at each program point. Each element of tuple represents the numbers that represent either the possible memory addresses or numeric values from the corresponding memory region that are held in a-loc.

For example, if there is single procedure then value set of any a-loc at any program point will be represented by some 2-tuple. Conventially first component of tuple holds addresses or numeric values from global region and second component holds values from activation region of corresponding single procedure. Let $a_1r_1, a_2r_1, ..., a_nr_1$ refers to addresses or integer values from region1 and $a_1r_2, a_2r2, ...a_nr2$ refers to addresses or integer values that are from

region2. Hence value set of any a-loc,say a, at any program point will be represetned as $a \mapsto (a_1r_1, a_2r_1, ...), (a_1r_2, a_2r2, ...)$.

Formally the listing of addresses and values in each component is represenetd by strided-interval, where k-bit strided interval s[l,u] represents a set of integers $\{i \mid l \leq i \leq u\}$ and such i's differ by stride of s. For example, strided interval $2[1, 9]$ in the 2-tuple $(2[1, 9], \perp)$ denotes the set of numeric values $\{1, 3, 5, 7, 9\}$ as well as the set of addresses $\{(Global, 1), (Global, 3), ..., (Global, 9)\}$. So value set will be nothing but an r-tuple of strided intervals, where r is the number of memory-regions for the executable.

The Value-Set Abstract Domain: Value-sets form a lattice with $\top$ of lattice representing any possible value of address or integer value and $\perp$ representing empty set. Some of the value set operators [3] are given below:.

- $(vs1 \sqsubseteq^{vs} vs2)$: Returns true if the value-set vs1 is a subset of vs2, false otherwise.
- $(vs1 \sqcap^{vs} vs2)$: Returns the meet (intersection) of value-sets vs1 and vs2.
- $(vs1 \sqcup^{vs} vs2)$: Returns the join (union) of value-sets vs1 and vs2.
- $(vs +^{vs} c)$: Returns the value-set obtained by adjusting all values in vs by the constant c, e.g., if $vs = (4, 4[4, 12])$ and c = 12, then $(vs +^{vs} c) = (16, 4[16, 24])$.
- $*(vs, s)$: Returns a pair of sets (F,P) where F and P are sets of a-locs. F represents set of "fully accessed" a-locs: a-locs that are of size s and whose starting addresses are in vs. P represents set of "partially accessed a-locs": it consists of (i)a-locs whose starting addresses are in vs but are not of size s, and (ii)a-locs whose addresses are in vs but whose starting addresses and sizes do not meet conditions to be in F.

3.2 Intraprocedural Analysis

This subsection describes an intraprocedural version of VSA. For the time being, consider programs that have no indirect jumps. Following kinds of instructions are discussed, where R1 and R2 are two registers of the same size, c, c1, and c2 are explicit integer constants, and $\leq$ and $\geq$ represent signed comparisons:

$$R1 = R2 + c \qquad\qquad R1 \leq c$$

$$*(R1 + c1) = R2 + c2 \qquad\qquad R1 \geq R2$$

$$R1 = *(R2 + c1) + c2$$

Almost all kinds of assembly instructions can be represented by using one of the above instructions. For example, instruction "mov R1, R2" can be represented by using $R1 = R2 + c$ where $c = 0$, while instruction "mov [R1-c], R2" can be represented by using $*(R1+c1) = R2+c2$ where $c1 = -c$ and $c2 = 0$. Some exceptions to this rule might be logical instructions. It is not clear how to deal with these instructions. Conditions of the two forms shown on the right are obtained from the instruction(s) that set condition codes used by branch instructions.

The analysis is performed on a control-flow graph (CFG) for the procedure. The CFG consists of one node per assembly instruction, and there is a directed edge n1 $\rightarrow$ n2 between a pair of nodes n1 and n2 in the CFG if there is a flow of control from n1 to n2. The edges are labeled with the instruction at the source of the edge. If the source of an edge is a branch instruction, then the edge is labeled according to the outcome of the branch. Each CFG has two special nodes: (1) an enter node that represents the entry point of the procedure, (2) an exit

node that represents the exit point of the procedure. Each edge in the CFG is associated with an abstract transformer that captures the semantics of the instruction represented by the CFG edge. Each abstract transformer takes input value-sets and returns new out value-sets. Sample abstract transformers [3] for various kinds of edges are listed in Table 1. Here 's' represents size of dereference performed by an instruction and [R] represents the value contained in location whose addreess is in R.

- Because each AR region of a procedure that may be called recursively-as well as each heap region-potentially represents more than one concrete data object, assignments to their a-locs must be modeled by weak updates, i.e., the new value-set must be joined with the existing one, rather than replacing it.

- Furthermore, unaligned writes can modify parts of various a-locs (which could possibly create forged addresses). In case 2 of Table 1, such writes are treated safely by setting the values of all partially modified a-locs to $\top^{vs}$. Similarly, case 3 treats a load of a potentially forged address as a load of $\top^{vs}$.

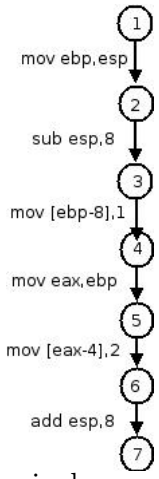
Instruction: R1 = R2 + c
Abstract Transformer:
Let $out := in$ and $vs_{R2} := in[R2]$
$out[R1] := vs_{R2} +^{vs} c$
return out
Description: Instruction copies $value(R2) + c$ to register R1.
That is $vs_{R2} +^{vs} c$ need to be copied to vs_{R1}
So first step is to determine vs_{R2} and $vs_{R2} := in[R2]$
Hence $vs_{R1} := vs_{R2} +^{vs} c$

Instruction: $*(R1 + c1) = R2 + c2$
Abstract Transformer:
Let $vs_{R1} := in[R1], vs_{R2} := in[R2], (F, P) = *(vs_{R1} +^{vs} c1, s)$ and out := in
$if(\mid F \mid = 1 and \mid P \mid = 0$ and (F has no heap a-locs or a-locs of recursive procedures)) then
    $out[v] := vs_{R2} +^{Vs} c2$, where $v \in F$
else
    for each $v \in F$ do
        $out[v] := out[v] \sqcup^{vs} (vs_{R2} +^{vs} c2)$
    end for
end if
for each $v \in P$ do
    $out[v] := \top^{vs}$
end for
return out
Description: Instruction moves the value (R2 + c2) to location [R1 + c1].
Hence the first step is to find out value set which represents value(R2 + c2).
This is obtained by expression $vs_{R2} +^{vs} c2$.
Now to move this value set to location [R1+c1], we need to find the a-locs
that represent this location. Operation (F,P) gives such a-locs.
$(F, P) = *(vs_{R1} +^{vs} c1, s)$. It returns two sets of a-locs: F and P.
F contains those a-locs which completely map the memory location [R1+c1]
(a-locs whose start address in $vs_{R1} +^{vs} c1$ and size = s),
while P contains the a-locs which partially map this location.
If there is only one such a-loc which completely maps the LHS location and
no partially mapped a-locs then set value set of such a-loc to $vs_{R2} +^{vs} c2$.
However if there are more than one such a-locs then we need to combine
value-sets of those a-locs.($\sqcup^{vs}$ operator)
In case of partially mapped a-locs, it is not possible to determine which portion
of a-loc will be overwritten and hence we set value set of such a-locs to $\top^{vs}$,
that is any possible value.

Instruction: $R1 = *(R2 + c1) + c2$
Abstract Transformer:
Let $vs_{R2} := in[R2], (F, P) = *(vs_{R2} +^{vs} c1, s)$ and out := in
$if(\mid P \mid = 0)$ then
    Let $vs_{rhs} := \sqcup^{vs} in[v] \mid v \in F$
    $out[R1] := vs_{rhs} +^{vs} c2$
else
        $out[R1] := \top vs$
end if
return out
Description:
Value [R2 + c1] + c2 is copied to register R1. Hence we need to find out value set of a-loc that represent the loction [R2 + c1] and adjust this value set by adding c2. So first we need to find out those a-locs that represent location [R2 + c1]. This is given by $(F, P) = *(vs_{R2} +^{vs} c1, s)$. So set F gives all those a-locs which are completely mapped and P gives partially mapped a-locs. If there are no partially mapped a-locs then we just combine value sets of all F a-locs and adjust it by adding c2. The resultant value set is nothing but newly computed value set for R1. $vs_{rhs} := \sqcup^{vs} in[v] \mid v \in F$
$out[R1] := vs_{rhs} +^{vs} c2$
If $\mid P \mid$ not equal to 0, this means some a-locs partially map with location whose start address = [R2 + c1] and size = s.
So it is not possible to determine which portion of such a-locs to be copied.
Hence we set value set of R1 to $\top vs$.

Table 1: Abstarct Transformers

Example 1
We will see how VSA is applied to Program 1 to get value sets at different program points. The very first step in algorithm is to find out the a-locs as described in section 2.2. So for Program 1 a-locs are shown in figure 2. We will now build a mapping from a-locs to triples(region, offset, size) as follows:



a-loc	(region, offset, size)
var_8	(AR_main,-8,8)

Once this mapping is done, we construct CFG for a given program. Above figure shows CFG for Program 1. Now VSA is applied to the edges one by one according to the edge transformers desribed in table1. This process is described below for each edge.

1. Edge 1-2 represents instruction := mov ebp, esp and $in := esp \mapsto (\perp, 0)$
   It can be represented by R1 = R2 + c where R1 = ebp; R2 = esp and c = 0.
   Now according to abstract transformer for R1 = R2 + c, out := in and
   $vs_{R1} := vs_{ebp} := vs_{R2} +^{vs} c$
   Hence, $vs_{ebp} := vs_{esp} +^{vs} 0 := esp \mapsto (\perp, 0) +^{vs} 0$
   Hence,$out := \{esp \mapsto (\perp, 0), ebp \mapsto (\perp, 0)\}$

2. Edge 2-3 represents instruction := sub esp, 8 and $in := \{esp \mapsto (\perp, 0), ebp \mapsto (\perp, 0)\}$
   It can be represented by R1 = R2 + c where R1 = esp; R2 = esp and c = -8.
   Now out := in and $vs_{R1} := vs_{esp} := vs_{R2} +^{vs} c$
   Hence, $vs_{esp} := vs_{esp} +^{vs} -8 := (\perp, 0) +^{vs} -8 := (\perp, -8)$
   $out := \{esp \mapsto (\perp, -8), ebp \mapsto (\perp, 0)\}$

3. Edge 3-4 represnts instruction mov[ebp-8], 1 and $in := \{esp \mapsto (\perp, -8), ebp \mapsto (\perp, 0)\}$
   It can be represented by $*(R1 + c1) = R2 + c2$ where R1 = ebp; R2 = dummyR ;
   c1 = -8; c2 = 1
   $vs_{R1} := vs_{[}ebp] := in[R1] := \{ebp \mapsto (\perp, 0)\}$
   $vs_{R2} := vs_{[}dummyR] := \{dummyR \mapsto (\top, \top)\}$
   $*(F, P) = *(vs_{R1} +^{vs} c1, s)$
   Hence, $*(F, P) = *((\perp, 0) +^{vs} -8, 4) = *((\perp, -8), 4)$
   Set F contains those a-locs whose start address in $(\perp, -8)$ and length = 4. That is, a-locs which are mapped exactly with a-loc on LHS.
   While set P contains those a-locs which are partially mapped with a-loc on LHS of instruction. Hence need to modify values contained in a-locs $\in$ F with value = $R2 + c2$
   Here $F = \oint$ and $P = var\_8$. Hence $out[var\_8] := \top$
   $out := \{esp \mapsto (\perp, -8), ebp \mapsto (\perp, 0), var\_8 \mapsto (\perp, \top)\}$

4. Edge 4-5 represents instruction mov eax, ebp and
   $in := \{esp \mapsto (\perp, -8), ebp \mapsto (\perp, 0), var\_8 \mapsto (\perp, \top)\}$
   It can be represented by $R1 := R2 + c$ where R1 = eax; R2 = ebp; c = 0
   out := in and $vs_{R1} := vs_{eax} := vs_{R2} +^{vs} c$
   Hence, $vs_{eax} := vs_{ebp} +^{vs} 0 := (\perp, 0) +^{vs} 0 := (\perp, 0)$
   $vs_{eax} := (\perp, 0)$
   $out := \{esp \mapsto (\perp, -8), ebp \mapsto (\perp, 0), var\_8 \mapsto (\perp, \top), eax \mapsto (\perp, 0)\}$

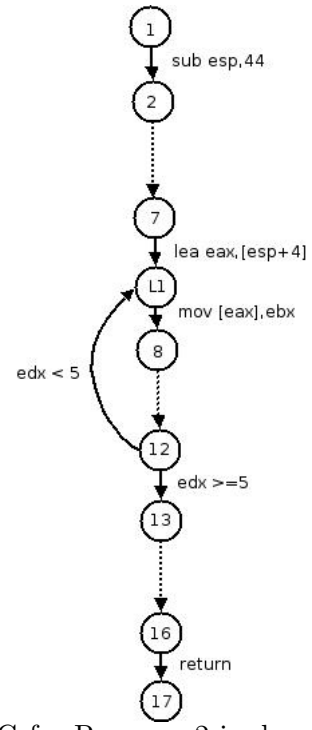
5. Edge 5-6 represnts instruction mov[eax-4], 2 and
   $in := \{esp \mapsto (\perp, -8), ebp \mapsto (\perp, 0), var\_8 \mapsto (\perp, \top), eax \mapsto (\perp, 0)\}$
   It can be represented by $*(R1 + c1) = R2 + c2$ where R1 = eax; R2 = dummyR;
   c1 = -4; c2 = 2
   $vs_{R1} := vs_{[}eax] := in[R1] := \{eax \mapsto (\perp, 0)\}$
   $vs_{R2} := vs_{[}dummyR] := \{dummyR \mapsto (\top, \top)\}$
   $*(F, P) = *(vs_{R1} +^{vs} c1, s)$
   Hence, $*(F, P) = *((\perp, 0) +^{vs} -4, 4) = *((\perp, -4), 4)$
   Set F contains those a-locs whose start address in $(\perp, -4)$ and length = 4.
   Set P contains a-locs which are partially mapped with a-loc on LHS of instruction. Hence need to modify values contained in a-locs $\in$ F with value = $R2 + c2$
   Here $F = \oint$ and $P = var\_8$. Hence $out[var\_8] := \top$
   $out := \{esp \mapsto (\perp, -8), ebp \mapsto (\perp, 0), var\_8 \mapsto (\perp, \top), eax \mapsto (\perp, 0)\}$

6. Edge 6-7 represents instruction add esp, 8 and
   $in := \{esp \mapsto (\perp, -8), ebp \mapsto (\perp, 0), var\_8 \mapsto (\perp, \top), eax \mapsto (\perp, 0)\}$

It can be represented by $R1 = R2 + c$ where R1 = esp; R2 = esp; c = 8
out := in and $vs_{R1} := vs_{esp} := vs_{R2} +^{vs} c$
Hence, $vs_{esp} := vs_{esp} +^{vs} 8 := (\perp, -8) +^{vs} -8 := (\perp, 0)$
$out := \{esp \mapsto (\perp, 0), ebp \mapsto (\perp, 0), eax \mapsto (\perp, 0), var\_8 \mapsto (\perp, \top)\}$

Example 2

Let's look at one more example where VSA is applied to Program 2 (figure 3) to get value sets at different program points. The a-locs for this program are shown in figure 4 and mapping from those a-locs to triples(region, offset, size) is as follows:



a-loc	(region, offset, size)
var_36	(AR_main,-36,36)
var_40	(AR_main,-40,4)
var_44	(AR_main,-44,4)
mem_8	(Global,8,4)
mem_4	(Global,4,4)

Partial CFG for Program 2 is shown above. It has not included all the edges but they are clear from the control flow of program. Now let's see the transformations along some edges. At the entry of a procedure we have $\{esp \mapsto (\perp, 0), mem\_4 \mapsto (1, \perp), mem\_8 \mapsto (2, \perp)\}$.

1. Edge 1-2 represents instruction sub esp, 44 and
   $in := \{esp \mapsto (\perp, 0), mem\_4 \mapsto (1, \perp), mem\_8 \mapsto (2, \perp)\}$
   It can be represented by $R1 = R2 + c$ where R1 = esp; R2 = esp and c = -44.
   $out := in$ and $vs_{R1} := vs_{esp} := vs_{R2} +^{vs} c$
   Hence, $vs_{esp} := vs_{esp} +^{vs} -44 := (\perp, 0) +^{vs} -44 := (\perp, -44)$
   $out := \{esp \mapsto (\perp, -44), mem\_4 \mapsto (1, \perp), mem\_8 \mapsto (2, \perp)\}$

2. Edge 2-3 represents instruction lea eax, esp+8, that is load eax with address = esp + 8 and $in := \{esp \mapsto (\perp, -44), mem\_4 \mapsto (1, \perp), mem\_8 \mapsto (2, \perp)\}$
   This can be represented by R1 = R2 + c where R1 = eax ; R2 = esp and c = 8.
   So out := in and $out[eax] = out[R1] = vs_{R2} +^{vs} c = vs_{esp} +^{vs} 8 = (\perp, -44) +^{vs} 8 = (\perp, -36)$
   Hence $out := \{esp \mapsto (\perp, -44), eax \mapsto (\perp, -36), (mem\_4 \mapsto 1), (mem\_8 \mapsto 2)\}$

3. Edge 3-4 represents instruction mov[esp+0], eax and
   $in := \{esp \mapsto (\perp, -44), eax \mapsto (\perp, -36), (mem\_4 \mapsto 1), (mem\_8 \mapsto 2)\}$
   This instruction can be represented by $*(R1 + c1) = R2 + c2$ where R1 = esp; c1 = 0;
   R2 = eax and c2 = 0

It moves the contents of eax to the location pointed to by expression $esp + 0$. That is we need to copy value set of eax to value sets of a-locs which represent the location $esp + 0$. Hence we need to find those a-locs which completely map the memory location at $esp + 0$ (memory locations with start address = esp+0 and size = 4) and then modify value-sets of those a-locs according to value set of eax. To find out comletely overlapping a-locs,
$*(F, P) = *(vs_{R1} +^{vs} c1, s) = *(vs_{esp} +^{vs} 0, 4) = *((\perp, -44) +^{vs} 0, 4) = *((\perp, -44), 4)$
Now we need to find such a-locs whose start address in $(\perp, -44)$ and whose size = 4. By looking at mapping table we find only var_44 satisfying this constraint and no a-loc satisfying partially mapping constraints. Hence $F = var\_44 and P = \oint$
Since $| F | = 1$ do strong update,
$out[var\_44] := vs_{R2} +^{vs} c2 := vs_{eax} +^{vs} 0 := (\perp, -36) +^{vs} 0 := (\perp, -36)$ Hence,
$out := \{esp \mapsto (\perp, -44), eax \mapsto (\perp, -36), var\_44 \mapsto (\perp, -36), (mem\_4 \mapsto 1), (mem\_8 \mapsto 2)\}$

4. Edge 4-5 represents instruction mov ebx, [4] and in is given by
$\{esp \mapsto (\perp, -44), eax \mapsto (\perp, -36), var\_44 \mapsto (\perp, -36), mem\_4 \mapsto (1, \perp), mem\_8 \mapsto (2, \perp)\}$
This instruction can be represented by $R1 = *(R2 + c1) + c2$ where R1 = ebx;
R2 = dummyR; c1 = 4 and c2 = 0.
Therefore, $vs_{R2} := in[R2]$ gives $vs_{dummR} := (\top, \top)$.
For the time being, ignore c2 in above instruction. Now the instruction is copying the contents at memory location (R2 + c1) to register R1. Then we need to find out value set of memory location (R2+c1) which should be then copied to value set of R1. To find out value set of memory location (R2+c1), we first need to find out which a-locs denote the memory location (R2+c1).
So let $*(F, P) = *(vs_{R2} +^{vs} c1, s) = *(vs_{dummyR} +^{vs} 4, 4) = *((\top, \top) +^{vs} 4, 4) = *((4, 4), 4)$
That is $*(F, P) = *((4, 4), 4)$ This operation will return the set F containing those a-locs whose start addresses in(4,4) and whose size = 4. By looking at mapping table we can say that $F = \{mem\_4\}$ and $P = \oint$ Since $| P | = 0, vs_{rhs} := in[mem\_4]$ As F contains only mem_4.
So $out[R1] := out[ebx] := vs_{rhs} +^{vs} c2 := in[mem\_4] +^{vs} 0 := in[mem\_4] := (1, \perp)$

$out := \{esp \mapsto (\perp, -44), eax \mapsto (\perp, -36), ebx \mapsto (1, \perp), var\_44 \mapsto (\perp, -36),$
$mem\_4 \mapsto (1, \perp), mem\_8 \mapsto (2, \perp)\}$

5. Continuing in a similar way, at the in of L1 we get:

$in := \{esp \mapsto (\perp, -44), eax \mapsto (\perp, -40), ebx \mapsto (1, \perp), ecx \mapsto (2, \perp),$
$edx \mapsto (0, \perp), var\_44 \mapsto (\perp, -36), mem\_4 \mapsto (1, \perp), mem\_8 \mapsto (2, \perp)\}$

Note here that, above value sets of eax and edx are the value-sets before completing single iteration. After the value-sets converge, final value-sets of eax and edx at node L1 are $(\perp, 8[-40, 0])$ and $(1[0, 4], \perp)$ respectively (value-sets that come along edge 12→ L1 are merged with value-sets along edge 7 → L1). Rest of the value-sets remain the same. To avoid complexity, just singleton values of eax and edx are considered in further calulations. Their correct value-sets are shown at the end of procedure.
Now edge L1-8 represents mov[eax],ebx which can be represented by instruction
$*(R1 + c1) = R2 + c2$ where R1 = eax; c1 = 0; R2 = ebx and c2 = 0.
$*(F, P) = *(vs_{R1} +^{vs} c1, s) = *(vs_{eax} +^{vs} 0, 4) = *((\perp, -40) +^{vs} 0, 4) = *((\perp, -40), 4)$
By looking at mapping table, $F = \{var\_40\}$ and $P = \oint$
Hence $out[var\_40] := vs_{R2} +^{vs} c2 := vs_{ebx} +^{vs} 0 := (1, \perp) +^{vs} 0 := (1, \perp)$
Here note that, as first component of value set tuple represents values in the global memory-region and second component represents values from activation region, var_40 contains

the value from global region and it does not contain any value from activation region of procedure(which is indicated by $\perp$ in second component)

$$out := \{esp \mapsto (\perp, -44), eax \mapsto (\perp, -40), ebx \mapsto (1, \perp), ecx \mapsto (2, \perp),$$
$$edx \mapsto (0, \perp), var\_44 \mapsto (\perp, -36), var\_40 \mapsto (1, \perp), mem\_4 \mapsto (1, \perp), mem\_8 \mapsto (2, \perp)\}$$

6. Edge 8-9 represents the instruction mov[eax+4], ecx which can be represented by $*(R1 + c1) = R2 + c2$ where R1 = eax; c1 = 4; R2 = ecx and c2 = 0. and

$$in := \{esp \mapsto (\perp, -44), eax \mapsto (\perp, -40), ebx \mapsto (1, \perp), ecx \mapsto (2, \perp),$$
$$edx \mapsto (0, \perp), var\_44 \mapsto (\perp, -36), var\_40 \mapsto (1, \perp) mem\_4 \mapsto (1, \perp), mem\_8 \mapsto (2, \perp)\}$$

$*(F, P) = *(vs_{R1} +^{vs} c1, s) = *(vs_{eax} +^{vs} 4, 4) = *((\perp, -40) +^{vs} 4, 4) = *((\perp, -36), 4)$
By looking at mapping table we find that $F = \oint$ and $P = \{var\_36\}$
Hence set $out[var\_36] := \top^{vs}$.

$$out := \{esp \mapsto (\perp, -44), eax \mapsto (\perp, -40), ebx \mapsto (1, \perp), ecx \mapsto (2, \perp), edx \mapsto (0, \perp),$$
$$var\_44 \mapsto (\perp, -36), var\_40 \mapsto (1, \perp), mem\_4 \mapsto (1, \perp), mem\_8 \mapsto (2, \perp), var\_36 \mapsto \top^{vs}\}$$

7. Edge 13-14 represent the instruction mov edi, [esp+0], that is contents at (esp+0) are copied into register edi. To do so we first need the contents i.e. value set of location [esp+0] which can be obtained from value set of a-loc representing location [esp+0]. *(F,P) operation gives such a-locs.

$$in := \{esp \mapsto (\perp, -44), eax \mapsto (\perp, -40), ebx \mapsto (1, \perp), ecx \mapsto (2, \perp), edx \mapsto (0, \perp),$$
$$var\_44 \mapsto (\perp, -36), var\_40 \mapsto (1, \perp), mem\_4 \mapsto (1, \perp), mem\_8 \mapsto (2, \perp), var\_36 \mapsto \top^{vs}\}$$

It can be represented by $R1 = *(R2 + c1) + c2$ where
R1 = edi; R2 = esp; c1 = 0 and c2 = 0.
Now $(F, P) = *(vs_{R2} +^{vs} c1, s) = *(vs_{esp} +^{vs} 0, 4) = *((\perp, -44) +^{vs} 0, 4)$
that is $(F, P) = *((\perp, -44), 4)$
From the mapping table we get $F = \{var\_44\}$ and $P = \oint$
That is var_44 maps the memory loction [esp + 0] and therefore its value set is copied into register edi.
$out[edi] := vs_{rhs} +^{vs} c2$ where $vs_{rhs} := in[var\_44]$
Therefore $out[edi] := in[var\_44] +^{vs} 0 := (\perp, -36) +^{vs} 0 := (\perp, -36)$

8. The edge 14-15 represents instruction mov eax, [edi] and can be represented by $R1 = *(R2 + c1) + c2$ where R1 = eax; R2 = edi; c1 = 0 and c2 = 0.
Applying the corresponding transformer, at the end of procedure we get following value sets for different a-locs:

$$out := \{esp \mapsto (\perp, -44), eax \mapsto (\perp, 8[-40, 2^{31} - 7]), ebx \mapsto (1, \perp), ecx \mapsto (2, \perp), edx \mapsto (5, \perp),$$
$$edi \mapsto (\perp, -36), var\_44 \mapsto (\perp, -36), var\_40 \mapsto (1, \perp), mem\_4 \mapsto (1, \perp), mem\_8 \mapsto (2, \perp),$$
$$var\_36 \mapsto \top^{vs}\}$$

9. Finally, edge 15 $\rightarrow$ 16 represents instruction add esp, 44 and can be represented by R1 = R2 + c. Applying the transformation, at the end of procedure we get following value-sets:

$$out := \{esp \mapsto (\perp, 0), eax \mapsto (\perp, 8[-40, 0]), ebx \mapsto (1, \perp), ecx \mapsto (2, \perp), edx \mapsto (5, \perp),$$
$$edi \mapsto (\perp, -36), var\_44 \mapsto (\perp, -36), var\_40 \mapsto (1, \perp), mem\_4 \mapsto (1, \perp), mem\_8 \mapsto (2, \perp),$$
$$var\_36 \mapsto \top^{vs}\}$$

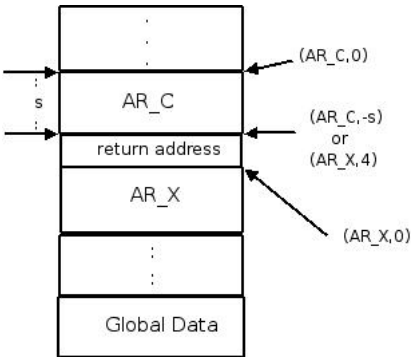


Figure 5: Observation for interprocedural analysis [3]

3.3 Interprocedural Analysis

Interprocedural analysis is carried out on super graph of executable. The algorithm for interprocedural analysis is similar to that for intraprocedural with two types of linkages added to the nodes used in intraprocedural analysis. Linkage edges:

- Call→Enter edge:
  For every call-site call P, an edge is added from the CFG node for call P to the enter node of procedure P.

- Exit→Endcall edge:
  For every procedure P, an edge is added from the exit node of P to the end-call node associated with every call to procedure P.
  See the CGF for Program 3 in figure7. The abstract transformers for non-linkage edges are same to those used for intraprocedural anlysis. Abstract transformers for linkage edges are given in section 3.3.1 and 3.3.2.

Observation for interprocedural analysis [3]
In abstract memory model, nothing was assumed about the relative positions of the memory regions. However, at a call, it is possible to establish the relative positions of the caller's AR-region (AR_C) and callee's AR-region (AR_X). Figure 5 illustrates this idea. At runtime, AR_C and AR_X overlap on the stack just before a call is executed. Specifically, the abstract address (AR_C,-s) in memory-region AR_C corresponds to the abstract address (AR_X,4) in memory region AR_X, where s = size of AR_C. This observation about the relative positions of AR_C and AR_X established at a call site is used while developing the abstract transformers for the linkage edges.
Actual parameters and register saves [1]
Stack operations like push/pop implicitly modify some locations in the AR of a procedure (say P). These locations correspond to the actual parameters of a call and to those used for register spilling and caller-saved registers. The locations accessed by push/pop instructions are not explicitly found as esp/ebp relative addresses, and so the algorithm that identifies a-locs will not introduce a-locs for the memory locations accessed by these stack operations; consequently, additional a-locs are introduced, which are called as extended a-locs, for memory locations that are implicitly accessed by such stack operations. To do this, the smallest sp delta for P is determined. This represents the maximum limit to which the stack can grow in a single invocation of P. If a finite minimum can't be found, the analysis issues a report. If there is a finite minimum, then extended a-locs are added to the AR on 4-byte boundaries to fill the space

between the lowest local a-loc and the minimum sp delta. Figure 8 shows the extended a-locs for procedure main and the formal parameters for procedure initArray for the Program 3.

3.3.1 Abstract Transformer for call → enter Edge

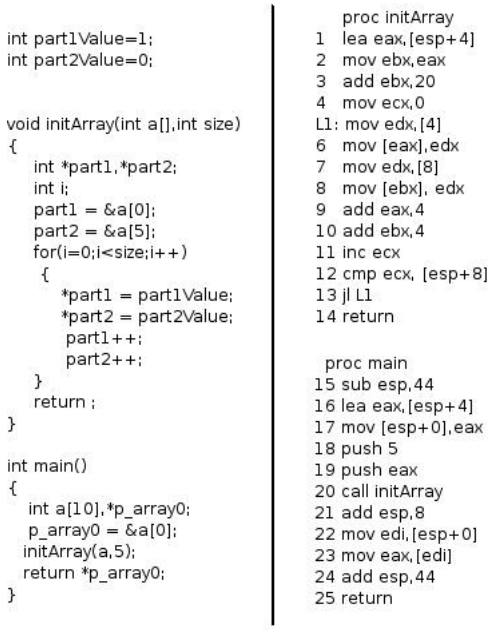
The transformer [3] takes the current value sets at the call node as an argument and returns new value sets for the call→enter edge.
1. First step is to set value set of esp to $(\perp, ..., 0, ..., \perp)$, where 0 occurs in the slot of AR_X. This step corresponds to changing current AR from that of AR_C to AR_X.(Line no. 3 in diagram)
2. After esp is initialized, for every a-loc a ∈ a-locs[AR_X], corresponding set of a-locs in AR_C is determined(line 7)and new value set for a (namely new_a) is computed from value sets of those a-locs. (lines 12 and 13) This step corresponds to copying the actual parameters of caller C to the formal parameters of callee X.
3. If procedure X is recursive, a weak update is performed rather than strong update.(lines 15 to 18).

1.Let C be the caller and X be the callee.
2.out := in
3.$out[esp] := (\perp, ...0, .. \perp)$
4.for each a-loc a ∈ a-locs$[AR_X]$ do
5. Let S_a be size of a-loc a.
6. // Find corresponding a-locs in AR_C as follows
7. $(F, P) := *(in[esp] +^{vs} offset(AR\_X, a), S_a)$
8. Set $new_a := \perp^{vs}$
9. if$(P \neq \oint)$ then // Partially accessed a-loc
10. $new_a := \top^{vs}$ // Any possible values
11. else
12. $vs_{actuals} := \sqcup \{in[v] \mid v \in F\}$
13. $new_a := vs_{actuals}$
14. end if
15. if X is recursive then
16. $out[a] := in[a] \sqcup^{vs} new_a$
17. else
18. $out[a] := new_a$
19. end if
20.end for
21.rerun out

Table 2: Abstarct Transformer for call → enter edge

3.3.2 Abstract Transformer for exit → end-call Edge

Let's apply interprocedural VSA algorithm to Program 3 in figure 6. The program consists of two procedures viz, main and initArray. First step of the algorithm is to identify a-locs for each region. Since there are two procedures, main and initArray, there are three memory regions: for global data, for AR_main and for AR_initArray. A-locs for these regions are identified separately and they are shown in figure 8. Next step is to create a table which maps the a-locs to their (region,offset,size) triple. Table 3 gives this mapping. Note that mapping table also contains



```
int part1Value=1;
int part2Value=0;


void initArray(int a[],int size)
{
    int *part1,*part2;
    int i;
    part1 = &a[0];
    part2 = &a[5];
    for(i=0;i<size;i++)
     {
        *part1 = part1Value;
        *part2 = part2Value;
         part1++;
         part2++;
    }
    return ;
}

int main()
{
   int a[10],*p_array0;
   p_array0 = &a[0];
  initArray(a,5);
  return *p_array0;
}
```

```
    proc initArray
1   lea eax,[esp+4]
2   mov ebx,eax
3   add ebx,20
4   mov ecx,0
L1: mov edx,[4]
6   mov [eax],edx
7   mov edx,[8]
8   mov [ebx], edx
9   add eax,4
10 add ebx,4
11 inc ecx
12 cmp ecx, [esp+8]
13 jl L1
14 return

 proc main
15 sub esp,44
16 lea eax,[esp+4]
17 mov [esp+0],eax
18 push 5
19 push eax
20 call initArray
21 add esp,8
22 mov edi,[esp+0]
23 mov eax,[edi]
24 add esp,44
25 return
```

Figure 6: Program 3 [1]

extended a-locs which are obtained as described already. Now construct CFG for individual procedures and then add linkage edges to get supergraph of executables. Figure 7 shows the supergraph for Program 3.

At the entry of supergraph we have $\{esp \mapsto (\perp, 0, \perp), mem\_4 \mapsto (1, \perp, \perp), mem\_8 \mapsto (0, \perp, \perp)\}$ Regions in the value sets are listed in the following order: (Global,AR_main,AR_initArray). Now start applying abstract transformers to supergraph of program starting with edge 15 →16.

1. After applying the intraprocedural abstract transformers along the edges, we get following value sets at node 20:

   $\{esp \mapsto (\perp, -52, \perp), mem\_4 \mapsto (1, \perp, \perp), mem\_8 \mapsto (0, \perp, \perp), eax \mapsto (\perp, -40, \perp),$
   $var\_44 \mapsto (\perp, -40, \perp), ext\_48 \mapsto (5, \perp, \perp), ext\_52 \mapsto (\perp, -40, \perp)\}$

   Two push operations have advanced esp by 8 bytes and set the value sets of extended a-locs ext_48 and ext_52.

2. When the call to procedure initArray is made along "call initArray" edge, esp is advanced by 4 bytes since return address of procedure main is pushed onto the stack. So now esp has value set: $esp \mapsto (\perp, -56, \perp)$
   Finally, we have following value sets at call initArray node before executing abstarct transformer for call→ enter edge:

   $\{esp \mapsto (\perp, -56, \perp), mem\_4 \mapsto (1, \perp, \perp), mem\_8 \mapsto (0, \perp, \perp), eax \mapsto (\perp, -40, \perp),$
   $var\_44 \mapsto (\perp, -40, \perp), ext\_48 \mapsto (5, \perp, \perp), ext\_52 \mapsto (\perp, -40, \perp)\}$

3. Now let's apply abstract transformer along the call→ enter edge to set value sets of formal parameters to that of actual parameters as described in section 3.3.1.

- First step is to set value set of esp to $(\perp, \perp, 0)$ which corresponds to changing current AR from that of procedure main to procedure initArray.
- After esp is initialized, for every a-loc of procedure initArray corresponding set of a-locs in procedure main is determined and new value set is computed as follows:
  There are 3 a-locs in procedure initArray as shown in mapping table. var_0 will always represent return addreess of caller procedure, hence let's ignore value set of this a-loc.
  Now for var_4:
  $*(F, P) = *(in[esp] +^{vs} \text{offset}(initArray, var\_4), 4) = *((\perp, -56, \perp) +^{vs} 4, 4)$
  $*(F, P) = *((\perp, -52, \perp), 4)$
  Here *(F,P) returns two sets of a-locs F and P where F contains those a-locs whose start addreeses are in $(\perp, -56, \perp)$ and size = 4. That is a-locs whose start addrees is -56 in AR_main. By looking at mapping table, we find a-loc ext_52 satisfying above conditions. Hence,s $F = ext\_52$ and $P = \oint$
  $vs_{actuals} := \sqcup^{vs} in[v] \mid v \in F$
  $vs_{actuals} := \sqcup^{vs} in[ext\_52] := in[ext\_52] := (\perp, -40, \perp)$
  $new_{var\_4} := vs_{actuals} := (\perp, -40, \perp)$
  $out[var\_4] := (\perp, -40, \perp)$
  Similarly for var_8:
  $*(F, P) = *(in[esp] +^{vs} \text{offset}(initArray, var\_8), 4) = *((\perp, -56, \perp) +^{vs} 8, 4)$
  $*(F, P) = *((\perp, -48, \perp), 4)$
  $F = ext\_48$ and $P = \oint$
  and $out[var\_8] := vs_{ext\_48} := (\perp, -40, \perp)$
- So after applying abstract transformer for call→ enter edge, we have following value sets at enter node of initArray (node 1):

  $\{esp \mapsto (\perp, \perp, 0), mem\_4 \mapsto (1, \perp, \perp), mem\_8 \mapsto (0, \perp, \perp), var\_8 \mapsto (5, \perp, \perp),$
  $var\_4 \mapsto (\perp, -40, \perp)\}$

4. Now we have reached in procedure initArray with initial value set as

   $\{esp \mapsto (\perp, \perp, 0), mem\_4 \mapsto (1, \perp, \perp), mem\_8 \mapsto (0, \perp, \perp), var\_8 \mapsto (5, \perp, \perp),$
   $var\_4 \mapsto (\perp, -40, \perp)\}$

   - Now start applying usual intraprocedural analysis along the edges of CFG of procedure initArray. For example, first instruction in the CFG of initArray is "mov eax,[esp+4]". We know the corresponding instruction is R1 = *(R2 + c1) + c2. Hence for the edge 1 → 2, we have following transformation:
     R1 = eax; R2 = esp and c1 = 4 and c2 = 0;
     $*(F, P) = *(vs_{R2} +^{vs} c1, s) = *(vs_{esp} +^{vs} 4, 4) = *((\perp, \perp, 4), 4)$
     From mapping table, $F = \{var\_4\}$ and $P = \oint$
     Hence $vs_{R1} := vs_{eax} := vs_{var\_4} := (\perp, -40, \perp)$ and

     $out = \{esp \mapsto (\perp, \perp, 0), eax \mapsto (\perp, -40, \perp), mem\_4 \mapsto (1, \perp, \perp), mem\_8 \mapsto (0, \perp, \perp),$
     $var\_8 \mapsto (5, \perp, \perp), var\_4 \mapsto (\perp, -40, \perp)\}$

   - Continuing in similar way, when we arrive at end node of initArray procedure we have following value sets :

     $out = \{esp \mapsto (\perp, \perp, 0), eax \mapsto (,,), ebx \mapsto (), var\_40 \mapsto (),$
     $ecx \mapsto (5, \perp, \perp), edx \mapsto (0, \perp, \perp), mem\_4 \mapsto (1, \perp, \perp), mem\_8 \mapsto (0, \perp, \perp),$
     $var\_8 \mapsto (5, \perp, \perp), var\_4 \mapsto (\perp, -40, \perp)\}$

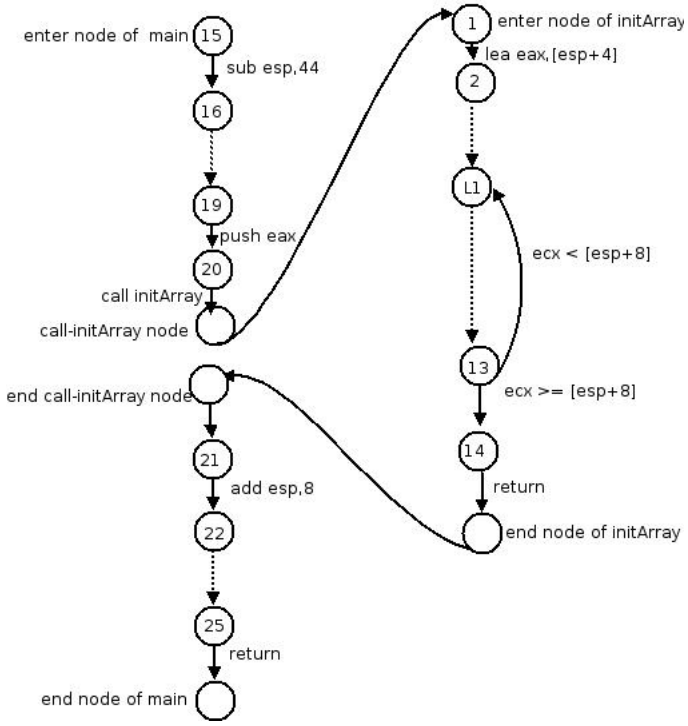


Figure 7: CFG for Program 3

5. Applying Exit end transformer: Now apply transformer Exit → end-call edge along the edge end node of initArray to end call initArray node. It takes two value-sets as input, in_c from call-initArray node and in_x from end node of initArray. The value-sets at the end of this edge is similar to in_x except for the value-sets of ebp, esp and the a-locs for AR_main.

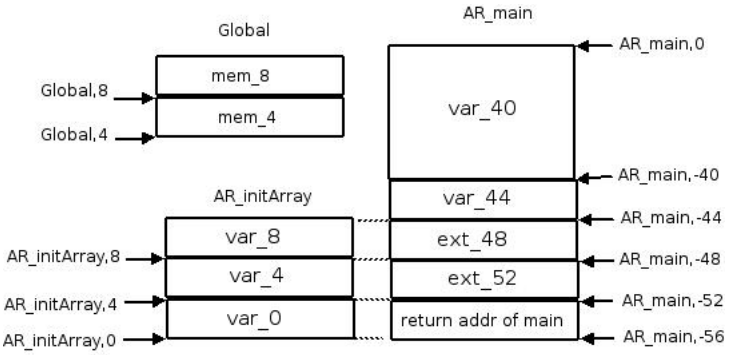


Figure 8: Memory Regions for Program 3

a-loc	(region,offset,size)
var_40	(AR_main,-40,40)
var_44	(AR_main,-44,4)
ext_48	(AR_main,-48,4)
ext_52	(AR_main,-52,4)
var_8	(AR_initArray,8,4)
var_4	(AR_initArray,4,4)
var_0	(AR_initArray,0,4)
mem_8	(Global,8,4)
mem_4	(Global,4,4)

Table 3: Mapping Table for Program 3

4 Overview of Aggregate Structure Identification

IDAPro's algorithm described in section 2.2 only considers accesses to global variables that appear as [absolute address], and accesses to local variables that appear as [esp + offset] or [ebp - offset] in the executable. It fails to count accesses to elements of arrays and variables that are accessed indirectly, and sometimes cannot take into account accesses to fields of structures, because these accesses are performed in ways that do not fall into any of the patterns that IDAPro considers. Therefore, it generally recovers only very coarse information about arrays and structures. This section gives a brief idea about Aggregate Structure Identification (ASI) [4] , which is an algorithm that infers the substructure of aggregates used in a program based on how the program accesses them. ASI helps to recover variables that are better than those recovered by IDAPro which in turns helps to get better results from analysis.

ASI is a unification-based, flow-insensitive algorithm [4] to identify the structure of aggregates (such as arrays, C structs, etc.) in a program. The algorithm ignores any type information known about aggregates, and considers each aggregate to be merely a sequence of bytes of a given length. Depending upon how this aggregate is accessed it is then broken up into smaller parts. The smaller parts are called atoms.

ASI requires data access paterns that are specified to the ASI algorithm through a data-access constraint language (DAC) [4]. There are two kinds of constructs in a DAC program: (1) DataRef is a reference to a set of bytes, and provides a means to specify how the data is accessed in the program; (2) UnifyConstraint provides a means to specify the flow of data in the program. ASI uses the these constraints mentioned in the DAC program to find the atoms and hence a coarsest refinement of the aggregates.

The atoms obtained from ASI are used as a-locs for (re-)analyzing the executable. When these atoms are used as a-locs in VSA, results of VSA can improve. These newly improved results can be used for next round of ASI. If the value-sets computed by VSA are improved from the previous round, the next round of ASI may also improve. This process can be repeated as long as desired, or until the process converges. To summarize, the algorithm for recovering a-locs [4] is

1. Run VSA using a-locs recovered by the Semi-Naïve approach
2. Generate data-access patterns from the results of VSA
3. Run ASI
4. Run VSA
5. Repeat steps 2, 3 and 4 until there are no improvements to the results of VSA

In short, a-locs-refinement principles [4] can be stated as

1. VSA results are used to interpret memory-access expressions in the executable.
2. ASI is used as a heuristic to determine the structure of each memory-region according to information recovered by VSA.
3. ASI calculates the atoms that are used as a-locs for the next round of VSA.

ASI alone is not a replacement for VSA. That is, ASI needs the information - namely value sets from VSA, before it can be applied to an executable.

5 Conclusion

In the security context, analyzing executables becomes necessary as opposed to analyzing source code of program. However there are several challenges in analyzing the executable code. The problem of lack of convenient abstract memory model was presented with the solution of presenting abstract memory model for executables which has the notion of a-locs which is similar to variables in programming language. Then Value Set Analysis was presented which captures the exhaustive set of addresses and integer values that can be present in each a-loc at every program point. VSA is followed by ASI to refine the a-locs.

Once VSA completes, the value-sets for the a-locs at each program point are used to determine each point's sets of used, killed and possibly killed a-locs. This information can very well be useful to understand the behavior of executable and hence the bugs and security vulnerabilities.

References

[1] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables.

[2] Gogul Balakrishnan Bgogul. Recovery of variables and heap structure in x86 executables, 2005.

[3] Balakrishnan Reps Melski. Wysinwyx: What you see is not what you execute, 2007.

[4] G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *POPL*, pages 119–132, 1999.

[5] Thomas W. Reps, Gogul Balakrishnan, and Junghee Lim. Intermediate-representation recovery from low-level code. In John Hatcliff and Frank Tip, editors, *PEPM*, pages 100–111. ACM, 2006.