Recovery of Variables and Heap Structure in x86 Executables

Gogul Balakrishnan
Univ. of Wisconsin
bgogul@cs.wisc.edu

Thomas Reps
Univ. of Wisconsin
reps@cs.wisc.edu

Abstract

This paper addresses two problems that arise when analyzing executables: (1) recovering variable-like quantities in the absence of symbol-table and debugging information, and (2) recovering useful information about objects allocated in the heap.

1. Introduction

There is an increasing need for tools to help programmers and security analysts understand executables. For instance, commercial companies and the military increasingly use Commercial Off-The Shelf (COTS) components to reduce the cost of software development. They are interested in ensuring that COTS components do not perform malicious actions (or can be forced to perform malicious actions). Viruses and worms have become ubiquitous. A tool that aids in understanding their behavior can ensure early dissemination of signatures, and thereby control the extent of damage caused by them. In both domains, the questions that need to be answered cannot be answered perfectly—the problems are undecidable—but static analysis provides a way to answer them conservatively.

In the past five 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 has 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: "**W**hat **Y**ou **S**ee **I**s **N**ot **W**hat **Y**ou e**X**ecute". 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, taken from a login program, is an example of such a mismatch [17]:

```
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. 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 is not just hypothetical; a similar vulnerability was discovered during the Windows security push in 2002 [17]. 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 WYSINWYX phenomenon is not restricted to the presence or absence of procedure calls; on the contrary, it is pervasive: security vulnerabilities can exist because of a myriad of platform-specific details due to features (and idiosyncrasies) of the compiler and the optimizer. These can include (i) memory-layout details (i.e., offsets of variables in the run-time stack's activation records and padding between fields of a struct), (ii) register usage, (iii) execution order, (iv) optimizations, and (v) artifacts of compiler bugs. Such information is hidden from tools that work on intermediate representations (IRs) that are built directly from the source code.

The goal of our work is to advance the state of the art of recovering, from executables, IRs that are (a) similar to those that would be available had one started from source code, but (b) expose the platform-specific details discussed above. Once such IRs are in hand, we will be in a position to leverage the substantial body of work on source-code-vulnerability analysis.[1]

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:

- For many kinds of potentially malicious programs, symbol-table and debugging information is entirely absent. In any case, even if it is present, it cannot be relied upon. For this reason, we have designed techniques that do not rely on symbol-table and debugging information being present. (Thus, throughout the paper, the term "executable" means a stripped executable.)
- To understand the memory-access operations in an executable, it is necessary to determine the set of addresses accessed by each memory-access operation. This is difficult because
  - While some memory operations use explicit memory addresses in the instruction (easy), others use indirect addressing via address expressions (difficult).
  - Arithmetic on addresses is pervasive. For instance, even when the value of a local variable is loaded from its slot in an activation record, address arithmetic is performed.
  - There is no notion of type at the hardware level, so address values cannot be distinguished from integer values.
  - Memory accesses do not have to be aligned, so word-sized address values could potentially be cobbled together from misaligned reads and writes.
- It is difficult to track the flow of data through memory. With the large address spaces of today's machines, it is infeasible to keep track statically of the contents of each memory address during the analysis. Source-code-analysis tools track the flow of data through variables, which provide a finite abstraction of the address space of the program. This observation suggests the need to recover variable-like entities from an executable. Without symbol-table and debugging information, a set of variable-like entities has to be inferred.

[1] A few words are in order regarding the scope of our ambitions. We assume that the executable that is being analyzed follows a "standard compilation model". By this, we mean that the executable has procedures, activation records, a global data region, and a heap; uses virtual functions and dynamically linked libraries; etc. During the analysis, we check that these assumptions hold. When violations are detected, they are reported, and the analysis proceeds, generally after making an optimistic choice. For instance, if the analysis finds that the return address can be modified within a procedure, it reports the violation to the user, but proceeds without modifying the control flow of the program.

- It is challenging to obtain reasonable information about the heap. Simple abstractions for the heap, such as assuming one summary node per malloc site [2, 31, 10] provide little useful information about the heap when applied to executables. Complex shape abstractions [28] cannot be applied to executables due to scalability reasons.

In [3], we presented a combined pointer-analysis and numeric-analysis algorithm called Value-Set Analysis (VSA) that takes into account pointer-arithmetic operations and determines an over-approximation of the set of addresses accessed by each memory operand in the program. However, the version of VSA presented in [3] only partially meets the challenges listed above. In the work described in [3], we used IDAPro [18], a commercial dissassembler, to infer variable-like entities based on the statically-known addresses and stack-frame offsets in the executable. We referred to such variable-like entities as *a-locs* in [3]. However, an a-loc recovered by IDAPro can only represent a set of contiguous locations, and cannot represent non-contiguous memory locations, such as the locations of (all instances of) a specific field in an array of structures, etc. The inability of IDAPro to represent sets of non-contiguous locations affects the accuracy of VSA, as well as clients of VSA.

In this paper, we present an algorithm that combines VSA [3] and Aggregate Structure Identification (ASI) [26] to recover a-locs that are strictly better than IDAPro's a-locs for tracking the flow of data through memory. ASI is an algorithm that determines the structure of aggregates in a program based on how the program accesses the aggregates. ASI assumes that the data-access patterns are readily apparent from the syntax of the program, which is not true for instructions in x86 executables. In this paper, we show how the information recovered by VSA can be used to communicate data-access patterns to ASI. The combination of VSA and ASI allows us (a) to recover a-locs that are based on *indirect* accesses to memory, rather than just the explicit addresses and offsets that occur in the program, and (b) to identify structures, arrays, and nestings of structures and arrays.

In addition, we present an abstraction for the heap that is somewhere in the middle between the extremes of one summary node per malloc site [2, 31, 10] and complex shape abstractions [28]—and is useful in the context of executables. In particular, the heap abstraction that we use makes it possible, in many circumstances, to establish a definite link between the set of objects allocated at a certain site and a particular virtual-function table.

The specific technical contributions of the paper are as follows:

- We show how to apply ASI to programming languages in which data-access patterns are not readily apparent from the syntax of the program. We present the results of applying this approach to x86 executables.
- We replace IDAPro's a-locs that are used during VSA in [3] with a refined notion of a-loc, and provide an algorithm to interpret indirect memory references using the refined notion.
- We develop an abstraction-refinement algorithm based on VSA and ASI to infer from an executable variable-like entities of an appropriate granularity. Our initial experiments show that the algorithm is successful in nearly 87% of the cases for local variables, and in nearly 72% of the cases for variables allocated in the heap.
- We propose an inexpensive abstraction for heap-allocated data structures that allows us to obtain some useful results for objects allocated in the heap. We show the effectiveness of the abstraction by measuring how well it resolves virtual-function calls in x86 executables obtained from C++ code.

The remainder of the paper is organized as follows: §2 provides a detailed explanation of several fundamental challenges that arise when analyzing executables. §3 provides background on VSA and ASI. §4 describes our abstraction-refinement algorithm to recover variable-like entities. §5 describes our abstraction for heap-allocated data structures. §6 provides experimental results evaluating these techniques. §7 and App. A discuss related work.

2. Challenges

Challenge 1: Recovering 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`—is captured by the fact that both statements access the same variable. However, in x86 executables, memory is accessed by specifying absolute addresses directly or indirectly through address expressions of the form "[*base* + *index* × *scale* + *offset*]", where *base* and *index* are registers and *scale* and *offset* are integer constants.

Because, x86 executables do not have intrinsic entities that are analogous to variables that can be used for analysis, the first step in executable analysis is to recover variable-like entities. In [3], we used a crude variable-recovery algorithm provided by the IDAPro disassembler tool [18]. The intuition behind the IDAPro approach is that the layout of memory is known at compile time or assembly time; the compiler or programmer decides a priori the locations of local variables, global variables, etc. Hence, direct accesses to program variables appear as absolute addresses, offsets relative to the frame pointer, etc. Such absolute addresses and offsets provide the starting addresses of program variables. In [3], the set of addresses between such explicitly occurring addresses/offsets was considered to be a variable-like entity referred to as an *a-loc* (for "abstract location").

The disadvantage of IDAPro's variable-recovery algorithm is that it only considers addresses and offsets that occur explicitly in the program. For example, consider the example program and the corresponding disassembly shown below.

```
                          proc main
void main(){          1   mov ebp, esp
   int x, y;          2   sub esp, 8
   x = 1;             3   mov [ebp-8], 1
   y = 2;             4   mov eax, ebp
   return;            5   mov [eax-4], 2
}                     6   add esp, 8
                      7   retn
```

Note that `x` is laid out at offset `-8`[2] and `y` is located at offset `-4` in the activation record of `main`. `x` is accessed relative to the frame pointer `ebp`, but `y` is accessed indirectly through register `eax`. IDAPro only recovers one a-loc of eight-bytes because `-8` is the only offset that is explicit in the executable. To recover variable `y`, the set of values that `eax` holds at `5` needs to be determined. Therefore, one has to look beyond the explicitly known addresses and stack-frame offsets to determine variable-like entities. (See §4.)

Challenge 2: Granularity of recovered variable-like entities

The granularity of variable-like entities that are recovered for an executable affects the complexity and accuracy of subsequent analyses that are based on the recovered variable-like entities. Consider the program shown below:

[2] We follow the convention that the value of `esp` at the beginning of the procedure marks the start of the activation record.

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

void main(){
   Point p, *pp;
   pp = &p;
   pp->x = 1;
   pp->y = 2;
   return pp->x;
}
```

```
   proc main
1  mov ebp, esp
2  sub esp, 8
3  lea eax, [ebp-8]
4  mov [eax], 1
5  mov [eax+4], 2
6  add esp, 8
7  mov eax, [ebp-8]
8  add esp,8
9  retn
```

The program initializes the two fields `x` and `y` of a local struct through the pointer pp and returns the value of field `x`. Observe that, in the executable, `eax` plays the role of the pointer `pp`. (Instruction "`3 lea eax, [ebp-8]`" is equivalent to the assignment `eax := ebp-8`.) Instructions `4` and `5` update the fields of `p`. The only statically known offset in the program is the starting address of `p`. Hence, IDAPro's variable-recovery algorithm identifies one a-loc of eight-bytes that covers both fields of `p`. A flow-dependence analysis based on this a-loc would create a spurious flow-dependence from instruction `5` to `7`. On the other hand, if flow-dependence analysis is performed with two a-locs that correspond to the fields of `p`, there will be no flow-dependence from `5` to `7`. (See §4.)

Challenge 3: The structure of heap-allocated objects

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. The abstraction-refinement algorithm presented in §4 can recover some information about the structure of heap-allocated objects when one summary object is used for each allocation site. §5 presents an improved heap abstraction that allows the abstraction-refinement algorithm to do an even better job.

Challenge 4: Resolving virtual-function calls

Even if the structure of heap-allocated memory blocks were known, the abstraction that is used to summarize the heap may not allow the analysis to do strong updates.[3] To illustrate this problem, consider C++ programs with inheritance and virtual functions. The first 4-bytes of an object contains the address of the virtual-function table. In many source-code-analysis algorithms, the heap is abstracted by associating a variable with each malloc site. This variable summarizes all blocks of memory that are allocated at the corresponding malloc site. However, such an abstraction is not sufficient for statically resolving any of the virtual function calls. This is illustrated in Fig. 1. The first instruction allocates a block of memory in the heap; the next instruction sets the virtual function pointer to the address of the virtual-function table. (This is usually done in the constructor.) If the one-variable-per-malloc-site abstraction is used, it would not be possible to establish the link between the object and the virtual-function table. Because the heap variable represents more than one block, the interpretation of the instruction that sets the virtual function pointer can only do a weak update, i.e., it can only join the virtual-function table address with the existing addresses, and not overwrite the virtual function pointer in the object with the address of the virtual-function table. After the call to malloc, the fields of the object can have any value (shown as ?); computing the join of ? with any value results in ?. Therefore, a definite link between the object and the virtual function table is never established.

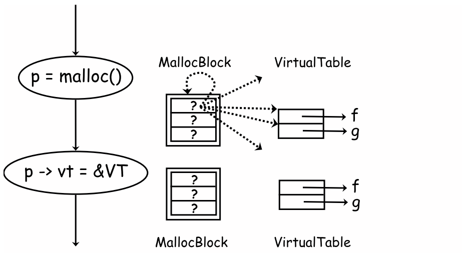

Figure 1. Weak update problem in x86 executables.

3. Background

In previous work [3], we developed a static-analysis algorithm, called *value-set analysis* (VSA), to recover information about the contents of memory locations and how they are manipulated by an executable. This section describes VSA and sets the context in which the techniques described in this paper are applied. Specifically, this section describes the following concepts: (1) abstract locations (a-locs), (2) value-set analysis (VSA), and (3) aggregate structure identification [26] (ASI). This material is related to the core of the paper as follows: In §4, we show how to use information gathered during VSA to harness ASI to the problem of identifying a-locs. This allows us to use a richer class of a-locs for subsequent runs of VSA.

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

int main(){
    int i;
    Point p[5];
    for(i=0;i<5;++i) {
       p[i].x = 1;
       p[i].y = 2;
    }
    return p[0].y;
}
```

```
     proc main
 0   mov ebp,esp
 1   sub esp,40
 2   mov ecx,0
 3   lea eax,[ebp-40]
L1:  mov [eax], 1
 5   mov [eax+4],2
 6   add eax, 8
 7   inc ecx
 8   cmp ecx, 5
 9   jl L1
10   mov eax,[ebp-36]
11   add esp,40
12   retn
```

Figure 2. A program with an array of structures.

EXAMPLE 3.1. The program shown in Fig. 2 will be used as an example in §3 and §4. The program initializes all elements of array `p[5]`. The `x`-members of each element are initialized with 1 and the `y`-members are initialized with 2. The disassembly is also shown. Instruction `L1` updates the `x`-members of the array elements, and instruction `5` updates the `y`-members. Fig. 3(a) shows how the variables are laid out in the activation record of `main`. □

3.1 Abstract Locations (A-locs)

When analyzing programs with source code, one associates information with variables. However, there is no explicit notion of a

[3] A strong update overwrites the contents of an abstract object, and represents a definite change in value to all concrete objects that the abstract object represents [5, 28]. Strong updates cannot generally be performed on summary objects because a (concrete) update usually affects only one of the summarized concrete objects.

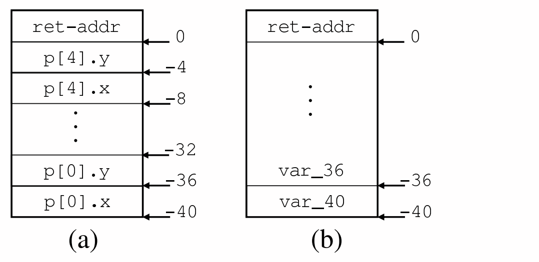

Figure 3. (a) Layout of the activation record for `main` in Ex. 3.1; (b) a-locs identified by IDAPro.

variable in an executable because an executable accesses memory by specifying addresses directly or indirectly through registers. Hence, the first step in analyzing an executable is to recover a set of variable-like quantities. This section described the variable-recovery algorithm used in [3]

In [3], we defined an abstraction of the concrete (runtime) address space. In the runtime address space, there is no separation of the activation records of various procedures, the heap, and the memory for global data. However, during the analysis of an executable, we break the address space into a set of disjoint memory areas, which are referred to as *memory-regions*. Each memory-region represents a group of locations that have similar runtime properties. For example, the runtime locations that belong to the activation record of the same procedure belong to a memory-region. For Ex. 3.1, there would be two memory-regions: (1) a *global*-region containing the locations that correspond to the global data, and (2) an *AR*-region containing the locations that belong to the activation record of `main`.

In [3], we recovered a set of variable-like entities for each memory-region using the method outlined in Challenge 1 (see §2): absolute addresses and offsets provide the starting addresses of the variable-like entities. In Ex. 3.1, the operand `[ebp-40]` in the instruction "`3 lea eax, [ebp-40]`" refers to offset -40 with respect to the start of the activation record for `main`. Similarly, the instruction "`10 mov eax, [ebp-36]`" refers to offset `-36` in the activation record of `main`. The algorithm used in [3] marks such statically determined addresses and offsets in the respective regions and considers the set of locations between two such static addresses/offsets as the analog of a C variable-like entity in the executable. [3] refers to such variable-like quantities as *a-locs*. We will use the same terminology in this paper (although the a-locs used in this paper will have a richer structure than the ones used in [3]). Fig. 3(b) shows the a-locs in `main`'s AR-region.

3.2 Value-Set Analysis (VSA)

A pointer-analysis algorithm is an important component of any program-analysis tool for programs with pointers. Unfortunately, pointer-analysis algorithms that have been developed for source-code analysis [2, 31, 10, 13, 6, 14, 32] are not applicable for analyzing executables. In particular, such algorithms typically ignore pointer arithmetic. However, pointer arithmetic is used extensively in an executable. For instance, even a direct access to a local variable is performed by dereferencing an address calculated as an offset relative to the stack pointer (or the frame pointer)—e.g., `mov eax, [ebp-36]` loads register `eax` with the value of the local variable at offset `-36` of the activation record of the procedure that the frame pointer `ebp` refers to.

The VSA algorithm described in [3] is a pointer-analysis algorithm suitable for executables. VSA is a combined numeric-analysis and pointer-analysis algorithm that determines an over-approximation of the set of numeric values or addresses that each a-loc holds at each program point. The set of addresses and numeric values is referred to as a *value-set*. 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 in an executable.

Suppose n is the number of regions in the executable. A value-set is a n-tuple of strided intervals of the form $s[l, u]$, with each component of the tuple representing the set of addresses in the corresponding region. A strided-interval $s[l, u]$ represents the set of integers $\{i \in \mathbb{Z} | l \leq i \leq u, i \equiv l (\texttt{mod } s)\}$.

- s is called the *stride*.
- $[l, u]$ is called the *interval*.
- $0[l, l]$ represents the singleton set $\{l\}$.

For Ex. 3.1, the value-sets are 2-tuples. We follow the convention that the first component always refers to the set of addresses (or numbers) in the global region and $\emptyset$ denotes an empty set. For instance, the tuple $(1[0, 9], \emptyset)$ represents the set of numbers $\{0, 1, \ldots, 9\}$ and the tuple $(\emptyset, 4[-40, -4])$ represents the set of offsets $\{-40, -36, \ldots, -4\}$ in the AR-region for `main`.

For Ex. 3.1, VSA determines that the value-set of `eax` at program point `L1` is $(\emptyset, 8[-40, -8])$, which means that `eax` holds the offsets $\{-40, -32, \ldots, -8\}$ in the AR-region corresponding to procedure `main`. Note that these offsets are the starting addresses of field `x` of the elements of array `p`.

VSA is a flow-sensitive, context-sensitive, abstract-interpretation algorithm (parameterized by call-string length [29]) that is based on an independent-attribute domain described below.

Let Proc denote the set of memory-regions associated with procedures in the program, AllocMemRgn denotes the set of memory regions associated with heap-allocation sites,[4] and Global denote the memory-region associated with the global data area. We work with the following basic domains:

$$\begin{aligned} \text{MemRgn} &= \{\text{Global}\} \cup \text{Proc} \cup \text{AllocMemRgn} \\ \text{ValueSet} &= \text{MemRgn} \rightarrow \text{StridedInterval}_\perp \\ \text{AlocEnv} &= \text{a-loc} \rightarrow \text{ValueSet}_\perp \end{aligned}$$

VSA associates each program point with an AbsMemConfig:

$$\begin{aligned} \text{AbsEnv} &= \begin{array}{l} (\text{register} \rightarrow \text{ValueSet}) \\ \times\ (\{\text{Global}\} \rightarrow \text{AlocEnv}) \\ \times\ (\text{Proc} \rightarrow \text{AlocEnv}_\perp) \\ \times\ (\text{AllocMemRgn} \rightarrow \text{AlocEnv}_\perp) \end{array} \\ \text{AbsMemConfig} &= (\text{CallString} \rightarrow \text{AbsEnv}_\perp) \end{aligned}$$

3.3 Aggregate Structure Identification (ASI)

In [26], Ramalingam et al. observed that there can be a loss of precision in the results that are computed by a static-analysis algorithm if it does not distinguish between accesses to different parts of the same aggregate (in Cobol programs). They developed the Aggregate Structure Identification (ASI) algorithm to distinguish among such accesses, and showed how the results of ASI can improve the results of dataflow analysis. This section briefly describes the ASI algorithm.

ASI is a unification-based, flow-insensitive algorithm to identify the structure of aggregates in a program. The algorithm ignores the type declarations for all aggregates, and considers each aggregate to be merely a sequence of bytes of a given length. The aggregate is then broken up into smaller parts depending upon how it is accessed by the program. These smaller parts are referred to as *atoms*.

The data-access patterns in the program are specified to the ASI algorithm through a data-access constraint language (DAC). The syntax of DAC programs is shown in Fig. 4. There are two kinds of constructs in a DAC program: (1) `DataRef` is a reference to a set of sequences of bytes, and provides a means to specify how the

[4] This aspect of the abstract domain will be augmented in §5.

data is accessed in the program; (2) UnifyConstraint provides a means to specify the flow of data in the program. Note that the direction of data flow in a UnifyConstraint is not considered because a UnifyConstraint merely establishes that there is a flow of data between the two sequences of bytes; consequently, they should have the same structure. ASI uses the constraints in the DAC program to find a coarsest refinement of the aggregates.

```
            Pgm   ::==  ε | UnifyConstraint Pgm
UnifyConstraint   ::==  DataRef ≈ DataRef
        DataRef   ::==  ProgVars |
                        DataRef[Int:Int] |
                        DataRef\Int+
```

Figure 4. Data-Access Constraint (DAC) language syntax. Int is the set of non-negative integers; $\texttt{Int}_+$ is the set of positive integers; and ProgVars is the set of program variables.

There are three kinds of data references:

- A variable P $\in$ ProgVar refers to all the bytes of variable P.
- DataRef[l:u] refers to bytes l through u in DataRef. For example, P[8:11] refers to the bytes 8..11 of variable P.
- DataRef$\backslash n$ is interpreted as follows. DataRef is an array of n elements and DataRef$\backslash n$ refers to the bytes of a statically indeterminate array element. For example, P[0:11]\3 refers to the sequences of bytes P[0:3], P[4:7], or P[8:11].

Instead of going into the details of the ASI algorithm, we provide the intuition behind the algorithm through an example. Let us consider the source-code program shown in Fig. 2. The data-access constraints for the program are shown below:

```
     p[0:39]\5[0:3]   ≈   const_1[0:3];
     p[0:39]\5[4:7]   ≈   const_2[0:3];
   return_main[0:3]   ≈   p[5:8];
```

The constraints assume that the size of Point is 8 bytes and that x and y are laid out next to each other. The first constraint encodes the initialization of the x members, namely, p[i].x = 1. The DataRef p[0:39]\5[0:3] refers to the bytes that correspond to the x members in array p. The last constraint corresponds to the return statement; it represents the fact that return_main, the return value of procedure main, is assigned bytes 5..8 of p, which corresponds to p[0].y.

The result of ASI is a DAG that shows the structure of each aggregate and the relationship among the atoms of the aggregate. The DAG for Ex. 3.1 is shown in Fig. 5. An ASI DAG has the following properties:

- Each node represents a set of sequences of bytes in an aggregate.
- A sequence of bytes that is accessed as an array in the program is represented by an *array* node. Array nodes are labeled with $\otimes$. The number in the array node represents the number of elements in the array. An array node has one child and the DAG rooted at the child represents the structure of the array element. In Fig. 5, the sequence of bytes represented by p[8:39] is identified as an array of four 8-byte elements. Each array element is a struct with two fields of four-bytes each.
- A sequence of bytes that is accessed like a C structure in the program is represented by a *struct* node. The number in the struct node represents the length of the structure and the children of a struct node represent the fields of the structure. In Fig. 5, the sequence of bytes p[0:39] is identified as a struct with three fields: two scalars and one array.
- Nodes are shared if there is a flow of data in the program involving the corresponding sequence of bytes either directly or indirectly. In Fig. 5, the nodes for the sequence of bytes corresponding to return_main[0:3] and p[5:8] are shared because of the return statement in main. Similarly the sequence of bytes that correspond to the y members of array p, namely p[0:39]\5, share the same node because they are all assigned the same constant at the same instruction.

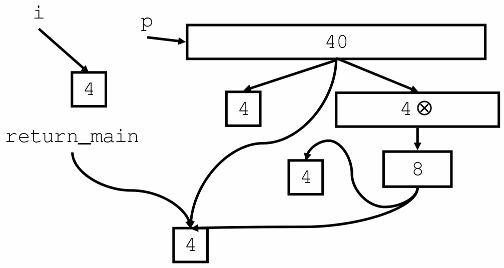

Figure 5. ASI DAG for the program in Fig. 2.

An alternative way to look at the results of ASI is through a C struct declaration as shown below.

```
struct {
   int m1; int m2;
   struct{
      int m3_m1; int m3_m2;
   } m3[4];
} AR_main;
```

4. Recovering A-locs via Iteration

The a-loc abstraction described in §3.1 is not powerful enough to represent arrays of structs (or, more precisely, the different sets of field-instances in an array of structs). An a-loc can only represent a contiguous sequence of memory locations in a memory-region, with no internal substructure. This limitation of IDAPro's a-locs can affect the accuracy of VSA as well as the clients of VSA.

For example, suppose that a procedure in a program has four local variables l_1, l_2, l_3, and l_4 that are laid out next to each other. Suppose that the compiler only generates explicit accesses to l_1 and l_3 and indirect accesses to l_2 and l_4 in terms of the addresses of l_1 and l_3. In this case, IDAPro will identify only two 8-byte a-locs: l_{12}, which spans l_1 and l_2, and l_{34}, which spans l_3 and l_4. During VSA, an update to l_4 is considered to be a weak update to l_{34}. Hence, the value-set computed for l_{34} is not as accurate as desired.

The atoms discovered by the ASI algorithm are similar to a-locs but more expressive; atoms can represent non-contiguous sequences of memory locations (such as the different sets of field-instances in an array of structs). When the atoms obtained for Ex. 3.1 are used as a-locs in value-set analysis, the a-locs defined by instruction L1 are {AR_main.m3[0..4].m3_m1, AR_main.m1}, and the a-locs used at 10 are {AR_main.m2}. Because these sets do not overlap (and there are no intermediate dependences that connect them transitively), this abstraction determines that 10 is not data dependent on L1.

One might hope to apply ASI to an x86 executable by treating each memory-region as an aggregate and determining the structure of each memory-region. However, applying ASI to x86 executables is problematic. One of the requirements for applying ASI is that it must be possible to extract data-access constraints from the program. When applying ASI to Cobol, the data-access patterns are apparent from the syntax of the constructs under consideration. Consequently, generating data-access constraints is possible. Unfortunately, this is not the case for x86 executables. For instance, the memory operand [eax] can either represent an access to a single variable or to all the elements of an array, as is evident in the assembly programs that have been shown earlier. Fortunately, value-sets provide the necessary information to generate data-access constraints. Recall that a value-set is an over-approximation of the set of offsets in each memory-region. This is exactly the information necessary to generate data-access constraints for the executable.

In this section, we show how the atoms in the ASI trees[5] that are obtained from ASI can be used in place of IDAPro's a-locs during VSA. Because ASI trees are not available initially, we use the a-locs identified by IDAPro during the first round of VSA. It is trivial to represent the structure induced by IDAPro's a-locs on the memory region using an ASI tree. Fig. 6 shows the ASI tree for Ex. 3.1. Each memory-region is assumed to be a sequence of bytes of length 2^{32} (2^{64} for a 64-bit machine). The negative offsets in the memory-region correspond to the offsets $\{0, 1, \ldots, (2^{31} - 1)\}$ of the aggregate, and the non-negative offsets in the memory-region correspond to the offsets $\{2^{31}, (2^{31} + 1), \ldots, (2^{32} - 1)\}$ of the aggregate. `ret_addr` is at offset 2^{31}, `var_40` is at offset $(2^{31} - 40)$, and `var_36` is at offset $(2^{31} - 36)$ of the aggregate associated with `AR_main`. The nodes labeled `LocalGuard` and `FormalGuard` represent out-of-bounds areas in the activation record of `_main`. Any access to these symbols is flagged by VSA as a possible memory-access violation.

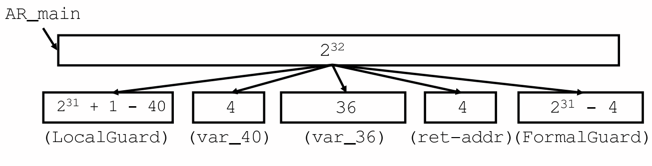

Figure 6. ASI tree constructed from IDAPro's a-locs for Ex. 3.1.

After the first round of value-set analysis has been carried out, the results of value-set analysis are used as a basis for running ASI. The results of ASI are used to refine the set of a-locs, and VSA is run again. This can be carried out for as many rounds as desired, or until no further changes occur. The process is illustrated in Fig. 7. ASI is used only as a heuristic to find a-locs for VSA; i.e., it is not necessary to generate data-access constraints for all memory accesses in the program. Because ASI is a unification-based algorithm, generating data-access constraints for certain kinds of instructions leads to undesirable results. §4.3 discusses some of these cases. Our abstraction-refinement principles can be summarized as follows:

1. VSA is used to obtain memory-access patterns in the executable;
2. ASI is used as a heuristic to determine a set of a-locs according to the memory-access patterns obtained from the information recovered by VSA.

It is important to understand that VSA generates sound results even if not all data-access patterns are taken into account during the previous round of ASI.

ASI is not a replacement for value-set analysis. ASI cannot be applied to x86 executables without the information that is obtained from value-set analysis—namely value-sets.

By propagating the types of parameters to library calls during ASI, we could obtain source-level types for other a-locs.

In the rest of this section, we describe the interplay between VSA and ASI: (1) we show how value-sets are used to generate data-access constraints for input to ASI, and (2) how the atoms in the ASI trees are used during the next round of VSA as a-locs.

4.1 Generating Data-Access Constraints from Value-Sets

This section describes the algorithm that generates ASI data-references for memory operands. First, we consider indirect operands of the form `[r]` and then indirect operands of the form [*base* + *index* × *scale* + *offset*]. To gain intuition about the algorithm, consider operand `[eax]` of instruction `L1` in Ex. 3.1. The value-set associated with `eax` is $(\emptyset, 8[-40, -4])$. The stride value of 8 and the interval $[-40, -4]$ in the activation record of `main` provides evidence that `[eax]` is an access to the elements of an array of 8-byte elements in the range $[-40, -4]$ of the activation record of `main`. Hence, an array access is generated for this operand.

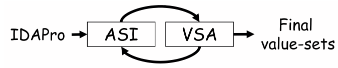

Figure 7. Iterating ASI and VSA to identify better a-locs.

Recall that a value-set is an n-tuple of strided intervals. The strided interval $s[l, u]$ in each component represents the offsets in the corresponding memory-region. Alg. 1 shows the pseudocode to convert offsets in a memory-region into an ASI reference. `SI2ASI` takes the name of a memory-region r, a strided interval $s[l, u]$, and the number of bytes accessed *length* as arguments. The *length* parameter is obtained from the instruction. For example, the *length* for `[eax]` is 4 because the instruction at `L1` in Ex. 3.1 is a four-byte data transfer. The algorithm returns a pair in which the first component is an ASI reference and the second component is a Boolean. The significance of the Boolean component is described below. The algorithm works as follows: If $s[l, u]$ is a singleton then the ASI reference is the one that accesses offsets l to $l + length - 1$ in the aggregate associated with the memory-region r. If $s[l, u]$ is not a singleton then the offsets represented by $s[l, u]$ are treated as references to an array. The size of the array element is the stride s whenever $(s \geq length)$. However, when $(s < length)$ an overlapping set of locations is accessed by the indirect memory-operand. Because an overlapping set of locations cannot be represented using an ASI reference, the algorithm chooses *length* to be the size of the array element. This is not a problem for the soundness of subsequent rounds of VSA because of refinement principle 2. The Boolean component of the pair denotes whether the algorithm generated an exact ASI reference or not. The number of elements in the array is the length of the interval divided by the size.

For operands of the form `[r]`, the set of ASI references is generated by invoking Alg. 1 for each non-empty memory-region in the value-set associated with `r`. In Ex. 3.1, the value-set associated with `eax` at `L1` is $(\emptyset, 8[-40, -4])$. Therefore, the set of ASI references is {AR_main[(-40):(-4)]\5[0:3]}.[6] There are no references to memory-region `Global` because the set of offsets in that region is empty.

One typical use of indirect operands of the form [*base* + *index* × *scale* + *offset*] is to access two-dimensional arrays. Note that *scale* and *offset* are statically-known constants. Because abstract values are strided intervals, we can absorb *scale* and *offset* into *base* and *index*. Hence, without loss of generality, we only discuss memory operands of the form [*base*+*index*]. Assuming that the two-dimensional array is stored in row-major format, one of the registers (usually the *base*) holds the starting addresses of the rows and the other register (usually the *index*) holds the indices of the elements in the row. Therefore, [*base*+*index*] refers to all the elements in the two-dimensional array. Alg. 2 shows the algorithm to generate an ASI reference, when the set of offsets in a memory-region is expressed as a sum of two strided intervals as in [*base*+*index*]. Note that we could use Alg. 1 by computing the abstract sum ($\oplus$) of the two strided intervals. However, doing so results in a loss of precision because strided intervals can only represent a single stride

[5] ASI trees are obtained from the DAG by duplicating shared nodes.

[6] Offsets in a `DataRef` cannot be negative. Negative offsets are used in the paper for ease of understanding. Negative offsets are mapped to the range $[0 : 2^{31} - 1]$; non-negative offsets are mapped to the range $[2^{31} : 2^{32} - 1]$.

exactly. Alg. 2 works as follows: First, it determines which of the two strided intervals is used as a *base* because it is not always apparent from the representation of the operand. The strided interval that is used as a *base* should have a stride that is greater than the length of the interval in the other strided interval. Once the roles of the strided intervals are established, the algorithm generates the ASI reference for *base* followed by the ASI reference for *index*. For example, consider the indirect memory operand `[eax+ecx]`. Suppose that the value-set of `ecx` is $(\emptyset, 16[-160, -16])$, the value-set of `eax` is $(1[0, 9], \emptyset)$, and *length* is 1. For this example, `eax` is the $index$ and `ecx` is the $base$. The ASI reference that is generated is `AR[-160:-1]\10[0:15][0:9]\10[0:0]`.

In some cases, the algorithm cannot establish either of the strided intervals as a base. In such cases, the algorithm computes the abstract sum ($\oplus$) of the two strided intervals and invokes `SI2ASI`.

Algorithm 1 `SI2ASI`: Algorithm to convert a given strided interval into an ASI reference.

Input: The name of a memory-region r, strided interval $s[l, u]$, number of bytes accessed *length*.

Output: A pair in which the first component is an ASI reference for the sequence of *length* bytes starting at offsets $s[l, u]$ in memory-region r and the second component is a Boolean that represents whether the ASI reference is an exact reference (true) or an approximate one (false).

if $s[l, u]$ is a singleton **then**
  return ⟨ “$r[l : l + \textit{length} - 1]$”, true⟩
else
  $size \leftarrow \max(s, \textit{length})$
  $n \leftarrow \lceil (u - l + size - 1)/size \rceil$
  ref ← “$r[l : u + size - 1] \backslash n[0 : size - 1]$”
  return ⟨ref, $(s = size)$⟩
end if

Algorithm 2 Algorithm to convert the set of offsets represented by the sum of two strided intervals into an ASI reference.

Input: The name of a memory-region r, two strided intervals $s_1[l_1, u_1]$ and $s_2[l_2, u_2]$, number of bytes accessed *length*.

Output: An ASI reference for the sequence of *length* bytes starting at offsets $s_1[l_1, u_1] + s_2[l_2, u_2]$ in memory region r.

if ($s_1[l_1, u_1]$ or $s_2[l_2, u_2]$ is singleton) **then**
  return `SI2ASI`(r, $s_1[l_1, u_1] \oplus s_2[l_2, u_2]$, *length*)
end if
if $s_1 \geq (u_2 - l_2 + \textit{length})$ **then**
  baseSI ← $s_1[l_1, u_1]$
  indexSI ← $s_2[l_2, u_2]$
else if $s_2 \geq (u_1 - l_1 + \textit{length})$ **then**
  baseSI ← $s_2[l_2, u_2]$
  indexSI ← $s_1[l_1, u_1]$
else
  return `SI2ASI`(r, $s_1[l_1, u_1] \oplus s_2[l_2, u_2]$, $size$)
end if
⟨baseRef, exactRef⟩ ← `SI2ASI`(r, baseSI, stride(baseSI))
if exactRef is false **then**
  return `SI2ASI`(r, $s_1[l_1, u_1] \oplus s_2[l_2, u_2]$, *length*)
else
  return `concat`(baseRef, `SI2ASI`(‘’, indexSI, *length*))
end if

4.2 Interpreting indirect memory-references

This section describes a lookup algorithm that finds the set of a-locs that are accessed by a memory operand. The lookup algorithm is used to interpret pointer-dereference operations during VSA. For instance, consider the assembly instruction “`mov [eax], 10`”. During VSA, the lookup algorithm is used to determine the a-locs accessed by `[eax]` and the value-sets for the a-locs are updated accordingly. In [3], the algorithm to determine the set of a-locs for a given value-set is trivial because each memory-region in [3] consists of a linear list of IDAPro a-locs. However, after ASI is performed the structure of each memory-region is an ASI tree.

In [26], Ramalingam et al. present a lookup algorithm to retrieve the set of atoms for an ASI expression. However, their lookup algorithm is not appropriate for use in VSA because the algorithm assumes that the only ASI expressions that can arise during lookup are the ones that were used during the atomization phase. Unfortunately, this is not the case during VSA, for the following reasons:

- ASI is used as a heuristic. As will be discussed in §4.3, some data-access patterns that arise during VSA might be ignored during the atomization phase.
- The executable can possibly access fields of those structures that have not yet been broken down into atoms. For example, the initial round of ASI based on IDAPro's information will not include accesses to the fields of structures. However, the first round of VSA may access structure fields.

We will use the tree obtained from Fig. 5 to describe the lookup algorithm. The tree is shown in Fig. 8. Every node in the tree is given a unique name shown within parentheses. The following terms are used in describing the lookup algorithm:

- `NodeDesc` is a descriptor for a part of an ASI tree node and is denoted by a pair $\langle name, \textit{length}\rangle$, where $name$ is the name associated with the ASI tree node and *length* is its length.
- `NodeDescList` is an ordered list of `NodeDesc` descriptors. A `NodeDescList` is represented as $[nd_1, nd_2, \ldots, nd_n]$. A `NodeDescList` represents a contiguous set of offsets in an aggregate. For example, the `NodeDescList` $[\langle a_3, 2\rangle, \langle a_4, 2\rangle]$ represents the offsets `2..5` of node i_1; the offsets `2..3` come from $\langle a_3, 2\rangle$ and the offsets `4..5` come from $\langle a_4, 2\rangle$.

The lookup algorithm is a traversal of the ASI tree guided by the ASI reference for the given memory operand. First, the memory operand is converted into an ASI reference using the algorithm described in §4.1, and the resulting ASI reference is parsed into a list of ASI operations. There are three kinds of ASI operations: (1) `GetChildren`($aloc$), (2) `GetRange`($start$, end), and (3) `GetArrayElements`(m). For example, the list of ASI operations for “`p[0:39]\10[0:1]`” is `[GetChildren(p), GetRange(0, 39), GetArrayElements(10), GetRange(0,1)]`. Each operation takes a `NodeDescList` as argument and returns a set of `NodeDescList` values. The operations are performed from left to right. The argument of each operation comes from the results of the operation that is immediately to its left. The a-locs that are accessed are all the a-locs in the final set of `NodeDesc` descriptors.

The `GetChildren`($aloc$) operation returns a `NodeDescList` that contains `NodeDesc` descriptors corresponding to the children of the root node of the tree associated with the aggregate `aloc`.

`GetRange`($start$, end) returns a `NodeDescList` that contains `NodeDesc` descriptors representing the nodes with offsets in the given range $[start : end]$.

`GetArrayElements`(m) treats the given `NodeDescList` as an array of m elements and returns a set of `NodeDescList` lists. Each `NodeDescList` list represents an array element. There can be more than one `NodeDescList` for the array elements because an array can be split during the atomization phase and different parts of the array might be represented by different nodes.

The following examples discuss traces of a few lookups.

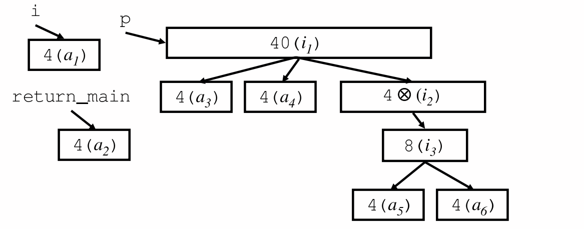

Figure 8. ASI tree for the example in Fig. 2. The name associated with the node is shown within parenthesis.

EXAMPLE 4.1. Lookup `p[0:3]`

$$\begin{array}{ll} & [\langle i_1, 40\rangle] \\ \texttt{GetChildren(p)} & \Downarrow \\ & [\langle a_3, 4\rangle, \langle a_4, 4\rangle, \langle i_2, 32\rangle] \\ \texttt{GetRange(0, 3)} & \Downarrow \\ & [\langle a_3, 4\rangle] \end{array}$$

`GetChildren(p)` returns the `NodeDescList` $[\langle a_3, 4\rangle$, $\langle a_4, 4\rangle$, $\langle i_2, 32\rangle]$. Applying `GetRange(0, 3)` returns $[\langle a_3, 4\rangle]$ because that describes offsets `0..3` in the given `NodeDescList`. The a-loc that is accessed by `p[0:3]` is a_3. □

EXAMPLE 4.2. Lookup `p[0:39]\5[0:3]`

$$\begin{array}{ll} & [\langle i_1, 40\rangle] \\ \texttt{GetChildren(p)} & \Downarrow \\ & [\langle a_3, 4\rangle, \langle a_4, 4\rangle, \langle i_2, 32\rangle] \\ \texttt{GetRange(0, 39)} & \Downarrow \\ & [\langle a_3, 4\rangle, \langle a_4, 4\rangle, \langle i_2, 32\rangle] \\ \texttt{GetArrayElements(5)} & \Downarrow \\ & [\langle a_3, 4\rangle, \langle a_4, 4\rangle], [\langle a_5, 4\rangle, \langle a_6, 4\rangle] \\ \texttt{GetRange(0, 3)} & \Downarrow \\ & [\langle a_3, 4\rangle], [\langle a_5, 4\rangle] \end{array}$$

Let us concentrate on `GetArrayElements(5)` because the other operations are similar to Ex. 4.1. `GetArrayElements(5)` is applied to $[\langle a_3, 4\rangle$, $\langle a_4, 4\rangle$, $\langle i_2, 32\rangle]$. The total length of the given `NodeDescList` is 40 and the number of required array elements is 5. Therefore, the size of the array element is 8. Intuitively, `GetArrayElements` unrolls the given `NodeDescList` and creates a `NodeDescList` for every unique n-bytes starting from the left, where n is the length of the array element. For this example, the unrolled `NodeDescList` is $[\langle a_3, 4\rangle$, $\langle a_4, 4\rangle$, $\langle a_5, 4\rangle$, $\langle a_6, 4\rangle$, $\ldots$, $\langle a_5, 4\rangle$, $\langle a_6, 4\rangle]$. The set of unique 8-byte `NodeDescList` lists for this example is $\{[\langle a_3, 4\rangle, \langle a_4, 4\rangle], [\langle a_5, 4\rangle, \langle a_6, 4\rangle]\}$. □

EXAMPLE 4.3. Lookup `p[8:37]\5[0:5]`

$$\begin{array}{ll} & [\langle i_1, 40\rangle] \\ \texttt{GetChildren(p)} & \Downarrow \\ & [\langle a_3, 4\rangle, \langle a_4, 4\rangle, \langle i_2, 32\rangle] \\ \texttt{GetRange(8, 37)} & \Downarrow \\ & [\langle i_2, 30\rangle] \\ \texttt{GetArrayElements(5)} & \Downarrow \\ & [\langle a_5, 4\rangle, \langle a_6, 2\rangle], [\langle a_6, 2\rangle, \langle a_5, 4\rangle], \\ & [\langle a_6, 4\rangle, \langle a_5, 2\rangle], [\langle a_5, 2\rangle, \langle a_6, 4\rangle] \\ \texttt{GetRange(0, 5)} & \Downarrow \\ & [\langle a_5, 4\rangle, \langle a_6, 2\rangle], [\langle a_6, 2\rangle, \langle a_5, 4\rangle], \\ & [\langle a_6, 4\rangle, \langle a_5, 2\rangle], [\langle a_5, 2\rangle, \langle a_6, 4\rangle] \end{array}$$

This example shows a slightly complicated case of the `GetArrayElements` operation. Unrolling of $[\langle i_2, 30\rangle]$ results in four distinct representations for 6-byte array elements, namely, $[\langle a_5, 4\rangle, \langle a_6, 2\rangle]$, $[\langle a_6, 2\rangle, \langle a_5, 4\rangle]$, $[\langle a_6, 4\rangle, \langle a_5, 2\rangle]$, and $[\langle a_5, 2\rangle, \langle a_6, 4\rangle]$.
□

4.3 Pragmatics

ASI takes into account all possible accesses and data transfers involving memory and finds a partition of the memory-regions that is consistent with these transfers. However, from the standpoint of accuracy of VSA and its clients, it is not always beneficial to take into account all possible accesses. Some of the reasons for this are

- VSA might obtain a very conservative estimate for the value-set of a register (say `R`). In some cases, the value-set for `R` could be $\top$, meaning that register `R` can possibly hold all addresses and numbers. For a memory operand `[R]`, we do not want to generate ASI references that refer to each memory-region as an array of 1-byte elements.
- Some compilers initialize the local stack frame with a known value to aid in debugging uninitialized variables at runtime. For instance, some versions of the Microsoft Visual Studio compiler initialize all bytes of a local stack frame with the value 0xc. The compiler might do this initialization by using a `memcpy`. Generating ASI references that mimic `memcpy` would cause the memory-region associated with this procedure to be broken down into an array of 1-byte elements, which is not desirable.

To deal with such cases, we provide some options for the user to tune the analysis:

- The user can supply an integer threshold. If the cardinality of the value-set associated with a register `R` is above the given threshold, no ASI reference is generated for the memory operand involving `R`.
- The user can supply a set of instructions for which ASI references should not be generated.
- The user can supply explicit references to be used during ASI.

4.4 Convergence

The first round of VSA uncovers memory accesses that are not explicit in the program, which results in the refinement of the a-locs for the next round of VSA. Consequently, the next round of VSA produces more precise value-sets because it is based on a better set of a-locs. Similarly, subsequent rounds of VSA uncover more memory accesses and hence refine the a-locs. Usually, every new round of VSA produces more precise value-sets. Usually, this process stabilizes; i.e., after a few iterations VSA results do not change with every round. However, in some cases, value-sets obtained during VSA may become worse than the value-sets obtained in the previous round. In such cases, the iteration can go on indefinitely. We ensure termination by limiting the iteration to the number of rounds specified by the user. Because, ASI is only used as a heuristic during VSA, the number of iteration rounds does not affect the soundness of the results of VSA.

5. An Abstraction for Heap-Allocated Storage

A great deal of work has been done on algorithms for flow-insensitive points-to analysis [2, 31, 10] (including algorithms that exhibit varying degrees of context-sensitivity [13, 6, 14, 32]). However, all of the aforementioned work uses a very simple abstraction of heap-allocated storage, which we call the *allocation-site abstraction* [19, 5]:

> *All of the nodes allocated at a given allocation site s are folded together into a single summary node n_s.*

In terms of precision, the allocation-site abstraction can often produce poor-quality information. If allocation site s is in a loop, or in a function that is called more than once, then s can allocate multiple nodes with different addresses. A points-to fact "p points to n_s" means that program variable p may point to *one* of the nodes that n_s represents. For an assignment of the form `p->selector1 = q`, points-to-analysis algorithms are ordinarily forced to perform a weak update: that is, selector edges emanating from the nodes that `p` points to are *accumulated*. Because imprecisions snowball as additional weak updates are performed (e.g., for assignments of the form `r->selector1 = p->selector2`), the use of weak updates has adverse effects on what a points-to-analysis algorithm can determine about the properties of heap-allocated data structures.

Even the use of multiple summary nodes per allocation site, where each summary node is qualified by some amount of calling context (as in [22, 15]), does not overcome the problem; that is, algorithms such as [22, 15] must still perform weak updates.

At the other extreme is a family of heap abstractions that have been introduced to discover information about the possible shapes of the heap-allocated data structures to which a program's pointer variables can point [28]. Those abstractions generally allow strong updates to be performed, and are capable of providing very precise characterizations of programs that manipulate linked data structures; however, the methods are also very costly in space and time.

The remainder of this section describes an abstraction for heap-allocated storage, called the *recency abstraction*, that lies in between these extremes. In particular, for allocation sites that arise because the source code contains a call `new C`, where `C` is a class that has virtual methods, the recency abstraction generally permits VSA and ASI to recover information about virtual-function tables.

The recency abstraction is similar in some respects to the allocation-site abstraction, in that each abstract node is associated with a particular allocation site; however, the recency abstraction uses two memory-regions per allocation site s:

$$\text{AllocMemRgn} = \{\text{MRAB}[s], \text{NMRAB}[s] \mid s \text{ an allocation site}\}$$

- MRAB$[s]$ represents the **m**ost-**r**ecently-**a**llocated **b**lock that was allocated at s. Because there is at most one such block in any concrete configuration, MRAB$[s]$ is *never* a summary memory-region.
- NMRAB$[s]$ represents the **n**on-**m**ost-**r**ecently-**a**llocated **b**locks that were allocated at s. Because there can be many such blocks in a given concrete configuration, NMRAB$[s]$ is generally a summary memory-region.

In addition, each MRAB$[s]$, NMRAB$[s]$ $\in$ AllocMemRgn is associated with a "count" value, denoted by MRAB$[s]$.count and NMRAB$[s]$.count, respectively, which is a value of type SmallRange $= \{[0,0], [0,1], [1,1], [0,\infty], [1,\infty], [2,\infty]\}$. The count value records a range for how many concrete blocks the memory-region represents. While NMRAB$[s]$.count can have any SmallRange value, MRAB$[s]$.count will be restricted to take on only values in $\{[0,0],[0,1],[1,1]\}$; consequently, under certain conditions, an abstract transformer can perform a strong update on an a-loc of MRAB$[s]$.

In addition to the count, each MRAB$[s]$, NMRAB$[s]$ $\in$ AllocMemRgn is also associated with a "size" value, denoted by MRAB$[s]$.size and NMRAB$[s]$.size, respectively, which is a value of type StridedInterval. The size value represents an over-approximation of the size of the block. This information is used to report memory-access violations that involve heap-allocated data. We now work with the following basic domains:

$$\begin{aligned}
\text{MemRgn} &= \{\text{Global}\} \cup \text{Proc} \cup \text{AllocMemRgn}\\
\text{ValueSet} &= \text{MemRgn} \to \text{StridedInterval}_\perp\\
\text{AlocEnv} &= \text{a-loc} \to \text{ValueSet}_\perp\\
\text{SmallRange} &= \{[0,0],[0,1],[1,1],[0,\infty],[1,\infty],[2,\infty]\}
\end{aligned}$$

The analysis associates each program point with an AbsMemConfig:

$$\begin{aligned}
\text{AllocAbsEnv} &= (\text{SmallRange} \times \text{StridedInterval} \times \text{AlocEnv})_\perp\\
\text{AbsEnv} &= \begin{array}{l}(\text{register} \to \text{ValueSet})\\ \times (\{\text{Global}\} \to \text{AlocEnv})\\ \times (\text{Proc} \to \text{AlocEnv}_\perp)\\ \times (\text{AllocMemRgn} \to \text{AllocAbsEnv})\end{array}\\
\text{AbsMemConfig} &= (\text{CallString} \to \text{AbsEnv}_\perp)
\end{aligned}$$

Like other memory-regions, MRAB$[s]$ and NMRAB$[s]$ have an associated set of a-locs that can be refined by ASI. Let count, size, and alocEnv, respectively, denote the SmallRange, StridedInterval, and AlocEnv associated with a given AllocMemRgn. A given absEnv $\in$ AbsEnv maps allocation memory-regions, such as MRAB$[s]$ or NMRAB$[s]$, to $\langle$count, size, alocEnv$\rangle$ triples.

The transformers for various operations are defined as follows:

- At the entry point of the program, the AbsMemConfig that describes the initial state records that, for each allocation site s, the AlocEnvs for both MRAB$[s]$ and NMRAB$[s]$ are $\perp$.
- In x86 code, return values are passed back in register `eax`. Let $size$ denote the size of block allocated at the allocation site. The value of $size$ is obtained from the value-set associated with the parameter of the allocation method. The transformer for allocation site s transforms absEnv to absEnv$'$, where absEnv$'$ is identical to absEnv, except that for all ValueSets of absEnv that contain $[..., \text{MRAB}[s] \mapsto si_1, \text{NMRAB}[s] \mapsto si_2, ...]$, become $[..., \emptyset, \text{NMRAB}[s] \mapsto si_1 \sqcup si_2, ...]$ in absEnv$'$. In addition, absEnv$'$ is updated on the following arguments:
$$\begin{aligned}
\text{absEnv}'(\text{MRAB}[s]) &= \langle [0,1], size, \lambda\text{a-loc}.\top_{\text{ValueSet}}\rangle\\
\text{absEnv}'(\text{NMRAB}[s]).\text{count} &= \begin{array}{l}\text{absEnv}(\text{NMRAB}[s]).\text{count}\\ + \text{absEnv}(\text{MRAB}[s]).\text{count}\end{array}\\
\text{absEnv}'(\text{NMRAB}[s]).\text{size} &= \begin{array}{l}\text{absEnv}(\text{NMRAB}[s]).\text{size}\\ \sqcup \text{absEnv}(\text{MRAB}[s]).\text{size}\end{array}\\
\text{absEnv}'(\text{NMRAB}[s]).\text{alocEnv} &= \begin{array}{l}\text{absEnv}(\text{NMRAB}[s]).\text{alocEnv}\\ \sqcup \text{absEnv}(\text{MRAB}[s]).\text{alocEnv}\end{array}\\
\text{absEnv}'(\texttt{eax}) &= [\text{MRAB}[s] \mapsto 0[0,0]]
\end{aligned}$$
In the present implementation, we assume that an allocation always succeeds; hence, in place of the first line above, we use
$$\text{absEnv}'(\text{MRAB}[s]) = \langle [1,1], size, \lambda\text{a-loc}.\top_{\text{ValueSet}}\rangle.$$
Consequently, the analysis only explores the behavior of the system on executions in which allocations always succeed.
- The join $\text{absEnv}_1 \sqcup \text{absEnv}_2$ of $\text{absEnv}_1, \text{absEnv}_2 \in$ AbsEnv is performed pointwise; in particular,
$$\begin{aligned}
\text{absEnv}'(\text{MRAB}[s]) &= \begin{array}{l}\text{absEnv}_1(\text{MRAB}[s])\\ \sqcup \text{absEnv}_2(\text{MRAB}[s])\end{array}\\
\text{absEnv}'(\text{NMRAB}[s]) &= \begin{array}{l}\text{absEnv}_1(\text{NMRAB}[s])\\ \sqcup \text{absEnv}_2(\text{NMRAB}[s])\end{array}
\end{aligned}$$

In all other abstract transformers (e.g., assignments, data movements, interpretation of conditions, etc.), as well as during ASI, MRAB$[s]$ and NMRAB$[s]$ are treated just like other memory regions—i.e., Global and the AR-regions—with two exceptions:

- During the phase that generates the data-access constraints for the executable, an ASI instruction is generated to equate MRAB$[s]$ and NMRAB$[s]$ so that they end up having the same set of a-locs with the same structure.
- During VSA, all abstract transformers are passed a *memory-region status map* that indicates which memory-regions, in the context of a given call-string suffix *cs*, are summary memory-regions. Whereas the Global region is always non-summary, to decide whether a procedure P's memory-region is a summary memory-region, first call-string *cs* is traversed, and then the call graph is traversed, to see whether the runtime stack could contain multiple pending activation records for P.

  The summary-status information for MRAB$[s]$ and NMRAB$[s]$ is obtained differently—from the values of absMemConfig(cs)(MRAB$[s]$).count and absMemConfig(cs)(NMRAB$[s]$).count, respectively.

The memory-region status map provides one of two pieces of information used to identify when a strong update can be performed. In particular, an abstract transformer can perform a strong update if the operation modifies (a) exactly one register, or (b) exactly one a-loc in a non-summary memory-region.

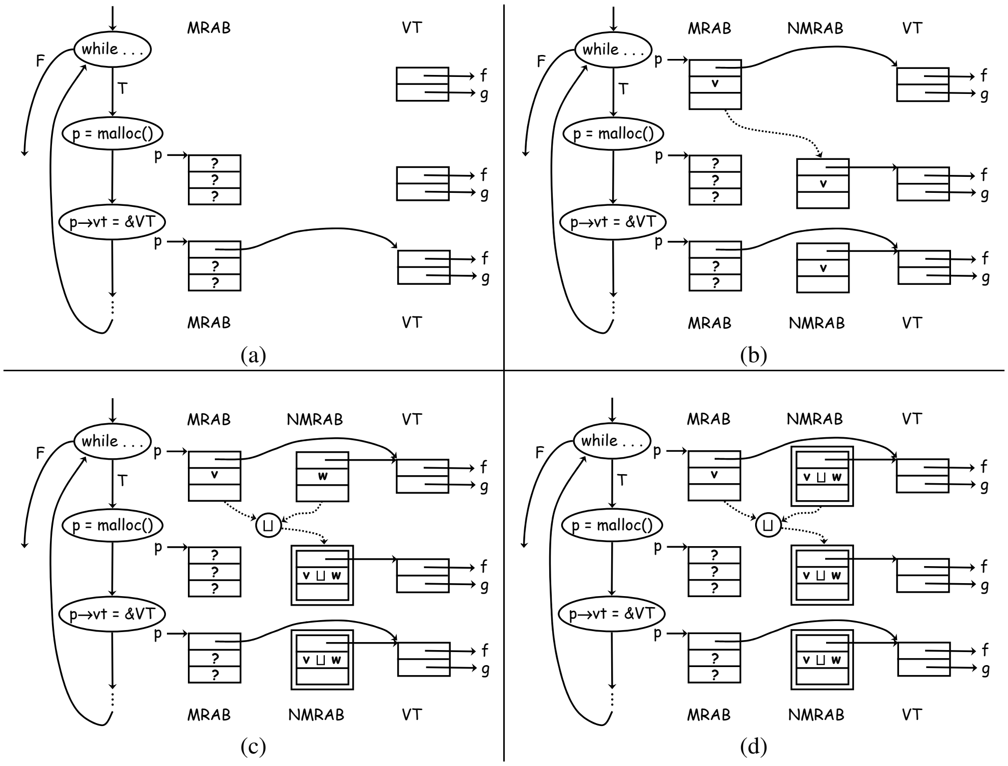

Figure 9. A trace of the evolution of parts of the AbsEnvs for three instructions in a loop.

EXAMPLE 5.1. Fig. 9 shows a trace of the evolution of parts of the AbsEnvs for three instructions in a loop, during the *second* round of VSA, after ASI has identified the a-locs of memory-regions MRAB and NMRAB (shown as the three rectangles within MRAB and NMRAB). Double boxes around NMRAB objects in Fig. 9(c) and (d) are used to indicate that they are summary memory-regions.

For brevity, in Fig. 9 the effect of each instruction is denoted using C syntax; the original source code in the loop body contains a C++ statement "`p = new C`", where `C` is a class that has virtual methods `f` and `g`. The symbols `f` and `g` that appear in Fig. 9 represent the addresses of methods `f` and `g`. The symbol `p` and the two fields of `VT` represent a-locs of the Global region. The dotted lines in Fig. 9(b)–(d) indicate how the value of NMRAB after the `malloc` statement depends on the value of MRAB and NMRAB before the `malloc` statement.

The AbsEnvs stabilize after four iterations. Note that in each of Fig. 9(a)–(d), it can be established that the instruction "`p->vt = &VT`" modifies exactly one a-loc in a non-summary memory-region, and hence a strong update can be performed on `p->vt`. This establishes a definite link between MRAB and VT—and hence between NMRAB and VT. □

6. Experiments

This section describes the results of our preliminary experiments. Tab. 1 shows the characteristics of the set of examples that we used in our evaluation. These programs were originally used by Pande and Ryder in [25] to evaluate their algorithm for resolving virtual-function calls in C++ programs. The programs in C++ were compiled without optimization[7] using the Microsoft Visual Studio 6.0 compiler and the .obj files obtained from the compiler were analyzed. The numbers in the column labeled as "Rounds" indicates the number of ASI-VSA iterations after which the value-sets obtained from VSA either stabilized or started to deteriorate. The column labeled "Coverage" is discussed below.

6.1 Results of virtual-function call resolution

Tab. 2 shows the results of applying VSA to resolve virtual-function calls. The column labeled $\perp$ shows the number of unreachable indirect call-sites. The column labeled $\top$ denotes the number of reachable indirect call-sites at which VSA could not determine the targets. The other columns show the distribution of the number of targets at the indirect call-sites.

A non-zero value in the $\top$-column means that VSA was not able to resolve some indirect calls in the executables. VSA reports such call-sites to the user, but does not explore any procedures from that call-site. The coverage column in Tab. 1 shows the fraction of the executable code that was analyzed during VSA. Note that for programs for which the $\top$-column is 0, the coverage reported in Tab. 1 is an over-approximation of the actual runtime coverage for all possible runs of the executable. However, for programs with non-zero values in the $\top$-column, the coverage metric reported in Tab. 1 is a possible under-approximation. Also, for programs with a 0 value in $\top$-column, the number of unreachable calls is an under-approximation. For instance, the calls that are flagged as unreachable (see the $\perp$-column) in `deriv1` are definitely unreachable.

We see that the algorithm resolves virtual-function calls for most of the call-sites. We examined the indirect call-sites that VSA was able to resolve and found that VSA had correctly resolved the targets of those calls. It is important to realize that all these virtual-function calls are resolved solely by tracking the flow of data through memory (including the heap). The analysis algorithm does not rely on symbol-table or debugging information.

[7] Note that unoptimized programs generally have more memory accesses, and hence represent a greater challenge than optimized programs.

VSA could not resolve the other indirect call-sites mostly because it could not establish that all the elements of an array are definitely initialized in a loop. The problem is as follows. In some of the example programs, an array of pointers to objects is initialized via a loop. These pointers are later used to invoke a virtual-function call. Even if VSA were successful in determining the addresses of the elements of the array, it would not establish that all elements of the array are definitely initialized by the instruction in the loop. Hence, the values of the pointers in the array remain $\top$.

	x86 Insts	CFGs	ICalls	Coverage(%)	Rounds	Time (s)
NP	252	5	6	98.99	2	1
primes	294	9	2	75.79	2	0
family	351	9	3	98.71	4	1
vcirc	407	14	5	78.23	3	0
fsm	502	13	1	87.57	2	5
office	592	22	4	60.34	1	0
trees	1299	29	3	68.08	6	9
deriv1	1369	38	18	56.11	9	4
chess	1662	41	1	84.21	3	16
objects	1739	47	23	37.12	3	2
simul	1920	60	3	22.84	3	6
greed	1945	47	17	72.87	4	10
shapes	1955	39	12	64.38	7	10
ocean	2552	61	5	44.65	3	17
deriv2	2639	41	56	32.26	7	2

Table 1. Characteristics of example programs. "Coverage" denotes the part of the program that is reachable during analysis (see §6.1). "Rounds" denotes the number of ASI-VSA rounds after which results stabilized.

6.2 Results of a-loc identification

To evaluate the results of the abstraction-refinement algorithm, we compared the results of the algorithm with the debugging information. We constructed an ASI tree for each activation record in the program using the debugging information, and determined the number of leaves that had similar structure in both the ASI tree obtained using debugging information and the one obtained using the abstraction-refinement algorithm. The results are in shown Fig. 6.2. On average, the abstraction-refinement algorithm determines the structure of 87% of the local variables correctly.

We did a similar evaluation of the structure of heap-allocated data objects. The results are shown in Fig. 6.2. For the heap, the abstraction-refinement algorithm determines the structure of 72% of these objects correctly. We only compared the structure of memory blocks that summarized reachable allocation-sites with the debugging information because the abstraction-refinement algorithm only provides information for such blocks.

7. Related Work

Other work on analyzing memory accesses in executables. Previous techniques deal with memory accesses very conservatively; generally, if a register is assigned a value from memory, it is assumed to take on any value. For instance, although the basic goal of the algorithm proposed by Debray et al. [11] is similar to that of VSA, their goal is to find an over-approximation of the set of values that each *register* can hold at each program point; for us, it is to find an over-approximation of the set of values that each (abstract) data object can hold at each program point, where data objects include *global, stack-allocated, and heap-allocated memory locations* in addition to registers. In the analysis proposed by

	$\perp$	1	2	≥ 3	$\top$
NP	0	0	6	0	0
primes	1	1	0	0	1
family	0	3	0	0	0
vcirc	0	5	0	0	0
fsm	0	1	0	0	0
office	0	4	0	0	0
trees	1	0	0	1	2
deriv1	8	8	2	0	0
chess	0	0	0	0	1
objects	18	0	4	0	1
simul	2	0	0	0	1
greed	6	10	0	0	1
shapes	4	4	3	0	1
ocean	3	0	0	0	2
deriv2	33	22	0	0	1

Table 2. Distribution of the number of callees at each indirect call.

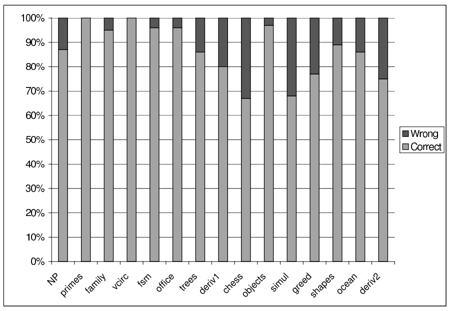

Figure 10. Percentage of the locals for which results of ASI matched the debugging information.

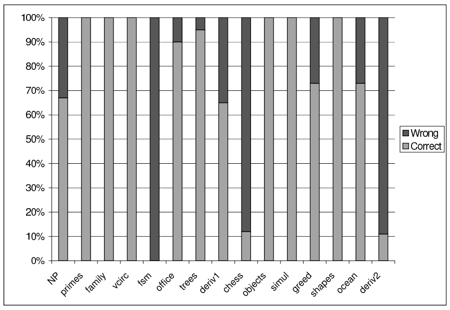

Figure 11. Percentage of the heap-allocated structure for which results of ASI matched the debugging information.

Debray et al., a set of addresses is approximated by a set of congruence values: they keep track of only the low-order bits of addresses. However, unlike VSA, their algorithm does not make any effort to track values that are not in registers. Consequently, it loses a great deal of precision whenever there is a load from memory.

Cifuentes and Fraboulet [7] give an algorithm to identify an intraprocedural slice of an executable by following the program's use-def chains. However, their algorithm also makes no attempt to track values that are not in registers, and hence cuts short the slice when a load from memory is encountered.

The two pieces of work that are most closely related to VSA are the algorithm for data-dependence analysis of assembly code of Amme et al. [1] and the algorithm for pointer analysis on a low-level intermediate representation of Guo et al. [15]. The algorithm of Amme et al. performs only an *intra*procedural analysis, and it is not clear whether the algorithm fully accounts for dependences between memory locations. The algorithm of Guo et al. [15] is only partially flow-sensitive: it tracks registers in a flow-sensitive manner, but treats memory locations in a flow-insensitive manner. The algorithm uses partial transfer functions [33] to achieve context-sensitivity. The transfer functions are parameterized by "unknown initial values" (UIVs); however, it is not clear whether the the algorithm accounts for the possibility of called procedures corrupting the memory locations that the UIVs represent.[8]

Identification of structures. Aggregate structure identification was devised by Ramalingam et al. to partition aggregates according to a Cobol program's memory-access patterns [26]. A similar algorithm was devised by Eidorff et al. [12] and incorporated in the AnnoDomani system. The original motivation for these algorithms was the Year 2000 problem; they provided a way to identify how date-valued quantities could flow through a program.

Mycroft [24] gave a unification-based algorithm for performing type reconstruction; for instance, when a register is dereferenced with an offset of 4 to perform a 4-byte access, the algorithm infers that the register holds a pointer to an object that has a 4-byte field at offset 4. The type system uses disjunctive constraints when multiple type reconstructions from a single usage pattern are possible.

Other work on pointer analyses The recency abstraction is similar in flavor to the allocation-site abstraction [19, 5], in that each abstract node is associated with a particular allocation site; however, the recency abstraction is designed to take advantage of the fact that VSA is a flow-sensitive, context-sensitive algorithm. Note that if the recency abstraction were used with a flow-insensitive algorithm, it would provide little additional precision over the allocation-site abstraction: because a flow-insensitive algorithm has just one abstract memory configuration that expresses a *program-wide* invariant, the algorithm would have to perform weak updates for assignments to MRAB nodes (as well as for assignments to NMRAB nodes); that is, edges emanating from an MRAB node would also have to be *accumulated*.

With a flow-sensitive algorithm, the recency abstraction uses twice as many abstract nodes as the allocation-site abstraction, but under certain conditions it is sound for the algorithm to perform strong updates for assignments to MRAB nodes, which is crucial to being able to establish a definite link between the set of objects allocated at a certain site and a particular virtual-function table.

Hackett and Rugina [16] describe a method that uses local reasoning about individual heap locations, rather than global reasoning about entire heap abstractions. In essence, they use an independent-attribute abstraction: each "tracked location" is tracked independently of other locations in concrete memory configurations. The recency abstraction is a different independent-attribute abstraction.

The use of count information on (N)MRAB nodes was inspired by the heap abstraction of Yavuz-Kahveci and Bultan [34], which also attaches numeric information to summary nodes to characterize the number of concrete nodes represented.

References

[1] W. Amme, P. Braun, E. Zehendner, and F. Thomasset. Data dependence analysis of assembly code. *Int. J. Parallel Proc.*, 2000.

[2] L. O. Andersen. Binding-time analysis and the taming of C pointers. In *Part. Eval. and Semantics-Based Prog. Manip.*, pages 47–58, 1993.

[3] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Comp. Construct.*, pages 5–23, 2004.

[4] J. Bergeron, M. Debbabi, J. Desharnais, M.M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. *Int. J. of Req. Eng.*, 2001.

[5] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *PLDI*, pages 296–310, 1990.

[6] B.-C. Cheng and W.W. Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *PLDI*, pages 57–69, 2000.

[7] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *Int. Conf. on Softw. Maint.*, pages 188–195, 1997.

[8] C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation. In *Int. Conf. on Softw. Maint.*, 1998.

[9] K.D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI*, pages 57–66, 1988.

[10] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI*, pages 35–46, 2000.

[11] S.K. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *POPL*, pages 12–24, 1998.

[12] P.H. Eidorff, F. Henglein, C. Mossin, H. Niss, M.H. Sørensen, and M. Tofte. Annodomini: From type theory to year 2000 conversion tool. In *POPL*, pages 1–14, 2005.

[13] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI*, 2000.

[14] J.S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *SAS*, 2000.

[15] B. Guo, M.J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D.I. August. Practical and accurate low-level pointer analysis. In *3nd IEEE/ACM Int. Symp. on Code Gen. and Opt.*, pages 291–302, 2005.

[16] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, pages 310–323, 2005.

[17] M. Howard. Some bad news and some good news. *MSDN*, October 2002. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure10102002.asp.

[18] IDAPro disassembler, http://www.datarescue.com/idabase/.

[19] N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *POPL*, pages 66–74, 1982.

[20] A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *Computer Aided Verif.*, 2005.

[21] J.R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *PLDI*, pages 291–300, 1995.

[22] A. Milanova, A. Rountev, and B.G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *TOSEM*, 2005.

[23] M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *ESOP*, 2005.

[24] A. Mycroft. Type-based decompilation. In *ESOP*, 1999.

[25] H. Pande and B. Ryder. Data-flow-based virtual function resolution. In *SAS*, pages 238–254, 1996.

[26] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *POPL*, 1999.

[27] X. Rival. Abstract interpretation based certification of assembly code. In *Int. Conf. on Verif., Model Checking, and Abs. Int.*, 2003.

[28] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst.*, 24(3):217–298, 2002.

[29] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[30] A. Srivastava and A. Eustace. ATOM - A system for building customized program analysis tools. In *PLDI*, 1994.

[31] B. Steensgaard. Points-to analysis in almost-linear time. In *POPL*, 1996.

[32] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. In *PLDI*, 2004.

[33] R.P. Wilson and M.S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, 1995.

[34] T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In *SAS*, 2002.

[8] The drawbacks of [1] and [15] mentioned above were acknowledged in conversations with authors of the two papers.

A. Additional Related Work

Other work on analyzing executables. Several platforms have been created for manipulating executables in the presence of additional information, such as source code, symbol-table information, and debugging information, including ATOM [30] and EEL [21].

Bergeron et al. [4] present a static-analysis technique to check if an executable with debugging information adheres to a user-specified security policy.

Rival [27] presents an analysis that uses abstract interpretation to check whether the assembly code of a program produced by a compiler possesses the same safety properties as the source code. The analysis assumes that source code and debugging information is available. First, the source code and the assembly code of the program are analyzed. Next, the debugging information is used to map the results of assembly-code analysis back to the source code. If the results for the corresponding program points in source and assembly code are compatible, then the assembly code possesses the same safety properties as the source code.

Identification of structures. Mycroft's algorithm for type-based decompilation has several weaknesses due to the absence of certain kinds of information. Some of these can be addressed using information obtained by the techniques described in this paper:

- Mycroft explains how several simplifications could be triggered if interprocedural side-effect information were available. Once the information computed by the methods presented in this paper is in hand, interprocedural side-effect information could be computed by standard techniques [9].
- Mycroft's algorithm is unable to recover information about the sizes of arrays that are identified. Although not described in this paper, our implementation incorporates a third analysis phase, called affine-relation analysis (ARA) [3, 23, 20], that, for each program point, identifies the affine relations that hold among the values of registers. In essence, this provides information about induction-variable relationships in loops, which, in turn, can allow VSA to recover information about array sizes when one register is used to sweep through an array under the control of a second loop-index register.
- Mycroft does not have stride information available; however, the basic abstract domain on which VSA is based is strided intervals.
- Mycroft excludes from consideration programs in which addresses of local variables are taken because "it can be unclear as to where the address-taken object ends—a `struct` of size 8 bytes followed by a coincidentally contiguously allocated `int` can be hard to distinguish from a `struct` of 12 bytes." This is a problematic restriction for a decompiler because it is a common idiom: in C programs, addresses of local variables are frequently used as explicit arguments to called procedures (when programmers simulate call-by-reference parameter passing), and C++ and Java compilers can use addresses of local variables to implement call-by-reference parameter passing.

  Because the methods presented in this paper provide information about the usage patterns of pointers into the stack, they would allow Mycroft's techniques to be applied in the presence of pointers into the stack.

It would be interesting to make use of Mycroft's techniques in conjunction with the techniques described in this paper.

Decompilation. Past work on decompiling assembly code to a high-level language [8] is also related to our work. However, the decompilers reported in the literature are somewhat limited in what they are able to do when translating assembly code to high-level code. For instance, Cifuentes's work [8] primarily concentrates on recovery of (a) expressions from instruction sequences, and (b) control flow.

We believe that many of the shortcomings of previous decompilers are caused by the fact that they do not incorporate methods to recover information about memory accesses. The memory-access-analysis methods that have been described in this paper can be performed prior to decompilation proper, to recover information about numeric values, address values, physical types, and definite links from objects to virtual-function tables.