DIVINE: DIscovering Variables IN Executables

Gogul Balakrishnan and Thomas Reps

Comp. Sci. Dept., University of Wisconsin; {bgogul,reps}@cs.wisc.edu

Abstract. This paper addresses the problem of recovering variable-like entities when analyzing executables in the absence of debugging information. We show that variable-like entities can be recovered by iterating *Value-Set Analysis* (VSA), a combined numeric-analysis and pointer-analysis algorithm, and *Aggregate Structure Identification*, an algorithm to identify the structure of aggregates. Our initial experiments show that the technique is successful in correctly identifying 88% of the local variables and 89% of the fields of heap-allocated objects. Previous techniques recovered 83% of the local variables, but 0% of the fields of heap-allocated objects. Moreover, the values computed by VSA using the variables recovered by our algorithm would allow any subsequent analysis to do a better job of interpreting instructions that use indirect addressing to access arrays and heap-allocated data objects: indirect operands can be resolved better at 4% to 39% of the sites of writes and up to 8% of the sites of reads. (These are the memory-access operations for which it is the most difficult for an analyzer to obtain useful results.)

1 Introduction

There is an increasing need for tools to help programmers and security analysts understand executables. For instance, 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 could ensure early dissemination of signatures, and thereby help 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.

The long-term goal of our work is to develop bug-detection and security-vulnerability analyses that work on executables. As a means to this end, our immediate goal is to advance the state of the art of recovering, from executables, Intermediate Representations (IRs) that are similar to those that would be available had one started from source code. We envisage the following uses for the IRs: (1) as an aid to a human analyst who is trying to understand the behavior of the program, and (2) as the basis for further static analysis of executables. Moreover, once such IRs are in hand, we will be in a position to leverage the substantial body of work on bug-detection and security-vulnerability analysis based on IRs built from source code.

One of the several obstacles in IR recovery is that a program's data objects are not easily identifiable in an executable. Consider, for instance, a data dependence from statement `a` to statement `b` that is transmitted by write/read accesses on some variable `x`. When performing source-code analysis, the programmer-defined variables provide us with convenient compartments for tracking such data manipulations. A dependence analyzer must show that `a` defines `x`, `b` uses `x`, and there is an `x`-def-free path from `a` to `b`. However, 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. If debugging information is available (and trusted), this provides one possibility; however, even if debugging information is available, analysis techniques have to account for bit-level, byte-level, word-level, and bulk-memory manipulations performed by programmers (or introduced by the compiler) that can sometimes violate variable boundaries [3, 18, 24]. If a program is suspected of containing malicious code, even if debugging information is present, it cannot be entirely relied upon. For these reasons, it is not always desirable to use debugging information—or at least to rely on it alone—for identifying a program's data objects. (Similarly, past work on source-code analysis has shown that it is sometimes valuable to ignore information available in declarations and infer replacement information from the actual usage patterns found in the code [12, 21, 23, 28, 30].)

Moreover, for many kinds of 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. While the reader may wonder about how effective one can be at determining information about a program's behavior from low-level code, a surprisingly large number of people—on a daily basis—are engaged in inspecting low-level code that is not equipped with debugging information. These include hackers of all hat shades (black, grey, and white), as well as employees of anti-virus companies, members of computer incident/emergency response teams, and members of the intelligence community.

Heretofore, the state of the art in recovering variable-like entities is represented by IDAPro [15], a commercial disassembly toolkit. IDAPro's algorithm is based on the observation that accesses to global variables appear as "[*absolute-address*]", and accesses to local variables appear as "[`esp` + *offset*]" or "[`ebp` − *offset*]" in the executable. That is, IDAPro recovers variables based on purely local techniques.[1] This approach has certain limitations. For instance, it does not take into account accesses to fields of structures, elements of arrays, and variables that are only accessed through pointers, because these accesses do not fall into any of the patterns that IDAPro considers. Therefore, it generally recovers only very coarse information about arrays and structures. Moreover, this approach fails to provide any information about the fields of heap-allocated objects, which is crucial for understanding programs that manipulate the heap.

The aim of the work presented in this paper is to improve the state of the art by using abstract interpretation [10] to replace local analyses with ones that take a more comprehensive view of the operations performed by the program. We present an algorithm that combines Value-Set Analysis (VSA) [4], which is a combined numeric-analysis and pointer-analysis algorithm that works on executables, and Aggregate Structure Identification (ASI) [23], which is an algorithm that infers the substructure of aggregates used in a program based on how the program accesses them, to recover variables that are better than those recovered by IDAPro. As explained in §5, the combination of VSA

[1] IDAPro does incorporate a few global analyses, such as one for determining changes in stack height at call-sites. However, the techniques are ad-hoc and based on heuristics.

and ASI allows us (a) to recover variables 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. Moreover, when the variables that are recovered by our algorithm are used during VSA, the precision of VSA improves. This leads to an interesting abstraction-refinement scheme; improved precision during VSA causes an improvement in the quality of variables recovered by our algorithm, which, in turn, leads to improved precision in a subsequent round of VSA, and so on.

The specific technical contributions of the paper are as follows:

- We present an abstract-interpretation-based algorithm for recovering variable-like entities from an executable. In particular, we show how information provided by VSA is used in combination with ASI for this purpose.
- We evaluate the usefulness of the variables recovered by our algorithm to a human analyst. We compare the variables recovered by our algorithm against the debugging information generated at compile time. Initial experiments show that the technique is successful in correctly identifying 88% of the local variables and 89% of the fields of heap-allocated objects. Previous techniques based on local analysis recovered 83% of the local variables, but 0% of the fields of heap-allocated objects.
- We evaluate the usefulness of the variables and values recovered by our algorithm as a platform for additional analyses. Initial experiments show that the values computed by VSA using the variables recovered by our algorithm would allow any subsequent analysis to do a better job of interpreting instructions that use indirect addressing to access arrays and heap-allocated data objects: indirect memory operands can be resolved better at 4% to 39% of the sites of writes and up to 8% of the sites of reads.

Our current implementation of the variable-recovery algorithm—which is incorporated in a tool called CodeSurfer/x86 [25]—works on x86 executables; however, the algorithms used are architecture-independent.

The remainder of the paper is organized as follows: §2 provides an abstract memory model for analyzing executables. §3 provides an overview of our approach to recover variable-like entities for use in analyzing executables. §4 provides background on VSA and ASI. §5 describes our abstraction-refinement algorithm to recover variable-like entities. §6 reports experimental results. §7 discusses related work.

2 An Abstract Memory Model

In this section, we present an abstract memory model for analyzing executables. A simple model is to consider memory as an array of bytes. Writes (reads) in this trivial memory model are treated as writes (reads) to the corresponding element of the array. However, there are some disadvantages in such a simple model:

- It may not be possible to determine specific address values for certain memory blocks, such as those allocated from the heap via `malloc`. For the analysis to be sound, writes to (reads from) such blocks of memory have to be treated as writes to (reads from) any part of the heap.
- The runtime stack is reused during each execution run; in general, a given area of the runtime stack will be used by several procedures at different times during execution. Thus, at each instruction a specific numeric address can be ambiguous

(because the same address may belong to different activation records at different times during execution): it may denote a variable of procedure `f`, a variable of procedure `g`, a variable of procedure `h`, etc. (A given address may also correspond to different variables of different activations of `f`.) Therefore, an instruction that updates a variable of procedure `f` would have to be treated as possibly updating the corresponding variables of procedures `g`, `h`, etc.

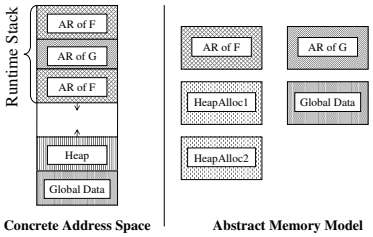

Fig. 1: Memory-regions

To overcome these problems, we work with the following abstract memory model [4]. Although in the concrete semantics the activation records (ARs) for procedures, the heap, and the memory for global data are all part of *one* address space, for the purposes of analysis, we separate the address space into a set of disjoint areas, which are referred to as *memory-regions* (see Fig. 1). Each memory-region represents a group of locations that have similar runtime properties. For example, the runtime locations that belong to the ARs of a given procedure belong to one memory-region. For a given program, there are three kinds of regions: (1) *global*-regions, for memory locations that hold global data, (2) *AR*-regions, each of which contains the locations of the ARs of a particular procedure, and (3) *malloc*-regions, each of which contains the locations allocated at a particular `malloc` site.

3 Overview of our Approach

Our goal is to subdivide the memory-regions of the executable into variable-like entities (which we call *a-locs*, for "abstract locations"). These can then be used as variables in tools that analyze executables. Memory-regions are subdivided using the information about how the program accesses its data. The intuition behind this approach is that data-access patterns in the program provide clues about how data is laid out in memory. For instance, the fact that an instruction in the executable accesses a sequence of four bytes in memory-region `M` is an indication that the programmer (or the compiler) intended to have a four-byte-long variable or field at the corresponding offset in `M`. In this section, we present the problems in developing such an approach, and the insights behind our solution, which addresses those problems. Details are provided in §5.

3.1 The Problem of Indirect Memory Accesses

Past work on analyzing executables [4, 15] uses the addresses and stack-frame offsets that occur explicitly in the program to recover variable-like entities. We will call this the *Semi-Naïve algorithm*. It is based on the observation that access to global variables appear as "[*absolute-address*]", and access to local variables appear as "[`esp` + *offset*]" or "[`ebp` − *offset*]" in the executable. Thus, absolute addresses and offsets that occur explicitly in the executable (generally) indicate the starting addresses of program variables. Based on this observation, the Semi-Naïve algorithm identifies each set of locations between two neighboring absolute addresses or offsets as a single variable. Such an approach produces poor results in the presence of indirect memory operands.

Example 1. The program initializes the two fields `x` and `y` of a local struct through the pointer `pp` and returns 0. `pp` is located at offset -12,[2] and struct `p` is located at offset -8 in the activation record of `main`. Address expression "`ebp-8`" refers to the address of `p`, and address expression "`ebp-12`" refers to the address of `pp`.

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

int main(){
   Point p, *pp;
   pp = &p;
   pp->x = 1;
   pp->y = 2;
   return 0;
}
```

```
  proc main
1 mov ebp, esp
2 sub esp, 12
3 lea eax, [ebp-8]
4 mov [ebp-12], eax
5 mov [eax], 1
6 mov [eax+4], 2
7 mov eax, 0
8 add esp, 12
9 retn
```

Instruction `4` initializes the value of `pp`. (Instruction "`3 lea eax, [ebp-8]`" is equivalent to the assignment `eax := ebp-8`.) Instructions `5` and `6` update the fields of `p`. Observe that, in the executable, the fields of p are updated via `eax`, rather than via the pointer `pp` itself, which resides at address `ebp-12`. □

In Ex. 1, `-8` and `-12` are the offsets relative to the frame pointer (i.e., `ebp`) that occur explicitly in the program. The Semi-Naïve algorithm would say that offsets `-12` through `-9` of the AR of `main` constitute one variable (say `var_12`), and offsets `-8` through `-1` of AR of `main` constitute another (say `var_8`). The Semi-Naïve algorithm correctly identifies the position and size of `pp`. However, it groups the two fields of `p` together into a single variable because it does not take into consideration the indirect memory operand `[eax+4]` in instruction `6`.

Typically, indirect operands are used to access arrays, fields of structures, fields of heap-allocated data, etc. Therefore, to recover a useful collection of variables from executables, one has to look beyond the explicitly occurring addresses and stack-frame offsets. Unlike the operands considered in the Semi-Naïve algorithm, local methods do not provide information about what an indirect memory operand accesses. For instance, an operand such as "[`ebp` − *offset*]" (usually) accesses a local variable. However, "`[eax + 4]`" may access a local variable, a global variable, a field of a heap-allocated data-structure, etc., depending upon what `eax` contains.

Obtaining information about what an indirect memory operand accesses is not straightforward. In this example, `eax` is initialized with the value of a register. In general, a register used in an indirect memory operand may be initialized with a value read from memory. In such cases, to determine the value of the register, it is necessary to know the contents of that memory location, and so on. Fortunately, Value-Set Analysis (VSA) described in [4, 24] (summarized in §4.1) can provide such information.

[2] We follow the convention that the value of `esp` (the stack pointer) at the beginning of a procedure marks the origin of the procedure's AR-region.

3.2 The Problem of Granularity and Expressiveness

The granularity and expressiveness of recovered variables can affect the precision of analysis clients that use the recovered variables as the executable's data objects.

Example 2. The program shown below initializes all elements of array `p`. The `x`-members of each element are initialized with 1; the `y`-members are initialized with 2. The disassembly is also shown. Instruction `L1` updates the `x`-members of the array elements; instruction `5` updates the `y`-members.

```
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
```

Fig. 2(a) shows how the variables are laid out in the AR of `main`. Note that there is no space for variable `i` in the AR for `main` because the compiler promoted `i` to register `ecx`. □

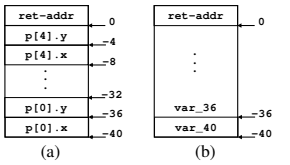

Fig. 2: AR of `main` for the program in Ex. 2: (a) actual layout, and (b) layout obtained from the Semi-Naïve approach.

As a specific example of an analysis client, consider a data-dependence analyzer, which answers such questions as: *"Does the write to memory at instruction `L1` affect the read from memory at instruction `10`"*. Note that in Ex. 2 the write to memory at instruction `L1` does not affect the read from memory at instruction `10` because `L1` updates the `x` members of the elements of array `p`, while instruction `10` reads the `y` member of array element `p[0]`. To simplify the discussion, assume that a data-dependence analyzer works as follows: (1) annotate each instruction with used, killed, and possibly-killed variables, and (2) compare the used variables of each instruction with killed or possibly-killed variables of every other instruction to determine data dependences.

Consider three different partitions of the AR of `main`:

$VarSet_1$: As shown in Fig. 2(b), the Semi-Naïve approach from §3.1 would say that the AR of `main` has two variables: `var_40` (4 bytes) and `var_36` (36 bytes). The variables that are possibly killed at `L1` are {`var_40`, `var_36`}, and the variable used at `10` is `var_36`. Therefore, the data-dependence analyzer reports that the write to memory at `L1` might affect the read at `10`. (This is sound, but imprecise.)

VarSet$_2$: As shown in Fig. 2(a), there are two variables for each element of array p. The variables possibly killed at L1 are {p[0].x, p[1].x, p[2].x, p[3].x, p[4].x}, and the variable used at instruction 10 is p[0].y. Because these sets are disjoint, the data-dependence analyzer reports that the memory write at instruction L1 definitely does not affect the memory read at instruction 10.

VarSet$_3$: Suppose that the AR of main is partitioned into just two (summary) variables: (1) p[?].x, which is a representative for the x members of the elements of array p, and (2) p[?].y, which is a representative for the y members of the elements of array p. The summary variable that is possibly killed at instruction L1 is p[?].x and the summary variable that is used at instruction 10 is p[?].y. These are disjoint; therefore, the data-dependence analyzer reports a definite answer, namely, that the write at L1 does not affect the read at 10.

Of the three alternatives presented above, VarSet$_3$ has several desirable features:

- It has a smaller number of variables than VarSet$_2$. When it is used as the set of variables in a data-dependence analyzer, it provides better results than VarSet$_1$.
- The variables in VarSet$_3$ are capable of representing a set of non-contiguous memory locations. For instance, p[?].x represents the locations corresponding to p[0].x, p[1].x, ..., p[4].x. The ability to represent non-contiguous sequences of memory locations is crucial for representing a specific field in an array of structures.
- The AR of main is only partitioned as much as necessary. In VarSet$_3$, only one summary variable represents the x members of the elements of array p, while each member of each element of array p is assigned a separate variable in VarSet$_2$.

A good variable-recovery algorithm should partition a memory-region in such a way that the set of variables obtained from the partition has the desirable features of VarSet$_3$. When debugging information is available, this is a trivial task. However, debugging information is often not available. Data-access patterns in the program provide information that can serve as a substitute for debugging information. For instance, instruction L1 accesses each of the four-byte sequences that start at offsets $\{-40, -32, \ldots, -8\}$ in the AR of main. The common difference of 8 between successive offsets is evidence that the offsets may represent the elements of an array. Moreover, instruction L1 accesses every four bytes starting at these offsets. Consequently, the elements of the array are judged to be structures in which the one of the fields is four bytes long.

4 Background

In this section, we describe (1) Value-Set Analysis (VSA) [4], and (2) Aggregate Structure Identification (ASI) [23]. This material is related to the core of the paper as follows:

- We use VSA as the mechanism to understand indirect memory accesses (see §4.1) and obtain data-access patterns (see §4.2) from the executable.
- In §5, we show how to use the information gathered during VSA to harness ASI to the problem of identifying variable-like entities in executables.

4.1 Value-Set Analysis (VSA)

VSA [4] is a combined numeric-analysis and pointer-analysis algorithm that determines an over-approximation of the set of numeric values or addresses that each register and

memory location holds at each program point. In particular, at each program point, VSA provides information about the contents of registers that appear in an indirect memory operand. 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. Moreover, unlike earlier algorithms that analyze executables [8, 11], VSA takes into account data manipulations involving memory locations also. To track the contents of memory locations, the initial run of VSA uses the variables recovered via the Semi-Naïve approach from §3.1.

For the program in Ex. 1, the initial run of VSA computes an over-approximation of the contents of the x86 registers (`eax`, `ax`, `ah`, `al`, `ebx`, etc.) and the memory-locations that correspond to `var_12` (4 bytes) and `var_8` (8 bytes). Similarly, for the program in Ex. 2, the initial run of VSA computes an over-approximation of the contents of the x86 registers and the memory-locations that correspond to `var_40` (4 bytes) and `var_36` (36 bytes). For both examples, the initial a-locs will be refined by our abstraction-refinement algorithm in §5. In the remainder of the paper, we overload the term "a-loc" both for the entities recovered by the Semi-Naïve algorithm (which are what we used in our previous work [4]), as well as for the entities identified by the abstraction-refinement algorithm of §5. (There should be no confusion, as it should always be clear from context which kind of a-loc is intended.)

VSA is a flow-sensitive, context-sensitive, interprocedural, abstract-interpretation algorithm (parameterized by call-string length [27]) that is based on an independent-attribute domain described below.

Call-Strings. The call-graph of a program is a labeled graph in which each node represents a procedure, each edge represents a call, and the label on an edge represents the call-site corresponding to the call represented by the edge. A call-string [27] is a sequence of call-sites $(c_1 c_2 \dots c_n)$ such that call-site c_1 belongs to the entry procedure, and there exists a path in the call-graph consisting of edges with labels c_1, c_2, ..., c_n. CallString is the set of all call-strings in the program.

A call-string suffix of length k is either $(c_1 c_2 \dots c_k)$ or $(*c_1 c_2 \dots c_k)$, where c_1, c_2, ..., c_k are call-sites. $(c_1 c_2 \dots c_k)$ represents the string of call-sites $c_1 c_2 \dots c_k$. $(*c_1 c_2 \dots c_k)$, which is referred to as a *saturated* call-string, represents the set $\{cs | cs \in \text{CallString}, cs = \pi c_1 c_2 \dots c_k, \text{and } |\pi| \geq 1\}$. CallString_k is the set of saturated call-strings of length k, plus non-saturated call-strings of length $\leq k$.

Value-Sets. During VSA, a set of numeric values and addresses is represented by a *value-set* that is a safe approximation of the actual set. Suppose that n is the number of memory-regions in the executable. A value-set is an 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 [24]. For a 32-bit machine, a strided-interval $s[l, u]$ represents the set of integers $\{i \in [-2^{31}, 2^{31} - 1] | l \leq i \leq u, i \equiv l (\texttt{mod } s)\}$.

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

For Ex. 2, 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 the empty set. For instance, the tuple $(1[0, 9], \emptyset)$ represents the set of numbers $\{0, 1, \dots, 9\}$

and the tuple $(\emptyset, 4[-40, -4])$ represents the set of offsets $\{-40, -36, \ldots, -4\}$ in the AR-region for `main`. (Although we refer to "tracking integer-valued and address-valued quantities simultaneously", the analysis makes no distinction between the two: values in the Global region could be either, and are treated appropriately according to what instruction is performed [4, 24].)

VSA Domain. Let Proc denote the set of memory-regions associated with procedures in the program; AllocMemRgn denote the set of memory-regions associated with heap-allocation sites;[3] Global denote the memory-region associated with the global data area; and a-loc[R] denote the a-locs that belong to memory-region `R`. We work with the following basic domains:

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

AbsEnv maps each region `R` to its corresponding AlocEnv[R] and each register to a ValueSet:

$$\begin{aligned} \mathsf{AbsEnv} = \; & (\mathsf{register} \rightarrow \mathsf{ValueSet}) \\ & \times (\{\mathsf{Global}\} \rightarrow \mathsf{AlocEnv[Global]}) \\ & \times (\mathsf{Proc} \rightarrow \mathsf{AlocEnv[Proc]}_{\perp}) \\ & \times (\mathsf{AllocMemRgn} \rightarrow \mathsf{AlocEnv[AllocMemRgn]}_{\perp}) \end{aligned}$$

VSA associates each program point with an AbsMemConfig:

$$\mathsf{AbsMemConfig} = (\mathsf{CallString}_{\mathsf{k}} \rightarrow \mathsf{AbsEnv}_{\perp})$$

In the above definitions, $\perp$ is used to denote a partial map. For instance, a ValueSet may not contain offsets in some memory-regions. Similarly, in AbsEnv, a procedure `P` whose activation record is not on the stack does not have an AlocEnv[`P`]. In addition to determining an over-approximation of the set of numeric values and addresses for each a-loc in the executable, VSA also finds a conservative estimate of the targets of indirect function-calls and indirect jumps—see [4]. Instead of describing VSA in detail, we highlight some of its features that are useful in a-loc recovery.

- *Information about indirect memory operands:* For the program in Ex. 1, VSA determines that the value-set of `eax` at instruction `6` is $(\emptyset, 0[-8, -8])$, which means that `eax` holds the offset -8 in the AR-region of `main`. Using this information, we can conclude that `[eax+4]` refers to offset -4 in the AR of `main`.
- *VSA provides data-access patterns:* For the program in Ex. 2, 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 of `main`. (These offsets are the starting addresses of field `x` of elements of array `p`.)

[3] The implementation actually uses an augmented abstract domain that overcomes some of the imprecision that arises due to the need to perform weak updates—i.e., accumulate information via join—on fields of summary malloc-regions. In particular, the augmented domain, which is described in [5], often allows our analysis to establish a definite link between a heap-allocated object of a class that uses 1 or more virtual functions and the appropriate virtual-function table. Due to space considerations, this aspect could not be described in the present paper. The results reported in §6 are based on the augmented domain.

– *VSA tracks updates to memory:* This is important because, in general, the registers used in an indirect memory operand may be initialized with a value read from memory. If updates to memory are not tracked, we may neither have useful information for indirect memory operands nor useful data-access patterns for the executable.

4.2 Aggregate Structure Identification (ASI)

ASI is a unification-based, flow-insensitive algorithm to identify the structure of aggregates in a program [23]. The algorithm ignores any type information known about 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 on how it is accessed by the program. The smaller parts are called *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. 3. 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. Note that the direction of data flow is not considered in a `UnifyConstraint`. The justification for this is that a flow of data from one sequence of bytes to another is evidence that 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[UInt:UInt] |
                    DataRef\UInt+
```

Fig. 3: Data-Access Constraint (DAC) language. `UInt` is the set of non-negative integers; $\texttt{UInt}_+$ is the set of positive integers; and `ProgVars` is the set of program variables.

There are three kinds of data references:

– A variable $\texttt{P} \in \texttt{ProgVar}$ 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 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 an element of array `DataRef`. For example, `P[0:11]`$\backslash 3$ 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 by means of an example. Consider the source-code program shown in Ex. 2. The data-access constraints for the program are

```
  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[4:7];
```

The constraints reflect the fact that the size of `Point` is `8` 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]`$\backslash$`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 the return value of `main` is assigned bytes `4..7` of `p`, which correspond to `p[0].y`.

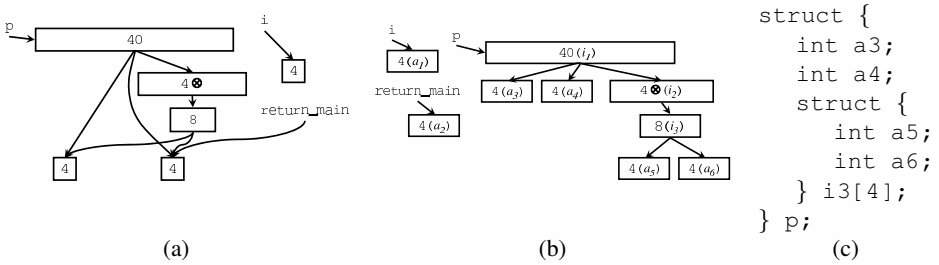

Fig. 4: (a) ASI DAG, (b) ASI tree, and (c) struct recovered for the program in Ex. 2.

The result of ASI is a DAG that shows the structure of each aggregate as well as relationships among the atoms of aggregates. The DAG for Ex. 2 is shown in Fig. 4(a). An ASI DAG has the following properties:

- A node represents a set of bytes.
- A sequence of bytes that is accessed as an array in the program is represented by an *array* node. Array nodes are labeled with $\bigotimes$. The number in an 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. 4(a), bytes `8..39` of array `p` are identified as an array of four 8-byte elements. Each array element is a struct with two fields of 4 bytes each.
- A sequence of bytes that is accessed like a C struct in the program is represented by a *struct* node. The number in the struct node represents the length of the struct; the children of a struct node represent the fields of the struct. In Fig. 4(a), bytes `0..39` of `p` are identified as a struct with three fields: two 4-byte scalars and one 32-byte 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. 4(a), the nodes for the sequences of bytes `return_main[0:3]` and `p[4:7]` 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[4:7]`, share the same node because they are all assigned the same constant at the same instruction.

The ASI DAG is converted into an ASI tree by duplicating shared nodes. The atoms of an aggregate are the leaves of the corresponding ASI tree. Fig. 4(b) shows the ASI tree for Ex. 2. ASI has identified that `p` has the structure shown in Fig. 4(c).

5 Recovering A-locs via Iteration

We use the atoms obtained from ASI as a-locs for (re-)analyzing the executable. The atoms identified by ASI for Ex. 2 are close to the set of variables $\texttt{VarSet}_3$ that was discussed in §3.2. One might hope to apply ASI to an executable by treating each memory-region as an aggregate and determining the structure of each memory-region (without using VSA results). However, 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 programs written in languages such as Cobol this is possible: the data-access patterns are apparent from the syntax of the constructs under consideration. Unfortunately, this is not the case for executables. For instance, the memory operand `[eax]` can either represent an access to a single variable or to the elements of an array. 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. Together with the information about the number of bytes accessed by each argument (which is available from the instruction), this provides the information needed to generate data-access constraints for the executable.

Furthermore, when we use the atoms of ASI as a-locs in VSA, the results of VSA can improve. Consider the program in Ex. 1. Recall from §3.1 that the length of `var_8` is 8 bytes. Because value-sets are only capable of representing a set of 4-byte addresses and 4-byte values, VSA recovers no useful information for `var_8`: it merely reports that the value-set of `var_8` is $\top$ (meaning any possible value or address). Applying ASI (using data-access patterns provided by VSA) results in the splitting of `var_8` into two 4-byte a-locs, namely, `var_8.0` and `var_8.4`. Because `var_8.0` and `var_8.4` are each four bytes long, VSA can now track the set of values or addresses in these a-locs. Specifically, VSA would determine that `var_8.0` (i.e., `p.x`) has the value `1` and `var_8.4` (i.e., `p.y`) has the value `2` at the end of `main`.

We can use the new VSA results to perform another round of ASI. If the value-sets computed by VSA are improved from the previous round, the next round of ASI may also improve. We can repeat this process as long as desired, or until the process converges (see §5.4).

Although not illustrated by Ex. 1, additional rounds of ASI and VSA can result in further improvements. For example, suppose that the program uses a chain of pointers to link `structs` of different types, e.g., variable `ap` points to a `struct A`, which has a field `bp` that points to a `struct B`, which has a field `cp` that points to a `struct C`, and so on. Typically, the first round of VSA recovers the value of `ap`, which lets ASI discover the a-loc for `A.bp` (from the code compiled for `ap->bp`); the second round of VSA recovers the value of `ap->bp`, which lets ASI discover the a-loc for `B.cp` (from the code compiled for `ap->bp->cp`); etc.

To summarize, the algorithm for recovering a-locs 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.[4]

It is important to understand that VSA generates sound results for *any* collection of a-locs with which it is supplied. However, if supplied very coarse a-locs, many a-locs will be found to have the value $\top$ at most points. By refining the a-locs in use, more precise answers are generally obtained. For this reason, 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. §5.5 discusses some of these cases.

In short, our abstraction-refinement principles are as follows:

1. VSA results are used to interpret memory-access expressions in the executable.

[4] Or, equivalently, until the set of a-locs discovered in step 3 is unchanged from the set previously discovered in step 3 (or step 1).

2. ASI is used as a heuristic to determine the structure of each memory-region according to information recovered by VSA.
3. Each ASI tree reflects the memory-access patterns in one memory-region, and the leaves of the ASI trees define the a-locs that are used for the next round of VSA.

ASI alone is not a replacement for VSA. That is, ASI cannot be applied to executables without the information that is obtained from VSA—namely value-sets.

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 as a-locs during the next round of VSA.

5.1 Generating Data-Access Constraints

This section describes the algorithm that generates ASI data-references for x86 operands. Three forms of x86 operands need to be considered: (1) register operands, (2) memory operands of form "[*register*]", and (3) memory operands of the form "[*base* + *index* $\times$ *scale* + *offset*]".

To prevent unwanted unification during ASI, we rename registers using live-ranges. For a register `r`, the ASI data-reference is $\texttt{r}_{lr}[0 : n-1]$, where *lr* is the live-range of the register at the given instruction and n is the size of the register (in bytes).

In the rest of the section, we describe the algorithm for memory operands. First, we consider indirect operands of the form `[r]`. To gain intuition about the algorithm, consider operand `[eax]` of instruction `L1` in Ex. 2. The value-set associated with `eax` is $(\emptyset, 8[-40, -8])$. The stride value of 8 and the interval $[-40, -8]$ in the AR of `main` provide evidence that `[eax]` is an access to the elements of an array of 8-byte elements in the range $[-40, -8]$ of the AR of `main`; an array access is generated for this operand.

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 *length* (the number of bytes accessed) 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. 2 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 later in this section. 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 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* as 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 $\lfloor (u - l)/\textit{size} \rfloor + 1$.

For operands of the form [`r`], the set of ASI references is generated by invoking Alg. 1 for each non-empty memory-region in `r`'s value-set. For Ex. 2, the value-set associated with `eax` at `L1` is $(\emptyset, 8[-40, -8])$. Therefore, the set of ASI references is

{AR_main[(-40):(-1)]\5[0:3]}.[5] There are no references to the `Global` region because the set of offsets in that region is empty.

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 $\langle$ "$r[l : l + \mathit{length} - 1]$", true$\rangle$
else
  $size \leftarrow \max(s, \mathit{length})$
  $n \leftarrow \lfloor (u - l)/size \rfloor + 1$
  ref $\leftarrow$ "$r[l : u + size - 1] \backslash n[0 : length - 1]$"
  return $\langle$ref, $(s < length)\rangle$
end if

The algorithm for converting indirect operands of the form [*base* + *index* $\times$ *scale* + *offset*] is given in Alg. 2. One typical use of indirect operands of the form [*base* + *index* $\times$ *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 *base*) holds the starting addresses of the rows and the other register (usually *index*) holds the indices of the elements in the row. 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 have used Alg. 1 by computing the abstract sum ($+^{si}$) of the two strided intervals. However, doing so results in a loss of precision because strided intervals can only represent a single stride exactly, and this would prevent us from recovering the structure of two-dimensional arrays. (In some circumstances, our implementation of ASI can recover the structure of arrays of 3 and higher dimensions.)

Alg. 2 works as follows: First, it determines which of the two strided intervals is used as the *base* because it is not always apparent from the representation of the operand. The strided interval that is used as the *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*. In some cases, the algorithm cannot establish either of the strided intervals as the base. In such cases, the algorithm computes the abstract sum ($+^{si}$) of the two strided intervals and invokes `SI2ASI`.

Alg. 2 generates a richer set of ASI references than Alg. 1. For example, consider the indirect memory operand `[eax+ecx]` from a loop that traverses a two-dimensional array of type `char`[5][10]. Suppose that the value-set of `ecx` is $(\emptyset, 10[-50, -10])$, the

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

value-set of `eax` is $(1[0,9], \emptyset)$, and *length* is 1. For this example, the ASI reference that is generated is "`AR[-50:-1]\5[0:9]\10[0:0]`". That is, AR is accessed as an array of five 10-byte entities, and each 10-byte entity is accessed as an array of ten 1-byte entities.

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 a singleton) **then**
  return `SI2ASI`(r, $s_1[l_1, u_1] +^{si} s_2[l_2, u_2]$, *length*)
end if
if $s_1 \geq (u_2 - l_2 + \mathit{length})$ **then**
  baseSI $\leftarrow s_1[l_1, u_1]$
  indexSI $\leftarrow s_2[l_2, u_2]$
else if $s_2 \geq (u_1 - l_1 + \mathit{length})$ **then**
  baseSI $\leftarrow s_2[l_2, u_2]$
  indexSI $\leftarrow s_1[l_1, u_1]$
else
  return `SI2ASI`(r, $s_1[l_1, u_1] +^{si} s_2[l_2, u_2]$, $size$)
end if
$\langle$baseRef, exactRef$\rangle \leftarrow$ `SI2ASI`(r, baseSI, stride(baseSI))
if exactRef is false **then**
  return `SI2ASI`(r, $s_1[l_1, u_1] +^{si} s_2[l_2, u_2]$, *length*)
else
  return `concat`(baseRef, `SI2ASI`("", indexSI, *length*))
end if

5.2 Interpreting Indirect Memory-References

This section describes a lookup algorithm that finds the set of a-locs accessed by a memory operand. The algorithm is used to interpret pointer-dereference operations during VSA. For instance, consider the 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 [4], the algorithm to determine the set of a-locs for a given value-set is trivial because each memory-region in [4] consists of a linear list of a-locs generated by the Semi-Naïve approach. However, after ASI is performed, the structure of each memory-region is an ASI tree.

In [23], 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 §5.5, some data-access patterns that arise during VSA should be ignored during ASI.

– 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, which is based on a-locs recovered by the Semi-Naïve approach, will not include accesses to the fields of structures. However, the first round of VSA may access structure fields.

We will use the tree shown in Fig. 4(b) to describe the lookup algorithm. Every node in the tree is given a unique name (shown within parentheses). The following terms are used in describing the lookup algorithm:

– `NodeFrag` is a descriptor for a part of an ASI tree node and is denoted by a triple $\langle name, start, \textit{length}\rangle$, where $name$ is the name of the ASI tree node, $start$ is the starting offset within the ASI tree node, and *length* is the length of the fragment.
– `NodeFragList` is an ordered list of `NodeFrag` descriptors, [nd_1, nd_2, ..., nd_n]. A `NodeFragList` represents a contiguous set of offsets in an aggregate. For example, [$\langle a_3, 2, 2\rangle$, $\langle a_4, 0, 2\rangle$] represents the offsets `2..5` of node i_1; offsets `2..3` come from $\langle a_3, 2, 2\rangle$ and offsets `4..5` come from $\langle a_4, 0, 2\rangle$.

The lookup algorithm traverses 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 §5.1, 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 `NodeFragList` as argument and returns a set of `NodeFragList` values. The operations are performed from left to right. The argument of each operation comes from the result 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 `NodeFrag` descriptors.

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

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

`GetArrayElements(`m`)` treats the given `NodeFragList` as an array of m elements and returns a set of `NodeFragList` lists. Each `NodeFragList` list represents an array element. There can be more than one `NodeFragList` 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 illustrate traces of a few lookups.

Example 3. Lookup `p[0:3]`

$$[\langle i_1, 0, 40\rangle]$$
`GetChildren(p)` $\Downarrow$
$$[\langle a_3, 0, 4\rangle, \langle a_4, 0, 4\rangle, \langle i_2, 0, 32\rangle]$$
`GetRange(0,3)` $\Downarrow$
$$[\langle a_3, 0, 4\rangle]$$

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

Example 4. Lookup `p[0:39]\5[0:3]`

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

Let us look at `GetArrayElements(5)` because the other operations are similar to Ex. 3. `GetArrayElements(5)` is applied to $[\langle a_3, 0, 4\rangle, \langle a_4, 0, 4\rangle, \langle i_2, 0, 32\rangle]$. The total length of the given `NodeFragList` is 40 and the number of required array elements is 5. Therefore, the size of the array element is 8. Intuitively, the operation unrolls the given `NodeFragList` and creates a `NodeFragList` for every unique n-byte sequence starting from the left, where n is the length of the array element. In this example, the unrolled `NodeFragList` is $[\langle a_3, 0, 4\rangle, \langle a_4, 0, 4\rangle, \langle a_5, 0, 4\rangle, \langle a_6, 0, 4\rangle, \ldots, \langle a_5, 0, 4\rangle, \langle a_6, 0, 4\rangle]$. The set of unique 8-byte `NodeFragLists` has two ordered lists: $\{[\langle a_3, 0, 4\rangle, \langle a_4, 0, 4\rangle], [\langle a_5, 0, 4\rangle, \langle a_6, 0, 4\rangle]\}$.□

Partial updates to a-locs. The abstract transformers in VSA are prepared to perform partial updates to a-locs (i.e., updates to *parts* of an a-loc) because `NodeFrag` elements in a `NodeFragList` may refer to parts of an ASI tree node. Consider "`p[0:1] = 0x10`".[6] The lookup operation for `p[0:1]` returns $[\langle a_3, 0, 2\rangle]$, where $\langle a_3, 0, 2\rangle$ refers to the first two bytes of a_3. An abstract transformer that "gives up" (because only part of a_3 is affected) and sets the value-set of a_3 to $\top$ in such cases would lead to imprecise results.

The value-set domain (see §4.1, [24]) provides bit-wise operations such as bit-wise and ($\&^{vs}$), bit-wise or ($|^{vs}$), left shift ($\ll^{vs}$), right shift ($\gg^{vs}$), etc. We use these operations to adjust the value-set associated with an a-loc when a partial update has to be performed during VSA. Assuming that the underlying architecture is little-endian, the abstract transformer for "`p[0:1] = 0x10`" updates the value-set associated with a_3 as follows:

$\mathsf{ValueSet}'(a_3) = (\mathsf{ValueSet}(a_3)\ \&^{vs}\ \text{0xffff0000})\ |^{vs}\ (\text{0x10})$.

5.3 Hierarchical A-locs

The iteration of ASI and VSA can over-refine the memory-regions. For instance, suppose that the 4-byte a-loc a_3 in Fig. 4(b) used in some round i is partitioned into two 2-byte a-locs, namely, $a_3.0$, and $a_3.2$ in round $i+1$. This sort of over-refinement can affect the results of VSA; in general, because of the properties of strided-intervals, a 4-byte value-set reconstructed from two adjacent 2-byte a-locs can be less precise than if the information was retrieved from a 4-byte a-loc. For instance, suppose that at some instruction `S`, a_3 holds either 0x100000 or 0x110001. In round i, this information is

[6] Numbers that start with "0x" are in C hexadecimal format.

exactly represented by the 4-byte strided interval 0x10001[0x100000, 0x110001] for a_3. On the other hand, the same set of numbers can only be over-approximated by two 2-byte strided intervals, namely, 1[0x0000, 0x0001] for $a_3.0$, and 0x1[0x10,0x11] for $a_3.2$ (for a little-endian machine). Consequently, if a 4-byte read of a_3 in round $i+1$ is handled by reconstituting a_3's value from $a_3.0$ and $a_3.2$, the result would be less precise:

$$\begin{aligned}\mathsf{ValueSet}(a_3) &= (\mathsf{ValueSet}(a_3.2) \ll^{vs} 16)|^{vs}\mathsf{ValueSet}(a_3.0)\\ &= \{0\text{x}100000, 0\text{x}100001, 0\text{x}110000, 0\text{x}110001\}\\ &\supset \{0\text{x}100000, 0\text{x}110001\}.\end{aligned}$$

We avoid the effects of over-refinement by keeping track of the value-sets for a-loc a_3 as well as a-locs $a_3.0$ and $a_3.2$ in round $i+1$. Whenever any of a_3, $a_3.0$, and $a_3.2$ is updated during round $i+1$, the overlapping a-locs are updated as well. For example, if $a_3.0$ is updated then the first two bytes of the value-set of a-loc a_3 are also updated (for a little-endian machine). For a 4-byte read of a_3, the value-set returned would be 0x10001[0x100000, 0x110001].

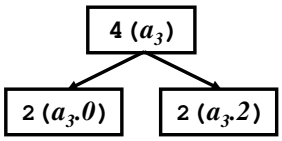

Fig. 5: Hierarchical a-locs

In general, if an a-loc a of length ≤ 4 gets partitioned into a sequence of a-locs $[a_1, a_2, \ldots, a_n]$ during some round of ASI, in the subsequent round of VSA, we use a as well as $\{a_1, a_2, \ldots, a_n\}$. We also remember the parent-child relationship between a and the a-locs in $\{a_1, a_2, \ldots, a_n\}$ so that we can update a whenever any of the a_i is updated during VSA and vice versa. In our example, the ASI tree used for round $i+1$ of VSA is identical to the tree in Fig. 4(b), except that the node corresponding to a_3 is replaced with the tree shown in Fig. 5.

One of the sources of over-refinement is the use of union types in the program. The use of hierarchical a-locs allows at least some degree of precision to be retained in the presence of unions.

5.4 Convergence

The first round of VSA uncovers memory accesses that are not explicit in the program, which allows ASI to refine the a-locs for the next round of VSA, which may produce more precise value-sets because it is based on a better set of a-locs. Similarly, subsequent rounds of VSA can uncover more memory accesses, and hence allow ASI to refine the a-locs. The refinement of a-locs cannot go on indefinitely because, in the worst case, an a-loc can only be partitioned into a sequence of 1-byte chunks. However, in most cases, the refinement process converges before the worst-case partitioning occurs. Also, the set of targets that VSA determines for indirect function-calls and indirect jumps may change when the set of a-locs (and consequently, their value-sets) changes between successive rounds. This process cannot go on indefinitely because the set of a-locs cannot change between successive rounds forever. Therefore, the iteration process converges when the set of a-locs, and the set of targets for indirect function calls and indirect jumps does not change between successive rounds.

5.5 Pragmatics

ASI takes into account the 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:

- VSA might obtain a very conservative estimate for the value-set of a register (say `R`). For instance, 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, some options are provided to tune the analysis:

- The user can supply an integer threshold. If the number of memory locations that are accessed by a memory operand is above the threshold, no ASI reference is generated.
- The user can supply a set of instructions for which ASI references should not be generated. One possible use of this option is to suppress `memcpy`-like instructions.
- The user can supply explicit references to be used during ASI.

In our experiments, we only used the integer-threshold option (which was set to 500).

6 Experiments

In this section, we present the results of our preliminary experiments, which were designed to answer the following questions:

1. How do the a-locs identified by abstraction refinement compare with the program's debugging information? This provides insight into the usefulness of the a-locs recovered by our algorithm for a human analyst.
2. How much more useful for static analysis are the a-locs recovered by an abstract-interpretation-based technique when compared to the a-locs recovered by purely local techniques?

6.1 Comparison of A-locs with Program Variables

To measure the quality of the a-locs identified by the abstraction-refinement algorithm, we used a set of C++ benchmarks collected from [1] and [22]. The characteristics of the benchmarks are shown in Tab. 1. The programs in Tab. 1 make heavy use of inheritance and virtual functions, and hence are a challenging set of examples for the algorithm.

We compiled the set of programs shown in Tab. 1 using the Microsoft VC 6.0 compiler with debugging information, and ran the a-loc recovery algorithm on the executables produced by the compiler until the results converged. After each round of ASI, for each program variable `v` present in the debugging information, we compared `v` with the structure identified by our algorithm (which did *not* use the debugging information), and classified `v` into one of the following categories:

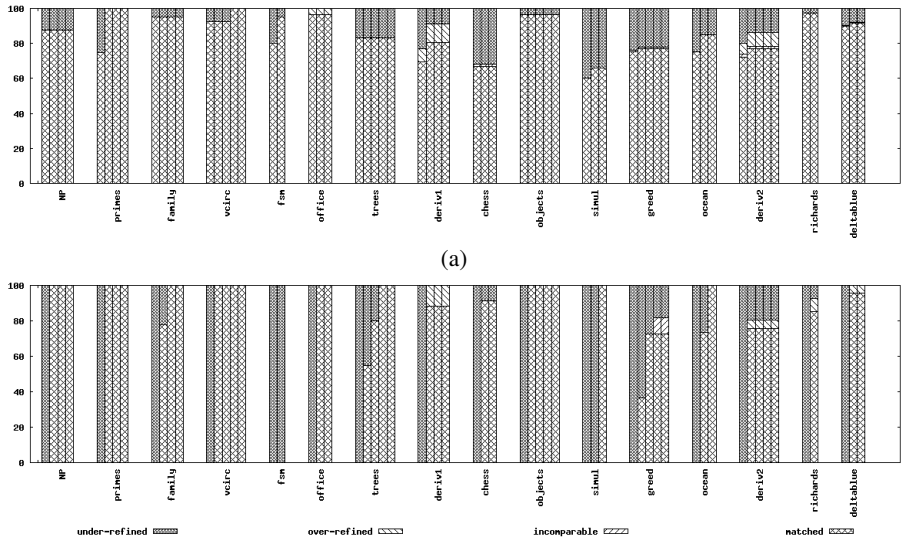

Fig. 6: Breakdown (as percentages) of how a-locs matched with program variables: (a) local variables, and (b) fields of heap-allocated data-structures.

- Variable `v` is classified as *matched* if the a-loc-recovery algorithm correctly identified the size and the offsets of `v` in the corresponding memory-region.
- Variable `v` is classified as *over-refined* if the a-loc-recovery algorithm partitioned `v` into smaller a-locs. For instance, a 4-byte `int` that is partitioned into an array of four `char` elements is classified as over-refined.
- Variable `v` is *under-refined* if the a-loc-recovery algorithm identified `v` to be a part of a larger a-loc. For instance, if the algorithm failed to partition a struct into its constituent fields, the fields of the struct are classified as under-refined.
- Variable `v` is classified as *incomparable* if `v` does not fall into one of the above categories.

The results of the classification process for the local variables and fields of heap-allocated data structures are shown in Fig. 6(a) and Fig. 6(b), respectively. The leftmost column for each program shows the results for the a-locs recovered using the Semi-Naïve approach, and the rightmost bar shows the results for the final round of the abstraction-refinement algorithm.

On average, our technique is successful in identifying correctly over 88% of the local variables and over 89% of the fields of heap-allocated objects (and was 100% correct for fields of heap-allocated objects in almost half of the examples). In contrast, the Semi-Naïve approach recovered 83% of the local variables, but 0% of the fields of heap-allocated objects.

Fig. 6(a) and Fig. 6(b) show that for some programs the results improve as more rounds of analysis are carried out. In most of the programs, only one round of ASI was required to identify all the fields of heap-allocated data structures correctly. In some of the programs, however, it required more than one round to find all the fields of heap-allocated data-structures. Those programs that required more than one round of ASI-

VSA iteration used a chain of pointers to link structs of different types, as discussed in §5.

	Insts	Procs	Mallocs
NP	252	5	2
primes	294	9	1
family	351	9	6
vcirc	407	14	1
fsm	502	13	1
office	592	22	4
trees	1299	29	10
deriv1	1369	38	16
chess	1662	41	24
objects	1739	47	5
simul	1920	60	2
greed	1945	47	1
ocean	2552	61	13
deriv2	2639	41	58
richards	3103	74	23
deltablue	5371	113	26

Table 1: C++ Examples

Most of the example programs do not have structures that are declared local to a procedure. This is the reason why the Semi-Naïve approach identified a large fraction of the local variables correctly. The programs `primes` and `fsm` have structures that are local to a procedure. As shown in Fig. 6(a), our approach identifies more local variables correctly for these examples.

6.2 Usefulness of the A-locs for Static Analysis

The aim of this experiment was to evaluate the quality of the variables and values discovered as a platform for performing additional static analysis. In particular, because resolution of indirect operands is a fundamental primitive that essentially any subsequent analysis would need, the experiment measured how well we can resolve indirect memory operands not based on global address or stack-frame offsets (e.g., accesses to arrays and heap-allocated data objects). We ran several rounds of VSA on the collection of commonly used Windows executables listed in Tab. 2, as well as the set of benchmarks from Tab. 1. For the programs in Tab. 1, we ran VSA-ASI iteration until convergence. For the programs in Tab. 2, we limited the number of VSA-ASI rounds to at most three. Round 1 of VSA performs its analysis using the a-locs recovered by the Semi-Naïve approach; the final round of VSA uses the a-locs recovered by the abstraction-refinement algorithm. After the first and final rounds of VSA, we labeled each memory operand as follows:

- A memory operand is *untrackable* if the size of all the a-locs accessed by the memory operand is greater than 4 bytes, or if the value-set associated with the address expression of the memory operand is $\top$.
- A memory operand is *weakly-trackable* if the size of *some* a-loc accessed by the memory operand is less than or equal to 4 bytes, and the value-set associated with the address expression of the memory operand is not $\top$.
- A memory operand is *strongly-trackable* if the size of *all* the a-locs accessed by the memory operand is less than or equal to 4 bytes, and the value-set associated with the address expression of the memory operand is not $\top$.

Recall that VSA can track value-sets for a-locs that are less than or equal to 4 bytes, but reports that the value-set for a-locs greater than 4 bytes is $\top$. Therefore, untrackable memory operands are the ones for which VSA provides no useful information at all, and strongly-trackable memory operands are the ones for which VSA definitely provides useful information. For a weakly-trackable memory operand, VSA provides some useful information if the operand is used to update the contents of memory; however, no useful information is obtained if the operand is used to read the contents of memory.

For instance, if `[eax]` in "`mov [eax], 10`" is weakly-trackable, then VSA would have updated the value-set for those a-locs that were accessed by `[eax]` and were of size less than or equal to 4 bytes. However, if `[eax]` in "`mov ebx, [eax]`" is weakly-trackable, the value-set of `ebx` is set to $\top$ because at least one of the a-locs accessed by `[eax]` is $\top$; this situation is not different from the case when `[eax]` is untrackable. We refer to a memory operand that is used to read the contents of memory as a *use-operand*, and a memory operand that is used to update the contents of memory as a *kill-operand*.

	Insts	Procs	Mallocs	n	Time
mplayer2	14270	172	0	2	0h 11m
smss	43034	481	0	3	2h 8m
print	48233	563	17	3	0h 20m
doskey	48316	567	16	3	2h 4m
attrib	48785	566	17	3	0h 23m
routemon	55586	674	6	3	2h 28m
cat	57505	688	24	3	0h 54m
ls	60543	712	34	3	1h 10m

Table 2: Windows Executables. (n is the number of VSA-ASI rounds.)

In Tab. 3, the "Weakly-Trackable Kills" column shows the fraction of kill-operands that were weakly-trackable during the first and final rounds of the abstraction refinement algorithm, and the "Strongly-Trackable Uses" column shows the fraction of use-operands that were strongly-trackable during the first and final round of the algorithm. In the table, we have classified memory operands as either *direct* or *indirect*. A *direct* memory operand is a memory operand that uses a global address or stack-frame offset. An *indirect* memory operand is a memory operand that does not use a global address or a stack-frame offset (e.g., a memory operand that accesses an array or a heap-allocated data object).

Both the Semi-Naïve approach and our abstract-interpretation-based a-loc-recovery algorithm provide good results for direct memory operands. However, the results for indirect memory operands are substantially better with the abstraction-interpretation-based method. For the set of C++ programs from Tab. 1, the results of VSA improve at 50% to 100% of the indirect kill-operands, and at 7% to 100% of the indirect use-operands. Similarly, for the Windows executables from Tab. 2, the results of VSA improve at 4% (`routemon`: $7\% \rightarrow 11\%$) to 39% (`mplayer2`: $12\% \rightarrow 51\%$) of the indirect kill-operands, and up to 8% (`attrib`, `print`: $4\% \rightarrow 12\%$, $6\% \rightarrow 14\%$) of the indirect use-operands.

We were surprised to find that the Semi-Naïve approach was able to provide a small amount of useful information for indirect memory operands. For instance, `trees`, `greed`, `ocean`, `deltablue`, and all the Windows executables have a non-zero percentage of trackable memory operands. On closer inspection, we found that these indirect memory operands access local or global variables that are also accessed directly elsewhere in the program. (In source-level terms, the variables are accessed both directly and via pointer indirection.) For instance, a local variable `v` of procedure `P` that is passed by reference to procedure `Q` will be accessed directly in `P` and indirectly in `Q`.

Several sources of imprecision in VSA prevent us from obtaining useful information at all of the indirect memory operands. One such source of imprecision is widening [10]. VSA uses a widening operator during abstract interpretation to accelerate fixpoint computation. Due to widening, VSA may fail to find non-trivial bounds for registers that are used as indices in indirect memory operands. These indirect memory operands are labeled as untrackable. The fact that the VSA domain is non-relational amplifies this

problem. (To a limited extent, we overcome the lack of relational information by obtaining relations among x86 registers from an additional analysis called affine-relation analysis. See §5 in [4] for details.) Note that the widening problem is orthogonal to the issue of finding the correct set of variables. Even if our a-loc recovery algorithm recovers all the variables correctly, imprecision due to widening persists. (Recently, using ideas from [7] and [13], we have implemented techniques to reduce the undesirable effects of widening, but do not yet have numbers to report.)

		Weakly-Trackable Kills (%)				Strongly-Trackable Uses (%)			
		Indirect		Direct		Indirect		Direct	
Round		1	n	1	n	1	n	1	n
NP	(4)	**0**	**100**	100	100	**0**	**100**	100	100
primes	(4)	**0**	**100**	100	100	0	83	100	100
family	(4)	**0**	**100**	100	100	**0**	**100**	100	100
vcirc	(5)	**0**	**100**	100	100	**0**	**100**	100	100
fsm	(2)	***0***	***50***	100	100	0	29	98	100
office	(3)	**0**	**100**	100	100	**0**	**100**	100	100
trees	(5)	10	100	98	100	25	61	96	100
deriv1	(4)	**0**	**100**	97	99	0	77	98	98
chess	(3)	0	60	99	99	0	25	100	100
objects	(5)	**0**	**100**	100	100	0	94	100	100
simul	(3)	**0**	**100**	71	100	0	38	57	100
greed	(5)	***3***	***53***	99	100	3	10	98	98
ocean	(3)	9	90	99	100	6	42	98	100
deriv2	(5)	**0**	**100**	100	100	0	97	95	100
richards	(2)	0	68	100	100	***0***	***7***	99	99
deltablue	(3)	1	57	99	100	0	16	99	99
mplayer2	(2)	**12**	**51**	89	97	***8***	***8***	89	92
smss	(3)	9	19	92	98	1	4	84	90
print	(3)	2	22	92	99	**6**	**14**	89	92
doskey	(3)	2	17	92	97	5	7	79	86
attrib	(3)	7	24	93	98	**4**	**12**	86	90
routemon	(3)	***7***	***11***	93	97	1	2	81	86
cat	(3)	12	22	93	97	1	4	79	84
ls	(3)	11	23	94	98	1	4	84	88

Table 3: Fraction of memory operands that are trackable after VSA. The number in parenthesis shows the number of rounds (n) of VSA-ASI iteration for each executable. (For Windows executables, the maximum number of rounds was set to 3.) **Boldface** and ***bold-italics*** in the Indirect columns indicate the maximum and minimum improvements, respectively.

Nevertheless, the results are encouraging. For the Windows executables, the number of memory operands that have useful information in round n is 2 to 4 *times* the number of memory operands that have useful information in round 1; i.e., the results of static analysis do significantly improve when a-locs recovered by the abstraction-interpretation-based algorithm are used in the place of a-locs recovered from purely local techniques. Our initial experiments show that the techniques are also feasible in terms of running time.

7 Related Work

In [18], Miné describes a combined data-value and points-to analysis that, at each program point, partitions the variables in the program into a collection of cells according to how they are accessed, and computes an over-approximation of the values in these cells. Miné's algorithm is similar in flavor to the VSA-ASI iteration scheme in that Miné finds his own variable-like quantities for static analysis. However, Miné's partitioning algorithm is still based on the set of variables in the program (which our algorithm assumes will not be available). His implementation does not support analysis of programs that use heap-allocated storage. Moreover, his techniques are not able to infer from loop access patterns—as ASI can—that an unstructured cell (e.g., `unsigned char z[32]` has internal array substructures, (e.g., `int y[8];` or `struct {int a[3]; int b;} x[2];`).

In [18], cells correspond to variables. The algorithm assumes that each variable is disjoint and is not aware of the relative positions of the variables. Instead, his algorithm issues an alarm whenever an indirect access goes beyond the end of a variable. Because our abstraction of memory is in terms of memory-regions (which can be thought of as cells for entire activation records), we are able to interpret an out-of-bound access precisely in most cases. For instance, suppose that two integers `a` and `b` are laid out next to each other. Consider the sequence of C statements "`p = &a; *(p+1) = 10;`". For the access `*(p+1)`, Miné's implementation issues an out-of-bounds access alarm, whereas we are able to identify that it is a write to variable `b`. (Such out-of-bounds accesses occur commonly during VSA because the a-loc-recovery algorithm can split a single source-level variable into more than one a-loc, e.g., array `p` in Ex. 2.)

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 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 [8] 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. [2] and the algorithm for pointer analysis on a low-level intermediate representation of Guo et al. [14]. 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. [14] 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 [31] 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.

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 [29] and EEL [17]. Bergeron et al. [6] present a static-analysis technique to check if an executable with debugging information adheres to a user-specified security policy.

Rival [26] 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. Aggregate structure identification was devised by Ramalingam et al. to partition aggregates according to a Cobol program's memory-access patterns [23]. 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 [20] 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. However, Mycroft's algorithm has several weaknesses. For instance, 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) [4, 16, 19], 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 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.

Decompilation. Past work on decompiling assembly code to a high-level language [9] is also peripherally 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 [9] primarily concentrates on recovery of (a) expressions from instruction sequences, and (b) control flow. We believe that decompilers would benefit from the memory-access-analysis method described in this paper, which 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 [5].

References

1. G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. In *European Conf. on Object-Oriented Programming*, 1996.
2. W. Amme, P. Braun, E. Zehendner, and F. Thomasset. Data dependence analysis of assembly code. *Int. J. Parallel Proc.*, 2000.
3. W. Backes. *Programmanalyse des XRTL Zwischencodes*. PhD thesis, Universitaet des Saarlandes, 2004. (In German.).
4. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Comp. Construct.*, 2004.
5. G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *SAS*, 2006.
6. 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.
7. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Int. Conf. on Formal Methods in Prog. and their Appl.*, 1993.
8. C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *ICSM*, pages 188–195, 1997.
9. C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation. In *ICSM*, 1998.
10. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
11. S.K. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *POPL*, 1998.
12. P.H. Eidorff, F. Henglein, C. Mossin, H. Niss, M.H. Sørensen, and M. Tofte. Anno Domini: From type theory to year 2000 conversion tool. In *POPL*, 1999.
13. D. Gopan and T. Reps. Lookahead widening. In *CAV*, 2006.
14. B. Guo, M.J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D.I. August. Practical and accurate low-level pointer analysis. In *Int. Symp. on Code Gen. and Opt.*, 2005.
15. IDAPro disassembler, http://www.datarescue.com/idabase/.
16. A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *CAV*, 2005.
17. J.R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *PLDI*, 1995.
18. A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *LCTES*, 2006.
19. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *ESOP*, 2005.
20. A. Mycroft. Type-based decompilation. In *ESOP*, 1999.
21. R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *Int. Conf. on Softw. Eng.*, 1997.
22. H. Pande and B. Ryder. Data-flow-based virtual function resolution. In *SAS*, 1996.
23. G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *POPL*, 1999.
24. T. Reps, G. Balakrishnan, and J. Lim. Intermediate representation recovery from low-level code. In *PEPM*, 2006.
25. T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum. A next-generation platform for analyzing executables. In *APLAS*, 2005.
26. X. Rival. Abstract interpretation based certification of assembly code. In *VMCAI*, 2003.
27. 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.
28. M. Siff and T.W. Reps. Program generalization for software reuse: From C to C++. In *Found. of Softw. Eng.*, 1996.
29. A. Srivastava and A. Eustace. ATOM - A system for building customized program analysis tools. In *PLDI*, 1994.
30. A. van Deursen and L. Moonen. Type inference for COBOL systems. In *WCRE*, 1998.
31. R.P. Wilson and M.S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, 1995.