Computer Sciences Department

Analyzing Memory Accesses in x86 Binary Executables

Gogul Balakrishnan
Thomas Reps

Technical Report #1486

July 2003

UNIVERSITY OF
WISCONSIN
MADISON

Analyzing Memory Accesses in x86 Executables

Gogul Balakrishnan and Thomas Reps

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

Abstract. This paper concerns static-analysis algorithms for analyzing x86 executables. The aim of the work is to recover intermediate representations that are similar to those that can be created for a program written in a high-level language. Our goal is to perform this task for programs such as plugins, mobile code, worms, and virus-infected code. For such programs, symbol-table and debugging information is either entirely absent, or cannot be relied upon if present; hence, the technique described in the paper makes no use of symbol-table/debugging information. Instead, an analysis is carried out to recover information about the contents of memory locations and how they are manipulated. The analysis, called *value-set analysis*, tracks address-valued and integer-valued quantities simultaneously.

1 Introduction

In recent years, there has been a growing need for tools that analyze executables. One would like to ensure that web-plugins, Java applets, etc., do not perform any malicious operations, and it is important to be able to decipher the behavior of worms and virus-infected code. Static analysis provides techniques that can help with such problems [30, 29]. A major stumbling block when developing binary-analysis tools is that machine-language instructions use explicit memory addresses and indirect addressing to manipulate data. In this paper, we present several techniques that overcome this obstacle to developing binary-analysis tools.

Just as source-code-analysis tools provide information about the contents of a program's variables and how variables are manipulated, a binary-analysis tool should provide information about the contents of memory locations and how they are manipulated. Existing techniques either treat memory accesses extremely conservatively [4, 6, 2], or assume the presence of symbol-table or debugging information [27]. Neither approach is satisfactory: the former produces very approximate results; the latter uses information that cannot be relied upon when analyzing viruses, worms, mobile code, etc. Our analysis algorithm can do a better job than previous work because it tracks the pointer-valued and integer-valued quantities that a program's data objects can hold, using a set of abstract data objects, called *a-locs* (for "abstract locations"). In particular, the analysis is not forced to give up all precision when a load from memory is encountered.

The idea behind the a-loc abstraction is to exploit the fact that accesses on the variables of a program written in a high-level language appear as either static addresses (for globals) or static stack-frame offsets (for locals). Consequently, we find all the statically known locations and stack offsets in the program, and define an a-loc to be the set of locations from one statically known location/offset up to, but not including the next statically known location/offset. (The registers and `malloc` sites are also a-locs.) As discussed in Sect. 3.2, the data object in the original source-code program that corresponds to a given a-loc can be one or more scalar, struct, or array variables, but can also consist of just a segment of a scalar, struct, or array variable.

Another problem that arises in analyzing executables is the use of indirect-addressing mode for memory operands. Machine-language instruction sets support two addressing modes for memory operands: direct and indirect. In direct addressing, the address is in the instruction itself; no analysis is required to determine the memory location (and hence the corresponding a-loc) referred to by the operand. On the other hand,

if the instruction uses indirect addressing, the address is specified through a register expression of the form *base* + *index* × *scale* + *offset* (where *base* and *index* are registers). In such cases, to determine the memory locations referred to by the operand, the values that the registers hold at this instruction need to be determined. We present a flow-sensitive, context-insensitive analysis that, for each instruction, determines an over-approximation to the set of values that each a-loc could hold.

The contributions of our work can be summarized as follows:

- We describe a static-analysis algorithm, *value-set analysis*, for tracking the values of data objects (other than just the hardware registers). Value-set analysis uses an abstract domain for representing an over-approximation of the set of values that each data object can hold at each program point. The algorithm tracks address-valued and integer-valued quantities simultaneously: it determines an over-approximation of the set of addresses that each data object can hold at each program point; at the same time, it determines an over-approximation of the set of integer values that each data object can hold at each program point.
- Value-set analysis can be used to obtain used, killed, and possibly-killed sets for each instruction in the program. These sets are similar to the sets of used, killed, and possibly-killed variables obtained by a compiler in some source-code analyses. They can be used to perform reaching-definitions analysis and to construct data-dependence edges.
- We have implemented the analysis techniques described in the paper. By combining this analysis with facilities provided by the IDAPro [17] and CodeSurfer® [7] toolkits, we have created CodeSurfer/x86, a prototype tool for browsing, inspecting, and analyzing x86 executables. This tool recovers IRs from x86 executables that are similar to those that can be created for a program written in a high-level language. The paper reports preliminary performance data for this implementation.

The information obtained from value-set analysis should also be useful in decompilation tools. Although the implementation is targeted for x86 executables, the techniques described in the paper should be applicable to other machine languages.

Some of the benefits of our approach are illustrated by the following example:

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

int main() {
    int *part1,*part2;
    int a[10],*p_array0;
    int i;
    part1=&a[0];
    p_array0=part1;
    part2=&a[5];
    for(i=0;i<5;++i) {
       *part1=part1Value;
       *part2=part2Value;
       part1++;
       part2++;
    }
    return *p_array0;
}
```

```
    proc main           ;
1   sub esp, 44         ;Adjust esp for locals
2   lea eax, [esp+4]    ;part1=&a[0]
3   lea ebx, [esp+24]   ;part2=&a[5]
4   mov [esp+0], eax    ;p_array0=part1
5   mov ecx, 0          ;i=0
L1: mov edx, [0]        ;
7   mov [eax], edx      ;*part1=part1Value
8   mov edx, [4]        ;
9   mov [ebx], edx      ;*part2=part2Value
10  add eax, 4          ;part1++
11  add ebx, 4          ;part2++
12  inc ecx             ;i++
13  cmp ecx, 5          ;
14  jl L1               ;(i<5)?loop:exit
15  mov edi,[esp+0]     ;
16  mov eax, [edi]      ;
17  add esp, 44         ;
18  retn                ;return *p_array0
```

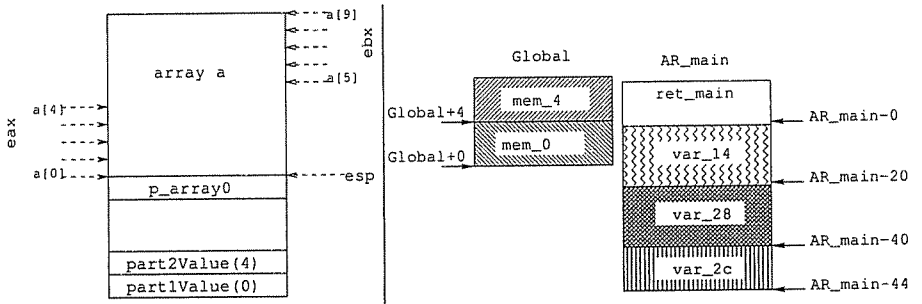
Fig. 1. A C program that initializes an array.

Example 1. Fig. 1 shows a simple C program and the corresponding disassembly. Procedure `main` declares an integer array `a` of ten elements. The program initializes the first five elements of `a` with the value of `part1Value`, and the remaining five with `part2Value`. It then returns `*p_array0`, i.e., the first element of `a`.

A diagram of how variables are laid out in the program's address space is shown in Fig. 2(a). To understand the assembly program in Fig. 1, it helps to know that

- The addresses of global variables `part1Value` and `part2Value` are 0 and 4, respectively.
- The local variables `part1`, `part2`, and `i` of the C program have been removed by the optimizer and are mapped to registers `eax`, `ebx`, and `ecx`.
- The instruction that modifies the first five elements of the array is "`7: mov [eax], edx`"; the one that modifies the last five elements is "`9: mov [ebx], edx`".

The statements that are underlined in Fig. 1 show the backward slice of the program with respect to `16 mov eax, [edi]`—which roughly corresponds to `return (*p_array0);` in the source code—that would be obtained using the sets of used, killed, and possibly-killed a-locs identified by value-set analysis. The slice obtained with this approach is actually smaller than the slice obtained by most source-code slicing tools. For instance, CodeSurfer/C does not distinguish accesses to different parts of an array. Hence, the slice obtained by CodeSurfer/C from C source code would include all of the statements in Fig. 1. ⊠



(a) Data layout (b) Memory-regions

Fig. 2. Data layout and memory-regions for Example 1.

The following insights shaped the design of value-set analysis:

- To prevent most indirect-addressing operations from appearing to be possible non-aligned accesses that span parts of two variables—and hence possibly forging new pointer values—it is important for the analysis to discover information about the alignments and strides of memory accesses.
- To prevent most loops that traverse arrays from appearing to be possible stack-smashing attacks, the analysis needs to use relational information so that the values of a-locs assigned to within a loop can be related to the values of the a-locs used in the loop's branch condition.
- It is desirable for the analysis to perform pointer analysis and numeric analysis simultaneously: information about numeric values can lead to improved tracking of pointers, and pointer information can lead to improved tracking of numeric values. This appears to be a crucial capability, because compilers use address arithmetic and indirect addressing to implement such features as pointer arithmetic, pointer dereferencing, array indexing, and accessing structure fields.

Value-set analysis produces information that is more precise than that obtained via several more conventional numeric analyses used in compilers, including constant propagation, range analysis, and integer-congruence analysis. At the same time, value-set analysis provides an analog of pointer analysis that is suitable for use on executables.

Debray et al. [11] proposed a flow-sensitive, context-insensitive algorithm for analyzing an executable to determine if two address expressions may be aliases. Our analysis yields more precise results than theirs: for the program shown in Fig. 1, their algorithm would be unable to determine the value of `edi`, and so the analysis would consider `[edi]`, `[eax]`, and `[ebx]` to be aliases of each other. Hence, the slice obtained using their alias analysis would also consist of the whole program. Cifuentes et al. [5] proposed a static-slicing algorithm for executables. They only consider programs with non-aliased memory locations, and hence would identify an unsafe slice of the program in Fig. 1, consisting only of the instructions `16`, `15`, `4`, `2`, and `1`. (See Sect. 9 for a more detailed discussion of related work.)

The remainder of the paper is organized as follows: Sect. 2 describes how value-set analysis fits in with the other components of CodeSurfer/x86, and discusses the assumptions that underlie our work. Sect. 3 describes the abstract domain used for value-set analysis. Sect. 4 describes the value-set analysis algorithm. Sect. 5 summarizes an auxiliary static analysis whose results are used during value-set analysis when interpreting conditions and when performing widening. Sect. 6 discusses indirect jumps and indirect function calls. Sect. 7 presents preliminary performance results. Sect. 8 discusses soundness issues. Sect. 9 discusses related work.

2 The Context of the Problem

CodeSurfer/x86 is the outcome of a joint project between the Univ. of Wisconsin and GrammaTech, Inc. CodeSurfer/x86 makes use of both IDAPro [17], a disassembly toolkit, and GrammaTech's CodeSurfer system [7], a toolkit for building program-analysis and inspection tools. This section describes how value-set analysis fits into the CodeSurfer/x86 implementation.

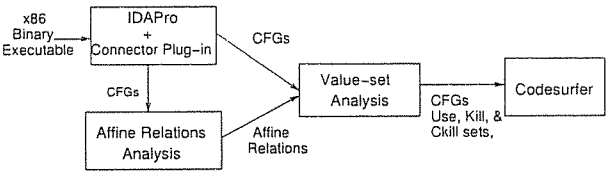


Fig. 3. Organization of CodeSurfer/x86.

The x86 executable is first disassembled using IDAPro. In addition to the disassembly listing, IDAPro also provides access to the following information:

Statically known memory addresses and offsets: IDAPro identifies the statically known memory addresses and stack offsets in the program, and renames all occurrences of these quantities with a consistent name. We use this database to define the a-locs.

Information about procedure boundaries: X86 executables do not have information about procedure boundaries. IDAPro identifies the boundaries of most of the procedures in an executable.[1]

[1] IDAPro does not identify the targets of all indirect jumps and indirect calls, and therefore the call graph and control-flow graphs that it constructs are not complete. Sect. 6 discusses techniques for using the abstract values computed during value-set analysis to augment the call graph and control-flow graphs on-the-fly to account for indirect jumps and indirect calls.

Calls to library functions: IDAPro discovers calls to library functions using an algorithm called the Fast Library Identification and Recognition Technology (FLIRT) [13]. This information is necessary to identify calls to `malloc`.

IDAPro provides access to its internal data structures via an API that allows users to create plug-ins to be executed by IDAPro. GrammaTech provided us with a plug-in to IDAPro (called the Connector) that constructs a variety of in-memory data structures, including CFGs, ASTs for the program's instructions, etc.

Value-set analysis is implemented using the data structures created by the Connector. As described in Sect. 5, value-set analysis makes use of the results of an additional preliminary analysis, which, for each program point, identifies the affine relations that hold among the values of registers. Once value-set analysis completes, the value-sets for the a-locs at each program point are used to determine each point's sets of used, killed, and possibly-killed a-locs; these are emitted in a format that is suitable for input to CodeSurfer.

CodeSurfer is a tool for code understanding and code inspection that supports both a GUI and a programmable interface for accessing a program's system dependence graph (SDG) [16], as well as other information stored in CodeSurfer's intermediate representations (IRs). CodeSurfer's GUI supports browsing ("surfing") of an SDG, along with a variety of operations for making queries about the SDG—such as slicing and chopping [24]. CodeSurfer's API provides a programmatic interface to these operations, as well as to lower-level information, such as the individual nodes and edges of the program's SDG, call graph, and control-flow graph, and a node's sets of used, killed, and possibly-killed a-locs. This API can be used to extend CodeSurfer's capabilities by writing programs that traverse CodeSurfer's IRs to perform additional program analyses.

A few words are in order about the goals, capabilities, and assumptions underlying our work:

- Given an executable as input, the goal is to check whether the executable conforms to a "standard" compilation model—i.e., a runtime stack is maintained; activation records are pushed on procedure entry and popped on procedure exit; each global variable resides at a fixed offset in memory; the occurrences of each local variable of a procedure f reside at a fixed offset in the activation records for f; the program's instructions occupy a fixed area of memory, are not self-modifying, and are separate from the program's data.

  If the executable does conform to this model, the system will create an IR for it. If it does not conform, then one or more violations will be discovered, and corresponding error reports will be issued (see Sect. 8).

  We envision CodeSurfer/x86 as providing (i) a tool for security analysis, and (ii) a general infrastructure for additional analysis of executables. Thus, in practice, when the system produces an error report, a choice is made about how to accommodate the error so that analysis can continue (i.e., the error is optimistically treated as a false positive), and an IR is produced; if the user can determine that the error report is indeed a false positive, then the IR is valid.
- The analyzer does not care whether the program was compiled from a high-level language, or hand-written in assembly. In fact, some pieces of the program may be the output from a compiler (or from multiple compilers, for different high-level languages), and others hand-written assembly.
- In terms of what features a high-level-language program is permitted to use, value-set analysis is capable of recovering information from programs that use global

variables, local variables, pointers, structures, arrays, heap-allocated storage, pointer arithmetic, indirect jumps, recursive procedures, and indirect calls through function pointers (but not runtime code generation or self-modifying code).
- As will become apparent, compiler optimizations often make value-set analysis *less* difficult, because more of the computation's critical data resides in registers, rather than in memory; register operations are more easily deciphered than memory operations.
- The major assumption that we make is that IDAPro is able to disassemble a program and build an adequate collection of *preliminary* IRs for it.

  We wish to stress that even though (i) the CFG created by IDAPro may be incomplete due to indirect jumps, and (ii) the call-graph created by IDAPro may be incomplete due to indirect calls, incomplete IRs do *not* trigger error reports. Both the CFG and the call-graph will be fleshed out according to information recovered during the course of value-set analysis (see Sect. 6).

  In fact, the relationship between value-set analysis and the preliminary IRs created by IDAPro is similar to the relationship between a points-to-analysis algorithm in a C compiler and the preliminary IRs created by the C compiler's front end. In both cases, the preliminary IRs are fleshed out during the course of analysis.

3 The Abstract Domain

The abstract stores used during value-set analysis over-approximate sets of concrete stores. Abstract stores are based on the concepts of *memory-regions* and *a-locs*, which are discussed first.

3.1 Memory-Regions

Memory addresses in an executable for an x-bit machine are x-bit numbers. Hence, one possible approach would be to use an existing numeric static-analysis domain, such as intervals [8], congruences [14], etc., to over-approximate the set of values (including addresses) that each data object can hold. However, there are several problems with such an approach: (1) addresses get reused, i.e., the same address can refer to different program variables at runtime; (2) a variable can have several runtime addresses; and (3) addresses cannot be determined statically in certain cases (e.g., memory blocks allocated from the heap via `malloc`).

Even though the same address can be shared by multiple activation records, it is possible to distinguish among these addresses based on what procedure is active at the time the address is generated (i.e., a reference to a local variable of `f` does not refer to a local variable of `g`). Value-set analysis uses an analysis-time analog of this: We assume that the address-space of a process consists of several non-overlapping regions called *memory-regions*. For a given executable, the set of memory-regions consists of one region per procedure, one region per heap-allocation statement, and a global region. We do not assume anything about the relative positions of these memory-regions. The region associated with a procedure represents all instances of the procedure's runtime-activation record. Similarly, the region associated with a heap-allocation statement represents all memory blocks allocated by that statement at runtime. The global region represents the uninitialized-data and initialized-data sections of the program.

Fig. 2(b) shows the memory-regions for the program from Fig. 1. The program has a single procedure, and hence has two regions: one corresponding to global data and the other corresponding to the activation record of `main`.

The analysis treats all data objects, whether local, global, or in the heap, in a fashion similar to the way compilers arrange to access variables in local activation records, namely, via an offset. We adopt this notion as part of our concrete semantics: a "concrete" memory address is represented by a pair: (memory-region, offset). (Thus, the concrete semantics already has a degree of abstraction built into it.) As explained below, an abstract memory address will track possible offsets using a numeric abstraction.

For the program from Fig. 1, the address of local variable `p_array0` is the pair `(AR_main, -44)`, and that of global variable `part2Value` is `(Global, 4)`.

At the enter node of a procedure `P`, register `esp` points to the start of the activation record of `P`. Therefore, the enter node of a procedure `P` is considered to be a statement that initializes `esp` with the address (`AR_P`, 0).

A call on `malloc` at program point `L` is considered to be a statement that assigns the address (`malloc_L`, 0).

3.2 A-Locs

Indirect addressing in x86 instructions involves only registers. However, it is not sufficient just to track values only for registers, because registers can be loaded with values from memory. If the analysis does not also track an approximation of the values that memory locations can hold, then the approximation produced for the registers will be very imprecise. Data dependences obtained from these sets will often be no better than those obtained by treating memory operations conservatively (i.e., every memory operation affects every other memory operation).

Instead, we use what we call the *a-loc* abstraction: An a-loc is roughly equivalent to a variable in a C program. The a-loc abstraction is based on the following observation: the data layout of the program is established at compile-time;[2] before generating the executable, the compiler decides where to place the global variables, local variables, etc. Globals will be accessed via direct operands in the executable. Similarly, locals will be accessed via indirect operands with `esp` (or `ebp`) as the base register, but a constant offset. Thus, examination of direct and indirect operands provides a rough idea of how the compiler (or the assembly programmer) laid out the data; i.e., it provides a rough idea of the base addresses and sizes of the program's variables. Consequently, we define an a-loc to be the set of locations between two such consecutive addresses or offsets.

For the program from Fig. 1, the direct operands are `[0]` and `[4]`. Therefore, we have two a-locs: `mem_0` (for addresses 0...3) and `mem_4` (for addresses 4...7). Also, the esp/ebp-based indirect operands are `[esp+0]`, `[esp+4]`, and `[esp+24]`. These operands are accesses on the local variables in the activation record of `main`. On entry to `main`, `esp` = `(AR_main, 0)`; the difference between the value of `esp` on entry to `main` and the value of `esp` when the local variables are accessed is `-44`. Thus, these memory references correspond to the offsets `-44`, `-40`, and `-20` in the memory-region for `AR_main`. These offsets will be referred to as `ar_offsets`. This gives rise to three more a-locs: `var_2c`, `var_28`, and `var_14`.[3] In addition to these a-locs, an a-loc for the return address is also defined; its offset in `AR_main` is `0`.

Note that `var_2c` corresponds to all of the source-code variable `p_array0`. In contrast, `var_28` and `var_14` correspond to disjoint segments of array `a[]`: `var_28` corresponds to `a[0..4]`; `var_14` corresponds to `a[5..9]`.

[2] Even if it is hand-written assembly, the programmer should have chosen some data-layout strategy.

[3] The numbers following "`var_`" are the offset in hexadecimal to the beginning of the a-loc.

Similarly, we have one a-loc per heap-region. In addition to these a-locs, registers are also considered to be a-locs. Note that an executable also has a section for read-only data. The values of these locations cannot change during program execution; hence, a-locs are not added for read-only data.

Offsets of an a-loc: Once the a-locs are identified, the relative positions of these a-locs in their respective regions are also recorded. This information will help us deal with pointer-arithmetic operations, as described in Sect. 4.1. The offset of an a-loc a in a region rgn will be represented as `offset`(rgn, a). For example, for the program from Fig. 1, `offset (AR_main, var_14)` is -20.

Addresses of an a-loc: The addresses that belong to an a-loc a can be represented by a pair $(rgn, [offset, offset + size - 1])$, where rgn represents the memory region to which it belongs to, *offset* is the offset of the a-loc within the region, and $size$ is the size of the a-loc. A pair of the form $[a, b]$ represents the set of integers $\{x | a \leq x \leq b\}$. For the program from Fig. 1, the addresses of a-loc `var_14` are $(\texttt{AR\_main}, [-40, -40 + 20 - 1]) = (\texttt{AR\_main}, [-40, -21])$. The $size$ of an a-loc may not be known for heap a-locs. In such cases, $size = \infty$.

3.3 Abstract Stores

An abstract store should over-approximate the set of memory addresses that each a-loc holds at a particular program point. As described in Sect. 3.1, every memory address is a pair (memory-region, offset). Therefore, a set of memory addresses in a memory region rgn is represented as $(rgn, \{o_1, o_2, \ldots, o_n\})$. The offsets $o_1, o_2, \ldots, o_n$ are numbers; they can be represented (i.e., over-approximated) using a numeric abstract domain, such as intervals, congruences, etc. We use a reduced interval congruence (RIC) for this purpose. A reduced interval congruence is the reduced cardinal product [9] of an interval domain and a congruence domain. For example, the set of numbers {1,3,5,7} can be represented as the RIC $(2\mathbb{Z} + 1) \cap [0, 7]$. Each RIC can be represented as a 4-tuple: the tuple (a,b,c,d) stands for $a \times [b, c] + d$, and denotes the set of integers $\{aZ + d | Z \in [b, c]\}$.[4] For instance, $\{1, 3, 5, 7\}$ is represented as the tuple $(2, 0, 3, 1)$.

An *abstract store* is a value of type a-loc $\rightarrow$ (memory-region $\rightarrow$ RIC). For instance, for the program from Fig. 1, at statement 7, `eax` holds the addresses of the first five elements of `main`'s local array, and thus the abstract store maps `eax` to [(`Global` $\mapsto \perp$), (`AR_main` $\mapsto 4[0, 4] - 40$)].

For conciseness, the abstract values that represent addresses in an a-loc for different memory-regions will be combined together into an r-tuple of RICs, where r is the number of memory regions. Such an r-tuple will be referred to as a *value-set*. Thus, an abstract store is a map from a-locs to value-sets: a-loc $\rightarrow$ RIC^r. At statement 7, the abstract store maps `eax` to the value-set $(\perp, 4[0, 4] - 40)$.

We chose to use RICs because in our context, it is important for the analysis to discover alignment and stride information so that it can interpret indirect-addressing operations that implement either (i) field-access operations in an array of structs, or (ii) pointer-dereferencing operations. That is, it is important to discover that a set of offsets in memory-region `f` consists of, say, $\{-36, -28, -20\}$, rather than merely the range $[-36, -20]$. If such offsets are used in an indirect-addressing operation, the former set permits us to determine that the memory accessed is 4-byte aligned.

When the contents of a pointer `p` is not aligned with the boundaries of variables, a memory access on `*p` can fetch portions of two variables; similarly, a write to `*p` can

[4] Because b is allowed to have the value $-\infty$, we cannot always adjust c and d so that b is 0.

overwrite portions of two variables. Such operations can be used to forge new addresses. For instance, suppose that the address of variable `a` is `1000`, the address of variable `b` is `1004`, and the value of `p` is `1001`. Then `*p` (as a 4-byte fetch) would retrieve 3 bytes of `a` and 1 byte of `b`. Thus, if value-set analysis were based on range information rather than RICs, it would either have to try to track *segments* of (possible) contents of data objects, or treat such dereferences conservatively by returning $\top$, thereby losing track of all information. (Even if the analysis tracked segments of possible contents of data objects, it would have to throw up its hands on any subsequent dereference: a static-analysis algorithm would have difficulties tracking the consequence of a dereference of a mixed-segment address, such as a dereference of the value fetched from `*p`.)

These issues motivated the use of RICs because RICs are capable of representing certain non-convex sets of integers. (In the example discussed above, $\{-36, -28, -20\}$ corresponds to the RIC $-8[0, 2] - 20$.)

Value-sets form a lattice. The following operators are defined for value-sets. All operators are pointwise applications of the corresponding RIC operator.

- $(vs_1 \sqsubseteq vs_2)$: Returns true if the value-set vs_1 is a subset of vs_2, false otherwise. (This defines the partial order on the value-set lattice).
- $(vs_1 \sqcap vs_2)$: Returns the intersection (meet) of value-sets vs_1 and vs_2.
- $(vs_1 \sqcup vs_2)$: Returns the union (join) of value-sets vs_1 and vs_2.
- $(vs_1 \nabla vs_2)$: Returns the value-set obtained by widening vs_1 with respect to vs_2. Suppose that $vs_1 = (10, 4[0, 1])$ and $vs_2 = (10, 4[0, 2])$, then $(vs_1 \nabla vs_2) = (10, 4[0, \infty])$.[5]
- $(vs \boxplus c)$: Returns the value-set obtained by adjusting all the values in vs by the constant c. Suppose that $vs = (4, 4[0, 2] + 4)$ and $c = 12$, then $(vs \boxplus c)$ returns $(16, 4[0, 2] + 16)$.
- $*(vs, s)$: Returns a pair of sets (F, P). F represents the set of "fully accessed" a-locs: it consists of the a-locs that are of size s and whose starting addresses are in vs. P represents the set of "partially accessed" a-locs: it consists of (i) a-locs whose starting addresses are in vs but are not of size s, and (ii) a-locs whose addresses are in vs but whose starting addresses and sizes do not meet the conditions to be in F.
- `RemoveLowerBounds` (vs) : Returns the value-set obtained by setting the lower bound of each component RIC to $-\infty$. For example, if $vs = ([0, 100], [100, 200])$, then `RemoveLowerBounds` $(vs) = ([-\infty, 100], [-\infty, 200])$.
- `RemoveUpperBounds` (vs) : Similar to `RemoveLowerBounds`, but sets the upper bound of each component to ∞.

To represent the abstract store at each program point efficiently, we use applicative dictionaries, which provide a space-efficient representation of a collection of dictionary values when many of the dictionary values have nearly the same contents as other dictionary values in the collection. Applicative dictionaries can be implemented using applicative balanced trees [26, 21], which are standard balanced trees on which all operations are carried out in the usual fashion, except that whenever one of the fields of an interior node M would normally be changed, a new node M' is created that duplicates M, and changes are made to the fields of M'. To be able to treat M' as the child of parent(M), it is necessary to change the appropriate child-field in parent(M), so a new node is created that duplicates parent(M), and so on, all the way to the root of the tree.

[5] Widening of two RICs is equivalent to the widening of the corresponding intervals [8] and congruences [14].

Thus, new nodes are introduced for each of the original nodes along the path from M to the root of the tree.

4 Value-Set Analysis

This section describes the value-set analysis algorithm. Value-set analysis is an abstract interpretation of the executable to find a safe approximation for the set of values that each data object holds at each program point. It uses the domain of abstract stores defined in Sect. 3. The present implementation of value-set analysis is flow-sensitive and context-insensitive.[6]

Value-set analysis has similarities with the pointer-analysis problem that has been studied in great detail for programs written in high-level languages. For each variable (say v), pointer analysis determines an over-approximation of the set of variables whose addresses v can hold. Similarly, value-set analysis determines an over-approximation of the set of addresses that each data object can hold at each program point. The results of value-set analysis can also be used to find the a-locs whose addresses a given a-loc a contains. On the other hand, value-set analysis also has some of the flavor of numeric static analyses, where the goal is to over-approximate the integer values that each variable can hold. In addition to information about addresses, value-set analysis determines an over-approximation of the set of integer values that each data object can hold at each program point.

4.1 Intraprocedural Analysis

Label on e	Transfer function for edge e
R1=R2+c	**let** $(\texttt{R2} \mapsto vs) \in$ e.Before e.After := e.Before $- [\texttt{R1} \mapsto *] \cup [\texttt{R1} \mapsto vs \boxplus c]$

*(R1+c_1)=R2+c_2	**let** $[\texttt{R1} \mapsto vs_1], [\texttt{R2} \mapsto vs_2] \in$ e.Before, $(F, P) = *(vs_1 \boxplus c_1, s)$,
	tmp = e.Before $- \{[p \mapsto *] \mid p \in P\} \cup \{[p \mapsto \top] \mid p \in P\}$, and
	Proc be the procedure containing the statement
	if ($\lvert F\rvert = 1$ and $\lvert P\rvert = 0$ and (*Proc* is not recursive) and (F has no heap objects)) **then**
	e.After := (tmp $- \{[v \mapsto *] \mid v \in F\} \cup \{[v \mapsto vs_2 \boxplus c_2] \mid v \in F\}$) // Strong update
	else
	e.After := (tmp $- \{[v \mapsto *] \mid v \in F\} \cup \{[v \mapsto (vs_2 \boxplus c_2) \sqcup vs_v] \mid v \in F, [v \mapsto vs_v] \in$ e.Before$\}$) // Weak update

R1= *(R2+c_1)+c_2	**let** $(\texttt{R2} \mapsto vs_{R2}) \in$ e.Before and $(F, P) = *(vs_{R2} \boxplus c_1, s)$
	if $\lvert P\rvert = 0$ **then**
	let $vs_{rhs} = \bigsqcup\{vs_v \mid v \in F, [v \mapsto vs_v] \in$ e.Before$\}$
	e.After := e.Before $- [\texttt{R1} \mapsto *] \cup [\texttt{R1} \mapsto (vs_{rhs} \boxplus c_2)]$
	else
	e.After := e.Before $- [\texttt{R1} \mapsto *] \cup [\texttt{R1} \mapsto \top]$

R1 $\leq c$	**let** $[\texttt{R1} \mapsto vs_{R1}] \in$ e.Before and $vs_c = ([-\infty, c], \top, \ldots, \top)$ e.After := e.Before $- [\texttt{R1} \mapsto *] \cup [\texttt{R1} \mapsto vs_{R1} \sqcap vs_c]$
R1 $\geq$ R2	**let** $[\texttt{R1} \mapsto vs_{R1}], [\texttt{R2} \mapsto vs_{R2}] \in$ e.Before and $vs_{lb} =$ RemoveUpperBounds(vs_{R2}) e.After := e.Before $- [\texttt{R1} \mapsto *] \cup [\texttt{R1} \mapsto vs_{R1} \sqcap vs_{lb}]$

Fig. 4. Transfer functions for value-set analysis. (In the second and third cases, s represents the size of the dereference performed by the instruction.)

This subsection describes an intraprocedural version of value-set analysis. For the time being, we will consider programs that have a single procedure and no indirect jumps. To aid in explaining the algorithm, we adopt a C-like notation for program statements. We will discuss the following kinds of instructions, where R1 and R2 are two

[6] In the near future, we plan to extend the implementation to have a degree of context-sensitivity, using the call-strings approach to interprocedural dataflow analysis [31].

registers of the same size, and c, c_1, and c_2 are explicit integer constants:

$$\begin{array}{rcll} \texttt{R1} & = & \texttt{R2} + c & \texttt{R1} \leq c \\ *(\texttt{R1} + c_1) & = & \texttt{R2} + c_2 & \texttt{R1} \geq \texttt{R2} \\ \texttt{R1} & = & *(\texttt{R2} + c_1) + c_2 & \end{array}$$

The analysis is performed on a CFG for the procedure. The CFG consists of one node per x86 instruction; the edges are labeled with the instruction at the source of the edge. If the source of an edge is a conditional, then the edge is labeled according to the outcome of the conditional. For instance, the edge `14→L1` will be labeled `ecx<5`, whereas the edge `14→15` will be labeled `ecx≥5`. Once we have the CFG, an abstract store is obtained for each program point by abstract interpretation [8]. The transformers for the various edges are listed in Fig. 4. Because the transformers for all conditionals are very similar, only some sample transformers are given for conditionals. Each transformer takes an abstract store and returns a new abstract store. Because each activation-record region of a procedure that may be called recursively—as well as each heap region—potentially represents more than one concrete data object, assignments to their a-locs must be modeled by weak updates, i.e., the new value-set must be unioned with the existing one, rather than replacing it (see case two of Fig. 4).

The abstract store for the entry node consists of the information about the initialized global variables and the initial value of the stack pointer (`esp`).

The abstract domain has infinite ascending chains. Hence, to ensure termination, widening needs to be performed. Widening needs to be carried out at least one node of every cycle in the CFG; however, the node at which widening is performed can affect the accuracy of the analysis. To choose widening points, our implementation of value-set analysis uses techniques from [3].

Example 2. For the program from Fig. 1, the abstract store for the entry node of `main` is $\{\texttt{esp} \mapsto (\bot, 0), \texttt{mem\_0} \mapsto (0, \bot), \texttt{mem\_4} \mapsto (1, \bot)\}$.

The fixpoint solution of value-set analysis for instruction `7` is $\{\texttt{esp} \mapsto (\bot, -44)$, $\texttt{mem\_0} \mapsto (0, \bot)$, $\texttt{mem\_4} \mapsto (1, \bot)$, $\texttt{eax} \mapsto (\bot, 4[0, \infty] - 40)$, $\texttt{ebx} \mapsto (\bot, 4[0, \infty] - 20)$, $\texttt{var\_2c} \mapsto (\bot, -40)$, $\texttt{ecx} \mapsto ([0, 4], \bot)\}$ and that of instruction `16` is $\{\texttt{esp} \mapsto (\bot, -44)$, $\texttt{mem\_0} \mapsto (0, \bot)$, $\texttt{mem\_4} \mapsto (1, \bot)$, $\texttt{eax} \mapsto (\bot, 4[1, \infty] - 40)$, $\texttt{ebx} \mapsto (\bot, 4[1, \infty] - 20)$, $\texttt{var\_2c} \mapsto (\bot, -40)$, $\texttt{ecx} \mapsto ([5, 5], \bot)$, $\texttt{edi} \mapsto (\bot, -40)\}$. The fixpoint solution for other instructions can be found in App. A.

Note that the value-sets obtained by the analysis can be used to discover the data dependence that exists between instructions `7` and `16`. At instruction `7`, $\texttt{eax} \mapsto (\bot, 4[0, \infty] - 40)$, and thus $*(as_{eax} \boxplus 0, 4)$ returns the set of a-locs {`var_28`, `var_14`, `ret_main`}. Similarly, at instruction `16` $*(as_{esp} \boxplus 8, 4)$ returns the set of a-locs {`var_28`}. Because the a-loc sets overlap, instruction `16` is data dependent on instruction `7`.

The results of value-set analysis also show that instruction `16` is not data dependent on `9`. At instruction `9`,$\texttt{ebx} \mapsto (\bot, 4[0, \infty] - 20)$, and thus $*(as_{ebx} \boxplus 0, 4)$ at instruction `9` returns {`var_14`, `ret_main`}. Because the a-loc sets do not overlap, `16` is not data dependent on `9`.

Note that the a-loc `ret_main` is also included in the set of variables accessed through `eax` at instruction `7`. This is because the analysis was not able to determine the upper bound for `eax`. Observe that `eax` is dependent on the loop variable `ecx`. We discuss in Sect. 5 how the implemented system actually finds upper or lower bounds for variables that are dependent on the loop variable. ⊠

4.2 Interprocedural Analysis

Let us now consider procedure calls, but for now ignore indirect jumps and calls. Interprocedural analysis presents new problems because the formals of a procedure and the actuals of a call need to be identified. This information is not directly available in the disassembly because parameters are typically passed on the stack in the x86 architecture. Moreover, the instructions that push the actual parameters on the stack need not occur immediately before the call. The following example will be used to explain the interprocedural case:

Example 3. Fig. 5 shows a program with two procedures, `main` and `initArray`. Procedure `main` has an integer array `a`, which is initialized by calling `initArray`. After initialization, `main` returns the second element of array `a`. The disassembly is also shown. ⊠

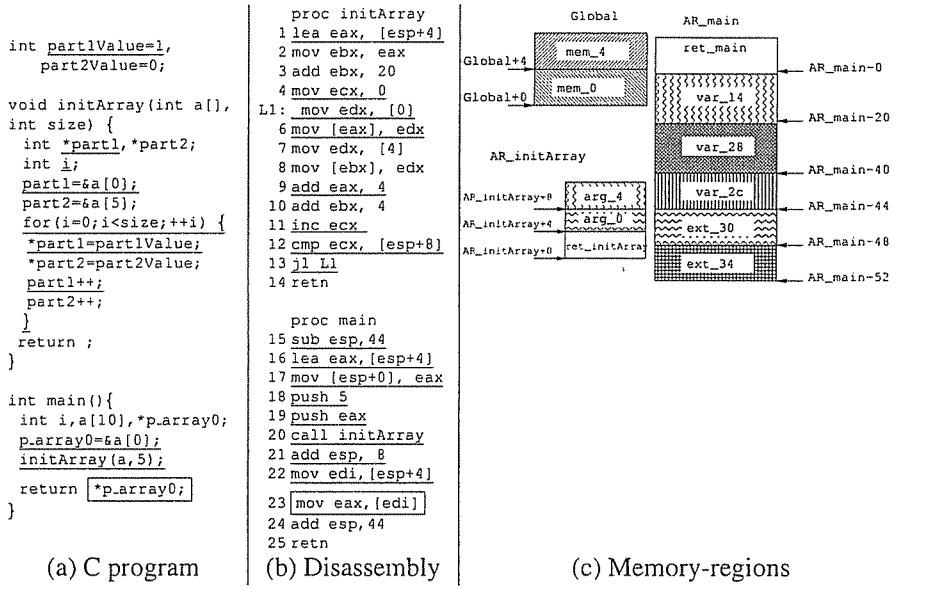


Fig. 5. Interprocedural example

Formal parameters On entry to a procedure, `esp` points to the return address, and the parameters to the procedure are the bytes beyond the return address (in the positive direction). Hence the ar_offsets for the formal parameters will be positive. Using this observation, the number of formal parameters for the procedure can be determined by

$$\text{Number of formals} = \left\lceil \frac{(\max(\text{ar\_offsets}) + \text{stack\_width} - \text{sizeof(return address)})}{\text{stack\_width}} \right\rceil$$

where stack_width is 4 for 32-bit executables and 2 for 16-bit executables. For our example, the maximum ar_offset in `initArray` is 8 (corresponding to the operand `[esp+8]`), and the size of the return address is 4; hence, there are two formals.

Actual parameters and register saves In an x86 program, stack operations like push/pop implicitly modify some locations in the activation record of a procedure (say `P`). These

locations correspond to the actual parameters of a call and to those used for register spilling and caller-saved registers. The locations accessed by push/pop instructions are not explicitly found as `esp`/`ebp`-relative addresses, and so the algorithm that identifies a-locs will not introduce variables for the memory locations accessed by these stack operations; consequently, we introduce additional variables, which we call *extended variables*, for memory locations that are implicitly accessed by such stack operations. To do this, the smallest `sp_delta` for `P` is determined. This represents the maximum limit to which the stack can grow in a single invocation of `P`.[7] If we are unable to find a finite minimum, the analysis stops and reports a problem. If there is a finite minimum, then *extended variables* are added to the activation record to fill the space between the lowest local variable and the minimum `sp_delta`.

At a call on a procedure that has k formals, the last k extended variables represent the actual parameters. Fig. 5(c) shows the extended variables for procedure `main` and the formal parameters for procedure `initArray` for the program in Example 3.

Handling of calls and returns The interprocedural algorithm is similar to the intraprocedural algorithm, but analyzes the supergraph of the executable. In the supergraph, each call site has two nodes: a call node and an end-call node. The only successor of the call node is the entry node of the called procedure and the only predecessor of the end-call node is the exit node of the procedure called by the corresponding call node. Nodes and edges for all other instructions are similar to the intraprocedural CFG. The call→entry and the exit→end-call edges will be refereed to as *linkage edges*.

The transformers for all the intraprocedural edges are the same as for the intraprocedural algorithm.

The transformer for the call→entry edge assigns actuals to formals and also changes `esp` to reflect the change in the current activation record. The abstract value for the entry node of a procedure `P` is determined as follows: The join of the value-sets associated with the first extended variable at calls to `P` is assigned to the first formal, and so on for each formal of `P`. The value-set for `esp` is set to $(\bot, \ldots, 0, \ldots, \bot)$, where the 0 occurs in the slot for `P`. In the fixpoint solution for Example 3, the abstract value for the enter node of `initArray` is: {`mem_0` $\mapsto (0, \bot, \bot)$, `mem_4` $\mapsto (1, \bot, \bot)$, `arg_0` $\mapsto (\bot, -40, \bot)$, `arg_4` $\mapsto (5, \bot, \bot)$, `eax` $\mapsto (\bot, -40, \bot)$, `esp` $\mapsto (\bot, \bot, 0)$, `ext_2c` $\mapsto (5, \bot, \bot)$, `ext_30` $\mapsto (\bot, -40, \bot)$ }.

Here the first component of each value-set corresponds to the `Global` region, the second to the `AR_main` region, and the last to the `AR_initArray` region.

The transformer for the exit→end-call edge ordinarily restores the value-set of `esp` to the value before the call. This corresponds to the normal case when the callee restores the value of `esp` to the value before the call. However, in some procedures the callee does not restore `esp`. For instance, `alloca` allocates memory on the stack by subtracting some number of bytes from `esp`. Value-set analysis takes care of those changes in `esp` that are just additions/subtractions to the initial value when it can determine that the change is always some constant amount. In such cases, `esp` is restored to the value before the call plus/minus the change. If value-set analysis cannot determine that the change is a constant, then it issues an error report.

[7] The stack can grow deeper due to calls made by `P`; however, these operations are not relevant because we are concerned merely with identifying the size of the activation record for `P`.

5 Affine Relations

Recall that in Example 2, value-set analysis was unable to find finite upper bounds for `eax` at instruction `7` and `ebx` at instruction `9`. This causes `ret_main` to be added to the possibly-killed sets for instructions `7` and `9`. This section describes how our implementation of value-set analysis obtains improved results, by identifying and then exploiting integer affine relations that hold among the program's registers, using an interprocedural algorithm for affine-relation analysis due to Müller-Olm and Seidl [19]. The algorithm is used to determine, for each program point, all affine relations that hold among an x86's 8 registers. More details about the algorithm can be found in App. B.

An integer affine relation among variables r_i $(i = 1 \ldots n)$ is a relationship of the form $a_0 + \sum_{i=1}^{n} a_i r_i = 0$, where the a_i $(i = 1 \ldots n)$ are integer constants. An affine relation can also be represented as an $(n+1)$-tuple, $(a_0, a_1, \ldots, a_n)$. There are two opportunities for incorporating information about affine relations: (i) in the interpretation of conditional instructions, and (ii) in an improved widening operation. Our implementation of value-set analysis incorporates both of these uses of affine relations.

At instruction `14` in the program in Fig. 1, `eax`, `esp`, and `ecx` are all related by the affine relation $\texttt{eax} = (\texttt{esp} + 4 \times \texttt{ecx}) + 4$. When the true branch of the conditional `jl L1` is interpreted, `ecx` is bounded on the upper end by 4, and thus the value-set `ecx` at `L1` is $([0,4], \bot)$. (A value-set in which all RICs are $\bot$ except the one for the `Global` region represents a set of pure numbers, as well as a set of global addresses.) In addition, the value-set for `esp` at `L1` is $(\bot, -44)$. Using these value-sets and solving for `eax` in the above relation yields

$$\texttt{eax} = (\bot, -44) + 4 \times ([0,4], \bot) + 4 = (\bot, -44) + 4 \times [0,4] + 4 = (\bot, 4[0,4] - 40).$$

In this way, a sharper value for `eax` at `L1` is obtained than would otherwise be possible.

Halbwachs et al. [15] introduced the "widening-up-to" operator (also called *limited widening*), which attempts to prevent widening operations from "over-widening" an abstract value to $+\infty$ (or $-\infty$). To perform limited widening, it is necessary to associate a set of inequalities M with each widening location. For polyhedral analysis, they defined $P \nabla_M Q$ to be the standard widening operation $P \nabla Q$, together with all of the inequalities of M that satisfy both P and Q. They proposed that the set M be determined by the linear relations that force control to remain in the loop. Our implementation of value-set analysis incorporates a limited-widening algorithm, adapted for reduced interval congruences. For instance, suppose that $P = (x \mapsto 3[0,2] + 5)$, $Q = (x \mapsto 3[0,3] + 5)$, and $M = \{x \leq 28\}$. Ordinary widening would produce $(x \mapsto 3[0, +\infty] + 5)$, whereas limited widening would produce $(x \mapsto 3[0,7] + 5)$. In some cases, however, the a-loc for which value-set analysis needs to perform limited widening is a register r_1, but not the register that controls the execution of the loop (say r_2). In such cases, the implementation of limited widening uses the results of affine-relation analysis—together with known constraints on r_2 and other register values—to determine constraints that must hold on r_1. For instance, if the loop back-edge has the label $r_2 \leq 20$, and affine-relation analysis has determined that $r_1 = 4 * r_2$ always holds at this point, then the constraint $r_1 \leq 80$ can be used for limited widening of r_1's abstract value.

6 Indirect Jumps and Indirect Calls

The supergraph of the program will not be complete in the presence of indirect jumps and indirect calls. Consequently, missing jump and call edges need to be inserted during

value-set analysis. For instance, suppose that value-set analysis is interpreting an indirect jump instruction `J1: jmp 1000[eax*4]`, and let the current abstract store at this instruction be $\{\texttt{eax} \mapsto ([0,9], \bot, \ldots, \bot)$. Edges need to be added from `J1` to the instructions whose addresses could be in memory locations {1000, 1004, ..., 1036}. If the addresses {1000, 1004, ..., 1036} refer to the read-only section of the program, then the addresses of the successors of `J1` can be read from the header of the executable. If not, the addresses of the successors of `J1` in locations {1000, 1004, ..., 1036} are determined from the current abstract value at `J1`. Due to possible imprecision in value-set analysis, it could be the case that value-set analysis reports that the locations {1000, 1004, ..., 1036} have all possible addresses. In such cases, value-set analysis proceeds without adding new edges. However, this could lead to an under-approximation of the value-sets at program points. Therefore, the analysis issues a report to the user whenever such decisions are made. We will refer to such instructions as *unsafe instructions*. Another issue with using the results of value-set analysis is that an address identified as a successor of `J1` might not be the start of an instruction. Such addresses are ignored, and the situation is reported to the user.

Indirect calls can be handled similarly, with a few additional complications. (At present, the techniques to handle indirect calls have not yet been incorporated in our implementation.)

- A successor instruction identified by the method outlined above may be in the middle of a procedure. In such cases, the analysis can report this to the user.
- The successor instruction may not be part of a procedure that was identified by IDAPro. This is due to the limitations of IDAPro's procedure-finding algorithm: IDAPro does not identify procedures that are called exclusively via indirect calls. In such cases, value-set analysis can invoke IDAPro's procedure-finding algorithm explicitly, to force a sequence of bytes from the executable to be decoded into a sequence of instructions and spliced into the IR for the program.

7 Performance Evaluation

Program	Procedures	Instructions	Malloc sites	Indirect jumps	Calls	Indirect calls	Memory usage (MB)	Value-set analysis (sec.)	Affine-relation analysis (sec.)
javac	36	3555	1	0	133	79	150	42	36
cat (2.0.14)	123	3892	1	3	138	4	175	51	32
cut (2.0.14)	129	4329	2	3	182	4	150	28	50
grep (2.4.2)	245	16808	18	4	654	6	450	85	78
flex (2.5.4)	239	23435	0	7	1200	3	850	200	376

Table 1. Running times and storage requirements for value-set analysis and affine-relation analysis.

Table 1 shows the running times and storage requirements of our prototype implementation for analyzing a set of Win32 and Linux/x86 programs; the program version is shown in parentheses. As a temporary expedient, calls to library functions are treated during analysis as identity transformers.

The analyses were performed on a Pentium-4 with a clock speed of 3.06GHz, equipped with a physical memory of 4GB and running Windows 2000. (The per-process address space was limited to 2GB.)

To contrast the capabilities of value-set analysis with analysis algorithms that treat memory accesses very conservatively—i.e., if a register is assigned a value from memory, it is assumed to take on any value—we compared it with a version of value-set analysis, called *crude value-set analysis*, that always sets the value-sets for all non-register a-locs to $\top$. Table 2 shows the number of flow-dependence edges obtained with three methods: (i) without using value-set analysis at all (which causes dependences to be missed); (ii) with value-set analysis; and (iii) with crude value-set analysis.

Program	No VSA	VSA	Crude VSA
javac	14973	40843	42186
cat (2.0.14)	14048	30922	31403
cut (2.0.14)	16591	29410	31770
grep (2.4.2)	80115	170648	191172

Table 2. Comparison of three variants of value-set analysis.

8 Soundness Issues

Soundness would mean that value-set analysis would identify used, killed, and possibly-killed sets that would never miss any data dependence, although they might cause spurious dependences to be reported. This is a lofty goal; however, it is not clear that a tool that achieves this goal would have practical value. (It is achievable trivially, merely by setting all value-sets to $\top$.)

There are less lofty goals that do not meet the standard articulated above—but may result in a more practical system. In particular, we may not care if the system is sound, as long as it can provide warnings about the situations that arise during the analysis that threaten the soundness of the results. This is the path that we are following in our work, and are in the process of adding such reports to our implementation.

Here are some of the cases in which the analysis can be unsound, but where the system can generate a report about the nature of the unsoundness:

- The program is vulnerable to a buffer-overrun attack. This can be detected by identifying a point at which there can be a write past the end of a memory-region.
- The control-flow graph and call-graph may not identify all successors of indirect jumps and indirect calls. Report generation for such cases is discussed in Sect. 6.
- A related situation is a jump to a code sequence concealed in the regular instruction stream; the alternative code sequence would decode as a legal code sequence when read out-of-registration with the instructions in which it is concealed. The analysis could detect this situation as an anomalous jump to an address that is in the code segment, but is not the start of an instruction.
- With self-modifying code, the control-flow graph and call-graph are not available for analysis. The analysis can detect the possibility that the program is self-modifying by identifying an anomalous jump or call to a location that can be modified.

9 Related Work

There is an extensive body of work on analyzing executables. The work that is most closely related to value-set analysis is the alias-analysis algorithm for executables proposed by Debray et al. [11]. The basic goal of their algorithm is similar to that of value-set analysis: for them, it 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 memory locations in addition to registers. In their analysis, 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 our algorithm, their algorithm does not make any effort to track values that are not in registers. Consequently, they lose a great deal of precision whenever there is a load from memory.

Cifuentes and Fraboulet [5] 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.

Past work on decompiling assembly code to a high-level language is also related to our goals [6, 4, 20]. Past work on decompilation has not done much to address the problem of recovering information about memory accesses. In our work, this is addressed via the notion of a-locs, plus additional information that is inferred as analysis progresses. Consequently, value-set analysis should be a useful algorithm to incorporate in a decompilation tool. By identifying points-to, range, and stride information prior to decompilation proper, value-set analysis provides a rich source of information about a program's data layout; this should allow a decompilation tool to do a better job of reverse translation for instruction sequences in which loads from and stores to memory are performed.

The idea of inferring the layout of a program's data structures based on the access patterns in the program is similar to the idea behind the Aggregate Structure Identification (ASI) algorithm of Ramalingam et al. [23]. However, ASI cannot be applied to x86 code without having the results of value-set analysis already in hand: ASI requires points-to, range, and stride information; however, this information is not available for an x86 executable until after value-set analysis. The good news is that ASI can be applied after value-set analysis to refine the program's a-locs, which can allow some clients of value-set analysis—such as dependence analysis—to compute more precise results. We plan to use ASI in conjunction with the results of value-set analysis in future work.

Xu et al. [33] also created a system that analyzed executables in the absence of symbol-table and/or debugging information. The goal of their system was to establish whether or not certain memory-safety properties held in SPARC executables. Initial inputs to the untrusted program were annotated with typestate information and linear constraints. The analyses developed by Xu et al. were based on classical theorem-proving techniques: the typestate-checking algorithm used the induction-iteration method [32] to synthesize loop invariants and Omega [22] to decide Presburger formulas. In contrast, the goal of the system described in the present paper is to recover information from an x86 executable that permits the creation of intermediate representations similar to those that can be created for a program written in a high-level language. Value-set analysis uses abstract-interpretation techniques to determine used, killed, and possibly-killed sets for each instruction in the program.

Several people have developed techniques to analyze executables in the presence of additional information, such as the source code, symbol-table information, or debugging information [18, 2, 1, 27]. Analysis techniques that assume access to such information are limited by the fact that it must not be relied on when dealing with programs such as viruses, worms, and mobile code (even if such information is present).

Dor et al. [12] present a static-analysis technique—implemented for programs written in C—whose aim is to identify string-manipulation errors, such as potential buffer overruns. In their work, a flow-insensitive pointer analysis is first used to detect pointers to the same base address; integer analysis is then used to detect relative-offset re-

lationships between values of pointer variables. The original program is translated to an integer program that tracks the string and integer manipulations of the original program; the integer program is then analyzed to determine relationships among the integer variables, which reflect the relative-offset relationships among the values of pointer variables in the original program. Because they are primarily interested in establishing that a pointer is merely *within the bounds* of a buffer, it is sufficient for them to use linear-relation analysis [10], in which abstract values are convex polyhedra defined by linear inequalities of the form $\sum_{i=1}^{n} a_i x_i \leq b$, where b and the a_i are integers, and the x_i are integer variables.

In our work, we are interested in discovering fine-grained information about the structure of memory-regions. As already discussed in Sect. 3.3, it is important for the analysis to discover alignment and stride information so that it can interpret indirect-addressing operations that implement field-access operations in an array of structs or pointer-dereferencing operations. Because we need to represent non-convex sets of numbers, linear-relation analysis is not appropriate. For this reason, the numeric component of value-set analysis is based on reduced interval congruences, which are capable of representing certain non-convex sets of integers.

Rugina and Rinard [28] have also used a combination of pointer and numeric analysis to determine information about a program's memory accesses. There are several reasons why their algorithm is not suitable for the problem that we face: (i) Their analysis assumes that the program's local and global variables are known before analysis begins: the set of "allocation blocks" for which information is acquired consists of the program's local and global variables, plus the dynamic-allocation sites. (ii) Their analysis determines range information, but does not determine alignment and stride information. (iii) Pointer and numeric analysis are performed separately: pointer analysis is performed first, followed by numeric analysis; moreover, it is not obvious that pointer analysis could be intertwined with the numeric analysis that is used in [28].

Our analysis *combines* pointer analysis with numeric analysis, whereas the analyses of Rugina and Rinard and Dor et al. use two separate phases: pointer analysis *followed by* numeric analysis. An advantage of combining the two analyses is that information about numeric values can lead to improved tracking of pointers, and pointer information can lead to improved tracking of numeric values. In our context, this kind of positive interaction is important for discovering alignment and stride information (cf. Sect. 3.3). Moreover, additional benefits can accrue to clients of value-set analysis; for instance, it can happen that extra precision will allow value-set analysis to identify that a strong update, rather than a weak update, is possible (i.e., an update can be treated as a kill rather than as a possible kill; cf. case two of Fig. 4).

References

1. 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.
2. J. Bergeron, M. Debbabi, M.M. Erhioui, and B. Ktari. Static analysis of binary code to isolate malicious behaviors. In *WETICE*, pages 184–189, 1999.
3. François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Int. Conf. on Formal Methods in Prog. and their Appl.*, Lec. Notes in Comp. Sci. Springer-Verlag, 1993.
4. C. Cifuentes and A. Fraboulet. Interprocedural data flow recovery of high-level language code from assembly. Technical Report 421, Univ. Queensland, 1997.

5. C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *Int. Conf. on Softw. Maint.*, pages 188–195, 1997.
6. C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation. In *Int. Conf. on Softw. Maint.*, pages 228–237, 1998.
7. CodeSurfer, GrammaTech, Inc., http://www.grammatech.com/products/codesurfer/.
8. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2nd Int. Symp. on Programming*, pages 106–130. Dunod, Paris, France, 1976.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Princ. of Prog. Lang.*, 1977.
10. P. Cousot and R. Cousot. Automatic discovery of linear restraints among variables of a program. In *Princ. of Prog. Lang.*, pages 84–97, 1978.
11. S.K. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Princ. of Prog. Lang.*, pages 12–24, 1998.
12. N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Prog. Lang. Design and Impl.*, pages 155–167, 2003.
13. Fast library identification and recognition technology, DataRescue sa/nv, Liège, Belgium, http://www.datarescue.com/idabase/flirt.htm.
14. P. Granger. Static analysis of arithmetic congruences. *Int. J. of Comp. Math.*, 1989.
15. N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.
16. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Trans. on Prog. Lang. and Syst.*, 12(1):26–60, January 1990.
17. IDAPro disassembler, DataRescue sa/nv, Liège, Belgium, http://www.datarescue.com/idabase/.
18. J.R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Prog. Lang. Design and Impl.*, pages 291–300, 1995.
19. M. Müller-Olm and H. Seidl. Computing interprocedurally valid relations in affine programs. In *Princ. of Prog. Lang.*, 2004.
20. A. Mycroft. Type-based decompilation. In *European Symp. on Programming*, 1999.
21. E.W. Myers. Efficient applicative data types. In *Princ. of Prog. Lang.*, pages 66–75, 1984.
22. W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.
23. G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *Princ. of Prog. Lang.*, pages 119–132, 1999.
24. T. Reps and G. Rosay. Precise interprocedural chopping. In *Found. of Softw. Eng.*, 1995.
25. T. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Static Analysis Symp.*, 2003.
26. T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *Trans. on Prog. Lang. and Syst.*, 5(3):449–477, July 1983.
27. X. Rival. Abstract interpretation based certification of assembly code. In *Int. Conf. on Verif., Model Checking, and Abs. Int.*, 2003.
28. R. Rugina and M.C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. New York, NY. ACM Press.
29. A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE J. Sel. Areas in Commun.*, 21(1):5–19, January 2003.
30. F.B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead*, pages 86–101, 2000.
31. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
32. N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *Princ. of Prog. Lang.*, pages 132–143, 1977.
33. Z. Xu, B. Miller, and T. Reps. Safety checking of machine code. In *Prog. Lang. Design and Impl.*, pages 70–82, 2000.

A Results of Value-Set Analysis for Fig. 1

```
1 esp ↦ (⊥,0), mem_0 ↦ (0,⊥),
  mem_4 ↦ (1,⊥)
2 esp ↦ (⊥,-44), mem_0 ↦ (0,⊥),
  mem_4 ↦ (1,⊥)
3 esp ↦ (⊥,-44), mem_0 ↦ (0,⊥),
  mem_4 ↦ (1,⊥), eax ↦ (0,-40)
4 esp ↦ (⊥,-44), mem_0 ↦ (0,⊥),
  mem_4 ↦ (1,⊥), eax ↦ (⊥,-40),
  ebx ↦ (⊥,-20)
5 esp ↦ (⊥,-44), mem_0 ↦ (0,⊥),
  mem_4 ↦ (1,⊥), eax ↦ (⊥,-40),
  ebx ↦ (⊥,-20), var_2c ↦ (⊥,-40)
7 esp ↦ (⊥,-44), mem_0 ↦ (0,⊥),
  mem_4 ↦ (1,⊥), eax ↦ (⊥,4[0,∞]-40),
  ebx ↦ (⊥,4[0,∞]-20), var_2c ↦ (⊥,-40),
  ecx ↦ ([0,4],⊥)
9 esp ↦ (⊥,-44), mem_0 ↦ (0,⊥),
  mem_4 ↦ (1,⊥), eax ↦ (⊥,4[0,∞]-40),
  ebx ↦ (⊥,4[0,∞]-20), var_2c ↦ (⊥,-40),
  ecx ↦ ([0,4],⊥)
14 esp ↦ (⊥,-44), mem_0 ↦ (0,⊥),
  mem_4 ↦ (1,⊥), eax ↦ (⊥,4[1,∞]-40),
  ebx ↦ (⊥,4[1,∞]-20), var_2c ↦ (⊥,-40),
  ecx ↦ ([1,5],⊥)
16 esp ↦ (⊥,-44), mem_0 ↦ (0,⊥),
  mem_4 ↦ (1,⊥), eax ↦ (⊥,4[1,∞]-40),
  ebx ↦ (⊥,4[1,∞]-20), var_2c ↦ (⊥,-40),
  ecx ↦ ([5,5],⊥), edi ↦ (⊥,-40)
```

B The Müller-Olm/Seidl Algorithm for Affine-Relation Analysis

Müller-Olm and Seidl [19] recently proposed an interprocedural dataflow-analysis algorithm to determine, for each program point, all affine relations that hold among a set of global variables.[8] The algorithm is both flow-sensitive and context-sensitive.

Let the n global variables of the program be $r_1, r_2, \ldots r_n$. An affine transformation is a pair (A, b), where $A \in \mathbb{Z}^{n \times n}$ and $b \in \mathbb{Z}^{n \times 1}$. If $\mathbf{r} = (r_1, r_2, \ldots r_n)$ is the initial value of the global variables and (A, b) is the affine transformation corresponding to the instructions along a path, then the values of the global variables after executing all the instructions along that path are given by $A\mathbf{r} + b$. The basic idea of the algorithm is to determine for each program point a summary of the affine transformations that involve global variables along all paths from program entry to that point. Let the set of affine transformations for a program point p be S. The possible values of global variables at p is given by the set $\mathbf{R}' = \{\mathbf{r}' \mid \mathbf{r}' = A\mathbf{r} + b, (A, b) \in S\}$.

Although $\mathbf{R}'$ is an infinite set, Müller-Olm and Seidl observe that it forms a finite-dimensional vector space, and thus can be represented in a finite way using any of its bases. Similarly, the set S of affine transformations that hold at p is infinite, but can also be represented by a basis of a vector space. These indirect representations are sufficient for our needs. As shown in [19], using such techniques it is possible to solve the problem of flow-sensitive, context-sensitive affine-relation analysis in time linear in the size of the program. The constant of proportionality is rather high, $O(k^8)$, where k is the number of variables for which affine relations are being tracked. However, for affine relations on x86 registers, k is 8. As shown in Sect. 7, the cost of computing affine relations for the case $k = 8$ is not prohibitive. Moreover, it is important to remember that k does not grow with program size; overall, the cost of the algorithm grows *linearly* in the size of the program.

Reps et al. [25] describe how interprocedural dataflow-analysis problems can be reduced to path queries in weighted pushdown systems. Our implementation of the Müller-Olm/Seidl algorithm uses a package that implements the techniques from [25].

[8] Extensions required to handle local variables are also described in [19]. The algorithm that deals with global variables is sufficient for our purposes, because we use it only to find affine relations involving registers.