Analyzing Memory Accesses in x86 Binary Executables*

Gogul Balakrishnan Thomas Reps

ABSTRACT

This paper concerns static-analysis algorithms for analyzing binary executables. The aim of the work is to recover intermediate representations (IRs) 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 viruses, worms, and mobile code. For such programs, symbol-table and debugging information is either entirely absent, or cannot be relied upon if present; hence, the analysis described in the paper makes no use of symbol-table/debugging information.

The main analysis discussed in the paper, called *value-set analysis*, tracks address-valued and integer-valued quantities simultaneously. It is related to pointer-analysis algorithms that have been developed for programs written in high-level languages, which determine an over-approximation of the set of variables whose addresses each pointer variable can hold. At the same time, value-set analysis is similar to range analysis and other numeric static-analysis algorithms that over-approximate the integer values that each variable can hold.

The techniques described in the paper have been implemented as part of CodeSurfer/x86, a prototype tool for browsing ("surfing"), inspecting, and analyzing x86 executables.

1. INTRODUCTION

In recent years, there has been a growing need for tools that analyze binary executables to aid in understanding them. For instance, when a new virus or worm shows up, anti-virus companies and agencies in charge of critical infrastructure protection need to understand its behavior and react accordingly. A tool that helps in understanding viruses can be of immense help because the rate at which viruses or worms spread is increasing rapidly. The recent Slammer worm was able to infect 90% of the vulnerable computers in the world [19]. Moreover, mobile code is becoming an effective and inexpensive means for improving or extending the functionality of software systems. Mobile code is obtained over a network from a possibly untrusted source and executed on a local machine. Plug-ins in web browsers, embedded scripts in HTML pages, Java applets, e-mail attachments, etc., are some examples. Before mobile code executes on our system, we would like to know that it does not do anything malicious or send out any private information. Static analysis provides techniques that can help answer such questions [30, 29].

A major stumbling block in 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 the 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 [28]. 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. We address these problems by defining an abstraction for the program's data objects, which we refer to as *a-locs* (for "abstract locations"); the analysis keeps track of sets of possible values for each a-loc.

The basic idea behind the a-loc abstraction is to find all the statically known locations in the program, and then map the range of locations from one statically known location up to, but not including the next statically known location, to one a-loc. (Registers and `malloc` sites are also considered to be a-locs.) The intuition is that because we do not know anything about the structure of the intervening region between two statically known locations, it will be tracked as one object. 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 binaries 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 $base + index \times scale + offset$ (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.

Several others have proposed techniques to obtain information from binary executables by means of static analysis [11, 6, 5]. However, previous techniques deal with memory accesses very conservatively; i.e., if a register is assigned a value from memory, it is assumed to take on any value. Our analysis technique can do a better job than previous work

*This work was supported in part by the Office of Naval Research under contracts N00014-01-1-0796 and N00014-01-1-0708, and by the Alexander von Humboldt Foundation. Address: Comp. Sci. Dept.; Univ. of Wisconsin; 1210 W. Dayton St.; Madison, WI 53706. E-mail: {bgogul,reps}@cs.wisc.edu.

because it tracks the pointer-valued and integer-valued quantities that the program's data objects (i.e., a-locs) can hold; in particular, it is not forced to give up all precision when a load from memory is encountered.

The contributions of our work can be summarized as follows:

- We identify issues germane to the problem of analyzing binary executables without the use of symbol-table/debugging information. In particular, we present a static analysis for tracking the values of data objects (other than just the hardware registers). This is an issue that has been ignored in past work [18, 11, 5, 4, 6, 2], leading to tools that are of limited utility for programs such as viruses, worms, and mobile code, for which symbol-table/debugging information cannot be relied upon.

- We present an abstract domain for representing a set of values that each data object can hold at each program point.

- We describe an algorithm, *value-set analysis*, that tracks address-valued and integer-valued quantities simultaneously: value-set analysis 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.

  The advantage of this approach is that it captures relationships among the program's address values and integer values. This appears to be a crucial capability, because binary executables use address arithmetic and indirect addressing to implement high-level-language features such as pointer arithmetic, pointer dereferencing, indexing into arrays, and accessing structure fields.

  The information that can be recovered from value-set analysis is more precise than several conventional numeric static analyses, including constant propagation, range analysis, and integer-congruence analysis; at the same time, value-set analysis provides an analogue of pointer-analysis information.

  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.

  The information obtained from value-set analysis should also be useful in decompilation tools.

- 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. (Although the implementation is targeted for x86 executables, the techniques described in the paper should be applicable to other machine languages.)

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

int main() {
    int *part1,*part2,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;
}
```

Figure 1: A C program that initializes an array.

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

EXAMPLE 1. Fig. 1 shows a simple C program. The program has a procedure `main` that declares an integer array `a` of 10 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`.

Fig. 2(a) shows the x86 disassembly of the program. A diagram of how variables are laid out in the program's address space is shown in Fig. 2(b). To understand Fig. 2(a), 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`", and the instruction that modifies the remaining five elements is "`9: mov [ebx], edx`".

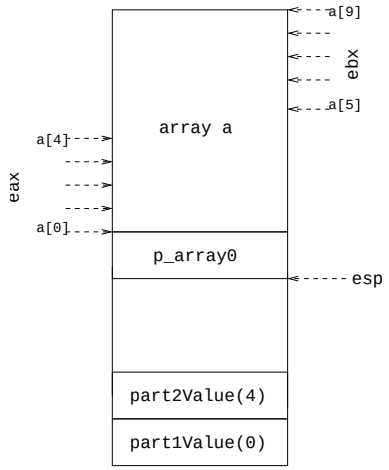
The statements that are underlined in Figs. 1 and 2 show the backward slice of the program with respect to `16 mov eax, [edi]`—which roughly corresponds to `return(*p_array0)` in the source code—that would be obtained using the sets of used, killed, and possibly-killed a-locs identified by value-set analysis. ⊠

Note that the slice obtained with this approach is 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.

Debray et al. [11] proposed a flow-sensitive, context-insensitive algorithm to determine if two memory operands are aliases. Our analysis yields more precise results than their analysis because our abstract domain generalizes the one that they use. For the program shown in Fig. 2(a), their

```
      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
```

(a) Disassembly



(b) Layout of variables in the program's address space and values of some registers at instruction 7

Figure 2: Assembly-language program for Example 1.

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 binary executables. They only consider programs with non-aliased memory locations, and hence would identify an unsafe slice of the program in Fig. 2(a), 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 discusses how value-set analysis fits in with the other tools used to implement CodeSurfer/x86. 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. Appendix A describes a technique that we plan to incorporate, which will improve the results of value-set analysis.

2. OVERVIEW

This section describes the overall organization of the CodeSurfer/x86 implementation. CodeSurfer/x86 was created as part 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.

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

Statically known memory addresses: IDAPro determines the statically known memory addresses in the program, and renames all occurrences of these addresses with a consistent name. We use this database to define a-locs (see Sect. 3.3).

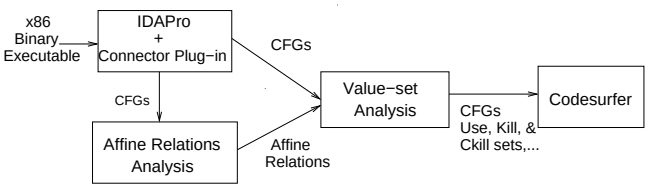


Figure 3: Overall tool organization

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

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

[1]IDAPro does not identify the targets of all indirect calls and jumps, and therefore the call graph and control-flow graphs constructed by IDAPro are not a safe representation of the program—although the heuristics used by IDAPro are at least as good as what other tools have implemented [18, 5].

Sect. 6 discusses techniques for using the abstract values computed during value-set analysis to alter the call graph and control-flow graphs on-the-fly to construct a safe representation of the program.

(called the *Connector*) that constructs in-memory data structures that represent the program's instructions, CFGs, ASTs, etc. (Access to the data structures is provided by an interface similar to EEL [18].) Value-set analysis is implemented using these in-memory data structures. 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 [25]. 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.

Although CodeSurfer was originally developed for ANSI C, its input language was recently generalized so that it could be retargeted to other languages. In our work, we have used this flexibility to create CodeSurfer/x86, a tool for code understanding and code inspection of x86 binaries; future work will develop additional analysis tools for binary executables based on the programmatic interface that is available in CodeSurfer/x86.

The task of creating CodeSurfer/x86 was greatly complicated by the restriction that we imposed on ourselves that symbol-table and debugging information were off-limits; it would have been relatively easy to create a CodeSurfer/x86 tool had we made use of such information; however, such a tool would not have been useful in situations in which symbol-table and debugging information is absent or untrusted.

The results of the value-set analysis algorithm described in this paper—together with the extension sketched out in App. A—provide a substitute for symbol-table and debugging information; this allowed us to create a tool that can be used when such information is absent or untrusted.

3. THE ABSTRACT DOMAIN

This section describes the abstract domain that is used by the value-set analysis algorithm to approximate sets of concrete memory addresses. The definition of the abstract domain uses the following concepts: *memory-regions* and *a-locs*. We describe these first.

3.1 Memory-Regions

Memory addresses in a binary 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:

- 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, a given numeric address is ambiguous: it may denote a variable of `f`, a variable of `g`, a variable of `h`, etc. (A given address may also correspond to different variables of different activations of `f`.)
- A given procedure `f` can be invoked in different calling contexts, and thus the activation record for `f` can appear in many different positions on the runtime stack. Different instances of the same variable have many different addresses.
- It may not be possible to determine specific address values for certain memory blocks, such as those allocated from the heap via `malloc`; these must be treated in some other fashion.

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 analogue of this: We assume that the address-space of a process consists of several non-overlapping regions called *memory-regions*. For a given binary 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.

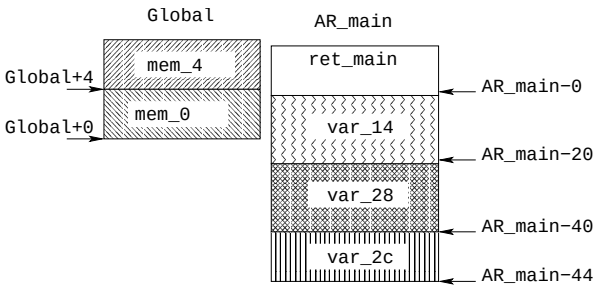


Figure 4: Memory-regions

Fig. 4 shows the memory-regions for the program from Fig. 2(a). 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. 2(a), 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 basic idea behind the a-loc abstraction is to find all the statically known locations in the program, and then map the range of locations from one statically known location up to, but not including the next statically known location, to one a-loc. (Registers and malloc sites are also considered to be a-locs.) Value-set analysis keeps track of value-set information for each a-loc.

A-locs are divided into four kinds: (1) global a-locs, (2) local a-locs, (3) heap a-locs, and (4) registers. (A binary 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.) Except for the register a-locs, an a-loc corresponds to a set of consecutive addresses in a memory-region.

Global a-locs: Global a-locs are determined as follows. All instructions using direct-addressing mode for memory operands are identified. For the program from Fig. 2(a), the instructions "mov edx, [0]" and "mov edx,[4]" have direct-addressing mode operands, namely, [0] and [4]. The set of addresses in between two such consecutive direct addresses in the program is a global a-loc. For the program from Fig. 2(a), there are two global a-locs: mem_0 (for offsets $0 \ldots 3$) and mem_4 (for offsets $4 \ldots 7$).

Local a-locs: Local a-locs are determined on a per-procedure basis. The x86 convention is to access local variables either via the stack pointer (esp) or the frame pointer (ebp). For each instruction in a procedure, the difference between the value of esp or ebp at that point and the value of esp at procedure entry is determined by interpreting the register-only instructions and assuming that calls do not change the values of registers. This difference is called the sp_delta (borrowing the terminology from IDAPro). In the program from Fig. 2(a), the sp_delta values for instructions 1 and 2 are 0 and -44, respectively.

The sp_delta values are not safe, because interprocedural side effects and memory reads/writes are not considered. However, the value-sets produced by the value-set analysis algorithm are still safe, because these sp_delta values are used only to get an approximate layout of the activation record of the procedure; during value-set analysis, the correct sp_delta values are determined, and hence the correct a-locs, too.

Once the sp_delta values are determined, all instructions that use indirect-addressing mode with ebp or esp as the base are identified. In the program from Fig. 2(a), the instructions "lea eax, [esp+4]", "lea eax, [esp+24]" and "mov edi, [esp+0]" use esp-relative addresses. The offsets in these operands are adjusted by the sp_delta values at the instruction. These adjusted offsets will be referred to as *ar_offsets*. The ar_offset for the instruction "mov edi, [esp+0]" is -44. Similarly, for instructions 2 and 3, the ar_offsets are -40 and -20, respectively. These ar_offsets represent the offset referred to by the operand in the memory-region corresponding to the procedure's activation record. The space in each region from one ar_offset up to, but not including the next ar_offset, is identified as a local a-loc. For instance, the local a-locs for procedure main are var_2c, var_28, and var_14.[2] In addition to these a-locs, an a-loc for the return address is also defined; its ar_offset 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].

Heap a-locs: Heap a-locs represent the memory locations in a heap region. One a-loc is defined per heap region.

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 accurately as possible, as described in Sect. 4.1. The offset of an a-loc a in a region rgn will be represented as $\texttt{offset}(rgn, a)$. For example, $\texttt{offset(AR\_main,var\_14)} = -20$ for the program from Fig. 2(a).

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. 2(a), 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, $size = \infty$.

The idea of finding a collection of address descriptors for each of the memory-regions based on the access patterns of the program is similar to the idea behind the Aggregate Structure Identification (ASI) algorithm of Ramalingam et al. [24]. However, ASI cannot be applied to binary code or assembly code without the results of value-set analysis; ASI requires points-to information, and this information is not available for a binary executable before value-set analysis. The good news is that ASI is complementary to value-set analysis; in Appendix A, we describe how we plan to use ASI in conjunction with value-set analysis so that clients of value-set analysis—such as dependence analysis—compute more precise results.

3.3 Abstract Stores

An abstract store should approximate the set of mem-

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

ory 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 $\{aZ+d | Z \in [b,c]\}$.[3] For instance, {1, 3, 5, 7} is represented as the tuple $(2,0,3,1)$.

An abstract store is a value of the following type:

$$\text{a-loc} \rightarrow (\text{memory-region} \rightarrow \text{RIC}).$$

For instance, for the program from Fig. 2(a), 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 $[(\texttt{Global} \mapsto \bot), (\texttt{AR\_main} \mapsto 4[0,4] - 40)]$.

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 $(\bot, 4[0,4] - 40)$.

We chose to use RICs because in our context, it is important for the analysis to discover stride and alignment 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 overwrite portions of two variables. 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 $-8[0,2] - 20$.)

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

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 address 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 $(vs_1 \nabla vs_2) = (10, 4[0,\infty])$.[4]
- $(vs \boxplus c)$: Returns the value-set obtained by adjusting all the addresses 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 ∞.

3.4 Representing Abstract Stores Efficiently

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 [27, 22], 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. Thus, new nodes are introduced for each of the original nodes along the path from M to the root of the tree. Because an operation that restructures a standard balanced tree may modify all of the nodes on the path to the root anyway, and because a single operation on a standard balanced tree that has n nodes takes at most $O(\log n)$ steps, the same operation on an applicative

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

balanced tree introduces at most $O(\log n)$ additional nodes and also takes at most $O(\log n)$ steps. The new tree resulting from the operation shares the entire structure of the original tree except for the nodes on a path from M' to the root, plus at most $O(\log n)$ other nodes that may be introduced to maintain the balance properties of the tree. In our implementation, the abstract values from the value-set analysis domain are implemented using applicative AVL trees [22].

4. VALUE-SET ANALYSIS

This section describes the value-set analysis algorithm. Value-set analysis is an abstract interpretation of the binary executable to find a safe approximation for the set of values that each data object holds at each program point. Value-set analysis uses the domain of abstract stores defined in Sect. 3. Value-set analysis is flow-sensitive and context-insensitive.[5]

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

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

1. `R1=R2+`c
2. `*(R1+`c_1`)=R2+`c_2
3. `R1=*(R2+`c_1`)+`c_2
4. `if(boolean expression) then Label1 else Label2`

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. 5. 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.

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 can be performed at any node of a cycle in the CFG. However, the node at which widening is done affects the accuracy of value-set analysis. Hence, widening points need to be chosen carefully. Our implementation uses some of the strategies outlined in [3] to choose widening points.

EXAMPLE 2. For the program from Fig. 2(a), the abstract value for the entry node of `main` is {`esp` $\mapsto (\perp, 0)$, `mem_0` $\mapsto (0, \perp)$, `mem_4` $\mapsto (1, \perp)$}.

Fig. 6 shows the results of value-set analysis applied to Fig. 2. 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`, `eax` $\mapsto (\perp, 4[0, \infty] - 40)$, and thus $*(as_{eax} \boxplus 0, 4)$ returns the set of a-locs {`var_28, var_14, ret_main`}. Similarly at statement `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`, `ebx` $\mapsto (\perp, 4[0, \infty] - 20)$, 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. 7 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. ⊠

4.2.1 *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

[5]The present implementation is context-insensitive. In the near future, we plan to extend it to have a degree of context-sensitivity, using the call-strings approach to interprocedural dataflow analysis [32].

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

*(R1+c_1)=R2+c_2:	let $[\texttt{R1} \mapsto vs_1], [\texttt{R2} \mapsto vs_2] \in \texttt{e.Before}, (F, P) = *(vs_1 \boxplus c_1, s)$
	and tmp $= \texttt{e.Before} - \{[p \mapsto *] \mid p \in P\} \cup \{[p \mapsto \top] \mid p \in P\}$
	if $\lvert F\rvert = 1$ and $\lvert P\rvert = 0$ then
	$\texttt{e.After} := \texttt{tmp} - \{[v \mapsto *] \mid v \in F\} \cup \{[v \mapsto vs_2 \boxplus c_2] \mid v \in F\}$
	else
	$\texttt{e.After} := \texttt{tmp} - \{[v \mapsto *] \mid v \in F\} \cup \{[v \mapsto (vs_2 \boxplus c_2) \sqcup vs_v] \mid v \in F, [v \mapsto vs_v] \in \texttt{e.Before}\}$

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

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

Figure 5: Transfer functions for value-set analysis. (In the second and third cases, s represents the size of the dereference performed by the instruction.)

procedure can be determined as follows:

$$\text{Number of formals} = \frac{\texttt{(max(ar\_offsets)-sizeof(return address))} \times 8}{\texttt{stack\_width}}$$

where `stack_width` is 32 for 32-bit executables and 16 for 16-bit executables.

For our example, the maximum ar_offset is 12, and the size of the return address is 4; hence, the number of formals is 2.

4.2.2 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`.[6] 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 of the call. Fig. 8 shows the extended variables for procedure `main` and the formal parameters for procedure `initArray` for the program in Example 3.

[6]The stack can grow deeper due to function 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`.

4.2.3 Handling of calls and returns

The interprocedural algorithm is similar to the intraprocedural algorithm, but analyzes the supergraph of the binary 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 fixed-point 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 al-

```
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)
```

Figure 6: Selected results of value-set analysis for Fig. 2(a). (A-locs with value ⊤ are not shown.)

.

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

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

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

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

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

Figure 7: Interprocedural example

ways some constant amount. In such cases, `esp` is restored to the value before the call plus/minus the change. If the analysis cannot determine that the change is a constant, then value-set analysis terminates with an error message.

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. 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 from Fig. 2(a), `eax`, `esp`, and `ecx` are all related by the affine relation `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

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

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

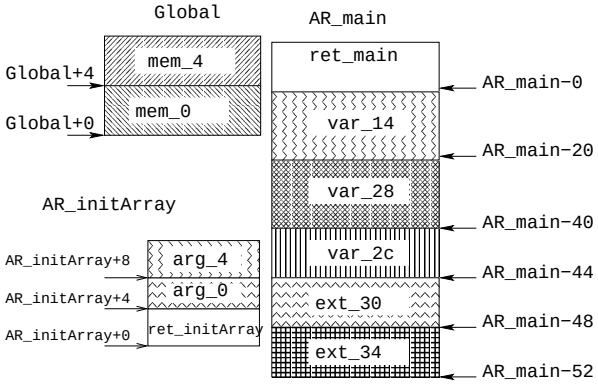


Figure 8: Interprocedural memory-regions

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.

Müller-Olm and Seidl [20] recently proposed an interprocedural dataflow-analysis algorithm to determine, for each program point, all affine relations that hold among a set of global variables.[7] Their 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 of affine relations 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 [20], 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, $O(k^{10})$, 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, a variant of the algorithm exists that reduces the constant to $O(k^8)$ [20].

It is important to remember, however, 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. [26] describe how interprocedural dataflow-analysis problems can be reduced to path queries in Weighted Pushdown Systems (WPDS). Our implementation of the Müller-Olm/Seidl algorithm makes use of the Weighted PDS Library [31], which implements the techniques from [26].

[7] Extensions required to handle local variables are also described in [20]. The algorithm that deals with global variables is sufficient for our purposes, because we use it only to find affine relations involving registers.

6. INDIRECT JUMPS AND CALLS

The supergraph of the program will not be complete in the presence of indirect jumps and 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 {`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 binary executable's headers. 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 are handled similarly, with the following exceptions:

- A successor instruction identified by the technique outlined above may be in the middle of a procedure. In such cases, a call edge is added to the starting instruction of the procedure rather than to the instruction itself. Because this is likely to cause the analysis results to be unsafe, the analysis reports this as an unsafe instruction.

- 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 invokes IDAPro's procedure-finding algorithm explicitly; this forces IDAPro to decode a sequence of bytes from the binary executable into a sequence of instructions and splice it into the intermediate representation of the program.

(The techniques described in this section have not yet been implemented in CodeSurfer/x86.)

7. PERFORMANCE EVALUATION

Table 1 shows the running times and storage requirements[8] of our prototype implementation for a set of common Linux programs (the program version is shown in parentheses).

[8] Our prototype implementation has a known storage leak in the affine-relations phase; hence, the figures for storage utilization are bloated. We will be switching to a new implementation of this phase in the near future.

Program	Procedures	Instructions	malloc	Indirect jumps	Indirect calls	Memory usage (MB)	Value-set analysis (s)	Affine Relations (s)	Coverage (%)
cat (2.0.14)	123	3814	1	7	4	49	0.48	5.21	49.14
cut (2.0.14)	129	4246	2	7	4	65	0.72	18.07	56.76
grep (2.4.2)	245	16682	18	14	6	380	1.89	83.65	27.43
gcc (2.96)	252	22842	8	7	5	1276	6.18	2229.08	97.05
flex (2.5.4)	239	23373	0	11	3	886	8.61	389.20	97.62

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

Measurements were taken on a Pentium-4 with a clock speed of 2.4GHz, equipped with a physical memory of 4GB and running Windows 2000. (The per-process address space was limited to 3GB.) The prototype implementation does not handle indirect calls, and uses the information provided by IDAPro for indirect jumps. (IDAPro identifies targets of indirect jumps if the jump involves a read-only jump table.) Hence the value-set analysis and affine-relations analysis cover only a part of the program (the coverage percentage is also shown). The performance evaluation of the prototype is encouraging, and suggests that the analysis presented in the paper can be used to build binary-analysis tools that have acceptable performance.

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 could 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 binary executables. The work that is most closely related to value-set analysis is the alias-analysis algorithm for executable programs 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 a binary 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. This greatly limits the utility of the approach.

Past work on decompiling assembly code to a high-level language is also related to our goals [6, 4, 21]. Unfortunately, the decompilers reported in the literature perform only a best-effort translation to high-level code, and hence the results from such tools are also of limited utility.

Several people have developed techniques to analyze binary executables in the presence of additional information, such as the source code, symbol-table information, or debugging information [18, 2, 1, 28]. 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. present a static-analysis technique—implemented for programs written in C—whose aim is to identify string-manipulation errors, such as potential buffer overruns [12]. As in our work, the analysis tracks both pointer-valued and integer-valued quantities; it is instructive to compare the two approaches.

In the work of Dor et al., 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 relationships 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 was already discussed in Sect. 3.3, it is important for the analysis to discover stride and alignment 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.

Our analysis combines pointer analysis with numeric analysis, whereas the analysis of Dor et al. uses 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 stride and alignment 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. 5).

In our work, we also face other challenges, such as not knowing anything about the program's variables when the analysis begins; this is addressed via the notion of a-locs. As discussed in App. A, we plan to use aggregate structure identification [24] in conjunction with the results of value-set analysis to refine the set of a-locs.

Thus, although both analyses use pointer analysis and numeric analysis, the requirements of the two analyses are much different—and hence the resulting static analyses are quite different as well.

Xu et al. [34] also created a system that analyzed binary code 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 [33] to synthesize loop invariants and Omega [23] 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.

10. ACKNOWLEDGMENT

We thank GrammaTech, Inc. for providing the basic infrastructure for CodeSurfer/x86, and R. Gruian for creating interfaces to IDAPro and CodeSurfer that allowed us to interface value-set analysis with those tools. We also thank H. Seidl for making available reference [20], and S. Schwoon for implementing the Weighted PDS Library [31].

11. REFERENCES

[1] J. Bergeron, M. Debbabi, J. Desharnais, 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. Erhioui, and B. Ktari. Static analysis of binary code to isolate malicious behaviors. In *WETICE*, pages 184–189, 1999.

[3] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of 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, Dept. of Comp. Sci. and Elec. Eng., Dec. 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, 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 *Symp. on Princ. of Prog. Lang.*, pages 238–252, 1977.

[10] P. Cousot and R. Cousot. Automatic discovery of linear restraints among variables of a program. In *Symp. on Princ. of Prog. Lang.*, pages 84–97, 1978.

[11] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Symp. on 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 *Conf. on Prog. Lang. Design and Impl.*, pages 155–167, 2003.

[13] Fast library identification and recognition technology, http://www.datarescue.com/idabase/flirt.htm.

[14] P. Granger. Static analysis of arithmetic congruences. *Int. J. of Comp. Math.*, pages 165–199, 1989.

[15] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. 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, Jan. 1990.

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

[18] J. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Conf. on Prog. Lang. Design and Impl.*, pages 291–300, 1995.

[19] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the Sapphire/Slammer worm. Technical report, CAIDA, ICSI, Silicon Defense, UC Berkeley EECS, and UC San Diego CSE, Jan. 2003.

[20] M. Müller-Olm and H. Seidl. Computing interprocedurally valid relations in affine programs. Tech. rep., Comp. Sci. Dept., Univ. of Trier, Trier, Ger., Jan. 2003. http://www.informatik.uni-trier.de/~seidl/papers/affine.ps.

[21] A. Mycroft. Type-based decompilation. In *European Symp. on Programming*, 1999.

[22] E. Myers. Efficient applicative data types. In *Symp. on Princ. of Prog. Lang.*, pages 66–75, 1984.

[23] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[24] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Symp. on Princ. of Prog. Lang.*, pages 119–132, 1999.

[25] T. Reps and G. Rosay. Precise interprocedural chopping. In *Symp. Found. of Softw. Eng.*, Oct. 1995.

[26] T. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Static Analysis Symp.*, 2003.

[27] 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.

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

[29] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE J. Sel. Areas in Commun.*, 21(1):5–19, Jan. 2003.

[30] F. 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] S. Schwoon. Weighted PDS library, 2003. "http://www.fmi.uni-stuttgart.de/szs/tools/wpds/".

[32] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[33] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *Symp. on Princ. of Prog. Lang.*, pages 132–143, 1977.

[34] Z. Xu, B. Miller, and T. Reps. Safety checking of machine code. In *Conf. on Prog. Lang. Design and Impl.*, pages 70–82, 2000.

APPENDIX

A. IMPROVING THE A-LOC ABSTRACTION

The a-loc abstraction is not powerful enough to represent structures. It can only represent a contiguous sequence of memory locations, with no known substructure. Therefore, the used, killed, and possibly-killed sets may include more a-locs than necessary, which may affect the accuracy of clients of this information.

EXAMPLE 4. Consider the variant of Example 1 shown in Fig. 9. The program initializes all elements of an array `p`. The `x` members of each element are initialized with 1 and the `y` members are initialized with 2. The corresponding assembly-language instructions are "`L1: mov [eax],1`" and "`5 mov [eax+4], 2`"; instruction `L1` updates the `x` members, and instruction `5` updates the `y` members.

The memory-regions and a-locs for the program are shown in Fig. 10. A backward slice with respect to "`10 mov eax,[esp+4]`" (i.e., "`return p[0].y`") should not include "`L1: mov [eax], 1`". However, slicing with the dependence graph that would be constructed by the version of value-set analysis that has been described thus far will include the instruction "`L1: mov [eax], 1`".

Note that the value-set for `eax` at instruction `L1` is $(\bot, 8[0,4]-40)$ and the value-set for `esp+4` at instruction `10` is $(\bot, -36)$. Therefore, value-set analysis correctly determines the fact that the locations accessed by `[esp+4]` in instruction `10` and `[eax]` in instruction `L1` must always be disjoint. However, the a-locs modified at `L1` are {`var_20`, `var_1c`}, and the a-loc used at `10` is `var_1c`. Consequently, `10` is considered to be data dependent on `L1`. The problem stems from the fact that, in cases like this example, a-locs are too coarse-grained a representation of used, killed, and possibly-killed memory locations, and this leads to a loss of precision. ⊠

```
typedef struct {                    proc main
    int x,y;                     1  sub esp,40
} Point;                         2  mov ecx,0
                                 3  lea eax,[esp]
int main(){                     L1: mov [eax],1
    int i;                       5  mov [eax+4],2
    Point px[10];                6  add eax, 8
    for(i=0;i<5;++i) {           7  inc ecx
       p[i].x=1;                 8  cmp ecx, 5
       p[i].y=2;                 9  jl L1
    }                           10  mov eax,[esp+4]
    return p[0].y;              11  add esp,40
}                               12  retn
```

Figure 9: A program that uses an array of structures.

A similar problem arises when analyzing source-code programs. As illustrated in Sect. 1, there can be a loss of precision if a static-analysis algorithm does not distinguish between accesses to different parts of the same aggregate (cf. Example 1). Ramalingam et al. [24] developed the Aggregate Structure Identification (ASI) algorithm to distinguish among accesses to different parts of an aggregate, and showed how the results of ASI can improve the results of dataflow analysis. Their technique ignores the type declaration for an aggregate and considers the aggregate to be merely a sequence of bytes of a given length. The aggregate is then broken up into smaller parts depending upon how the aggregate is accessed by the program. These smaller parts of the aggregate are referred to as atoms.

Atoms are similar to a-locs, but the unification process used in ASI allows atoms to better reflect the access patterns

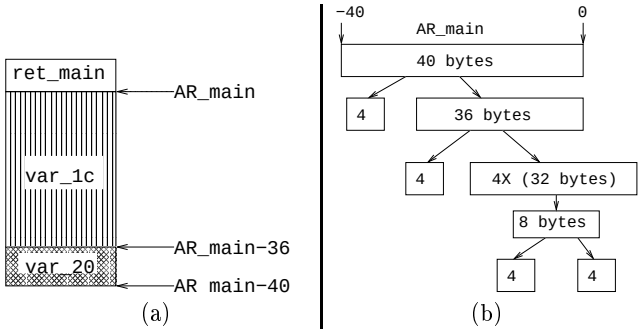


Figure 10: (a) a-locs for example 4; (b) results of aggregate structure identification applied to Example 4.

of the program. However, ASI is not a replacement for value-set analysis; ASI requires more information to be at hand than is available at the beginning of value-set analysis—such as points-to information, sizes of arrays, and iteration counts for loops. Fortunately, information of this sort is available *after* value-set analysis has been carried out, which means that ASI can be used in conjunction with value-set analysis to obtain improved results: after a first round of value-set analysis, the results of ASI are used to refine the a-loc abstraction, after which value-set analysis is run again. This can be carried out for as many rounds as desired, or until no further changes occur.

EXAMPLE 5. Fig. 10 shows the structure identified for the array p in the program from Fig. 9 (see Example 4). The algorithm would identify array p to have the following structure:

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

When the results of the ASI algorithm are used to identify a more refined set of a-locs, the a-locs defined by instruction `L1` are {`AR_main.m3[0..4].m3_m1`, `AR_main.m1`}, and the a-locs used at `10` are {`AR_main.m2`}. Because these sets do not overlap, this abstraction determines that `10` is not data dependent on `L1`. (In this example, it was not even necessary to re-run value-set analysis to obtain improved results.) ⊠