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 by the executable.

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. A major stumbling block when developing binary-analysis tools is that it is difficult to understand memory operations because machine-language instructions use explicit memory addresses and indirect addressing. 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 §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 normally support two addressing modes for memory operands: direct and indirect. In direct addressing, the

⋆ Supported by ONR contracts N00014-01-1-{0708,0796} and NSF grant CCR-9986308.

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 typically 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.

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:

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 address of global variable `part1Value` is 4 and that of `part2Value` is 8.
- 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

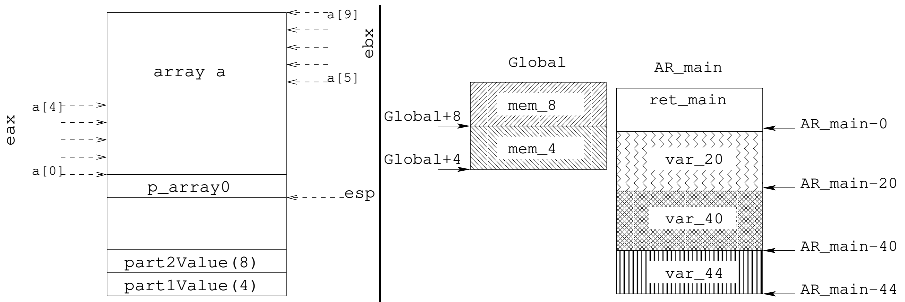
```
int part1Value=0;
int part2Value=1;

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          ;
    sub esp, 44        ;Adjust esp for locals
    lea eax, [esp+4]   ;part1=&a[0]
    lea ebx, [esp+24]  ;part2=&a[5]
    mov [esp+0], eax   ;p_array0=part1
    mov ecx, 0         ;i=0
L1: mov edx, [4]       ;
    mov [eax], edx     ;*part1=part1Value
    mov edx, [8]       ;
    mov [ebx], edx     ;*part2=part2Value
    add eax, 4         ;part1++
    add ebx, 4         ;part2++
    inc ecx            ;i++
    cmp ecx, 5         ;
    jl L1              ;(i<5)?loop:exit
    mov edi,[esp+0]    ;
    mov eax, [edi]     ;set return value
    add esp, 44        ;
    retn               ;return *p_array0
```

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

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, not just the underlined ones. ⊠



(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 §9 for a more detailed discussion of related work.)

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

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 VSA fits into the CodeSurfer/x86 implementation.

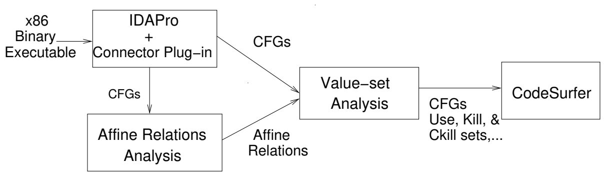


Fig. 3. Organization of CodeSurfer/x86.

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. §6 discusses techniques

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 augments IDAPro's data structures. VSA is implemented using the data structures created by the Connector. As described in §5, VSA 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 VSA completes, the value-sets for the a-locs at each program point are used to determine each point's sets of used, killed, and possibly-killed a-locs; 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 an API 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 [16] and chopping [25]. The 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 (ARs) are pushed on procedure entry and popped on procedure exit; each global variable resides at a fixed offset in memory; each local variable of a procedure f reside at a fixed offset in the ARs for f; actual parameters of f are pushed onto the stack by the caller so that the corresponding formal parameters reside at fixed offsets in the ARs 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 §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, VSA is capable of recovering information from programs that use global variables, local variables, pointers, structures, arrays, heap-allocated storage, pointer arithmetic,

for using the abstract stores computed during VSA to augment the call graph and control-flow graphs on-the-fly to account for indirect jumps and indirect calls.

indirect jumps, recursive procedures, and indirect calls through function pointers (but not runtime code generation or self-modifying code).

- Compiler optimizations often make VSA *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. 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 VSA (see §6). In fact, the relationship between VSA 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 VSA 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 ARs, 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`). VSA 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-AR. 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. There is a single procedure, and hence two regions: one for global data and one for the AR 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 ARs, 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,8)`.

At the enter node of a procedure `P`, register `esp` points to the start of the AR 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 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 memory operations would have to be treated conservatively, which would lead to very imprecise data dependences. Instead, we use what we call the *a-loc* abstraction to track (an over-approximation of) the values of memory locations.

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 before generating the executable, i.e., the compiler or the assembly-programmer 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 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 `[4]` and `[8]`. Therefore, we have two a-locs: `mem_4` (for addresses $4..7$) and `mem_8` (for addresses $8..11$). 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 AR 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` at these operands is `-44`. Thus, these memory references correspond to the offsets `-44`, `-40`, and `-20` in the memory-region for `AR_main`. This gives rise to three more a-locs: `var_44`, `var_40`, and `var_20`. 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_44` corresponds to all of the source-code variable `p_array0`. In contrast, `var_40` and `var_20` correspond to disjoint segments of array `a[]`: `var_40` corresponds to `a[0..4]`; `var_20` corresponds to `a[5..9]`.

Similarly, we have one a-loc per heap-region. In addition to these a-locs, registers are also considered to be a-locs.

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. The offset of an a-loc a in a region rgn will be denoted by `offset`(rgn, a). For example, for the program from Fig. 1, `offset(AR_main,var_20)` is `-20`.

Addresses of an a-loc: The addresses that belong to an a-loc a can be represented by a pair $(rgn, [\textit{offset}, \textit{offset} + size - 1])$, 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_20` are $(\texttt{AR\_main}, [-40, -21])$. 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 must over-approximate the set of memory addresses that each a-loc holds at a particular program point. As described in §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 {1,3,5,9} can be over-approximated as the RIC $(2\mathbb{Z} + 1) \cap [0, 9]$. 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 $\{aZ + d | Z \in [b, c]\}$.[2] For instance, {1, 3, 5, 9} is over-approximated by the tuple $(2, 0, 4, 1)$, which equals $\{1, 3, 5, 7, 9\}$.

An *abstract store* is a value of type a-loc → (memory-region → RIC). For conciseness, the abstract stores 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 → RIC^r. 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 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.

When the contents of an a-loc `a` is not aligned with the boundaries of a-locs, a memory access on `*a` can fetch portions of two a-locs; similarly, a write to `*a` can overwrite portions of two a-locs. Such operations can be used to forge new addresses. For instance, suppose that the address of a-loc `x` is `1000`, the address of a-loc `y` is `1004`, and the contents of `a` is `1001`. Then `*a` (as a 4-byte fetch) would retrieve 3 bytes of `x` and 1 byte of `y`.

This issue motivated the use of RICs because RICs are capable of representing certain non-convex sets of integers, and ranges (alone) are not. Suppose that the contents set of `a` is {`1000, 1004`}; then `*a` (as a 4-byte fetch) would retrieve `x` or `y`. The range [1000, 1004] includes the addresses `1001`, `1002`, and `1003`, and hence *[1000, 1004] (as a 4-byte fetch) could result in a forged address. However, because VSA is based on RICs, {`1000, 1004`} is represented exactly, as the RIC 4[0,1]+1000. If VSA 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.

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.
- $(vs_1 \sqcap vs_2)$: Returns the intersection (meet) of value-sets vs_1 and vs_2.

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

– $(vs_1 \sqcup vs_2)$: 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, e.g., if $vs_1 = (10, 4[0, 1])$ and $vs_2 = (10, 4[0, 2])$, then $(vs_1 \nabla vs_2) = (10, 4[0, \infty])$.
– $(vs \boxplus c)$: Returns the value-set obtained by adjusting all values in vs by the constant c, e.g., if $vs = (4, 4[0, 2] + 4)$ and $c = 12$, then $(vs \boxplus c) = (16, 4[0, 2] + 16)$.
– $*(vs, s)$: 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 [26, 21].

4 Value-Set Analysis (VSA)

This section describes the value-set analysis algorithm. VSA 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 §3. The present implementation of VSA is flow-sensitive and context-insensitive.[3]

VSA 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, VSA determines an over-approximation of the set of addresses that each data object can hold at each program point. The results of VSA can also be used to find the a-locs whose addresses a given a-loc a contains. On the other hand, VSA 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, VSA 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 VSA. 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:

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

[3] 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 [29].

Label on e	Transfer function for edge e
R1=R2+c	**let** (R2 $\mapsto vs) \in$ e.Before e.After $:=$ e.Before $-$ [R1 $\mapsto *] \cup$ [R1 $\mapsto vs \boxplus c$]
*(R1+c_1)=R2+c_2	**let** [R1 $\mapsto vs_{R1}$], [R2 $\mapsto vs_{R2}$] $\in$ e.Before, $(F, P) = *(vs_{R1} \boxplus c_1, s)$, tmp $=$ e.Before $- \{[a \mapsto *] \mid a \in P \cup F\} \cup \{[p \mapsto \top] \mid p \in P\}$, 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 $\cup \{[v \mapsto vs_{R2} \boxplus c_2] \mid v \in F\}$) // Strong update **else** // Weak update e.After $:=$ (tmp $\cup \{[v \mapsto (vs_{R2} \boxplus c_2) \sqcup vs_v] \mid v \in F, [v \mapsto vs_v] \in$ e.Before$\}$)

R1=*(R2+c_1)+c_2	**let** (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 $-$ [R1 $\mapsto *] \cup$ [R1 $\mapsto (vs_{rhs} \boxplus c_2)$]
	else
	e.After $:=$ e.Before $-$ [R1 $\mapsto *] \cup$ [R1 $\mapsto \top$]

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

Fig. 4. Transfer functions for VSA. (In cases 2 and 3, s represents the size of the dereference performed by the instruction.)

Conditions of the last two forms are obtained from the predecessor(s) of conditional jump instructions that affect condition codes.

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`$\rightarrow$`L1` will be labeled `ecx`$<$`5`, whereas the edge `14`$\rightarrow$`15` will be labeled `ecx`$\geq$`5`. Once we have the CFG, an abstract store is obtained for each program point by abstract interpretation [8]. Sample transformers for various kinds of edges are listed in Fig. 4. Each transformer takes an abstract store and returns a new abstract store. Because each AR region of a procedure that may be called recursively—as well as each heap region—potentially represents more than one concrete data object, assignments to their a-locs must be modeled by weak updates, i.e., the new value-set must be unioned with the existing one, rather than replacing it (see case two of Fig. 4). Furthermore, unaligned writes can modify parts of various a-locs (which could possibly create forged addresses). In case 2 of Fig. 4, such writes are treated safely by setting the values of all partially modified a-locs to $\top$. Similarly, case 3 treats a load of a potentially forged address as a load of $\top$.

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 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 VSA uses techniques from [3].

Example 2. For the program from Fig. 1, the abstract store for the entry node of `main` is $\{$`esp` $\mapsto (\bot, 0)$, `mem_4` $\mapsto (0, \bot)$, `mem_8` $\mapsto (1, \bot)\}$.

The fixpoint solution of VSA for instruction `7` is $\{$`esp` $\mapsto (\bot, -44)$, `mem_4` $\mapsto (0, \bot)$, `mem_8` $\mapsto (1, \bot)$, `eax` $\mapsto (\bot, 4[0, \infty]-40)$, `ebx` $\mapsto (\bot, 4[0, \infty]-20)$, `var_44`

$\mapsto (\bot, -40)$, `ecx` $\mapsto ([0,4], \bot)\}$ and that of instruction `16` is $\{$`esp` $\mapsto (\bot, -44)$, `mem_4` $\mapsto (0, \bot)$, `mem_8` $\mapsto (1, \bot)$, `eax` $\mapsto (\bot, 4[1,\infty] - 40)$, `ebx` $\mapsto (\bot, 4[1,\infty] - 20)$, `var_44` $\mapsto (\bot, -40)$, `ecx` $\mapsto ([5,5], \bot)$, `edi` $\mapsto (\bot, -40)\}$.

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` $\mapsto (\bot, 4[0,\infty] - 40)$, and thus $*($`eax` $\boxplus\, 0, 4)$ returns the possibly-killed set as $\{$`var_40, var_20, ret_main`$\}$. Similarly, at instruction `16`, $*($`esp` $\boxplus\, 8, 4)$ returns the use set as $\{$`var_40`$\}$. Reaching-definitions analysis based on this information reveals that instruction `16` is data dependent on instruction `7`. Similarly, reaching-definitions analysis reveals that instruction `16` is not data dependent on `9`.

Note that the a-loc `ret_main` is also included in the set of a-locs accessed through `eax` at instruction `7`. This is because the analysis was not able to determine an upper bound for `eax`. Observe that `eax` is dependent on the loop variable `ecx`. We discuss in §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. Further, the instructions that push the actual parameters on the stack need not occur immediately before the call. Example 3 will be used to explain the interprocedural case.

Example 3. Fig. 5 shows a program with two procedures, `main` and `initArray` (see also Fig. 6). Procedure `main` has an integer array `a`, which is initialized by calling `initArray`. After initialization, `main` returns the second element of array `a`. ⊠

Actual parameters and register saves In an x86 program, stack operations like push/pop implicitly modify some locations in the AR of a procedure (say `P`). These locations correspond to the actual parameters of a call and to those used for register spilling and caller-saved registers. The locations accessed by push/pop instructions are not explicitly found as `esp`/`ebp`-relative addresses, and so the algorithm that identifies a-locs will not introduce a-locs for the memory locations accessed by these stack operations; consequently, we introduce additional a-locs, which we call *extended a-locs*, for memory locations that are implicitly accessed by such stack operations. To do this, the smallest `sp_delta` for `P` is determined. This represents the maximum limit to which the stack can grow in a single invocation of `P`. (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 AR for `P`.) If we are

```
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;
}
```

(a) C program

```
    proc initArray
 1  lea eax, [esp+4]
 2  mov ebx, eax
 3  add ebx, 20
 4  mov ecx, 0
L1: mov edx, [4]
 6  mov [eax], edx
 7  mov edx, [8]
 8  mov [ebx], edx
 9  add eax, 4
10  add ebx, 4
11  inc ecx
12  cmp ecx, [esp+8]
13  jl L1
14  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+0]
23  mov eax,[edi]
24  add esp,44
25  retn
```

(b) Disassembly

Fig. 5. Interprocedural example

unable to find a finite minimum, the analysis issues a report. If there is a finite minimum, then *extended a-locs* are added to the AR on 4-byte boundaries to fill the space between the lowest local a-loc and the minimum `sp_delta`.

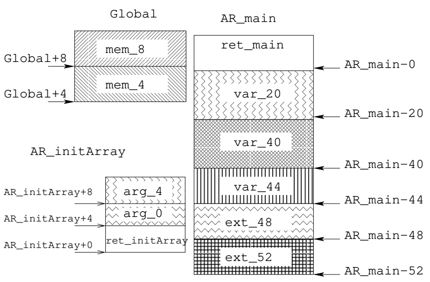


Fig. 6. Memory-regions

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 offsets for the formal parameters will be positive. Hence, a-locs with positive offsets are considered to be the formal parameters.

At a call on a procedure that has k formals, the last k extended a-locs represent the actual parameters. Fig. 6 shows the extended a-locs 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. The call→entry and the exit→end-call edges will be refereed to as *linkage edges*. Nodes, edges and edge-transformers for all other instructions are similar to the intraprocedural CFG.

The transformer for the call→entry edge assigns actuals to formals and also changes `esp` to reflect the change in the current AR. First the join of the abstract stores at the call-sites of `P` is computed; then the value-set of `esp` in the newly computed value is set to $(\bot, \ldots, 0, \ldots, \bot)$, where the 0 occurs in the slot for `P`. In addition, each formal parameter $\texttt{Formal}_i$ is initialized as follows:

$$(F_i^c, P_i^c) = *(as_c[\texttt{esp}] \boxplus (\texttt{offset}(\texttt{AR\_P}, \texttt{Formal}_i) - 4), S_i)$$

$$as_{\texttt{enter\_P}}[\texttt{Formal}_i] = \begin{cases} \top & \#\text{ if } \bigcup\limits_{c \in \texttt{callsites(P)}} P_i^c \neq \phi \\ \bigsqcup\limits_{c \in \texttt{callsites(P)}, v \in F_i^c} as_c[v] & \#\text{ otherwise} \end{cases}$$

where as_c is the abstract store at call-site c of `P` and S_i is the size of $\texttt{Formal}_i$. That is, the value-set for a formal on entry is the join of the value-sets of the corresponding actuals at the callers. The offset of the actual in the AR of the caller is determined from the offset of the formal parameter. In the fixpoint solution for Example 3, the abstract store for the enter node of `initArray` is: $\{\texttt{mem\_4} \mapsto (0, \bot, \bot)$, $\texttt{mem\_8} \mapsto (1, \bot, \bot)$, $\texttt{arg\_0} \mapsto (\bot, -40, \bot)$, $\texttt{arg\_4} \mapsto (5, \bot, \bot)$, $\texttt{eax} \mapsto (\bot, -40, \bot)$, $\texttt{esp} \mapsto (\bot, \bot, 0)$, $\texttt{ext\_48} \mapsto (5, \bot, \bot)$, $\texttt{ext\_52} \mapsto (\bot, -40, \bot)\ \}$. The regions in the value-sets are listed in the following order: (`Global`, `AR_main`, `AR_initArray`).

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. VSA 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 VSA cannot determine that the change is a constant, then it issues an error report.

5 Affine Relations

Recall that in Example 2, VSA 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 VSA 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 [19].

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 VSA 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 $([0, 4], \perp)$. (A value-set in which all RICs are $\perp$ 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 $(\perp, -44)$. Using these value-sets and solving for eax in the above relation yields

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

In this way, a sharper value for eax at L1 is obtained than would otherwise be possible; Such bounds cannot be obtained for loops that are controlled by a condtion that is not based on indices; however, the analysis is still safe in such cases.

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 store 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 VSA 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 VSA 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 store.

The performance evaluation in §7 uses a version of affine-relation analysis that models the restoration of callee-save registers across calls. (At present, certain technical difficulties preclude a similar treatment of caller-save registers. We have also not yet implemented a check to determine that the code obeys the calling conventions for caller-save and callee-save registers.)

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 VSA. For instance, suppose that VSA is interpreting an indirect jump instruction `J1: jmp 1000[eax*4]`, and let the current abstract store at this instruction be $\{$`eax` $\mapsto ([0,9], \perp, \ldots, \perp)$. 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 store at `J1`. Due to possible imprecision in VSA, it could be the case that VSA reports that the locations {1000, 1004, ..., 1036} have all possible addresses. In such cases, VSA 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 VSA 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 a few additional complications.

- A successor instruction identified by the method outlined above may be in the middle of a procedure. In such cases, the analysis reports 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, VSA 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. (At present, this technique has not yet been incorporated in our implementation.)

7 Performance Evaluation

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.)

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	51	76	29
cat (2.0.14)	123	3892	1	3	138	4	42	9	26
cut (2.0.14)	129	4329	2	3	182	4	48	7	42
grep (2.4.2)	245	16808	18	4	654	6	102	117	75
gcc (2.96)	252	22984	8	3	1048	4	232	108	295
tar (1.13.19)	581	47739	11	21	2553	29	258	220	156
awk (3.1.0)	595	69927	84	33	3669	152	623	1017	1011
winhlp32 (5.0.2195.2014)	1018	108380	0	10	6002	1005	737	1712	1290

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

To contrast the capabilities of VSA 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 VSA, called *crude VSA*, 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 VSA at all (which causes dependences to be missed); (ii) with VSA; and (iii) with crude VSA.

8 Soundness Issues

Program	No VSA	VSA	Crude VSA
javac	21597	52884	54996
cat (2.0.14)	17932	32826	33632
cut (2.0.14)	23116	37834	39116
grep (2.4.2)	123293	201584	217003
gcc (2.96)	320089	5921020	5970559
tar (1.13.19)	644518	4088659	4305446

Table 2. Comparison of 3 variants of VSA.

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. There are less lofty goals that do not meet this standard—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.

Here are some of the cases in which the analysis can be unsound, but where the system generates 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 §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 modifiable location.

9 Related Work

There is an extensive body of work on analyzing executables. The work that is most closely related to VSA is the alias-analysis algorithm for executables proposed by Debray et al. [11]. The basic goal of their algorithm is similar to that of VSA: 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]. However, that work has also not done much to address the problem of recovering information about memory accesses.

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. [24]. However, ASI cannot be applied to x86 code without having the results of VSA already in hand: ASI requires points-to, range, and stride information; however, this information is not available for an x86 executable until after VSA. The good news is that ASI can be applied after VSA 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. [31] 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 [30] 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. VSA 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 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 stores 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 §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. xFor this reason, the numeric component of VSA 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. §3.3). Moreover, additional benefits can accrue to clients of VSA; for instance, it can happen that extra precision will allow VSA 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). The advantages of combining pointer analysis with numeric analysis have been studied in [22]. In the context of [22], combining the two analysis only improves precision. However, in our context, a combined analysis is needed to ensure safety.

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. F. 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, 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. Precise interprocedural analysis through linear algebra. 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. A. Pioli and M. Hind. Combining interprocedural pointer analysis and conditional constant propagation. Tech. Rep. RC 21532(96749), IBM T.J. Watson Research Center, March 1999.
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 *Princ. of Prog. Lang.*, pages 119–132, 1999.
25. T. Reps and G. Rosay. Precise interprocedural chopping. In *Found. of Softw. Eng.*, 1995.
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. 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.
30. N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *Princ. of Prog. Lang.*, pages 132–143, 1977.
31. Z. Xu, B. Miller, and T. Reps. Safety checking of machine code. In *Prog. Lang. Design and Impl.*, pages 70–82, 2000.