Intermediate-Representation Recovery from Low-Level Code *

Thomas Reps Gogul Balakrishnan Junghee Lim
University of Wisconsin-Madison
{reps,bgogul,junghee}@cs.wisc.edu

Abstract

The goal of our work is to create tools that an analyst can use to understand the workings of COTS components, plugins, mobile code, and DLLs, as well as memory snapshots of worms and virus-infected code. This paper describes how static analysis provides techniques that can be used to recover intermediate representations that are similar to those that can be created for a program written in a high-level language.

1. Introduction

In the past five years, there has been a considerable amount of research activity to develop static-analysis tools to find bugs and security vulnerabilities [27, 45, 25, 20, 12, 8, 14, 28, 22]. However, most of the effort has been on static-analysis of source code, and the issue of analyzing executables has largely been ignored. In the security context, this is particularly unfortunate because source-code analysis can fail to detect certain vulnerabilities due to the WYSINWYX phenomenon: "**W**hat **Y**ou **S**ee **I**s **N**ot **W**hat **Y**ou e**X**ecute". That is, there can be a mismatch between what a programmer intends and what is actually executed on the processor.

The following source-code fragment, taken from a login program, is an example of such a mismatch [30]:

```
memset(password, '\0', len);
free(password);
```

The login program temporarily stores the user's password—in clear text—in a dynamically allocated buffer pointed to by the pointer variable `password`. To minimize the lifetime of the password, which is sensitive information, the code fragment shown above zeroes-out the buffer pointed to by `password` before returning it to the heap. Unfortunately, a compiler that performs useless-code elimination may reason that the program never uses the values written by the call on `memset` and therefore the call on `memset` can be removed, thereby leaving sensitive information exposed in the heap. This is not just hypothetical; a similar vulnerability was discovered during the Windows security push in 2002 [30]. This vulnerability is invisible in the source code; it can only be detected by examining the low-level code emitted by the optimizing compiler.

* Portions of this paper have appeared in [4, 7, 38]. This research was supported in part by NSF under grants CCF-0524051 and CCR-9986308, and by ONR under contracts N00014-01-1-{0796,0708}.

PEPM '06 January 9–10, 2006, Charleston, South Carolina, USA.

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

There are a number of reasons why analyses based on source code do not provide the right level of detail for checking certain kinds of properties:

- Source-level tools are only applicable when source is available, which limits their usefulness in security applications (e.g., to analyzing code from open-source projects).
- Analyses based on source code typically make (unchecked) assumptions, e.g., that the program is ANSI-C compliant. This often means that an analysis does not account for behaviors that are allowed by the compiler (e.g., arithmetic is performed on pointers that are subsequently used for indirect function calls; pointers move off the ends of arrays and are subsequently dereferenced; etc.)
- Programs typically make extensive use of libraries, including dynamically linked libraries (DLLs), which may not be available in source-code form. Typically, source-level analyses are performed using code stubs that model the effects of library calls. Because these are hand-crafted, they are likely to contain errors, which may cause an analysis to return incorrect results.
- Programs are sometimes modified subsequent to compilation, e.g., to perform optimizations or insert instrumentation code [46]. (They may also be modified to insert malicious code.) Such modifications are not visible to tools that analyze source.
- The source code may have been written in more than one language. This complicates the life of designers of tools that analyze source code because multiple languages must be supported, each with its own quirks.
- Even if the source code is primarily written in one high-level language, it may contain inlined assembly code in selected places. Source-level analysis tools typically either skip over inlined assembly code [18] or do not push the analysis beyond sites of inlined assembly code [1].

Thus, even if source code is available, a substantial amount of information is hidden from analyses that start from source code, which can cause bugs, security vulnerabilities, and malicious behavior to be invisible to such tools. Moreover, a source-level analysis tool that strives to have greater fidelity to the program that is actually executed would have to duplicate all of the choices made by the compiler and optimizer; such an approach is doomed to failure.

The long-term goal of our work is to develop bug-detection and security-vulnerability analyses that work on executables. The advantage of this approach is that an executable contains the actual instructions that will be executed, and hence provides information that reveals the actual behavior that arises during program execution. Access to such information can be crucial; for instance, many security exploits depend on platform-specific features, such as the structure of activation records. Vulnerabilities can escape notice when a tool does not have information about adjacency relationships among variables.

To be able to apply techniques like the ones used in [27, 45, 25, 20, 12, 8, 14, 28, 22, 13], one already encounters a challenging program-analysis problem. From the perspective of the compiler community, one would consider the problem to be "IR recovery": one needs to recover *intermediate representations* from the executable that are similar to those that would be available had one started from source code. From the perspective of the model-checking community, one would consider the problem to be that of "model extraction": one needs to extract a suitable *model* from the executable. Thus, our immediate goal is to advance the state of the art of recovering, from executables, IRs that are (a) similar to those that would be available had one started from source code, but (b) expose the platform-specific details discussed above. Specifically, we are interested in recovering IRs that represent the following information:

- control-flow graphs (CFGs), with indirect jumps resolved
- a call graph, with indirect calls resolved
- information about the program's variables
- possible values of pointer variables
- sets of used, killed, and possibly-killed variables for each CFG node
- data dependences (including dependences between instructions that involve memory accesses)
- type information (e.g., base types, pointer types, and structs)

Once such IRs are in hand, we will be in a position to leverage the substantial body of work on source-code-vulnerability analysis.

In IR recovery, there are numerous obstacles that must be overcome. In particular, in many situations debugging information is not available. Even if debugging information is present, it cannot be relied upon if the program is potentially malicious. For this reason, we have designed IR-recovery techniques that do not rely on debugging information being present. (Thus, throughout the paper, the term "executable" means a stripped executable.) Our current implementation of IR recovery—which is incorporated in a tool called CodeSurfer/x86 [38]—works on x86 executables; however, the algorithms used are language-independent.

When debugging information is absent, an executable's data objects are not easily identifiable. Consider, for instance, a data dependence from statement `a` to statement `b` that is transmitted by write/read accesses on some variable `x`. When performing source-code analysis, the programmer-defined variables provide us with convenient compartments for tracking such data manipulations. A dependence analyzer must show that `a` defines `x`, `b` uses `x`, and there is an `x`-def-free path from `a` to `b`. However, in executables, memory is accessed either directly—by specifying an absolute address—or indirectly—through an address expression of the form "[*base* + *index* $\times$ *scale* + *offset*]", where *base* and *index* are registers and *scale* and *offset* are integer constants. It is not clear from such expressions what the natural compartments are that should be used for analysis. Because, executables do not have *intrinsic* entities that can be used for analysis (analogous to source-level variables), a crucial step in IR recovery is to identify variable-like entities.

Past work on IR recovery from executables has relied on some simple techniques for identifying variable-like entities. For instance, IDAPro [31], a commercial disassembly toolkit, recovers variables based on statically known-addresses and stack-frame offsets. However, this approach has certain limitations. For instance, it generally recovers only very coarse information about arrays.

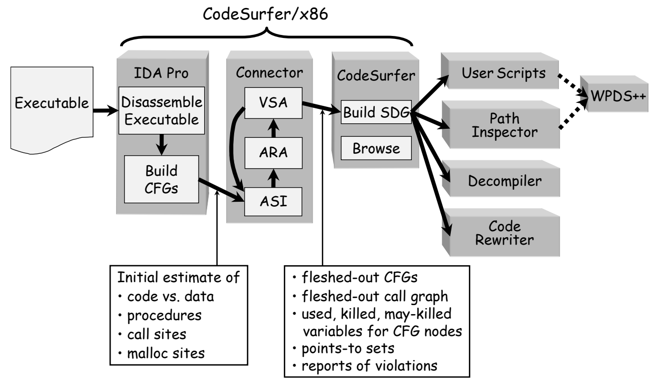

Figure 1. Organization of CodeSurfer/x86 and companion tools.

Moreover, this approach cannot provide any information about the fields of heap-allocated objects, which is crucial for understanding programs that manipulate the heap. (See §2.1.1.)

One of the main challenges in static analysis of low-level code is to recover information about memory-access operations (e.g., the set of addresses accessed by each operation). This is difficult because

- While some memory operations use explicit memory addresses in the instruction (easy), others use indirect addressing via address expressions (difficult).
- Arithmetic on addresses is pervasive. For instance, even when the value of a local variable is loaded from its slot in an activation record, address arithmetic is performed.
- There is no notion of type at the hardware level: address values are not intrinsically different from integer values.
- Memory accesses do not have to be aligned, so word-sized address values could potentially be cobbled together from misaligned reads and writes.

In our research on static analysis of low-level code, we have developed techniques that cope with such issues.

The tool set that we have developed for analyzing executables builds on (i) recent advances in static analysis of program executables [4, 6, 5], and (ii) new techniques for software model checking and dataflow analysis [11, 39, 40, 33]. The main components of the tool set are *CodeSurfer/x86*, *WPDS++*, and the *Path Inspector*:

- CodeSurfer/x86 recovers IRs from an executable that are similar to the IRs that source-code-analysis tools create—but, in many respects, the IRs that CodeSurfer/x86 builds are more precise. (For instance, code from DLLs is imported into the IRs that CodeSurfer/x86 builds, whereas the IRs that source-code-analysis tools create typically require hand-written stubs for library routines.) CodeSurfer/x86 also provides an API to these IRs.
- WPDS++ [32] is a library for answering generalized reachability queries on *weighted pushdown systems* (WPDSs) [11, 39, 40, 33]. This library provide a mechanism for defining and solving model-checking and dataflow-analysis problems. To extend CodeSurfer/x86's analysis capabilities, the CodeSurfer/x86 API can be used to extract a WPDS model from an executable and to run WPDS++ on the model.
- The Path Inspector is a software model checker built on top of CodeSurfer and WPDS++. It supports safety queries about a program's possible control configurations.

In addition, by writing scripts that traverse the IRs that CodeSurfer/x86 recovers, the tool set can be extended with further capabilities (e.g., decompilation, code rewriting, etc.).

Fig. 1 shows how these components fit together. CodeSurfer/x86 makes use of both IDAPro [31], a disassembly toolkit, and GrammaTech's CodeSurfer system [18], a toolkit originally developed for building program-analysis and inspection tools that analyze source code. These components are glued together by a piece called the Connector, which uses three static analyses—aggregate-structure identification (ASI) [37], affine-relation analysis (ARA) [35, 33], and value-set analysis (VSA) [4]—to recover information about the contents of memory locations and how they are manipulated by an executable.

Material about CodeSurfer/x86 has been presented in several previous papers [4, 33, 6, 38, 5]. The present paper describes enhancements to the abstract domains and abstract arithmetic that we have developed since the publication of [4].

The remainder of the paper is organized as follows: §2 provides an overview of the three main analyses used in CodeSurfer/x86: variable and type discovery (ASI), VSA, and ARA. §3 describes some of the work we have done to enhance VSA since the publication of [4]. §4 discusses related work.

2. Overview of CodeSurfer/x86

The analyses in CodeSurfer/x86 are a great deal more ambitious than even relatively sophisticated disassemblers, such as IDAPro. Previous work on analyzing executables has dealt with memory accesses very conservatively: generally, if a register is assigned a value from memory, it is assumed to take on any value [23, 16]. Other research concerning data-dependence analysis on executables has a variety of shortcomings: either the analysis is for single-procedures only [2], or handles memory locations in a flow-insensitive manner [26], or does not account for the possibility of called procedures corrupting the memory locations being tracked [26].

To address such issues, we have brought to bear a variety of static-analysis techniques. At the technical level, our work addresses the following problem:

> Given a stripped executable E (i.e., with all debugging information removed), identify the
> - procedures, data objects, types, and libraries that it uses
>
> and
> - for each instruction I in E and its libraries
> - for each interprocedural calling context of I
> - for each machine register and variable V
>
> statically compute an accurate over-approximation to
> - the set of values that V may contain when I executes

The constraint that debugging information is unavailable complicated the task of creating CodeSurfer/x86; however, the results from its various static-analysis phases provide a substitute for such information. This allowed us to create a tool that can be used when debugging information is absent or untrusted.

A few words are in order about the scope of ambition, capabilities, and assumptions underlying CodeSurfer/x86.

We assume that the executable that is being analyzed follows a "standard compilation model". By this, we mean that the executable has procedures, ARs, a global data region, and a heap; might use virtual functions and DLLs; maintains a runtime stack; each global variable resides at a fixed offset in memory; each local variable of a procedure f resides 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, and are not self-modifying.

During the analysis, these assumptions are checked. When violations are detected, an error report is issued, and the analysis proceeds, generally after making an optimistic choice. For instance, if the analysis finds that the return address might be modified within a procedure, it reports the potential violation, but proceeds without modifying the control flow of the program. If the analyst who is using the tool can determine that the error report is a false positive, then the IR is valid.

The major assumption that we make about IDAPro is that it is able to disassemble a program and build an adequate collection of *preliminary* IRs for it. Even though (i) the CFGs 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. The CFGs and the call-graph are fleshed out according to information recovered during the course of memory-access analysis. The relationship between memory-access analysis and the preliminary IRs created by IDAPro is similar to the relationship between a points-to-analysis algorithm in a compiler and the preliminary IRs created by the compiler's front end. In both cases, the preliminary IRs are fleshed out using the results of static analysis.

The analyzer does not care whether the program was compiled from a high-level language, or hand-written in assembly code. 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 code. Still, it is easiest to talk about the information that the tool is capable of recovering in terms of the kinds of features that high-level languages support: it is capable of recovering information from programs that use global variables, local variables, pointers, structures, arrays, heap-allocated storage, pointer arithmetic, indirect jumps, recursive procedures, virtual functions, and indirect calls through function pointers (but, at present, not run-time code generation or self-modifying code).

Compiler transformations do not confuse the analysis as long as they conform to the aforementioned compilation model. The analysis is capable of handling

- tail recursion (it sees the loop that results from tail-call optimization)
- local variables accessed using `esp`-relative offsets or Pascal-style displays
- applications with custom allocators (although the user has to identify the allocators)

Moreover, optimizations often make the task of memory-access analysis *less* difficult: unoptimized programs generally have more memory accesses than optimized programs. Optimizations typically arrange to keep more of the computation's critical data in registers, rather than in memory. Operations on registers are easier to analyze than operations that access memory because a register cannot be the target of a pointer.

2.1 Variable and Type Discovery

One of the major stumbling blocks in analysis of executables is the difficulty of recovering information about variables and types, especially for aggregates (i.e., structures and arrays). The variable and type-discovery phase of CodeSurfer/x86 recovers such information for variables that are allocated globally, locally (i.e., on the run-time stack), and dynamically (i.e., from the heap). An iterative strategy is used; with each round of the analysis (consisting of ASI, ARA, and VSA), the notion of the program's variables and types is refined.

The memory model that we use is an abstraction of the concrete (runtime) address space, and has two parts:

Memory-regions. Although in the concrete semantics the ARs for procedures, the heap, and the memory for global data are all part of *one* address space, for the purposes of analysis, we separate the address space into a set of disjoint areas, which are referred to as *memory-regions*. Each memory-region represents a group of locations that have similar runtime properties. For example, the runtime locations that belong to the ARs of a given procedure belong to one memory-region. For a given program, there are three kinds of regions: (1) *global*-regions, which represent memory locations that hold global data, (2) *AR*-regions, which represent the locations in the ARs of the different procedures, and (3) *malloc*-regions, which represent the locations allocated at different malloc sites.

A-locs. The second part of the memory model uses a set of (proxies for) variables, which are inferred for each memory-region. Such objects are called *a-locs*, which stands for "abstract locations". In addition to the a-locs identified for each memory-region, the registers represent an additional class of a-locs.

Initially, CodeSurfer/x86 uses a set of variables (i.e., a-locs) that are obtained from IDAPro. Because IDAPro has relatively limited information available at the time that it applies its variable-discovery heuristics (i.e., it only knows about statically known memory addresses and stack offsets), what it can do is rather limited, and generally leads to a very coarse-grained approximation of the program's variables. (See §2.1.1.)

Once a given round of VSA completes, the value-sets for the a-locs at each instruction provide a way to identify an over-approximation of the memory accesses performed at that instruction. This information is used to refine the current set of a-locs by running a variant of the ASI algorithm [37], which identifies commonalities among accesses to different parts of an aggregate data value. (See §2.1.2.)

2.1.1 Limitations of IDAPro's A-loc Identification Algorithm

In the version of our work described in [4], IDAPro's approach to recovering variables was used to instantiate the memory model with a-locs. IDAPro's approach to recovering variables is based on the following observations:

> The layout of memory is known at compile time: the compiler decides *a priori* the locations of global variables, local variables, etc. Direct accesses to program variables are performed using either absolute addresses (for globals) or offsets relative to the frame pointer or stack pointer (for locals). Thus, absolute addresses and offsets (generally) indicate the starting addresses of program variables.

Thus, in the version of our work described in [4], an a-loc consisted of the set of locations between two consecutive statically known addresses or stack-frame offsets.

This approach has several limitations. In particular, because only addresses and offsets that occur explicitly in the program are considered, this approach does not identify variables that are accessed only indirectly. For instance, IDAPro can never identify fields of heap-allocated data objects because they are always accessed by memory-access expressions that lie outside of the class considered: fields of dynamically allocated objects are accessed in terms of offsets relative to the base address of the object itself, which is something that IDAPro knows nothing about. Moreover, IDAPro can also have trouble with locals and globals. For example, IDAPro would not discover fields of elements of an array when they are accessed relative to `eax`, e.g., by `[eax]` and `[eax+4]`, rather than being accessed relative to the stack pointer (`esp`) or frame pointer (`ebp`). To recover such fields, the set of values that `eax` can hold needs to be determined; i.e., the analysis has to look at more than just the explicitly known addresses and stack-frame offsets.

Because IDAPro does not take into account all memory-access operations in the executable, it may produce a too-coarse set of a-locs, which can affect the precision of VSA. For example, suppose that a procedure in a program has four 4-byte local variables l_1, l_2, l_3, and l_4 that are laid out next to each other. Suppose that the compiler generates explicit accesses to l_1 and l_3 and only generates indirect accesses to l_2 and l_4 (in terms of the addresses of l_1 and l_3). In this case, IDAPro will only take into account the explicit accesses to l_1 and l_3—hence, it identifies two 8-byte a-locs: l_{12}, which spans l_1 and l_2, and l_{34}, which spans l_3 and l_4. Because (for a 32-bit machine) value-sets only represent addresses and numeric values up to 32 bits, VSA will not be able to represent the addresses and numeric values that l_{12} and l_{34} hold. Reads from and writes to (parts of) l_{12} and l_{34} would be treated conservatively, and VSA would report that l_{12} and l_{34} hold $\top$ (any possible address or numeric value) at all program points. On the other hand, when l_1, l_2, l_3, and l_4 are used as four 4-byte a-locs, VSA will produce more precise (non-$\top$) value-sets.

Another limitation of relying on IDAPro's variable-identification algorithm is that the a-locs recovered are not expressive enough:

- IDAPro's a-locs cannot capture information about the repeating structure of arrays; an array is identified merely as a contiguous block of data.
- IDAPro does not identify fields of heap-allocated objects. Such information is crucial for tracking the contents of heap a-locs.
- An IDAPro a-loc can only represent a contiguous sequence of memory locations in a memory-region, with no internal substructure. It cannot represent non-contiguous memory locations, such as the locations of (all instances of) a specific field in an array of structures.

This lack of expressiveness of IDAPro's a-locs can affect the precision of clients of VSA, such as the dependence analyzer in CodeSurfer/x86. This uses sets of used, killed, and possibly-killed a-locs for each program point, generated from the results of VSA, to build a system dependence graph [29] for the executable. In particular, IDAPro's a-locs are too coarse-grained a representation of used, killed, and possibly-killed memory locations, and this can lead to extra edges in the dependence graph.

2.1.2 Aggregate Structure Identification (ASI)

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

One might hope to apply ASI to an executable by treating each memory-region as an aggregate and applying ASI (without using VSA results). However, one of the requirements for applying ASI is that it must be possible to extract data-access constraints from the program. This is possible when ASI is applied to programs written in a language such as Cobol: the data-access patterns are apparent from the syntax of the constructs under consideration. However, for executables the data-access patterns are not readily apparent. For instance, the memory operand `[eax]` can represent an access to either a single variable or to the elements of an array. Fortunately, the value-sets recovered by VSA furnish information that can be used to generate ASI data-access constraints; information about the values that a-locs can hold (in terms of range and stride information—see §3.2) provides information not only about points-to relationships, but also about the extent and repeating structure of a memory-access operation.

Our extension to ASI exploits the information made available by VSA to create data structures that record the structure of each memory-region and relationships among the memory-regions' atoms (which correspond to the newly discovered a-locs). This generally leads to a much more accurate set of a-locs than the initial set of a-locs discovered by IDAPro. For instance, consider a simple loop, implemented in source code as

```
int a[10], i;
for (i = 0; i < 10; i++)
  a[i] = i;
```

From the executable, IDAPro will determine that there are two variables, one of size 4 bytes and one of size 40 bytes, but will provide no information about the substructure of the 40-byte variable. In contrast, in addition to the 4-byte variable, ASI will correctly identify that the 40 bytes are an array of ten 4-byte quantities.

The current version of the Connector uses a refinement loop that performs repeated phases of ASI, ARA, and VSA (see Fig. 1). The first round of ASI is performed using the variables discovered by IDAPro. On each subsequent round, ASI is used to refine the previous set of a-locs, and the refined set of a-locs is then used to analyze the program during the next round of VSA. The number of iterations is controlled by a command-line parameter.

After the second round of ASI, the a-locs in hand permit VSA to start to analyze the contents of dynamically allocated objects (i.e., memory locations allocated using malloc or new) [5]. VSA considers each malloc site m to be a "memory-region" (consisting of the objects allocated at m), and the memory-region for m serves as a representative for the base addresses of those objects.[1] This lets ASI handle the use of an offset from an object's base address similar to the way that it handles a stack-frame offset—with the net result that the second round of ASI starts to identify the fine-grained structure of dynamically allocated objects. The object fields discovered in this way become a-locs for the second round of VSA, which will then discover an over-approximation of their contents.

2.2 Value-Set Analysis (VSA)

The goal of VSA is to determine, at each program point, an over-approximation of the set of numeric values and addresses (or *value-set*) that each register and memory location (a-loc) holds. The information computed during VSA is also used to augment the call graph and control-flow graphs to account for indirect jumps and indirect function calls.

VSA is a combined numeric and pointer-analysis algorithm. VSA 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 each program point, VSA determines an over-approximation of the set of addresses that each data object can hold.*

At the same time, VSA is similar to range analysis and other numeric static-analysis algorithms that over-approximate the integer values that each variable can hold:

> *At each program point, VSA determines an over-approximation of the set of integer values that each data object can hold.*

The following insights shaped the design of VSA:

- A *non-aligned access* to memory—e.g., an access via an address that is not aligned on a 4-byte word boundary—spans parts of two words, and provides a way to forge a new address from *parts* of old addresses. It is important for VSA to discover information about the alignments and strides of memory accesses, or else most indirect-addressing operations appear to be possibly non-aligned accesses (see the discussion in §4).
- To prevent most loops that traverse arrays from appearing to corrupt the stack, 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 (see §2.3 and [4, 35, 33]).
- It is desirable for VSA to track integer-valued and address-valued quantities *simultaneously*. This is crucial for analyzing executables because
  - integers and addresses are indistinguishable at execution time, and
  - compilers use address arithmetic and indirect addressing to implement such features as pointer arithmetic, pointer dereferencing, array indexing, and accessing structure fields.

  Moreover, information about integer values can lead to improved tracking of address-valued quantities, and information about address values can lead to improved tracking of integer-valued quantities.

VSA 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, VSA provides an analog of pointer analysis that is suitable for use on executables.

2.3 Affine-Relation Analysis (ARA)

VSA is not relational; that is, it does not keep track of the relationships that hold among registers and memory locations. However, when interpreting conditional branches, specifically those that implement loops, it is important to know such relationships [4]. Hence, a separate affine-relation analysis (ARA) [35, 33] is performed to recover affine relations that hold at conditional branch points; those affine relations are then used by VSA when interpreting conditional branches. (Currently, ARA recovers affine relations that only involve registers.) ARA implements the affine-relation domain from [35], which is based on arithmetic modulo 2^{32} and hence accounts for arithmetic overflow.

Before each call instruction, a subset of the registers is saved on the stack, either by the caller or the callee, and restored at the return. Such registers are called the *caller-save* and *callee-save* registers. To preserve their values across a call, ARA treats caller-save and callee-save registers as local variables of the calling procedure [33]; i.e., the values of caller-save and callee-save registers after the call are set to the values before the call and the values of other registers are set to the values at the exit node of the callee.

3. Enhancements to Value-Set Analysis

VSA is a combined numeric-analysis and pointer-analysis algorithm that determines, at each program point, an over-approximation of the set of numeric values and addresses that each a-loc holds. One of the basic abstract domains used during VSA is the *value-set domain* (see §3.3), which is a safe approximation to a set of concrete addresses and numeric values. A value-set is a map from memory-regions to *strided intervals* (see §3.2), and associates each memory-region m with a strided interval that represents a set of offsets in m. In this section, we describe strided intervals and value-sets, and sketch how they are used to define abstract transformers for x86 instructions.

3.1 Notational Conventions

We use different typefaces to make the following distinctions: integers ($\mathbb{Z}$) and other mathematical expressions are written in ordinary mathematical notation (e.g., 1, -2^{31}, $2^{31}-1$, $1 \leq 2$, etc.); variables that hold integers appear in italics (e.g., x). Bounded integers, such as unsigned numbers and signed two's-complement numbers, as well as variables that hold such quantities, appear in bold (e.g., $\mathbf{1}$, $-\mathbf{2^{31}}$, $\mathbf{2^{31}-1}$). Fragments of C code appear in Courier (e.g., $\mathtt{1}$, $\mathtt{-2^{31}}$, $\mathtt{2^{31}-1}$, $\mathtt{a}$, $\mathtt{if(a < b)\{\ldots\}}$, $\mathtt{z = x + y;}$).

When the same name appears in different typefaces, our convention is that the meaning changes appropriately: $\mathtt{x}$, $\mathbf{x}$, and x refer to program variable $\mathtt{x}$, whose signed two's-complement value (or unsigned value, if appropriate) is $\mathbf{x}$, and whose integer value is x.

Names that denote strided intervals are also written in bold.

[1] The implementation actually uses a more precise abstraction of dynamically allocated memory; see footnote 6 and [5].

Let $[x]_m$ denote the congruence class of x mod m, defined as $[x]_m \stackrel{\text{def}}{=} \{x + i \times m \mid i \in \mathbb{Z}\}$; note that $[x]_0 = \{x\}$.

3.2 Strided-Interval Arithmetic

A k-bit *strided interval* is a triple $\mathbf{s}[\mathbf{lb}, \mathbf{ub}]$ such that $-2^k \leq lb \leq ub \leq 2^k - 1$. The meaning of a strided interval is defined as follows:

DEFINITION 3.1. **[Meaning of a strided interval]**. *A k-bit strided interval $\mathbf{s}[\mathbf{lb}, \mathbf{ub}]$ represents the set of integers*

$$\gamma(\mathbf{s}[\mathbf{lb}, \mathbf{ub}]) = \{i \in [-2^k, 2^k - 1] \mid lb \leq i \leq ub, i \in [lb]_s\}.$$

□

Note that a strided interval of the form $\mathbf{0}[\mathbf{a}, \mathbf{a}]$ represents the singleton set $\{a\}$. Except where noted, we will assume that we are working with 32-bit strided intervals.

In a strided interval $\mathbf{s}[\mathbf{lb}, \mathbf{ub}]$, $\mathbf{s}$ is called the *stride*, and $[\mathbf{lb}, \mathbf{ub}]$ is called the *interval*. Stride $\mathbf{s}$ is unsigned; bounds $\mathbf{lb}$ and $\mathbf{ub}$ are signed.[2] The stride is unsigned so that each two-element set of 32-bit numbers, including such sets as $\{-2^{31}, 2^{31}-1\}$, can be denoted exactly. For instance, $\{-2^{31}, 2^{31} - 1\}$ is represented exactly by the strided interval $(\mathbf{2^{32}} - \mathbf{1})[-\mathbf{2^{31}}, \mathbf{2^{31}} - \mathbf{1}]$.

As defined above, some sets of numbers can be represented by more than one strided interval. For instance, $\gamma(\mathbf{4}[\mathbf{4}, \mathbf{14}]) = \{4, 8, 12\} = \gamma(\mathbf{4}[\mathbf{4}, \mathbf{12}])$. Without loss of generality, we will assume that all strided intervals are *reduced* (i.e., upper bounds are tight, and whenever the upper bound equals the lower bound the stride is $\mathbf{0}$). For example, $\mathbf{4}[\mathbf{4}, \mathbf{12}]$ and $\mathbf{0}[\mathbf{12}, \mathbf{12}]$ are reduced strided intervals; $\mathbf{4}[\mathbf{4}, \mathbf{14}]$ and $\mathbf{4}[\mathbf{12}, \mathbf{12}]$ are not.

The remainder of this subsection describes abstract arithmetic and bit-level operations on strided intervals for use in abstract interpretation [21].

DEFINITION 3.2. **[Soundness criterion]**. *For each op^{si}, if $\mathbf{si_3} = \mathbf{si_1}\ op^{si}\ \mathbf{si_2}$, then $\gamma(\mathbf{si_3}) \supseteq \{a\ op\ b \mid a \in \gamma(\mathbf{si_1})$ and $b \in \gamma(\mathbf{si_2})\}$.*
□

Sound algorithms for performing arithmetic and bit-level operations on *intervals* (i.e., strided intervals with stride 1) are described in a book by H. Warren [47]. They provided a starting point for the operations that we define for strided intervals, which extend Warren's operations to take strides into account.

Below, we summarize several of Warren's interval operations, and describe how a sound stride for $\mathbf{si_3}$ can be obtained for each operation $op^{si} \in \{+^{si}, -_{\mathbf{u}}^{si}, -^{si}, ++^{si}, --^{si}, |^{si}, \sim^{si}, \&^{si}, \wedge^{si}\}$.

Addition ($+^{si}$)

Suppose that we have the following bounds on two two's-complement values $\mathbf{x}$ and $\mathbf{y}$: $\mathbf{a} \leq \mathbf{x} \leq \mathbf{b}$ and $\mathbf{c} \leq \mathbf{y} \leq \mathbf{d}$. With 32-bit arithmetic, the result of $\mathbf{x} + \mathbf{y}$ is not always in the interval $[\mathbf{a} + \mathbf{c}, \mathbf{b} + \mathbf{d}]$ because the bound calculations $\mathbf{a} + \mathbf{c}$ and $\mathbf{b} + \mathbf{d}$ can overflow in either the positive or negative direction. Warren provides the method shown in Tab. 1 to calculate a bound on $\mathbf{x} + \mathbf{y}$.

Case (3) of Tab. 1 is the case in which neither bound calculation overflows. In cases (1) and (5) of Tab. 1, the result of $\mathbf{x} + \mathbf{y}$ is bounded by $[\mathbf{a} + \mathbf{c}, \mathbf{b} + \mathbf{d}]$ even though *both* bound calculations overflow. Thus, we merely need to identify cases (2) and (4), in which case the bounds imposed are the extreme negative and positive numbers (see lines [9]–[11] of Fig. 2). This can be done by the code that appears on lines [4]–[7] of Fig. 2: if $\mathtt{u}$ is negative, then case (2) holds; if $\mathtt{v}$ is negative, then case (4) holds [47, p. 57].

[2] To reduce notation, we rely on context to indicate whether a typeface conversion denotes a conversion to a signed two's-complement value or to an unsigned value: if x is a stride, $\mathbf{x}$ denotes an unsigned value; if y is an interval bound, $\mathbf{y}$ denotes a signed two's-complement value.

(1)	$a + c < -2^{31}, b + d < -2^{31}$	$\Rightarrow \mathbf{a} + \mathbf{c} \leq \mathbf{x} + \mathbf{y} \leq \mathbf{b} + \mathbf{d}$
(2)	$a + c < -2^{31}, b + d \geq -2^{31}$	$\Rightarrow -\mathbf{2^{31}} \leq \mathbf{x} + \mathbf{y} \leq \mathbf{2^{31}} - \mathbf{1}$
(3)	$-2^{31} \leq a + c < 2^{31}, b + d < 2^{31}$	$\Rightarrow \mathbf{a} + \mathbf{c} \leq \mathbf{x} + \mathbf{y} \leq \mathbf{b} + \mathbf{d}$
(4)	$-2^{31} \leq a + c < 2^{31}, b + d \geq 2^{31}$	$\Rightarrow -\mathbf{2^{31}} \leq \mathbf{x} + \mathbf{y} \leq \mathbf{2^{31}} - \mathbf{1}$
(5)	$a + c \geq 2^{31}, b + d \geq 2^{31}$	$\Rightarrow \mathbf{a} + \mathbf{c} \leq \mathbf{x} + \mathbf{y} \leq \mathbf{b} + \mathbf{d}$

Table 1. Cases to consider for bounding the result of adding two signed two's-complement numbers [47, p. 56].

```
[1]  void addSI(int a, int b, unsigned s1,
[2]             int c, int d, unsigned s2,
[3]             int& lbound, int& ubound, unsigned& s) {
[4]    lbound = a + c;
[5]    ubound = b + d;
[6]    int u = a & c & ~lbound & ~(b & d & ~ubound);
[7]    int v = ((a ^ c) | ~(a ^ lbound)) & (~b & ~d & ubound);
[8]    if(u | v < 0) { // case (2) or case (4)
[9]      s = 1;
[10]     lbound = 0x80000000;
[11]     ubound = 0x7FFFFFFF;
[12]   }
[13]   else s = gcd(s1, s2);
[14] }
```

Figure 2. Implementation of abstract addition ($+^{si}$) for strided intervals.

In the proof of Thm. 3.4 (see below), we will make use of the following observation:

OBSERVATION 3.3. *In case (1) of Tab. 1, all three sums $\mathbf{a} + \mathbf{c}$, $\mathbf{x} + \mathbf{y}$, and $\mathbf{b} + \mathbf{d}$ yield values that are too high by 2^{32} (compared to $a + c$, $x + y$, and $b + d$, respectively) [47, p. 56]. Similarly, in case (5), all three sums yield values that are too low by 2^{32}.* □

Fig. 2 shows a C procedure that uses these ideas to compute $\mathbf{s1}[\mathbf{a}, \mathbf{b}] +^{si} \mathbf{s2}[\mathbf{c}, \mathbf{d}]$, but also takes the strides $\mathbf{s1}$ and $\mathbf{s2}$ into account. The `gcd` (greatest common divisor) operation is used to find a sound stride for the result.[3]

THEOREM 3.4. **[Soundness of $+^{si}$]**. *If $\mathbf{si_3} = \mathbf{si_1} +^{si} \mathbf{si_2}$, then $\gamma(\mathbf{si_3}) \supseteq \{a + b \mid a \in \gamma(\mathbf{si_1})$ and $b \in \gamma(\mathbf{si_2})\}$.* □

Proof: The soundness of the interval of $\mathbf{si_3}$ follows from the arguments given in [47]. We need to show that the stride computed by procedure `addSI` from Fig. 2 is sound.

In lines [9]–[11] of Fig. 2, which correspond to cases (2) and (4), the answer is the entire interval $[-2^{31}, 2^{31} - 1]$, so the stride of 1 is obviously sound. In all other situations, `gcd` is used to find the stride.

Let $\mathbf{si_1} = \mathbf{s_1}[\mathbf{lb_1}, \mathbf{ub_1}]$, $SI_1 = \gamma(\mathbf{si_1})$, $\mathbf{si_2} = \mathbf{s_2}[\mathbf{lb_2}, \mathbf{ub_2}]$, and $SI_2 = \gamma(\mathbf{si_2})$. We consider the cases where

$$\mathbf{si_3} = \text{gcd}(\mathbf{s_1}, \mathbf{s_2})[\mathbf{lb_1} + \mathbf{lb_2}, \mathbf{ub_1} + \mathbf{ub_2}].$$

Let[4]

$$b_1 = \begin{cases} 0 & \text{if } s_1 = 0 \\ (ub_1 - lb_1)/s_1 & \text{otherwise} \end{cases}$$
$$b_2 = \begin{cases} 0 & \text{if } s_2 = 0 \\ (ub_2 - lb_2)/s_2 & \text{otherwise} \end{cases}$$

Thus,

$$\begin{aligned} SI_1 &= \{lb_1 + i \times s_1 \mid 0 \leq i \leq b_1\} \\ &= \{lb_1, lb_1 + s_1, \ldots, lb_1 + b_1 \times s_1\} \\ SI_2 &= \{lb_2 + j \times s_2 \mid 0 \leq j \leq b_2\} \\ &= \{lb_2, lb_2 + s_2, \ldots, lb_2 + b_2 \times s_2\} \end{aligned}$$

and $SI_1 + SI_2 = \{lb_1 + lb_2, \ldots, lb_1 + lb_2 + i \times s_1 + j \times s_2, \ldots, lb_1 + lb_2 + b_1 \times s_1 + b_2 \times s_2\}$.

[3] By convention, $\text{gcd}(0, a) = a$, $\text{gcd}(a, 0) = a$, and $\text{gcd}(0, 0) = 0$.

[4] By the assumption that we work only with *reduced* strided intervals, in a strided interval $\mathbf{s}[\mathbf{lb}, \mathbf{ub}]$, $s \neq 0$ implies that s divides evenly into $(ub - lb)$.

```
unsigned minOR(unsigned a, unsigned b,
                unsigned c, unsigned d) {
  unsigned m, temp;
  m = 0x80000000;
  while(m != 0) {
    if(~a & c & m) {
      temp = (a | m) & -m;
      if(temp <= b) {
        a = temp;
        break;
      }
    }
    else if(a & ~c & m) {
      temp = (c | m) & -m;
      if(temp <= d) {
        c = temp;
        break;
      }
    }
    m = m >> 1;
  }
  return a | c;
}
```

Figure 3. Implementation of minOR [47, p. 59].

```
unsigned maxOR(unsigned a, unsigned b,
                unsigned c, unsigned d) {
  unsigned m, temp;
  m = 0x80000000;
  while(m != 0) {
    if(b & d & m) {
      temp = (b - m) | (m - 1);
      if(temp >= a) {
        b = temp;
        break;
      }
      temp = (d - m) | (m - 1);
      if(temp >= c) {
        d = temp;
        break;
      }
    }
    m = m >> 1;
  }
  return b | d;
}
```

Figure 4. Implementation of maxOR [47, p. 60].

Let $s = \gcd(s_1, s_2)$, $s_1 = s \times m$, and $s_2 = s \times n$. We wish to show that s divides evenly into the difference between an arbitrary element e in $SI_1 + SI_2$ and the lower-bound value $lb_1 + lb_2$. Let e be some element of $SI_1 + SI_2$: $e = (lb_1 + lb_2) + i \times s_1 + j \times s_2$, where $0 \le i \le b_1$ and $0 \le j \le b_2$. The difference $e - (lb_1 + lb_2)$ is non-negative and equals $i \times s_1 + j \times s_2$, which equals $s \times (i \times m + j \times n)$, and hence is divisible by s.

Moreover, by Obs. 3.3, when we compare the values in $SI_1 + SI_2$ to the interval $[-2^{31}, 2^{31} - 1]$, they are either

- all too low by 2^{32} (case (1) of Tab. 1),
- in the interval $[-2^{31}, 2^{31} - 1]$ (case (3) of Tab. 1), or
- all too high by 2^{32} (case (5) of Tab. 1).

Let $-2^{31} \le e' \le 2^{31} - 1$ be e adjusted by an appropriate multiple of 2^{32}. Similarly, let $-2^{31} \le lb \le 2^{31} - 1$ be $lb_1 + lb_2$ adjusted by the same multiple of 2^{32}. (Note that, by Obs. 3.3, lb is the minimum element if all elements of $SI_1 + SI_2$ are similarly adjusted.) The argument that $e - (lb_1 + lb_2)$ is divisible by s carries over in each case to an argument that $e' \in [lb]_s$. Consequently, $\gamma(\mathbf{si_3}) \supseteq SI_1 + SI_2$, as was to be shown. □

Unary Minus ($-^{\mathrm{si}}_{\mathbf{u}}$)

Suppose that we have the following bounds on the two's-complement value $\mathbf{y}$: $\mathbf{c} \le \mathbf{y} \le \mathbf{d}$. Then $-d \le -y \le -c$. The number -2^{31} is representable as a 32-bit two's-complement number, but 2^{31} is not. Moreover, if $\mathbf{c} = -2^{31}$, then $-\mathbf{c} = -2^{31}$ as well, which means that we do not necessarily have $-\mathbf{y} \le -\mathbf{c}$ (note that $-\mathbf{c}$ is a two's-complement value in this expression). However, in all other cases we have $-\mathbf{d} \le -\mathbf{y} \le -\mathbf{c}$; by the assumption that we work only with *reduced* strided intervals, in $\mathbf{s}[\mathbf{c}, \mathbf{d}]$ the upper bound $\mathbf{d}$ is achievable, which allows us to retain $\mathbf{s}$ as the stride of $-^{\mathrm{si}}_{\mathbf{u}}(\mathbf{s}[\mathbf{c}, \mathbf{d}])$:

$$-^{\mathrm{si}}_{\mathbf{u}}(\mathbf{s}[\mathbf{c}, \mathbf{d}]) = \begin{cases} \mathbf{0}[-\mathbf{2^{31}}, -\mathbf{2^{31}}] & \text{if } \mathbf{c} = \mathbf{d} = -2^{31} \\ \mathbf{s}[-\mathbf{d}, -\mathbf{c}] & \text{if } \mathbf{c} \ne -2^{31} \\ \mathbf{1}[-\mathbf{2^{31}}, \mathbf{2^{31}} - \mathbf{1}] & \text{otherwise} \end{cases}$$

Subtraction ($-^{\mathrm{si}}$), Increment ($++^{\mathrm{si}}$), and Decrement ($--^{\mathrm{si}}$)

The $+^{\mathrm{si}}$ and $-^{\mathrm{si}}_{\mathbf{u}}$ operations on strided intervals can be used to implement other arithmetic operations, such as subtraction ($-^{\mathrm{si}}$), increment ($++^{\mathrm{si}}$), and decrement ($--^{\mathrm{si}}$), as follows:

$$\begin{aligned} \mathbf{x} -^{\mathrm{si}} \mathbf{y} &= \mathbf{x} +^{\mathrm{si}} (-^{\mathrm{si}}_{\mathbf{u}} \mathbf{y}) \\ ++^{\mathrm{si}} \mathbf{x} &= \mathbf{x} +^{\mathrm{si}} \mathbf{0}[\mathbf{1}, \mathbf{1}] \\ --^{\mathrm{si}} \mathbf{x} &= \mathbf{x} +^{\mathrm{si}} \mathbf{0}[-\mathbf{1}, -\mathbf{1}] \end{aligned}$$

Bitwise Or ($|^{\mathrm{si}}$)

Following Warren, [47, p. 58–63], we develop the algorithm for $|^{\mathrm{si}}$ (bitwise-or on strided intervals) by first examining how to bound bitwise-or on *unsigned* values and then using this as a subroutine in the algorithm for $|^{\mathrm{si}}$. Suppose that we have the following bounds on unsigned values $\mathbf{x}$ and $\mathbf{y}$: $\mathbf{a} \le \mathbf{x} \le \mathbf{b}$ and $\mathbf{c} \le \mathbf{y} \le \mathbf{d}$. The two algorithms from Warren's book given in Figs. 3 and 4 provide bounds on the minimum and maximum possible values, respectively, that $\mathbf{x} \,|\, \mathbf{y}$ can attain.

Warren argues [47, p. 58–59], that the minimum possible value of $\mathbf{x} \,|\, \mathbf{y}$ can be found by scanning $\mathbf{a}$ and $\mathbf{c}$ from left-to-right, and finding the leftmost position at which either

- a $\mathbf{0}$ of $\mathbf{a}$ can be changed to $\mathbf{1}$, and all bits to the right set to $\mathbf{0}$, yielding a number $\mathbf{a}'$ such that $\mathbf{a}' \le \mathbf{b}$ and $(\mathbf{a}' \,|\, \mathbf{c}) < (\mathbf{a} \,|\, \mathbf{c})$, or
- a $\mathbf{0}$ of $\mathbf{c}$ can be changed to $\mathbf{1}$, and all bits to the right set to $\mathbf{0}$, yielding a number $\mathbf{c}'$ such that $\mathbf{c}' \le \mathbf{d}$ and $(\mathbf{a} \,|\, \mathbf{c}') < (\mathbf{a} \,|\, \mathbf{c})$.

This is implemented by function minOR of Fig. 4. For instance, suppose that we have

$$\begin{aligned} \mathbf{0000101} &= \mathbf{a} \le \mathbf{x} \le \mathbf{b} = \mathbf{0001001} \\ \mathbf{0010011} &= \mathbf{c} \le \mathbf{y} \le \mathbf{d} = \mathbf{0101001}. \end{aligned}$$

We reject $\mathbf{a}' = \mathbf{0010000}$ because $\mathbf{0010000} \not\le \mathbf{0001001} = \mathbf{b}$; however, we find that $\mathbf{c}' = \mathbf{0010100}$ meets the condition $\mathbf{0010100} \le \mathbf{0101001} = \mathbf{d}$, and

$$\begin{aligned} (\mathbf{a} \,|\, \mathbf{c}') &= (\mathbf{0000101} \,|\, \mathbf{0010100}) \\ &= \mathbf{0010101} \\ &< \mathbf{0010111} \\ &= (\mathbf{0000101} \,|\, \mathbf{0010011}) \\ &= (\mathbf{a} \,|\, \mathbf{c}). \end{aligned}$$

Note that Warren's algorithm relies on the assumption that it is working on intervals with strides of 1. For instance, we could select the new contribution to the lower bound, $\mathbf{c}' = \mathbf{0010100} > \mathbf{0010011} = \mathbf{c}$, without having to worry about whether a stride value repeatedly added to $\mathbf{c}$ might miss $\mathbf{c}'$.

The algorithm to find the maximum possible value of $\mathbf{x} \,|\, \mathbf{y}$ (Fig. 4) has a similar flavor [47, p. 60].

Tab. 2 shows the method that Warren gives for finding bounds on the bitwise-or of two signed two's-complement values $\mathbf{x}$ and $\mathbf{y}$, where $\mathbf{a} \le \mathbf{x} \le \mathbf{b}$ and $\mathbf{c} \le \mathbf{y} \le \mathbf{d}$ [47, p. 63]. The method calls the procedures from Figs. 3 and 4, which find bounds on the bitwise-or of two unsigned values.

Function ntz of Fig. 5 counts the number of trailing zeroes of its argument x. At line [3] of Fig. 5, y is set to a mask that identifies the trailing zeroes of $\mathbf{x}$ [47, p. 11], i.e., y is set to the binary number $\mathbf{0}^i\mathbf{1}^j$, where $i + j = 32$ and j equals the number of trailing zeroes

a	b	c	d	signed **minOR**	signed **maxOR**
< 0	< 0	< 0	< 0	$\mathbf{minOR(a,b,c,d)}$	$\mathbf{maxOR(a,b,c,d)}$
< 0	< 0	< 0	≥ 0	$\mathbf{a}$	$\mathbf{-1}$
< 0	< 0	≥ 0	≥ 0	$\mathbf{minOR(a,b,c,d)}$	$\mathbf{maxOR(a,b,c,d)}$
< 0	≥ 0	< 0	< 0	$\mathbf{c}$	$\mathbf{-1}$
< 0	≥ 0	< 0	≥ 0	$\mathbf{min(a,c)}$	$\mathbf{maxOR(0,b,0,d)}$
< 0	≥ 0	≥ 0	≥ 0	$\mathbf{minOR(a,0xFFFFFFFF,c,d)}$	$\mathbf{maxOR(0,b,c,d)}$
≥ 0	≥ 0	< 0	< 0	$\mathbf{minOR(a,b,c,d)}$	$\mathbf{maxOR(a,b,c,d)}$
≥ 0	≥ 0	< 0	≥ 0	$\mathbf{minOR(a,b,c,0xFFFFFFFF)}$	$\mathbf{maxOR(a,b,0,d)}$
≥ 0	≥ 0	≥ 0	≥ 0	$\mathbf{minOR(a,b,c,d)}$	$\mathbf{maxOR(a,b,c,d)}$

Table 2. Signed $\mathbf{minOR(a,b,c,d)}$ and $\mathbf{maxOR(a,b,c,d)}$ [47, p. 63]. Warren's method uses unsigned `minOR` and `maxOR` to find bounds on the bitwise-or of two signed two's-complement values $\mathbf{x}$ and $\mathbf{y}$, where $\mathbf{a} \leq \mathbf{x} \leq \mathbf{b}$ and $\mathbf{c} \leq \mathbf{y} \leq \mathbf{d}$. (Because $\mathbf{a} \leq \mathbf{b}$ and $\mathbf{c} \leq \mathbf{d}$, the nine cases shown above are exhaustive.)

```
[1]  int ntz(unsigned x) {
[2]    int n;
[3]    int y = -x & (x-1);
[4]    n = 0;
[5]    while(y != 0) {
[6]      n = n + 1;
[7]      y = y >> 1;
[8]    }
[9]    return n;
[10] }
```

Figure 5. Counting trailing 0's of `x` [47, p. 86].

in $\mathbf{x}$. The trailing ones of $\mathbf{y}$ are then counted in the while-loop in lines [5]–[8].

We now turn to the algorithm for bitwise-or on strided intervals ($|^{\text{si}}$). Suppose that we want to perform $\mathbf{s_1[a,b]} \,|^{\text{si}}\, \mathbf{s_2[c,d]}$. As illustrated by the topmost set shown in Fig. 6, all elements in $\gamma(\mathbf{s_1[a,b]})$ share the same $t_1 = \text{ntz}(s_1)$ low-order bits: because the t_1 low-order bits of stride $\mathbf{s_1}$ are all $\mathbf{0}$, repeated addition of $\mathbf{s_1}$ to $\mathbf{a}$ cannot affect the t_1 low-order bits. Similarly, all elements in $\gamma(\mathbf{s_2[c,d]})$ share the same $t_2 = \text{ntz}(s_2)$ low-order bits.

As illustrated by the third set shown in Fig. 6, all values in the answer strided interval share the same t low-order bits, where $t = \min(t_1, t_2)$. Consequently, we may take $s = 2^t$ $(= \mathbf{1} << \mathbf{t})$ as the stride of the answer, and the value of the shared t low-order bits can be calculated by $\mathbf{r} = (\mathbf{a} \,\&\, \mathbf{mask}) \,|\, (\mathbf{c} \,\&\, \mathbf{mask})$, where $\mathbf{mask} = (\mathbf{1} << \mathbf{t}) - \mathbf{1}$.

The $32-t$ high-order bits are handled by masking out the t low-order bits, and then applying the method from Tab. 2 for finding bounds on the bitwise-or of two signed two's-complement values.

Thus, to compute $\mathbf{s_1[a,b]} \,|^{\text{si}}\, \mathbf{s_2[c,d]}$, we perform the following steps:

- Set $t := \min(\text{ntz}(s_1), \text{ntz}(s_2))$.
- Set $s := 2^t$.
- Calculate the value of the shared t low-order bits as $r := (\mathbf{a} \,\&\, \mathbf{mask}) \,|\, (\mathbf{c} \,\&\, \mathbf{mask})$, where $\mathbf{mask} = (\mathbf{1} << \mathbf{t}) - \mathbf{1}$.
- Use the method from Tab. 2 to bound the value of $\mathbf{x}' \,|\, \mathbf{y}'$ for $(\mathbf{a} \,\&\, \sim\mathbf{mask}) \leq \mathbf{x}' \leq (\mathbf{b} \,\&\, \sim\mathbf{mask})$ and $(\mathbf{c} \,\&\, \sim\mathbf{mask}) \leq \mathbf{y}' \leq (\mathbf{d} \,\&\, \sim\mathbf{mask})$. Call these bounds $\mathbf{lb}$ and $\mathbf{ub}$.
- Return the strided interval
  $\mathbf{s}[((\mathbf{lb} \,\&\, \sim\mathbf{mask}) \,|\, \mathbf{r}), ((\mathbf{ub} \,\&\, \sim\mathbf{mask}) \,|\, \mathbf{r})]$.

Bitwise not ($\sim^{si}$), And ($\&^{si}$), and Xor ($^{\wedge si}$)

Suppose that we have bounds on $\mathbf{x}$: $\mathbf{a} \leq \mathbf{x} \leq \mathbf{b}$. A bound on the result of applying $\sim$ to $\mathbf{x}$ is $\sim\mathbf{b} \leq \sim\mathbf{x} \leq \sim\mathbf{a}$ [47, p. 58]. Similarly, for strided intervals, we have

$$\sim^{si}(\mathbf{s[lb,ub]}) = \mathbf{s}[\sim\mathbf{ub}, \sim\mathbf{lb}]. \qquad (1)$$

$\gamma(\mathbf{s_1[a,b]}) = \{a,\, a+s_1,\, a+2s_1,\, \ldots,\, b\}$
$\gamma(\mathbf{s_2[c,d]}) = \{c,\, c+s_2,\, c+2s_2,\, \ldots,\, d\}$
$t_1 = \text{ntz}(s_1)$
$t_2 = \text{ntz}(s_2)$
$t = \min(t_1, t_2)$

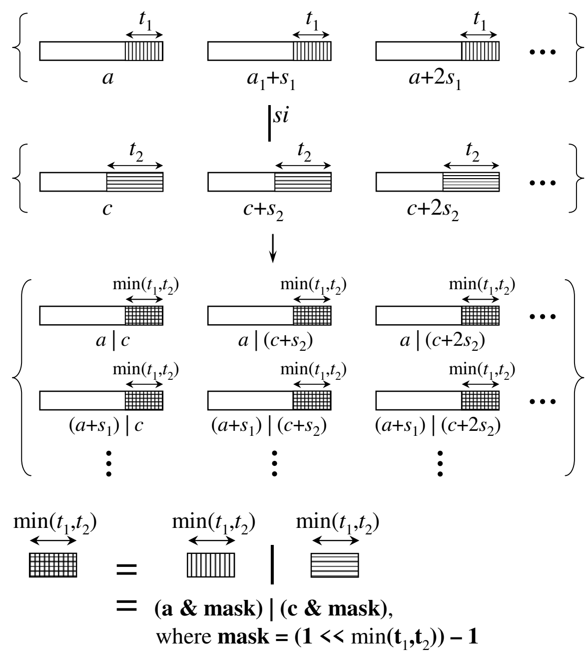

Figure 6. Justification of the use of $t = \min(\text{ntz}(s_1), \text{ntz}(s_2))$ in the stride calculation of the abstract bitwise-or operation ($|^{\text{si}}$). In particular, all values in the answer strided interval share the same t low-order bits.

Eqn. (1) relies on the assumption that strided intervals are reduced: the assumption guarantees that $ub \in \gamma(\mathbf{s[lb,ub]})$, and hence that $\sim ub$ is the least element of $\gamma(\sim^{si}(\mathbf{s[lb,ub]}))$.

By De Morgan's Laws, and by the fact that $\sim^{si}(\sim^{si}(\mathbf{s[lb,ub]})) = \mathbf{s[lb,ub]}$, $\&^{si}$ and $^{\wedge si}$ can be computed using $|^{\text{si}}$ and $\sim^{si}$:

$$\begin{aligned} \mathbf{si_1} \,\&^{si}\, \mathbf{si_2} &= \sim^{si}(\sim^{si}\mathbf{si_1} \,|^{\text{si}} \sim^{si}\mathbf{si_2}) \\ \mathbf{si_1} \,^{\wedge si}\, \mathbf{si_2} &= (\mathbf{si_1} \,\&^{si} \sim^{si}\mathbf{si_2}) \,|^{\text{si}} (\sim^{si}\mathbf{si_1} \,\&^{si}\, \mathbf{si_2}) \\ &= \sim^{si}(\sim^{si}\mathbf{si_1} \,|^{\text{si}}\, \mathbf{si_2}) \,|^{\text{si}} \sim^{si}(\mathbf{si_1} \,|^{\text{si}} \sim^{si}\mathbf{si_2}) \end{aligned}$$

Strided-Interval Arithmetic for Different Radices

An arithmetic operation in one radix can lead to a different result from the same operation performed in another radix. Even when all radices are different powers of 2, not all effects can be fixed up merely by applying a mask to the result. In particular, the values of the x86 flags (condition codes) depend upon the radix in which an operation is performed.[5]

EXAMPLE 3.5. Suppose that the 16-bit register `ax` has the value **0xffff**. The abstract transformer for a 16-bit addition operation, say `ADD ax,1`, must account for the effect on the CF (carry) flag.

[5] Most of the arithmetic and bit-level instructions in the x86 instruction set affect some subset of the flags (condition codes), which are stored in the processor's `EFLAGS` register. For example, the x86 `CMP` instruction subtracts the second operand from the first operand, and sets the `EFLAGS` register according to the results. The values of certain bits of `EFLAGS` are used by other instructions (such as the `Jcc`, `CMOVcc`, and `SETcc` families of instructions) to direct the flow of control in the program.

&& (and)	FALSE	MAYBE	TRUE
FALSE	FALSE	FALSE	FALSE
MAYBE	FALSE	MAYBE	MAYBE
TRUE	FALSE	MAYBE	TRUE

|| (or)	FALSE	MAYBE	TRUE
FALSE	FALSE	MAYBE	TRUE
MAYBE	MAYBE	MAYBE	TRUE
TRUE	TRUE	TRUE	TRUE

^ (xor)	FALSE	MAYBE	TRUE
FALSE	FALSE	MAYBE	TRUE
MAYBE	MAYBE	MAYBE	MAYBE
TRUE	TRUE	MAYBE	FALSE

¬ (not)	
FALSE	TRUE
MAYBE	MAYBE
TRUE	FALSE

⊔ (join)	FALSE	MAYBE	TRUE
FALSE	FALSE	MAYBE	MAYBE
MAYBE	MAYBE	MAYBE	MAYBE
TRUE	MAYBE	MAYBE	TRUE

Figure 7. Operations on Bool3s.

In this example, CF needs to be set to $\mathbf{1}$, which is a different value than CF would have if we modeled the 16-bit addition as a 32-bit addition of $\mathbf{0x0000ffff}$ and $\mathbf{0x00000001}$ and masked out the lower 16 bits: the 32-bit addition would set CF to $\mathbf{0}$ because no carry results from the 32-bit operation. □

To make it convenient to define abstract transformers that track x86 flag values, the operations of strided-interval arithmetic also compute abstract condition-code values that over-approximate the values computed by the CPU, including CF (carry), ZF (zero), SF (sign), PF (parity), AF (auxiliary carry) and OF (overflow). Each strided-interval operation op^{si} returns a descriptor of the possible condition-code values that could result from applying the corresponding concrete operation op to the concretizations of the arguments of op^{si}. To represent multiple possible Boolean values, we use the abstract domain Bool3:

$$\text{Bool3} = \{\text{FALSE}, \text{MAYBE}, \text{TRUE}\}.$$

In addition to the Booleans FALSE and TRUE, Bool3 has a third value, MAYBE, which means "may be FALSE or may be TRUE". Fig. 7 shows tables for the Bool3 operations && (and), || (or), ^ (xor), ¬ (not), and ⊔ (join).

To account for effects like the one illustrated in Ex. 3.5, strided-interval arithmetic is implemented as a template that is parameterized on the number of bits. Zero-extend and sign-extend operations are also provided to convert 8-bit strided intervals to 16-bit and 32-bit strided intervals, and 16-bit strided intervals to 32-bit strided intervals.

3.3 Value-Set Arithmetic

During VSA, a set of addresses and numeric values is represented by a value-set, which is a map from memory-regions to strided intervals. A value-set associates each memory-region m with an abstract value that represents a set of offsets in m. Let Proc denote the set of AR memory-regions associated with procedures in the program, AllocMemRgn denote the set of memory-regions associated with heap-allocation sites,[6] and Global denote the memory-region associated with the global data area. We work with the following basic domains:

$$\text{MemRgn} = \{\text{Global}\} \cup \text{Proc} \cup \text{AllocMemRgn}$$
$$\text{ValueSet} = \text{MemRgn} \to \text{StridedInterval}_{\perp}$$

In this section, we give a sketch of the abstract value-set arithmetic used in CodeSurfer/x86.

[6] The implementation actually uses an augmented abstract domain that overcomes some of the imprecisions that arise due to the need to perform weak updates—i.e., accumulate information via join—on fields of summary malloc-regions. In particular, the augmented domain often allows our analysis to establish a definite link between a heap-allocated object of a class that uses one or more virtual functions and the appropriate virtual-function table (see [5]).

For brevity, we usually write value-sets as tuples. We follow the convention that the first component of a value-set refers to the set of addresses (or numbers) in Global, and $\emptyset$ denotes an empty set. For instance, the tuple $(\mathbf{1[0,9]}, \emptyset, \ldots)$ represents the set of numbers $\{0, 1, \ldots, 9\}$ and the tuple $(\emptyset, \mathbf{4[-40,-4]}, \emptyset, \ldots)$ represents the set of offsets $\{-40, -36, \ldots, -4\}$ in the first AR-region.

It is useful to classify value-sets in terms of four value-set *kinds*:

Kind	Form of value-set	
VS_{glob}	$(si_0, \emptyset, \ldots)$	si_0 is a set of offsets in the Global memory-region
VS_{single}	$(\emptyset, \ldots, si_l, \emptyset, \ldots)$	si_l is a set of offsets in the l-th memory-region ($l \neq$ Global)
VS_{arb}	$(si_0, \ldots, si_k, \ldots)$	si_k is a set of offsets in the k-th memory-region
$\top^{vs}$	$(\top^{si}, \top^{si}, \ldots)$	all addresses and numeric values

Note that a value-set $vs = (si_0, si_1, \ldots)$ has an implicit set of concrete addresses associated with each of the si_k, $k > 0$: if k corresponds to an AR-region for procedure p, these are the possible concrete stack-frame base addresses for p (relative to which local-variable offsets are calculated); if k corresponds to a malloc-region, these are the possible concrete base addresses of heap-allocated memory objects. Consequently, value-set operations cannot be performed component-wise. For instance, it would be unsound to use $(-^{\mathrm{si}}_{\mathbf{u}}\, si_0, -^{\mathrm{si}}_{\mathbf{u}}\, si_1, \ldots)$ as the value-set for the negation of vs (i.e., $-^{\mathrm{vs}}_{\mathbf{u}}\, vs$) because the implicit set of concrete addresses in $(-^{\mathrm{si}}_{\mathbf{u}}\, si_0, -^{\mathrm{si}}_{\mathbf{u}}\, si_1, \ldots)$ would not have been negated. On the contrary, the implicit set of concrete addresses in $(-^{\mathrm{si}}_{\mathbf{u}}\, si_0, -^{\mathrm{si}}_{\mathbf{u}}\, si_1, \ldots)$ would be the *same* as the implicit set of concrete addresses in $vs = (si_0, si_1, \ldots)$. Similar considerations hold for other arithmetic and bit-level operations on value-sets (cf. the entries with $\top^{vs}$ in the tables for $+^{\mathrm{vs}}$ and $-^{\mathrm{vs}}$, below).

Addition ($+^{\mathrm{vs}}$)

The following table shows the value-set kinds produced by $+^{\mathrm{vs}}$ for different kinds of arguments:

$+^{\mathrm{vs}}$	VS_{glob}	VS_{single}	VS_{arb}	$\top^{vs}$
VS_{glob}	VS_{glob}	VS_{single}	VS_{arb}	$\top^{vs}$
VS_{single}	VS_{single}	$\top^{vs}$	$\top^{vs}$	$\top^{vs}$
VS_{arb}	VS_{arb}	$\top^{vs}$	$\top^{vs}$	$\top^{vs}$
$\top^{vs}$	$\top^{vs}$	$\top^{vs}$	$\top^{vs}$	$\top^{vs}$

The value-set operation $+^{\mathrm{vs}}$ is symmetric in its arguments, and can be defined as follows:

$\underline{VS_{glob} +^{\mathrm{vs}} VS_{glob}}$: Let $vs_1 = (si_0^1, \emptyset, \ldots)$ and $vs_2 = (si_0^2, \emptyset, \ldots)$. Then $vs_1 +^{\mathrm{vs}} vs_2 = (si_0^1 +^{\mathrm{si}} si_0^2, \emptyset, \ldots)$.

$\underline{VS_{glob} +^{\mathrm{vs}} VS_{single}}$: Let $vs_1 = (si_0^1, \emptyset, \ldots)$ and $vs_2 = (\emptyset, \ldots, si_l^2, \emptyset, \ldots)$. Then $vs_1 +^{\mathrm{vs}} vs_2 = (\emptyset, \ldots, si_0^1 +^{\mathrm{si}} si_l^2, \emptyset, \ldots)$.

$\underline{VS_{single} +^{\mathrm{vs}} VS_{glob}}$: Let $vs_1 = (\emptyset, \ldots, si_l^1, \emptyset, \ldots)$ and $vs_2 = (si_0^2, \emptyset, \ldots)$. Then $vs_1 +^{\mathrm{vs}} vs_2 = (\emptyset, \ldots, si_l^1 +^{\mathrm{si}} si_0^2, \emptyset, \ldots)$.

$\underline{VS_{glob} +^{\mathrm{vs}} VS_{arb}}$: Let $vs_1 = (si_0^1, \emptyset, \ldots)$ and $vs_2 = (si_0^2, \ldots, si_k^2, \ldots)$. Then $vs_1 +^{\mathrm{vs}} vs_2 = (si_0^1 +^{\mathrm{si}} si_0^2, \ldots, si_0^1 +^{\mathrm{si}} si_k^2, \ldots)$.

$\underline{VS_{arb} +^{\mathrm{vs}} VS_{glob}}$: Let $vs_1 = (si_0^1, \ldots, si_k^1, \ldots)$ and $vs_2 = (si_0^2, \emptyset, \ldots)$. Then $vs_1 +^{\mathrm{vs}} vs_2 = (si_0^1 +^{\mathrm{si}} si_0^2, \ldots, si_k^1 +^{\mathrm{si}} si_0^2, \ldots)$.

Subtraction ($-^{\mathrm{vs}}$)

The following table shows the value-set kinds produced by $-^{\mathrm{vs}}$ for different kinds of arguments:

$-^{\mathrm{vs}}$	VS_{glob}	VS_{single}	VS_{arb}	$\top^{vs}$
VS_{glob}	VS_{glob}	$\top^{vs}$	$\top^{vs}$	$\top^{vs}$
VS_{single}	VS_{single}	$\top^{vs}$	$\top^{vs}$	$\top^{vs}$
VS_{arb}	VS_{arb}	$\top^{vs}$	$\top^{vs}$	$\top^{vs}$
$\top^{vs}$	$\top^{vs}$	$\top^{vs}$	$\top^{vs}$	$\top^{vs}$

The operation $-^{\mathrm{vs}}$ is not symmetric in its arguments; it produces $\top^{vs}$ in all but the following three cases:

$\underline{VS_{glob} -^{vs} VS_{glob}}$: Let $vs_1 = (si_0^1, \emptyset, \ldots)$ and $vs_2 = (si_0^2, \emptyset, \ldots)$. Then $vs_1 -^{vs} vs_2 = (si_0^1 -^{si} si_0^2, \emptyset, \ldots)$.

$\underline{VS_{single} -^{vs} VS_{glob}}$: Let $vs_1 = (\emptyset, \ldots, si_l^1, \emptyset, \ldots)$ and $vs_2 = (si_0^2, \emptyset, \ldots)$. Then $vs_1 -^{vs} vs_2 = (\emptyset, \ldots, si_l^1 -^{si} si_0^2, \emptyset, \ldots)$.

$\underline{VS_{arb} -^{vs} VS_{glob}}$: Let $vs_1 = (si_0^1, \ldots, si_k^1, \ldots)$ and $vs_2 = (si_0^2, \emptyset, \ldots)$. Then $vs_1 -^{vs} vs_2 = (si_0^1 -^{si} si_0^2, \ldots, si_k^1 -^{si} si_0^2, \ldots)$.

Bitwise And ($\&^{vs}$), Or ($|^{vs}$), and Xor ($\wedge^{vs}$)

Let $op^{vs} \in \{\&^{vs}, |^{vs}, \wedge^{vs}\}$ denote one of the binary bitwise value-set operations, and let $op^{si} \in \{\&^{si}, |^{si}, \wedge^{si}\}$ denote the corresponding strided-interval operation. Let $\mathbf{id}$ and $\mathbf{annihilator}$ denote the following value-sets:

op^{vs}	$\mathbf{id}$	$\mathbf{annihilator}$
$\&^{vs}$	$(\mathbf{0}[-\mathbf{1},-\mathbf{1}], \emptyset, \ldots)$	$(\mathbf{0}[\mathbf{0},\mathbf{0}], \emptyset, \ldots)$
$|^{vs}$	$(\mathbf{0}[\mathbf{0},\mathbf{0}], \emptyset, \ldots)$	$(\mathbf{0}[-\mathbf{1},-\mathbf{1}], \emptyset, \ldots)$
$\wedge^{vs}$	$(\mathbf{0}[\mathbf{0},\mathbf{0}], \emptyset, \ldots)$	

$\underline{VS_{glob}\, op^{vs}\, VS_{glob}}$: Let $vs_1 = (si_0^1, \emptyset, \ldots)$ and $vs_2 = (si_0^2, \emptyset, \ldots)$. Then $vs_1\, op^{vs}\, vs_2 = (si_0^1\, op^{si}\, si_0^2, \emptyset, \ldots)$.

$\underline{VS_{glob}\, op^{vs}\, VS}$: Let vs denote a value-set of any kind. Then

$$\mathbf{id}\, op^{vs}\, vs = vs$$
$$\mathbf{annihilator}\, op^{vs}\, vs = \mathbf{annihilator}$$

Otherwise, $(si_0, \emptyset, \ldots)\, op^{vs}\, vs = \top^{vs}$.

$\underline{VS\, op^{vs}\, VS_{glob}}$: Let vs denote a value-set of any kind. Then

$$vs\, op^{vs}\, \mathbf{id} = vs$$
$$vs\, op^{vs}\, \mathbf{annihilator} = \mathbf{annihilator}$$

Otherwise, $vs\, op^{vs}(si_0, \emptyset, \ldots) = \top^{vs}$.

Value-Set Arithmetic for Different Radices

Value-set arithmetic is templatized to account for different radices. Operations on component strided intervals are performed using the strided-interval arithmetic of the appropriate radix.

3.4 Abstract Operations for the X86 Instruction Set

VSA is a flow-sensitive, context-sensitive, abstract-interpretation algorithm (parameterized by call-string length [42]) that is based on the independent-attribute domain described below.

VSA associates each instruction with an AbsMemConfig, which, for each call-string, maps each register to a ValueSet, each flag to a Bool3, and the global-region, each AR-region, and each malloc-region to an AlocEnv (or $\bot$):

$$\begin{aligned}
\text{Flag} &= \{\text{CF, ZF, SF, PF, AF, OF}\} \\
\text{AlocEnv} &= \text{a-loc} \to \text{ValueSet} \\
\text{AbsEnv} &= \begin{array}{l} (\text{register} \to \text{ValueSet}) \\ \times\, (\text{Flag} \to \text{Bool3}) \\ \times\, (\{\text{Global}\} \to \text{AlocEnv}) \\ \times\, (\text{Proc} \to \text{AlocEnv}_\bot) \\ \times\, (\text{AllocMemRgn} \to \text{AlocEnv}_\bot) \end{array} \\
\text{AbsMemConfig} &= (\text{CallString} \to \text{AbsEnv}_\bot)
\end{aligned}$$

VSA obtains an AbsMemConfig for each instruction by an abstract interpretation [21] performed on an interprocedural CFG. The interprocedural CFG contains one node for each instruction in the executable, as well as one node for each instruction in the DLLs used by the executable. The edges of the interprocedural CFG are labeled with the instruction at the source of the edge. If the source of an edge is a branch instruction, then the edge is also labeled with the outcome of the branch.

Intraprocedural Analysis

AbsEnvs for different call-strings are propagated separately through a CFG by applying the appropriate abstract transformer at the AbsEnv level (i.e., the transformer is of type AbsEnv $\to$ AbsEnv). For instance, for a simple move instruction that has the form `MOV reg1,reg2`, the abstract transformer at the AbsEnv level can be expressed as follows (where $\uparrow i$ denotes selection of the i-th component of an AbsEnv tuple, and $m[k \leftarrow v]$ denotes the update of map m to associate key k with value v):

$\lambda env.\mathbf{let}\ r = env\uparrow 1\ \mathbf{in}\ (r[\mathtt{reg1} \leftarrow r(\mathtt{reg2})], env\uparrow 2, env\uparrow 3, env\uparrow 4, env\uparrow 5)$.

A discussion of some of the issues that arise in intraprocedural propagation, including how VSA handles memory-access operations, is found in [4, §4.1].

Interprocedural Analysis

The basic steps taken to handle parameter passing, calls, and returns are discussed in [4, §4.2]. Since the publication of [4], the implementation of CodeSurfer/x86 has been extended to have a degree of context-sensitivity, using the call-strings approach to interprocedural dataflow analysis [42].

We have also used the call-strings information to improve *intra*procedural propagation: VSA uses the call-string associated with the current AbsEnv, together with the executable's call graph, to determine which of the possible pending ARs in the current (abstract) calling context represent procedures that could have been called recursively. The distinction is important because a-locs in AR-regions of potentially recursive procedures represent more than one concrete memory location—our term for them is *summary a-locs*—and hence assignments to them must be modeled by weak updates, i.e., the new value-set computed for the a-loc by the abstract transformer must be joined with the value-set that the a-loc has in the incoming AbsEnv, rather than replacing it. This use of call-string and call-graph information can allow some a-locs in some abstract calling contexts to be identified as non-summary a-locs—in which case strong updates, rather than weak updates, are possible (i.e., in such abstract calling contexts, an assignment to the a-loc can be treated as a kill, rather than as just a possible kill).

This improves on the treatment reported in [4], where call-strings information was not available, and thus in all abstract calling contexts weak updates were always performed for a-locs of procedures that could be called recursively in *some* calling context.

Idioms

Before applying an abstract transformer, the instruction is checked to see if it matches a pattern for which we know how to carry out abstract interpretation more precisely than if value-set arithmetic is performed directly. Some examples are given below.

`XOR reg,reg`. The `XOR` instruction sets its first operand to the bitwise exclusive-or ($\wedge$) of the instruction's two operands. The idiom catches the case when `XOR` is used to set a register to $\mathbf{0}$; hence, the a-loc for register `reg` is set to the value-set $(\mathbf{0}[\mathbf{0},\mathbf{0}], \emptyset, \ldots)$.

`TEST reg,reg`. The `TEST` instruction computes the bitwise and (`&`) of its two operands, and sets the SF, ZF, and PF flags according to the result. The idiom addresses how the value of ZF is set when the value-set of `reg` has the form $(\mathbf{si}, \emptyset, \ldots)$:

$$\text{ZF} := \begin{cases} \text{TRUE} & \text{if } \gamma(\mathbf{si}) = \{0\} \\ \text{FALSE} & \text{if } \gamma(\mathbf{si}) \cap \{0\} = \emptyset \\ \text{MAYBE} & \text{otherwise} \end{cases}$$

`CMP a,b` or `CMP b,a`. In the present implementation, we assume that an allocation always succeeds (and hence value-set analysis only explores the behavior of the system on executions in which allocations always succeed). Under this assumption, we can apply the following idiom: Suppose that $k1, k2, \ldots$ are malloc-regions, the value-set for `a` is $(\emptyset, \ldots, si_{k1}, si_{k2}, \ldots)$, and the value-set for `b` is $(\mathbf{0}[\mathbf{0},\mathbf{0}], \emptyset, \ldots)$. Then ZF is set to FALSE.

4. Related Work

Other work on analyzing memory accesses in executables. Several others have proposed techniques to obtain information from executables by means of static analysis, including [34, 23, 16, 15, 17, 10, 36, 2, 9, 3, 26]. This work is summarized below.

The xGCC tool [3] analyzes XRTL intermediate code with the aim of verifying safety properties, such as the absence of buffer overflow, division by zero, and the use of uninitialized variables. The tool uses an abstract domain based on sets of intervals; it supports an arithmetic on this domain that takes into account the properties of signed two's-complement numbers. However, the domain used in xGCC does not support the notion of strides—i.e., the intervals are strided intervals with strides of 1. Because on many processors memory accesses do not have to be aligned on word boundaries, an abstract arithmetic based solely on intervals does not provide enough information to check for non-aligned accesses.

For instance, a 4-byte fetch from memory where the starting address is in the interval $[1020, 1028]$ must be considered to be a fetch of any of the following 4-byte sequences: $(1020, \ldots, 1023)$, $(1021, \ldots, 1024)$, $(1022, \ldots, 1025)$, $\ldots$, $(1028, \ldots, 1031)$. Suppose that the program writes the addresses a_1, a_2, and a_3 into the words at $(1020, \ldots, 1023)$, $(1024, \ldots, 1027)$, and $(1028, \ldots, 1031)$, respectively. Because the abstract domain cannot distinguish an unaligned fetch from an aligned fetch, a 4-byte fetch where the starting address is in the interval $[1020, 1028]$ will appear to allow address forging: e.g., a 4-byte fetch from $(1021, \ldots, 1024)$ contains the three high-order bytes of a_1, concatenated with the low-order byte of a_2.

In contrast, if an analysis knows that the starting address of the 4-byte fetch is characterized by the strided interval $\mathbf{4[1020, 1028]}$, it would discover that the set of possible values is restricted to $\{a_1, a_2, a_3\}$. Moreover, a tool that uses intervals rather than strided intervals is likely to suffer a catastrophic loss of precision when there are chains of indirection operations: if the first indirection operation fetches the possible values at $(1020, \ldots, 1023)$, $(1021, \ldots, 1024)$, $\ldots$, $(1028, \ldots, 1031)$, the second indirection operation will have to follow nine possibilities—including all addresses potentially forged from the sequence a_1, a_2, and a_3. Consequently, the use of intervals rather than strided intervals in a tool that attempts to identify potential bugs and security vulnerabilities is likely to cause a large number of false alarms to be reported.

Other work deals with memory accesses very conservatively: if a register is assigned a value from memory, it is assumed to take on any value. For instance, although the basic goal of the algorithm proposed by Debray et al. [23] is similar to that of VSA, their goal is to find an over-approximation of the set of values that each *register* can hold at each program point; for us, it is to find an over-approximation of the set of values that each (abstract) data object can hold at each program point, where data objects include global, stack-allocated, and heap-allocated memory locations, in addition to registers. In the analysis proposed by Debray et al., a set of addresses is approximated by a set of congruence values: they keep track of only the low-order bits of addresses. However, unlike VSA, their algorithm does not make any effort to track values that are not in registers. Consequently, it loses a great deal of precision whenever there is a load from memory.

Cifuentes and Fraboulet [16] give an algorithm to identify an intraprocedural slice of an executable by following the program's use-def chains. However, their algorithm also makes no attempt to track values that are not in registers, and hence cuts short the slice when a load from memory is encountered.

The two pieces of work that are most closely related to VSA are the algorithm for data-dependence analysis of assembly code of Amme et al. [2] and the algorithm for pointer analysis on a low-level intermediate representation of Guo et al. [26]. The algorithm of Amme et al. performs only an *intra*procedural analysis, and it is not clear whether the algorithm fully accounts for dependences between memory locations. The algorithm of Guo et al. is only partially flow-sensitive: it tracks registers in a flow-sensitive manner, but treats memory locations in a flow-insensitive manner. The algorithm uses partial transfer functions [48] to achieve context-sensitivity. The transfer functions are parameterized by "unknown initial values" (UIVs); however, the algorithm does not account for the possibility of called procedures corrupting the memory locations that the UIVs represent.

Several platforms have been created for manipulating executables in the presence of additional information, such as source code and debugging information, including ATOM [44], EEL [34], and Vulcan [43].

Bergeron et al. [9] present a static-analysis technique to check if an executable with debugging information adheres to a user-specified security policy.

Rival [41] presents an analysis that checks whether the assembly code produced by a compiler possesses the same safety properties as the original source code. The analysis assumes that source code and debugging information are available. After the source code is analyzed, the source-level invariants are translated into low-level invariants, which the system then attempts to prove for the low-level code.

Identification of structures. Aggregate structure identification (ASI) was devised by Ramalingam et al. to partition aggregates according to a Cobol program's memory-access patterns [37]. A similar algorithm was devised by Eidorff et al. [24] and incorporated in the Anno Domini system. The original motivation for these algorithms was the Year 2000 problem; they provided a way to identify how date-valued quantities could flow through a program.

In our work, ASI complements VSA: ASI addresses the issue of identifying the structure of aggregates, whereas VSA addresses the issue of over-approximating the contents of memory locations. ASI provides an improved method for the variable-identification facility of IDAPro, which uses only much cruder techniques (and only takes into account statically known memory addresses and stack offsets). Moreover, ASI requires more information to be on hand than is available in IDAPro (such as the range and stride of a memory-access operation). Fortunately, this is exactly the information that is available after VSA has been carried out, which means that ASI can be used in conjunction with VSA to obtain improved results: after each round of VSA, the results of ASI are used to refine the a-loc abstraction, after which VSA is run again—generally producing more precise results.

Mycroft gives a unification-based algorithm for performing type reconstruction, including identifying structures [36]. For instance, when a register is dereferenced with an offset of 4 to perform a 4-byte access, the algorithm infers that the register holds a pointer to an object that has a 4-byte field at offset 4. The type system uses disjunctive constraints when multiple type reconstructions from a single usage pattern are possible.

Mycroft points out several weaknesses of the algorithm due to the absence of certain information. Some of these could be addressed using information obtained by the techniques described in this paper:

- Mycroft explains how several simplifications could be triggered if interprocedural side-effect information were available. Once the information computed by the methods used in CodeSurfer/x86 is in hand, interprocedural side-effect information could be computed by standard techniques [19].
- Mycroft's algorithm is unable to recover information about the sizes of arrays that are identified. In our work, affine-relation analysis (ARA) [35, 33] is used to identify, for each program point, affine relations that hold. In essence, this provides information about induction-variable relationships in loops, which, in turn, can allow VSA to recover information about array sizes

when, e.g., one register is used to sweep through an array under the control of a second loop-index register.

- Mycroft does not have stride information available; however, VSA's abstract domain is based on strided intervals.
- Mycroft excludes from consideration programs in which addresses of local variables are taken because "it can be unclear as to where the address-taken object ends—a `struct` of size 8 bytes followed by a coincidentally contiguously allocated `int` can be hard to distinguish from a `struct` of 12 bytes." This is a problematic restriction for a decompiler because it is a common idiom: in C programs, addresses of local variables are frequently used as explicit arguments to called procedures (when programmers simulate call-by-reference parameter passing), and C++ and Java compilers can use addresses of local variables to implement call-by-reference parameter passing.

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

Decompilation. Past work on decompiling assembly code to a high-level language [17] is also related to our work. However, the decompilers reported in the literature are somewhat limited in what they are able to do when translating assembly code to high-level code. For instance, Cifuentes's work [17] primarily concentrates on recovery of (a) expressions from instruction sequences, and (b) control flow. We believe that the memory-access-analysis methods described in this paper would enable a decompiler to do a better job. By performing these analyses prior to decompilation proper, information about numeric values, address values, physical types, and definite links from objects to virtual-function tables [5] would be available to the decompiler.

References

[1] PREfast with driver-specific rules, October 2004. WHDC, Microsoft Corp., http://www.microsoft.com/whdc/devtools/tools/PREfast-drv.mspx.

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

[3] W. Backes. *Programmanalyse des XRTL Zwischencodes*. PhD thesis, Universitaet des Saarlandes, 2004. (In German.).

[4] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, 2004.

[5] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. TR 1548, UW-Madison, December 2005.

[6] G. Balakrishnan and T. Reps. Recovery of variables and heap structure in x86 executables. TR 1533, UW-Madison, 2005.

[7] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *VSTTE*, 2005.

[8] T. Ball and S.K. Rajamani. The SLAM toolkit. In *CAV.*, 2001.

[9] 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.

[10] J. Bergeron, M. Debbabi, M.M. Erhioui, and B. Ktari. Static analysis of binary code to isolate malicious behaviors. In *WETICE*, 1999.

[11] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, 2003.

[12] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *S–P&E*, 30, 2000.

[13] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *NDSS*, 2004.

[14] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *CCS*, 2002.

[15] C. Cifuentes and A. Fraboulet. Interprocedural data flow recovery of high-level language code from assembly. TR 421, U. Queensland, 1997.

[16] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *ICSM*, 1997.

[17] C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation. In *ICSM*, 1998.

[18] CodeSurfer, GrammaTech, Inc. "http://www.grammatech.com".

[19] K.D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI*, 1988.

[20] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE*, 2000.

[21] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL*, 1977.

[22] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, 2002.

[23] S.K. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *POPL*, 1998.

[24] P.H. Eidorff, F. Henglein, C. Mossin, H. Niss, M.H. Sørensen, and M. Tofte. Anno Domini: From type theory to year 2000 conversion tool. In *POPL*, 1999.

[25] D.R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

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

[27] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *STTT*, 2(4), 2000.

[28] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

[29] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *TOPLAS*, 12(1):26–60, January 1990.

[30] M. Howard. Some bad news and some good news. October 2002. MSDN, Microsoft Corp.

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

[32] N. Kidd, T. Reps, D. Melski, and A. Lal. WPDS++: A C++ library for weighted pushdown systems, 2004. http://www.cs.wisc.edu/wpis/wpds++/.

[33] A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *CAV*, 2005.

[34] J.R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *PLDI*, 1995.

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

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

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

[38] T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum. A next-generation platform for analyzing executables. In *APLAS*, 2005.

[39] T. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *SAS*, 2003.

[40] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *SCP*, 58(1–2):206–263, October 2005.

[41] X. Rival. Abstract interpretation based certification of assembly code. In *VMCAI*, 2003.

[42] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, chapter 7. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[43] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. TR 2001-50, Microsoft Research, 2001.

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

[45] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, 2000.

[46] D.W. Wall. Systems for late code modification. In R. Giegerich and S.L. Graham, editors, *Code Generation – Concepts, Tools, Techniques*, pages 275–293. Springer-Verlag, 1992.

[47] H.S. Warren, Jr. *Hacker's Delight*. Addison-Wesley, 2003.

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