Intermediate-Representation Recovery from Low-Level Code

Thomas Reps  Gogul Balakrishnan  Junghee Lim

University of Wisconsin-Madison
{reps,bgo,unjh}@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: “What You See Is Not What You eXecute”. 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]:

```c
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.

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
### Report Documentation Page

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

<table>
<thead>
<tr>
<th>1. REPORT DATE</th>
<th>2. REPORT TYPE</th>
<th>3. DATES COVERED</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006</td>
<td></td>
<td>00-00-2006 to 00-00-2006</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4. TITLE AND SUBTITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intermediate-Representation Recovery from Low-Level Code</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5a. CONTRACT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5b. GRANT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5c. PROGRAM ELEMENT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5d. PROJECT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5e. TASK NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5f. WORK UNIT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6. AUTHOR(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>University of Wisconsin , Computer Sciences Department, 716 Langdon Street, Madison, WI, 53706</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8. PERFORMING ORGANIZATION REPORT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9. SPONSORING/MonitorING AGENCY NAME(S) AND ADDRESS(ES)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10. SPONSOR/Monitor’S ACRONYM(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>11. SPONSOR/Monitor’S REPORT NUMBER(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12. DISTRIBUTION/AVAILABILITY STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approved for public release; distribution unlimited</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>13. SUPPLEMENTARY NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>14. ABSTRACT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>15. SUBJECT TERMS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16. SECURITY CLASSIFICATION OF:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. REPORT unclassified</td>
</tr>
<tr>
<td>b. ABSTRACT unclassified</td>
</tr>
<tr>
<td>c. THIS PAGE unclassified</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>17. LIMITATION OF ABSTRACT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>18. NUMBER OF PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>19a. NAME OF RESPONSIBLE PERSON</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Standard Form 298 (Rev. 8-98)  
Prescribed by ANSI Std Z39-18
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-killled 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 x 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.
maTech’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 $V$
- the set of values $V$ that 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 re-
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 offset-sets 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 $l_3$ and $l_4$ 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 $l_2$ and $l_4$ hold. Reads from and writes to (parts of) $l_2$ and $l_4$ would be treated conservatively, and VSA would report that $l_{12}$ and $l_{34}$ hold $\bot$ (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-$\bot$) 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

```c
int a[10], i;
for (i = 0; i < 10; i++)
a[i] = i;
```
At the same time, VSA is similar to range analysis and other nu-
cularly allocated memory; see footnote 6 and [5].

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
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-
gained 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 de-
veloped for programs written in high-level languages, which deter-
mine 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 nu-
cumeric 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 ad-
  dress that is not aligned on a 4-byte word boundary—spans
  parts of two words, and provides a way to forge a new ad-
  dress from parts of old addresses. It is important for VSA
to discover information about the alignments and strides of mem-
ory accesses, or else most indirect-addressing operations appear
as 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 informa-
tion 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

\(^1\) The implementation actually uses a more precise abstraction of dynami-
cally allocated memory; see footnote 6 and [5].
Let $[x]_m$ denote the congruence class of $x$ mod $m$, defined as $[x]_m = \{x + i \times m \ | \ i \in \mathbb{Z}\}$; note that $[x]_0 = \{x\}$.

### 3.2 Strided-Interval Arithmetic

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

**Definition 3.1. [Meaning of a strided interval].** A k-bit strided interval $[lb, ub]$, represents the set of integers

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

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

In a strided interval $[lb, ub, s]$, $s$ is called the stride, and $[lb, ub]$ is called the interval. Stride $s$ is unsigned; bounds $lb$ and $ub$ are signed.\(^7\) The strided interval 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 $[2^{32} - 1, -2^{31} 2^{31} - 1]$. As defined above, some sets of numbers can be represented by more than one strided interval. For instance, $\gamma([4, 14]) = \{4, 8, 12\}$. 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 0). For example, $[4, 12]$ and $[0, 12]$ are reduced strided intervals; $[4, 14]$ and $[12, 12]$ 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 operation $\oplus$, if $s_{i3} = s_{i1} \oplus s_{i2}$, then $\gamma(s_{i3}) \supseteq \{a \oplus b | a \in \gamma(s_{i1}) \text{ and } b \in \gamma(s_{i2})\}$.

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 $s_{i3}$ can be obtained for each operation $\oplus$. $\oplus \in \{+u, -u, +u, -u, u, u, u, u, u\}$

### Addition ($+u$)

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

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 $x + y$ is bounded by $[a + c, b + d]$ even though both bound calculations overflow. Thus, we merely need to identify cases (2) and (4), in which 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 $u$ is negative, then case (2) holds; if $v$ is negative, then case (4) holds ([47], p. 57).

\(^7\)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, $x$ denotes an unsigned value; if $y$ is an interval bound, $y$ denotes a signed two’s-complement value.

---

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

<table>
<thead>
<tr>
<th>Case</th>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a + c &lt; -2^k$, $b + d &lt; 2^k$</td>
<td>$a + c \leq x + y \leq b + d$</td>
</tr>
<tr>
<td>2</td>
<td>$a + c &lt; -2^k$, $b + d &gt; 2^k$</td>
<td>$a + c \leq x + y \leq b + d$</td>
</tr>
<tr>
<td>3</td>
<td>$-2^k &lt; a + c &lt; 2^k$, $b + d &lt; 2^k$</td>
<td>$a + c \leq x + y \leq b + d$</td>
</tr>
<tr>
<td>4</td>
<td>$-2^k &lt; a + c &lt; 2^k$, $b + d &gt; 2^k$</td>
<td>$a + c \leq x + y \leq b + d$</td>
</tr>
<tr>
<td>5</td>
<td>$a + c &gt; 2^k$, $b + d &gt; 2^k$</td>
<td>$a + c \leq x + y \leq b + d$</td>
</tr>
</tbody>
</table>

**Figure 2.** Implementation of abstract addition ($+u$) 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 $a + c$, $x + y$, and $b + d$ yield values that are too high by $2^k$ (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 $2^k$.\(^8\)

Fig. 2 shows a C procedure that uses these ideas to compute $s_{i1}[a, b] + s_{i2}[c, d]$, but also takes the strides $s_{i1}$ and $s_{i2}$ into account. The gcd (greatest common divisor) operation is used to find a sound stride for the result.\(^9\)

**Theorem 3.4. [Soundness of $+u$].** If $s_{i3} = s_{i1} + s_{i2}$, then $\gamma(s_{i3}) \supseteq \{a + b | a \in \gamma(s_{i1}) \text{ and } b \in \gamma(s_{i2})\}$.\(^9\)

**Proof:** The soundness of the interval $s_{i3}$ follows from the arguments given in [47]. We need to show that the stride computed by procedure $\text{addSI}$ 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 $[-2^k, 2^k - 1]$, so the stride of $l$ is obviously sound. In all other situations, gcd is used to find the stride.

Let $s_{i1} = s_{i1}[lb_1, ub_1]$, $s_{i2} = s_{i2}[lb_2, ub_2]$, and $s_{i3} = s_{i3}[lb_3, ub_3]$. We consider the cases where $s_{i3} = \text{gcd}(s_{i1}, s_{i2})[lb_1 + lb_2, ub_1 + ub_2]$.

\(^8\)By convention, gcd(0, $a$) = $a$, gcd($a$, 0) = $a$, and gcd(0, 0) = 0.

\(^9\)By the assumption that we work only with *reduced* strided intervals, in a strided interval $s_{i1}[lb, ub]$, $s \neq 0$ implies that $s$ divides evenly into $(ub - lb)$.
Let $s = \gcd(s_1, s_2), s_1 = x \times m$, and $s_2 = x \times n$. We wish to show that $s$ divides evenly into the difference between an arbitrary element $e$ in $S_1 + S_2$ and the lower-bound value $b_1 + b_2$. Let $e$ be some element of $S_1 + S_2$: $e = (b_1 + b_2) + i \times s_1 + j \times s_2$, where $0 \leq i \leq b_1$ and $0 \leq j \leq b_2$. The difference $e - (b_1 + b_2)$ 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 $S_1 + S_2$ 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 $-2^{31} \leq e' \leq 2^{31} - 1$ be $e$ adjusted by an appropriate multiple of $2^{32}$. Similarly, let $-2^{31} \leq lb \leq 2^{31} - 1$ 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 $S_1 + S_2$ are similarly adjusted.) The argument that $e - (lb_1 + lb_2)$ is divisible by $s$ carries over in each case to an argument that $e' \in [lb]$. Consequently, $\gamma(sA) \subseteq S_1 + S_2$, as was to be shown. \(\square\)

**Unary Minus (\(-^u\))**

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

$$-^u(s[e,d]) = \begin{cases} [0,-2^{31}, -2^{31}] & \text{if } c = d = -2^{31} \\
[s-d, -c] & \text{if } c \neq -2^{31} \\
[1-2^{31}, 2^{31} - 1] & \text{otherwise} \end{cases}$$

**Subtraction (\(-^d\)), Increment (\(+^d\)), and Decrement (\(-^d\))**

The $+^d$ and $-^d$ operations on strided intervals can be used to implement other arithmetic operations, such as subtraction ($-^d$), increment ($+^d$), and decrement ($-^d$), as follows:

$$x -^d y = x -^d (e^d y)$$

$$+^d x = x + ^d 0[1, 1]$$

$$-^d x = x + ^d 0[-1, -1]$$

**Bitwise Or (\(^{\text{bitwise-or}}\))**

Following Warren [47, p. 58–63], we develop the algorithm for $|^{\text{bitwise-or}}$ (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 $|^{\text{bitwise-or}}$. Suppose that we have the following bounds on unsigned values $x$ and $y$: $0 \leq x \leq b$ and $0 \leq y \leq d$.

We reject $a$ if $a$ can be changed to 1, and all bits to the right set to 0; yielding a number $a'$ such that $a' \leq b$ and $(a' | c) < (a | c)$, or a $b$ if $b$ can be changed to 1, and all bits to the right set to 0; yielding a number $c'$ such that $c' \leq d$ and $(a | c') < (a | c)$.

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

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

0000101 = $a = 0 \leq x = b = 0001001$  
0010011 = $c = y \leq d = 0101001$.

We reject $a' = 0010000$ because $0010000 \leq 0001001 = b$; however, we find that $a' = 0010010$ meets the condition  

0010100 = $0101001 - 1 = d$, and  

$$(a | c') = (0001001, 0010100) = 0010101 = 0010111 = 000101000101110 = (a | c).$$

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, $c' = 0010100 > 0010011 = c$, without having to worry about whether a stride value repeatedly added to $c$ might miss $c'$.

The algorithm to find the maximum possible value of $x \land y$ (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 $x$ and $y$, where $a \leq x \leq b$ and $c \leq y \leq d$ [47, p. 63]. The method calls the procedures from Figs. 3 and 4, which find bounds on the bitwise-or of two unsigned values.

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

Function minOR 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 $x$ [47, p. 11], i.e., $y$ is set to the binary number $0^{12}$, where $i + j = 32$ and $j$ equals the number of trailing zeroes
in x. The trailing ones of y are then counted in the while-loop in lines [5]–[8].

We now turn to the algorithm for bitwise-or on signed integers (10). Suppose that we want to perform $s_1 \ll [a, b] \ll s_2 \ll [c, d]$. As illustrated by the topmost set shown in Fig. 6, all elements in $\gamma(s_1 \ll [a, b])$ share the same $t_1 = ntz(s_1)$ low-order bits; because the $t_1$ low-order bits of stride $s_1$ are all 0, repeated addition of $t_1$ to $a$ cannot affect the $t_1$ low-order bits. Similarly, all elements in $\gamma(s_2 \ll [c, d])$ share the same $t_2 = ntz(s_2)$ 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 $s = 2^t = (1 << t)$ as the stride of the answer, and the value of the shared $t$ low-order bits can be calculated by $r = (a \& \text{mask}) \ll (c \& \text{mask})$, where

$$\text{mask} = (1 << t) - 1.$$

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 $s_1 \ll [a, b] \ll s_2 \ll [c, d]$, we perform the following steps:

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

**Bitwise not ($\sim$)**, And ($\&$), and Xor ($\oplus$)

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

$$\sim(s[l_b \& \sim \text{mask} \ll r], (r_b \& \sim \text{mask} \ll r]).$$


gamma(s_1 \ll [a, b]) = \{a, a + s_1, a + 2s_1, \ldots, b\}

gamma(s_2 \ll [c, d]) = \{c, c + s_2, c + 2s_2, \ldots, d\}

t_1 = ntz(s_1)

t_2 = ntz(s_2)

t = \min(t_1, t_2)

$$\gamma(s_1 \ll [a, b]) \ll \gamma(s_2 \ll [c, d])$$

(1)

**Figure 5. Counting trailing 0’s of x [47, p. 86].**
In this example, CF needs to be set to 1, which is a different value than CF would have if we modeled the 16-bit addition as a 32-bit addition of 0x000000ff and 0x00000001 and masked out the lower 16 bits: the 32-bit addition would set CF to 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 $op^\mathit{vs}$ 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 $op^\mathit{vs}$. 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), $^\wedge$ (xor), $\neg$ (not), and $\cup$ (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, and Global denote the memory-region associated with the global data area. We work with the following basic domains:

- $\text{MemRgn} = \{\text{Global}\} \cup \text{Proc} \cup \text{AllocMemRgn}$
- $\text{ValueSet} = \text{MemRgn} \rightarrow \text{StridedInterval}$

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

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 $\langle \{0, 9\}, \emptyset, \ldots \rangle$ represents the set of numbers $\{0, 1, 2, 3, 4, 5\}$ and the tuple $\langle \{0, 4\}, \emptyset, \ldots \rangle$ represents the set of offsets $\{-40, -36, \ldots \}$ in the first AR-region.

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

<table>
<thead>
<tr>
<th>Kind</th>
<th>Form of value-set</th>
</tr>
</thead>
<tbody>
<tr>
<td>$VS_{\text{glob}}$</td>
<td>$\langle s_{\text{glob}}, \emptyset, \ldots \rangle$</td>
</tr>
<tr>
<td>$VS_{\text{single}}$</td>
<td>$\langle \emptyset, \ldots, s_{\text{single}}, \emptyset, \ldots \rangle$</td>
</tr>
<tr>
<td>$VS_{\text{arb}}$</td>
<td>$\langle s_{\text{arb}}, \ldots, s_{\text{arb}}, \emptyset, \ldots \rangle$</td>
</tr>
<tr>
<td>$^\wedge$</td>
<td>$\langle 1,\ldots,1,0,\ldots \rangle$</td>
</tr>
</tbody>
</table>

Note that a value-set $vs = \langle s_{\text{arb}}, s_{\text{arb}}, \ldots \rangle$ has an implicit set of concrete addresses associated with each of the $s_{\text{arb}}$, $k > 0$: 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 mallocation, 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 $\langle 1, -2, 3 \rangle$ as the value-set for the negation of $vs$ (i.e., $-vs$) because the implicit set of concrete addresses in $\langle -s_{\text{arb}}, -s_{\text{arb}}, \ldots \rangle$ would not have been negated. On the contrary, the implicit set of concrete addresses in $\langle -s_{\text{arb}}, -s_{\text{arb}}, \ldots \rangle$ would be the same as the implicit set of concrete addresses in $vs = \langle s_{\text{arb}}, s_{\text{arb}}, \ldots \rangle$. Similar considerations hold for other arithmetic and bit-level operations on value-sets (cf. the entries with $^\wedge$ in the tables for $^\wedge$ and $^\wedge$, below).

### Addition ($+^\wedge$)

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>$VS_{\text{glob}}$</th>
<th>$VS_{\text{single}}$</th>
<th>$VS_{\text{arb}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$^\wedge$</td>
<td>$+$</td>
<td>$+$</td>
<td>$+$</td>
</tr>
</tbody>
</table>

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

- $VS_{\text{glob}} +^\wedge VS_{\text{glob}}$:
  - Let $vs_1 = \langle s_{\text{glob}}^1, \emptyset, \ldots \rangle$ and $vs_2 = \langle s_{\text{glob}}^2, \emptyset, \ldots \rangle$. Then $vs_1 +^\wedge vs_2 = \langle s_{\text{glob}}^1 + s_{\text{glob}}^2, \emptyset, \ldots \rangle$.

- $VS_{\text{single}} +^\wedge VS_{\text{single}}$:
  - Let $vs_1 = \langle s_{\text{single}}^1, \emptyset, \ldots \rangle$ and $vs_2 = \langle s_{\text{single}}^2, \emptyset, \ldots \rangle$. Then $vs_1 +^\wedge vs_2 = \langle s_{\text{single}}^1 + s_{\text{single}}^2, \emptyset, \ldots \rangle$.

- $VS_{\text{arb}} +^\wedge VS_{\text{arb}}$:
  - Let $vs_1 = \langle s_{\text{arb}}^1, \ldots, s_{\text{arb}}^1, \emptyset, \ldots \rangle$ and $vs_2 = \langle s_{\text{arb}}^2, \ldots, s_{\text{arb}}^2, \emptyset, \ldots \rangle$. Then $vs_1 +^\wedge vs_2 = \langle s_{\text{arb}}^1 + s_{\text{arb}}^2, \ldots, s_{\text{arb}}^1 + s_{\text{arb}}^2, \emptyset, \ldots \rangle$.

The operation $-^\wedge$ is not symmetric in its arguments; it produces $T^\wedge$ in all but the following three cases:

### Subtraction ($-^\wedge$)

The following table shows the value-set kinds produced by $-^\wedge$ for different kinds of arguments:

<table>
<thead>
<tr>
<th>Operation</th>
<th>$VS_{\text{glob}}$</th>
<th>$VS_{\text{single}}$</th>
<th>$VS_{\text{arb}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-^\wedge$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
</tbody>
</table>
absenvs for different call-strings are propagated separately through intraprocedural analysis labeled with the instruction at the source of the edge. If the source is a-loc for register \( r \), then the target a-loc can be treated as a kill, rather than as just a possible kill. Otherwise, the a-loc can be treated as an unknown or a-loc—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 (\( \land \)) of the instruction’s two operands. The idiom catches the case when XOR is used to set a register to \( 0 \); hence, the a-loc for register reg is set to the value-set \( \langle \{0[0, 0], \ldots \} \rangle \).

**TEST reg, reg.** The TEST instruction computes the bitwise and (\& or \( \land \)) of its two operands, and sets the SF, PF, and ZF flags according to the result. The idiom describes how the value of ZF is set when the value-set of reg has the form \( \langle si, \ldots \rangle \):

\[
ZF := \begin{cases} 
\text{TRUE} & \text{if } [\gamma[si]] = [0] \\
\text{FALSE} & \text{if } [\gamma[si]] \cap [0] = [0] \\
\text{MAYBE} & \text{otherwise}
\end{cases}
\]

**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 \( \langle 0, \ldots, s1a, s2a, \ldots \rangle \), and the value-set for b is \( \langle 0[0, 0], \ldots \rangle \). 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 stripped 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: $(1020, \ldots, 1023)$, $(1021, \ldots, 1024)$, $(1022, \ldots, 1025)$, $(1028, \ldots, 1031)$. 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 stripped interval $[1020, 1028]$, 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 stripped 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 $(1020, \ldots, 1023)$, $(1021, \ldots, 1024)$, $(1028, \ldots, 1031)$, the second indirection operation will have to follow nine possibilities—including all addresses potentially forged from the sequence $a_1, a_2, a_3$. Consequently, the use of intervals rather than stripped 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 by 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 intraprocedural 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 contiguous 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.

Decomputation

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