WYSINWYX:
What You See Is Not What You eXecute

GOGUL BALAKRISHNAN
NEC Laboratories America, Inc.
and
THOMAS REPS
University of Wisconsin and GrammaTech, Inc.

Over the last seven years, we have developed static-analysis methods to recover a good approximation to the variables and dynamically-allocated memory objects of a stripped executable, and to track the flow of values through them. The paper presents the algorithms that we developed, explains how they are used to recover intermediate representations (IRs) from executables that are similar to the IRs that would be available if one started from source code, and describes their application in the context of program understanding and automated bug hunting.

Unlike algorithms for analyzing executables that existed prior to our work, the ones presented in this paper provide useful information about memory accesses, even in the absence of debugging information. The ideas described in the paper are incorporated in a tool for analyzing Intel x86 executables, called CodeSurfer/x86. CodeSurfer/x86 builds a system dependence graph for the program, and provides a GUI for exploring the graph by (i) navigating its edges, and (ii) invoking operations, such as forward slicing, backward slicing, and chopping, to discover how parts of the program can impact other parts.

To assess the usefulness of the IRs recovered by CodeSurfer/x86 in the context of automated bug hunting, we built a tool on top of CodeSurfer/x86, called Device-Driver Analyzer for x86 (DDA/x86), which analyzes device-driver executables for bugs. Without the benefit of either source code or symbol-table/debugging information, DDA/x86 was able to find known bugs (that had been discovered previously by source-code-analysis tools), along with useful error traces, while having a low false-positive rate. DDA/x86 is the first known application of program analysis/verification techniques to industrial executables.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verifi-

Authors' addresses: G. Balakrishnan, NEC Laboratories America, Inc., 4 Independence Way, Princeton, NJ 08540; `bgogul@nec-labs.com`. T. Reps, Computer Sciences Dept., Univ. of Wisconsin, 1210 W. Dayton St., Madison, WI 53703, and GrammaTech, Inc., 317 N. Aurora St., Ithaca, NY 14850; `reps@cs.wisc.edu`. At the time the research reported in the paper was carried out, G. Balakrishnan was affiliated with the University of Wisconsin.

The work was supported in part by ONR under grants N00014-01-1-{0796, 0708}, by NSF under grants CCR-9986308 and CCF-{0540955, 0524051}, by HSARPA under AFRL contract FA8750-05-C-0179, and by AFRL under contract FA8750-06-C-0249.

T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

Portions of the work appeared in the 13th, 14th, and 17th Int. Confs. on Compiler Construction [Balakrishnan and Reps 2004; Balakrishnan et al. 2005; Reps and Balakrishnan 2008], the 17th Int. Conf. on Computer Aided Verification [Balakrishnan et al. 2005], the 3rd Asian Symp. on Prog. Langs. and Systems [Reps et al. 2005], the 2006 Workshop on Part. Eval. and Semantics-based Prog. Manip. [Reps et al. 2006], the 13th Int. Static Analysis Symp. [Balakrishnan and Reps 2006], the 8th Int. Conf. on Verif., Model Checking, and Abs. Interp. [Balakrishnan and Reps 2007], and the 14th Int. Conf. on Tools and Algs. for the Const. and Analysis of Systems [Balakrishnan and Reps 2008], as well as in G. Balakrishnan's Ph.D. dissertation [Balakrishnan 2007].

cation—*Assertion checkers; model checking*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering* ; D.3.2 [**Programming Languages**]: Language Classifications—*Macro and assembly languages*; D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*; E.1 [**Data**]: Data Structures—*arrays; lists, stacks, and queues; records*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms: Algorithms, Security, Theory, Verification

Additional Key Words and Phrases: Abstract interpretation, context-sensitive analysis, data-structure recovery, interprocedural dataflow analysis, pointer analysis, reverse engineering, static analysis

1. INTRODUCTION

Recent research in programming languages, software engineering, and computer security has led to new kinds of tools for analyzing programs for bugs and security vulnerabilities [Havelund and Pressburger 2000; Wagner et al. 2000; Engler et al. 2000; Corbett et al. 2000; Bush et al. 2000; Ball and Rajamani 2001; Chen and Wagner 2002; Henzinger et al. 2002; Das et al. 2002]. In these tools, static analysis is used to determine a conservative answer to the question "Can the program reach a bad state?"[1] Some of this work has already been transitioned to commercial products for source-code analysis (see Ball et al. [2006], [Coverity], and [CodeSonar]).

However, these tools all focus on analyzing *source code* written in a high-level language. Unfortunately, most programs that an individual user will install on his computer, and many commercial off-the-shelf (COTS) programs that a company will purchase, are delivered as stripped machine code (i.e., neither source code nor symbol-table/debugging information is available). If an individual or company wishes to vet such programs for bugs, security vulnerabilities, or malicious code (e.g., back doors, time bombs, or logic bombs) the availability of good source-code-analysis products is irrelevant.

Less widely recognized is that even when the original source code is available, source-code analysis has certain drawbacks [Howard 2002; WHDC 2007]. The reason is that computers do not execute source code; they execute *machine-code* programs that are generated from source code. The transformation from source code to machine code can introduce subtle but important differences between what a programmer intended and what is actually executed by the processor. For instance, the following compiler-induced vulnerability was discovered during the Windows security push in 2002 [Howard 2002]: the Microsoft C++ .NET compiler reasoned that because the program fragment shown below on the left never uses the values written by `memset` (intended to scrub the buffer pointed to by `password`), the

[1]Static analysis provides a way to obtain information about the possible states that a program reaches during execution, but without actually running the program on specific inputs. Static-analysis techniques explore the program's behavior for *all* possible inputs and *all* possible states that the program can reach. To make this feasible, the program is "run in the aggregate"—i.e., on descriptors that represent *collections* of memory configurations [Cousot and Cousot 1977].

`memset` call could be removed—thereby leaving sensitive information exposed in the freelist at runtime.

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

Such a vulnerability is invisible in the original source code; it can only be detected by examining the low-level code emitted by the optimizing compiler. We call this the WYSINWYX phenomenon (pronounced "wiz-in-wicks"): **W**hat **Y**ou **S**ee [in source code] **I**s **N**ot **W**hat **Y**ou e**X**ecute [Reps et al. 2005; Balakrishnan et al. 2007; Balakrishnan 2007].

WYSINWYX is not restricted to the presence or absence of procedure calls; on the contrary, it is pervasive. Some of the reasons why analyses based on source code can provide the wrong level of detail include

—Many security exploits depend on platform-specific details that exist because of features and idiosyncrasies of compilers and optimizers. These include memory-layout details (such as the positions—i.e., offsets—of variables in the runtime stack's activation records and the padding between structure fields), register usage, execution order (e.g., of actual parameters at a call), optimizations performed, and artifacts of compiler bugs. Bugs and security vulnerabilities can escape notice when a tool is unable to take into account such fine-grained details.

—Analyses based on source code[2] 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 and that can lead to bugs or security vulnerabilities (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 are sometimes modified subsequent to compilation, e.g., to perform optimizations or insert instrumentation code [Wall 1992]. They may also be modified to insert malicious code. Such modifications are not visible to tools that analyze source code.

In short, even when source code is available, a substantial amount of information is hidden from source-code-analysis tools, which can cause bugs, security vulnerabilities, and malicious behavior to be invisible to such tools.

The alternative is to perform static analysis at the machine-code level. The advantage of this approach is that the machine code contains the actual instructions that will be executed; this addresses the WYSINWYX phenomenon because it provides information that reveals the actual behavior that arises during program execution.

Although having to perform static analysis on machine code represents a daunting challenge, there is also a possible silver lining: by analyzing an artifact that is closer to what is actually executed, a static-analysis tool may be able to obtain a *more accurate* picture of a program's properties. The reason is that—to varying degrees—the semantic definition of every programming language leaves certain details unspecified. Consequently, for a source-code analyzer to be sound, it must

[2]Terms like "analyses based on source code" and "source-code analyses" are used as a shorthand for "analyses that work on intermediate representations (IRs) built from source code."

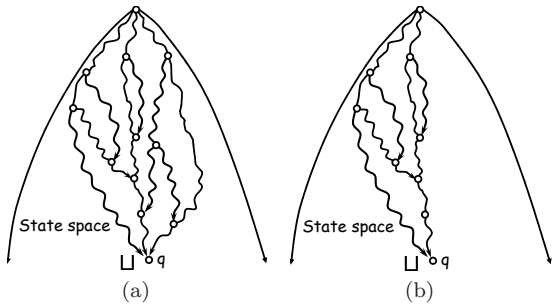


Fig. 1. Source-code analysis, which must account for all possible choices made by the compiler, must summarize more paths (see (a)) than machine-code analysis (see (b)). Because the latter can focus on fewer paths, it can yield more precise results.

account for *all* possible implementations, whereas a machine-code analyzer only has to deal with *one* possible implementation—namely, the one for the code sequence chosen by the compiler.

For instance, in C and C++ the order in which actual parameters are evaluated is not specified: actuals may be evaluated left-to-right, right-to-left, or in some other order; a compiler could even use different evaluation orders for different functions. Different evaluation orders can give rise to different behaviors when actual parameters are expressions that contain side effects. For a source-level analysis to be sound, at each call site it must take the join ($\sqcup$) of the results from analyzing each permutation of the actuals.[3] In contrast, an analysis of an executable only needs to analyze the particular sequence of instructions that lead up to the call.

Static-analysis tools are always fighting imprecision introduced by the join operation. One of the dangers of static-analysis tools is that loss of precision by the analyzer can lead to the user being swamped with a huge number of reports of potential errors, most of which are false positives. As illustrated in Fig. 1, because a source-code-analysis tool summarizes more behaviors than a tool that analyzes machine code, the join performed at q must cover more abstract states. This can lead to less-precise information than that obtained from machine-code analysis. Because more-precise answers mean a lower false-positive rate, machine-code-analysis tools have the potential to report fewer false positives.

There are other trade-offs between performing analysis at source level versus the machine-code level: with source-code analysis one can hope to learn about bugs and vulnerabilities that exist on multiple platforms, whereas analysis of the machine code only provides information about vulnerabilities on the specific platform on which the executable runs. From that standpoint, source-code analysis and machine-code analysis are complementary.

Although it is possible to create source-code tools that strive to have greater fidelity to the program that is actually executed—examples include Chandra and Reps [1999] and Nita et al. [2008]—in the limit, the tool would have to incorporate

[3]We follow the conventions of abstract interpretation [Cousot and Cousot 1977], where the lattice of properties is oriented so that the confluence operation used where paths come together is join ($\sqcup$). In dataflow analysis, the lattice is often oriented so that the confluence operation is meet ($\sqcap$). The two formulations are duals of one another.

all the platform-specific decisions that would be made by the compiler. Because such decisions depend on the level of optimization chosen, to build these choices into a tool that works on a representation that is close to the source level would require simulating much of the compiler and optimizer inside the analysis tool. Such an approach is impractical.

In addition to addressing the WYSINWYX issue, performing analysis at the machine-code level provides a number of other benefits:

—Programs typically make extensive use of libraries, including dynamically linked libraries (DLLs), which may not be available as source code. Typically, source-code analyses are performed using code stubs that model the effects of library calls. Because these are created by hand, they may contain errors, which can cause an analysis to return incorrect results. In contrast, a machine-code-analysis tool can analyze the library code directly [Gopan and Reps 2007].

—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-code-analysis tools typically either skip over inlined assembly [CodeSurfer] or do not push the analysis beyond sites of inlined assembly [PREfast 2004]. To a machine-code-analysis tool, inlined assembly just amounts to additional instructions to analyze.

—Source-code-analysis tools are only applicable when source is available, which limits their usefulness in security applications (e.g., to analyzing code from open-source projects).

Research carried out during the last decade by our research group [Xu et al. 2000; 2001; Balakrishnan and Reps 2004; Reps et al. 2005; Reps et al. 2006; Balakrishnan and Reps 2006; 2007; Gopan and Reps 2007; Balakrishnan 2007; Lim and Reps 2008; Balakrishnan and Reps 2008] as well as by others [Larus and Schnarr 1995; Cifuentes and Fraboulet 1997b; Debray et al. 1998; Bergeron et al. 1999; Amme et al. 2000; De Sutter et al. 2000; Bergeron et al. 2001; Kiss et al. 2003; Debray et al. 2004; Backes 2004; Regehr et al. 2005; Guo et al. 2005; Christodorescu et al. 2005; Kruegel et al. 2005; Cova et al. 2006; Chang et al. 2006; Brumley and Newsome 2006; Emmerik 2007; Zhang et al. 2007] has developed the foundations for performing static analysis at the machine-code level. The machine-code-analysis problem comes in three versions: (i) in addition to the executable, the program's source code is also available; (ii) the source code is unavailable, but the executable includes symbol-table/debugging information ("unstripped executables"), and (iii) the executable has no symbol-table/debugging information ("stripped executables"). The appropriate variant to work with depends on the intended application. Many techniques apply to multiple variants, but are severely hampered when symbol-table/debugging information is absent.

In 2004, we supplied a key missing piece, particularly for analysis of stripped executables [Balakrishnan and Reps 2004]. Previous to that work, static-analysis tools for machine code had rather limited abilities: it was known how to (i) track values in registers and, in some cases, the stack frame [Larus and Schnarr 1995], and

(ii) analyze control flow (sometimes by applying local heuristics to try to resolve indirect calls and indirect jumps, but otherwise ignoring them).

The work presented in our 2004 paper [Balakrishnan and Reps 2004] provided a way to apply the tools of abstract interpretation [Cousot and Cousot 1977] to the problem of analyzing stripped executables, and we followed this up with other techniques to complement and enhance the approach [Reps et al. 2005; Lal et al. 2005; Reps et al. 2006; Balakrishnan and Reps 2006; 2007; Balakrishnan 2007; Balakrishnan and Reps 2008]. This body of work has resulted in a method to recover a good approximation to an executable's variables and dynamically allocated memory objects, and to track the flow of values through them. These methods are incorporated in a tool called CodeSurfer/x86 [Balakrishnan et al. 2005].

CodeSurfer/x86: A Platform for Recovering IRs from Stripped Executables. Given a stripped executable as input, CodeSurfer/x86 [Balakrishnan et al. 2005] recovers IRs that are similar to those that would be available had one started from source code. The recovered IRs include control-flow graphs (CFGs), with indirect jumps resolved; a call graph, with indirect calls resolved; information about the program's variables; possible values for scalar, array, and pointer variables; sets of used, killed, and possibly-killed variables for each CFG node; and data dependences. The techniques employed by CodeSurfer/x86 do not rely on debugging information being present, but can use available debugging information (e.g., Windows `.pdb` files) if directed to do so.

The analyses used in CodeSurfer/x86 are a great deal more ambitious than even relatively sophisticated disassemblers, such as IDAPro [IDAPro]. At the technical level, they address the following problem:

> *Given a (possibly stripped) executable E, 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, and*
> *—for each machine register and variable V in scope at I,*
> *statically compute an accurate over-approximation to the set of values that V may contain when I executes.*

It is useful to contrast this approach against the approach used in much of the other work that now exists on analyzing executables. Many research projects have focused on *specialized* analyses to identify aliasing relationships [Debray et al. 1998], data dependences [Amme et al. 2000; Cifuentes and Fraboulet 1997b], targets of indirect calls [De Sutter et al. 2000], values of strings [Christodorescu et al. 2005], bounds on stack height [Regehr et al. 2005], and values of parameters and return values [Zhang et al. 2007]. In contrast, CodeSurfer/x86 addresses all of these problems by means of a set of analyses that focuses on the problem stated above. In particular, CodeSurfer/x86 discovers an over-approximation of the set of states that can be reached at each point in the executable—where a *state* means *all* of the state: values of registers, flags, and the contents of memory—and thereby provides information about aliasing relationships, targets of indirect calls, etc.

One of the goals of CodeSurfer/x86 is to be able to detect whether an executable conforms to a standard compilation model. By "standard compilation model" we

mean that the executable has procedures, activation records (ARs), a global data region, and a free-storage pool; 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 performed by CodeSurfer/x86, these aspects of the program are checked. When violations are detected, an error report is issued, and the analysis proceeds. In doing so, however, we generally choose to have the analyzer only explore behaviors that stay within those of the desired execution model. 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. Consequently, if the executable conforms to the standard compilation model, CodeSurfer/x86 creates a valid IR for it; if the executable does not conform to the model, then one or more violations will be discovered, and corresponding error reports will be issued; if the (human) analyst can determine that the error report is indeed a false positive, then the IR is valid. The advantages of this approach are (i) it provides the ability to analyze some aspects of programs that may deviate from the desired execution model; (ii) it generates reports of possible deviations from the desired execution model; (iii) it does not force the analyzer to explore all of the consequences of each (apparent) deviation, which may be a false positive due to loss of precision that occurs during static analysis.

The contributions of our work can be summarized as follows:

—We devised an abstract memory model that is suitable for analyzing executables (see §2).

—We developed (several variations of) a static-analysis algorithm that—without relying on symbol-table or debugging information—is able to track the flow of values through memory (see §3).

—We devised an algorithm to recover variable-like entities from an executable that can serve as proxies for the missing source-level variables in algorithms for further analysis of executables (see §4 and §5). The algorithm addresses the problem of recovering such entities regardless of whether they are local, global, or allocated in the heap.

—We used these methods to create the first program-slicing tool for executables that can help with understanding dependences across memory updates and memory accesses [Balakrishnan et al. 2005].

—We used these methods to create the first automatic program-verification tool for stripped executables: it allows one to check that a stripped executable conforms to an API-usage rule specified as a finite-state machine [Balakrishnan et al. 2005; Balakrishnan and Reps 2008] (see §6).

Legal Issues. Machine-code analysis has different usage scenarios depending, for instance, on whether

—the user has source code for the entire application, including the libraries and OS utilities that it uses

—the user has source code for the application, but only machine code for the libraries and OS utilities

—the user has only machine code

In most parts of the world, the latter two situations may be subject to legal restrictions. Most end-user license agreements (EULAs) contain provisions that prohibit disassembly, decompilation, and reverse engineering of the licensed program. Wikipedia's page on "Software License Agreements" [Wikipedia: Shrink-Wrap and Click-Wrap Licenses] notes, "Whether shrink-wrap licenses are legally binding [in the United States] differs between jurisdictions, though a majority of jurisdictions hold such licenses to be enforceable." It adds,

> ... publishers have begun to encrypt their software packages to make it impossible for a user to install the software without either agreeing to the license agreement or violating the Digital Millennium Copyright Act (DMCA) and foreign counterparts. [Wikipedia: Enforceability]

In the United States, the DMCA prohibits the circumvention of access-control technologies [DMCA §1201]. However, there are several statutory exceptions for law enforcement, intelligence, and other government activities (§1201(e)), reverse engineering/interoperability (§1201(f)), encryption research (§1201(g)), and security testing (§1201(j)). In addition to the statuatory exceptions, other exemptions can be granted by the Librarian of Congress. As to whether EULA clauses that prohibit reverse engineering for interoperability purposes are enforceable,

> The 8th Circuit case of Blizzard v. BnetD determined that such clauses are enforceable, following the Federal Circuit decision of Baystate v. Bowers. [Wikipedia: Shrink-Wrap and Click-Wrap Licenses]

To us, however, the boundary between permitted activities and excluded activities is not entirely clear. The point of a EULA is to allow users to execute the application's machine code. These days many applications are run in non-standard ways—e.g., in a guest virtual machine, using runtime optimization [Bala et al. 2000], or under program shepherding [Kiriansky et al. 2002]. Static analysis of machine code is yet another non-standard form of execution—one based on abstract interpretation [Cousot and Cousot 1977]. In general, as in ordinary execution of machine code, abstract execution of machine code involves repeatedly decoding an instruction and performing a state change according to the instruction's semantics. The only difference from standard execution is that a non-standard semantics is used: in essence, instead of running the program on single concrete states, the program is "run in the aggregate"; i.e., it is executed over descriptors that represent *collections* of states.

Organization and Roadmap to the Paper. The remainder of the paper is organized as follows: §2 presents the abstract memory model used in CodeSurfer/x86, and an algorithm to recover variable-like entities, referred to as a-locs (for abstract locations), from an executable. §3 presents an abstract-interpretation-based algorithm, referred to as value-set analysis (VSA), to recover information about the contents of machine registers and memory locations at every program point in an executable. §4 presents an improved a-loc recovery algorithm. §5 describes how the

various algorithms used in CodeSurfer/x86 interact with each other. §6 presents Device-Driver Analyzer for x86 (DDA/x86)—a tool built on top of CodeSurfer/x86 to analyze device-driver executables for bugs—and presents a case study in which DDA/x86 was used to find bugs in Windows device drivers. §7 discusses related work. §8 presents our conclusions and directions for further work.

Even though this is a lengthy paper, it was necessary to be selective about the material presented, or the paper would have been considerably longer. The paper concentrates on the essential core of our work; in a few places we refer the reader to other papers, either for additional details (see footnote 4, §3.1, and §3.4) or for information about variations and enhancements of the techniques presented in this paper (see the list at the beginning of §7).

For readers more familiar with source-code analysis, we have tried to make the paper as accessible as possible. Such readers may wish to keep two counterbalancing themes in mind (concentrating on whichever is of greater interest to them):

—To a considerable degree, we were able to make our analysis problems closely resemble standard source-code-analysis problems. To a large extent, the algorithms we describe are adaptations and variations on well-known techniques. This theme runs through §2, §3, and §6, which provide a flavor of the algorithms used in CodeSurfer/x86 and how their capabilities compare with source-code analyses.

—At the same time, considerable work was necessary to map ideas from source-code analysis over to machine-code analysis, due to several reasons:
  —We work with stripped executables, so our analyses start with no information about the program's variables.
  —Even control-flow information presents difficulties: (i) branch conditions are implicit because in x86 separate instructions are used for setting flags based on some condition, and a subsequent conditional-jump instruction performs the branch according to flag values; and (ii) it is often difficult to identify the targets of indirect jumps and indirect function calls.
  —As discussed in §6.2, when creating finite-state machines for property checking, the vocabulary of events in executables differs from the vocabulary of events in source code.

  The theme of how our analyzer can bootstrap itself from preliminary IRs that record fairly basic information about the code of a stripped executable to IRs on which it is possible to run analyses that resemble standard source-code analyses is the subject of §2.2, §4, and §5.

In source-code analysis, abstraction refinement [Kurshan 1994; Clarke et al. 2000] is a well-known technique for enhancing precision. §5 describes our abstraction-refinement loop, which not only improves precision but also orchestrates the analysis phases that allow us to overcome the lack of any initial information about a program's variables.

2. AN ABSTRACT MEMORY MODEL

One of the major stumbling blocks in analyzing executables is the difficulty of recovering information about variables and types, especially for aggregates (i.e., structures and arrays). 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* × *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 the analysis of executables is to identify variable-like entities.

If debugging information is available (and trusted), this provides one possibility; however, even if debugging information is available, analysis techniques have to account for bit-level, byte-level, word-level, and bulk-memory manipulations performed by programmers (or introduced by the compiler) that can sometimes violate variable boundaries [Backes 2004; Miné 2006; Reps et al. 2006]. If a program is suspected of containing malicious code, even if debugging information is present, it cannot be entirely relied upon. For these reasons, it is not always desirable to use debugging information—or at least to rely on it alone—for identifying a program's data objects. (Similarly, past work on source-code analysis has shown that it is sometimes valuable to ignore information available in declarations and infer replacement information from the actual usage patterns found in the code [Eidorff et al. 1999; O'Callahan and Jackson 1997; Ramalingam et al. 1999; Siff and Reps 1996; van Deursen and Moonen 1998].)

Example 2.1. The two programs shown in Fig. 2 will be used in this section to illustrate the issues involved in recovering a suitable set of variable-like entities from a machine-code program. The C program shown in Fig. 2(a) initializes all elements of array `pts[5]` and returns `pts[0].y`. The `x`-members of each element are initialized with the value of the global variable `a` and the `y`-members are initialized with the value of global variable `b`. The initial values of the global variables `a` and `b` are 1 and 2, respectively.

Fig. 2(b) shows the corresponding x86 program (in Intel assembly-language syntax). By convention, `esp` is the stack pointer in the x86 architecture. Instruction `1` allocates space for the locals of `main` on the stack. Fig. 3(a) shows how the variables are laid out in the activation record of `main`. Note that there is no space for variable `i` in the activation record because the compiler promoted `i` to register `edx`. Similarly, there is no space for pointer `p` because the compiler promoted it to register `eax`.

Instructions `L1` through `12` correspond to the `for`-loop in the C program. Instruction `L1` updates the `x`-members of the array elements, and instruction `8` updates the `y`-members. Instructions `13` and `14` correspond to initializing the return value for `main`. ■

```
typedef struct {
     int x,y;
} Point;

int a = 1, b = 2;

int main(){
     int i, *py;
     Point pts[5], *p;
     py = &pts[0].y;
     p  = &pts[0];
     for(i = 0; i < 5; ++i) {
         p->x = a;
         p->y = b;
         p += 8;
     }
     return *py;
}
```

(a)

```
      proc main         ;
   1  sub esp, 44       ;Allocate locals
   2  lea eax, [esp+8]  ;t1 = &pts[0].y
   3  mov [esp+0], eax  ;py = t1
   4  mov ebx, [4]      ;ebx = a
   5  mov ecx, [8]      ;ecx = b
   6  mov edx, 0        ;i = 0
   7  lea eax,[esp+4]   ;p = &pts[0]
 L1:  mov [eax], ebx    ;p->x = a
   8  mov [eax+4],ecx   ;p->y = b
   9  add eax, 8        ;p += 8
  10  inc edx           ;i++
  11  cmp edx, 5        ;
  12  jl L1             ;(i < 5)?L1:exit loop
  13  mov edi, [esp+0]  ;t2 = py
  14  mov eax, [edi]    ;set return value (*t2)
  15  add esp, 44       ;Deallocate locals
  16  retn              ;
```

(b)

Fig. 2. (a) A C program that initializes an array of structs; (b) the corresponding x86 program (in Intel assembly-language syntax).

2.1 Memory-Regions and Abstract Addresses

This section presents the basic abstract memory model that is used in CodeSurfer/x86's analyses. One simple model considers memory to be an array of bytes. Writes (reads) in this model are treated as writes (reads) to the corresponding element of the array. However, there are some disadvantages in such an approach:

—It may not be possible to determine specific address values for certain memory blocks, such as those allocated from the heap via `malloc`. For the analysis to be sound, writes to (reads from) such blocks of memory have to be treated as writes to (reads from) any part of the heap, which leads to imprecise (and mostly useless) information about memory accesses.

—The runtime stack is reused during each execution run; in general, a given area of the runtime stack will be used by several procedures at different times during execution. Thus, at each instruction a specific numeric address can be ambiguous (because the same address may belong to different Activation Records (ARs) at different times during execution): it may denote a variable of procedure `f`, a variable of procedure `g`, a variable of procedure `h`, etc. (A given address may also correspond to different variables of different activations of `f`.) Therefore, an instruction that updates a variable of procedure `f` would have to be treated as possibly updating the corresponding variables of procedures `g`, `h`, etc., which also leads to imprecise information about memory accesses.

To overcome these problems, we work with the following abstract memory model [Balakrishnan and Reps 2004]. Although in the concrete semantics the activation records for procedures, the heap, and the memory area 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* (see Fig. 3(b)). Each memory-region represents a group of locations that have similar runtime properties: in particular, the runtime locations that belong to the ARs of a given procedure belong to one memory-region. Each (abstract) byte in a memory-region represents

a set of concrete memory locations. For a given program, there are three kinds of regions: (1) the *global*-region, for memory locations that hold initialized and uninitialized global data, (2) *AR*-regions, each of which contains the locations of the ARs of a particular procedure, and (3) *malloc*-regions, each of which contains the locations allocated at a particular `malloc` site. We do not assume anything about the relative positions of these memory-regions.

For an n-bit architecture, the size of each memory-region in the abstract memory model is 2^n. For each region, the range of offsets within the memory-region is $[-2^{n-1}, 2^{n-1}-1]$. Offset 0 in an AR-region represents all concrete starting addresses of the ARs that the AR-region represents. Offset 0 in a malloc-region represents all concrete starting addresses of the heap blocks that the malloc-region represents. Offset 0 of the global-region represents the concrete address 0.

The analysis treats all data objects, whether local, global, or in the heap, in a fashion similar to the way compilers arrange to access variables in local ARs, namely, via an offset. We adopt this notion as part of our abstract semantics: an abstract address in a memory-region is represented by a pair: (memory-region, offset).

By convention, `esp` is the stack pointer in the x86 architecture. On entry to a procedure `P`, `esp` points to the top of the stack, where the new activation record for `P` is created. Therefore, in our abstract memory model, `esp` holds abstract address `(AR_P, 0)` on entry to procedure `P`, where `AR_P` is the activation-record region associated with procedure `P`. Similarly, because `malloc` returns the starting address of an allocated block, the return value for `malloc` (if allocation is successful) is the abstract address (`Malloc_n`, 0), where `Malloc_n` is the memory-region associated with the n^{th} call-site on `malloc`.[4]

EXAMPLE 2.2. Fig. 3(c) shows the memory-regions for the program in Ex. 2.1. There is a single procedure, and hence two regions: one for global data and one for the AR of `main`. Furthermore, the abstract address of local variable `py` is the pair `(AR_main,-44)` because it is at offset -44 with respect to the AR's starting address. Similarly, the abstract address of global variable `b` is `(Global,8)`. ■

2.2 Abstract Locations (A-Locs)

As pointed out earlier, executables do not have intrinsic entities like source-code variables that can be used for analysis; therefore, the next step is to recover variable-like entities from the executable, which will serve as proxies for the program's actual variables (e.g., the variables declared in the source code from which the executable was created). We refer to such variable-like entities as *a-locs* (for "abstract locations").

Heretofore, the state of the art in recovering variable-like entities is represented by IDAPro [IDAPro], a commercial disassembly toolkit. IDAPro's algorithm is

[4]CodeSurfer/x86 actually uses an abstraction of heap-allocated storage, called the *recency abstraction*, that involves more than one memory-region per call-site on `malloc` [Balakrishnan and Reps 2006]. The recency abstraction overcomes some of the imprecision that arises 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 pointer field of a heap-allocated object and objects pointed-to by the pointer field.

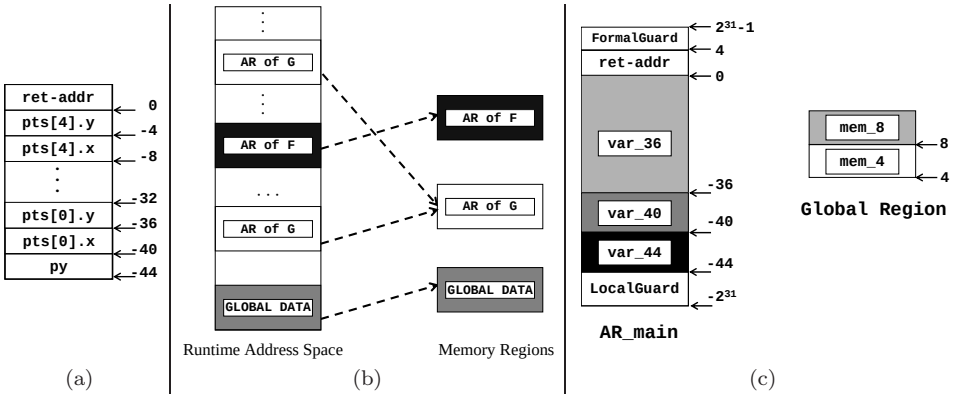


Fig. 3. (a) Layout of the activation record for procedure `main` in Ex. 2.1; (b) abstract memory model; (c) a-locs identified by the Semi-Naïve algorithm.

based on the observation that the data layout generally follows certain conventions: accesses to global variables appear as "[*absolute-address*]", and accesses to local variables appear as "[`esp` + *offset*]" or "[`ebp` − *offset*]". IDAPro identifies such statically-known absolute addresses, `esp`-based offsets, and `ebp`-based offsets in the program, and treats the set of locations in between two such absolute addresses or offsets as one entity. We refer to this method of recovering a-locs as the *Semi-Naïve algorithm*. The Semi-Naïve algorithm is based on purely local techniques. (IDAPro does incorporate a few global analyses, such as one for determining changes in stack height at call-sites. However, the techniques are ad-hoc, heuristic methods.)

In CodeSurfer/x86, the Semi-Naïve algorithm is used to identify the *initial* set of a-locs; several global analyses based on abstract interpretation are then used to obtain an improved set of a-locs. The latter methods are discussed in §4.

Let us look at the a-locs identified by the Semi-Naïve algorithm for the program in Ex. 2.1.

Global A-Locs. In Ex. 2.1, instructions "`mov ebx, [4]`" and "`mov ecx,[8]`" have direct memory operands, namely, `[4]` and `[8]`. IDAPro identifies these statically-known absolute addresses as the starting addresses of global a-locs and treats the locations between these addresses as one a-loc. Consequently, IDAPro identifies addresses 4..7 as one a-loc, and the addresses 8..11 as another a-loc. Therefore, we have two a-locs: `mem_4` (for addresses 4..7) and `mem_8` (for addresses 8..11). (An executable can have sections for read-only data. The global a-locs in such sections are marked as read-only a-locs.)

Local A-Locs. Local a-locs are determined on a per-procedure basis as follows. At each instruction in the procedure, IDAPro computes the difference between the value of `esp` (or `ebp`) at that point and the value of `esp` at procedure entry. These computed differences are referred to as `sp_delta`.[5] After computing `sp_delta` values, IDAPro identifies all `esp`-based indirect operands in the proce-

[5]When IDAPro computes the `sp_delta` values, it uses heuristics to identify changes to `esp` (or `ebp`) at procedure calls and instructions that access memory, and therefore the `sp_delta` values may be incorrect. Consequently, the layout obtained by IDAPro for an AR may not be in agreement with

dure. In Ex. 2.1, instructions "`lea eax, [esp+8]`", "`mov [esp+0], eax`", "`lea eax, [esp+4]`", and "`mov edi, [esp+0]`" have `esp`-based indirect operands. Recall that on entry to procedure `main`, `esp` contains the abstract address `(AR_main, 0)`. Therefore, for every `esp`/`ebp`-based operand, the computed `sp_delta` values give the corresponding offset in `AR_main`. For instance, `[esp+0]`, `[esp+4]`, and `[esp+8]` refer to offsets `-44`, `-40`, and `-36`, respectively, in `AR_main`. This gives rise to three local a-locs: `var_44`, `var_40`, and `var_36`. Note that `var_44` corresponds to all of the source-code variable `py`. In contrast, `var_40` and `var_36` correspond to disjoint segments of array `pts[]`: `var_40` corresponds to program variable `pts[0].x`; `var_36` corresponds to the locations of program variables `pts[0].y`, `p[1..4].x`, and `p[1..4].y`. In addition to these a-locs, an a-loc for the return address is also defined; its offset in `AR_main` is `0`.

In addition to the a-locs identified by IDAPro, two more a-locs are added: (1) a `FormalGuard` that spans the space beyond the topmost a-loc in the AR-region, and (2) a `LocalGuard` that spans the space below the bottom-most a-loc in the AR-region. `FormalGuard` and `LocalGuard` delimit the boundaries of an activation record; therefore, a memory write to `FormalGuard` or `LocalGuard` represents a write beyond the end of an activation record.

Heap A-Locs. In addition to globals and locals, we have one a-loc per heap-region. There are no heap a-locs in Ex. 2.1 because it does not use the heap.

Registers. In addition to the global, heap, and local a-locs, registers are also considered to be a-locs.

After the a-locs are identified, we create a mapping from a-locs to (*rgn*, *off*, *size*) triples, where *rgn* represents the memory-region to which the a-loc belongs, *off* is the starting offset of the a-loc in *rgn*, and *size* is the size of the a-loc. The starting offset of an a-loc a in a region *rgn* is denoted by `offset`(*rgn*, a). For Ex. 2.1, `offset(AR_main,var_40)` is `-40` and `offset(Global, mem_4)` is 4. The a-loc layout map can also be queried in the opposite direction: for a given region, offset, and size, what are the overlapping a-locs? As described in §3.4, such information is used to interpret memory-dereferencing operations during VSA.

3. VALUE-SET ANALYSIS (VSA)

Another significant obstacle in analyzing executables is that it is difficult to obtain useful information about memory-access expressions in the executable. Information about memory-access expressions is a crucial requirement for any tool that works on executables. Consider the problem of identifying possible data dependences between instructions in executables. An instruction i_1 is data dependent on another instruction i_2 if i_1 might read the data that i_2 writes. For instance, in Ex. 2.1, instruction `14` is data dependent on instruction `8` because instruction `8` writes to `pts[0].y` and instruction `14` reads from `pts[0].y`. On the other hand, instruction `14` is *not* data dependent on instruction `L1`.

---

the way that memory is actually accessed during execution runs. This can have an impact on the *precision* of the results obtained by our abstract-interpretation algorithms; however, as discussed in §4.4, the results obtained by the algorithms are still *sound*, even if the initial set of a-locs is suboptimal because of incorrect `sp_delta` values.

There has been work in the past on analysis techniques to obtain such information. However, prior techniques are either overly-conservative or unsound in their treatment of memory accesses. The alias-analysis algorithm proposed by Debray et al. [1998] assumes that any memory write can affect any other memory read. Therefore, their algorithm reports that instruction `14` is data dependent on both `L1` and `8`—i.e., it provides an overly-conservative treatment of memory operations. On the other hand, Cifuentes and Fraboulet [1997b] use heuristics to determine if two memory operands are aliases of one another, and hence may fail to identify the data dependence between instruction `8` and instruction `14`.

To obtain information about memory-access expressions, CodeSurfer/x86 makes use of a number of analyses, and the sequence of analyses performed is itself iterated (for reasons discussed in §5). The variable and type-discovery phase of CodeSurfer/x86 recovers information about variables that are allocated globally, locally (i.e., on the stack), and dynamically (i.e., from the freelist); see §2.2 and §4. The recovered variables (a-locs) are the basic variables used in CodeSurfer/x86's value-set-analysis (VSA) algorithm, which statically identifies the set of values that the a-locs may contain when an instruction I executes. This section describes the VSA algorithm.

VSA is a combined numeric-analysis and pointer-analysis algorithm that determines a safe approximation of the set of numeric values or addresses that each register and a-loc holds at each program point. In particular, at each instruction I that contains an indirect memory operand, VSA provides information about the contents of the registers that are used. This permits it to determine the (abstract) addresses that are potentially accessed—and hence the a-locs that are potentially accessed—which, in turn, permits it to determine the potential effects of I on the state.

The problem that VSA addresses has similarities with the *pointer-analysis* problem that has been studied in great detail for programs written in high-level languages [Hind 2001]. For each variable (say `v`), pointer analysis determines an over-approximation of the set of variables whose addresses `v` can hold. Similarly, VSA determines an over-approximation of the set of addresses that a register or a memory location holds at each program point. For instance, VSA determines that at instruction `L1` in Ex. 2.1 `eax` holds one of the offsets $\{-40, -32, -24, \ldots, -8\}$ in the activation record of procedure `main`, which corresponds to the addresses of field `x` of the elements of array `pts[0..4]`.

On the other hand, VSA also has some of the flavor of *numeric static analyses*, where the goal is to over-approximate the integer values that each variable can hold; in addition to information about addresses, VSA determines an over-approximation of the set of integer values that each data object can hold at each program point. For instance, VSA determines that at instruction `L1`, `edx` holds numeric values in the range $0, \ldots, 4$.

A key feature of VSA is that it *tracks integer-valued and address-valued quantities simultaneously*. This is crucial for analyzing executables because numeric operations and address-dereference operations are inextricably intertwined even in the instruction(s) generated for simple source-code operations. For instance, consider the operation of loading the value of a local variable `v` into register `eax`. If `v` has offset `-12` in the current AR, the load would be performed by the instruction

`mov eax, [ebp-12]`. This involves a *numeric* operation (`ebp-12`) to calculate an address whose value is then *dereferenced* (`[ebp-12]`) to fetch the value of `v`, after which the value is placed in `eax`. A second key feature of VSA is that, unlike earlier algorithms [Cifuentes and Fraboulet 1997a; 1997b; Cifuentes et al. 1998; Debray et al. 1998], it *takes into account data manipulations that involve memory locations.*

VSA is based on abstract interpretation [Cousot and Cousot 1977], where the aim is to determine the possible states that a program reaches during execution, but without actually running the program on specific inputs. The set of descriptors of memory configurations used in abstract interpretation is referred to as an *abstract domain*. An element of an abstract domain represents a set of concrete (i.e., run-time) states of a program. An element of the abstract domain for VSA associates each a-loc with a set of (abstract) memory addresses and numeric values.

VSA is a flow-sensitive, context-sensitive, interprocedural, abstract-interpretation algorithm (parameterized by call-string length [Sharir and Pnueli 1981]). In the rest of this section, we formalize the VSA domain and describe the VSA algorithm in detail.

3.1 Value-Sets

A value-set represents a set of memory addresses and numeric values. Recall from §2.1 that each abstract address is a pair (*memory-region*, *offset*). Therefore, a set of abstract addresses can be represented by a set of tuples of the form $(rgn_i \mapsto \{o_1^i, o_2^i, ..., o_{n_i}^i\})$. A value-set uses a k-bit strided-interval (SI) [Reps et al. 2006] to represent the set of offsets in each memory-region. Let γ denote the concretization function for the strided-interval domain; a k-bit strided interval $\mathbf{s}[\mathbf{l}, \mathbf{u}]$ represents the set of integers

$$\gamma(\mathbf{s}[\mathbf{l}, \mathbf{u}]) = \{i \in [-2^{k-1}, 2^{k-1} - 1] \mid l \leq i \leq u, i \equiv l (\texttt{mod } s)\}, \text{ where}$$

—$\mathbf{s}$ is called the *stride*.

—$[\mathbf{l}, \mathbf{u}]$ is called the *interval*.

—$\mathbf{0}[\mathbf{l}, \mathbf{l}]$ represents the singleton set $\{l\}$.

We also call $\perp$ a strided interval; it denotes the empty set of offsets (i.e., $\emptyset$).

Consider the set of addresses $S = \{(\texttt{Global} \mapsto \{1, 3, 5, 9\}), (\texttt{AR\_main} \mapsto \{-48, -40\})\}$. The value-set for S is the set $\{(\texttt{Global} \mapsto \mathbf{2}[\mathbf{1}, \mathbf{9}]), (\texttt{AR\_main} \mapsto \mathbf{8}[-\mathbf{48}, -\mathbf{40}])\}$. Note that the value-set for S is an over-approximation; the value-set includes the global address 7, which is not an element of S. For conciseness, a value-set will be shown as an r-tuple of SIs, where r is the number of memory-regions for the executable. By convention, the first component of the r-tuple represents addresses in the `Global` memory-region. Using this notation, the value-set for S is the 2-tuple, $(\mathbf{2}[\mathbf{1}, \mathbf{9}], \mathbf{8}[-\mathbf{48}, -\mathbf{40}])$.

A value-set is capable of representing a set of memory addresses as well as a set of numeric values (which is a crucial requirement for analyzing executables because numbers and addresses are indistinguishable at runtime). For instance, the 2-tuple $(\mathbf{2}[\mathbf{1}, \mathbf{9}], \perp)$ denotes the set of numeric values $\{1, 3, 5, 7, 9\}$ as well as the set of addresses $\{(\texttt{Global}, 1), (\texttt{Global}, 3), \ldots, (\texttt{Global}, 9)\}$; however, because the second component is $\perp$, it does not represent any addresses in memory-region `AR_main`. The 2-tuple $(\perp, \mathbf{8}[-\mathbf{48}, -\mathbf{40}])$ represents the set of addresses $\{(\texttt{AR\_main}, -48),$

(`AR_main`, −40)}; however, because the first component is ⊥, it does not represent any pure numeric values nor any addresses in the `Global` memory-region.

Advantages of Strided Intervals for Analysis of Executables. We chose to use SIs instead of ranges because alignment and stride information allow indirect-addressing operations that implement either (i) field-access operations in an array of structs, or (ii) pointer-dereferencing operations, to be interpreted more precisely.

Let `*a` denote a dereference of a-loc `a`. Suppose that the contents of `a` is not aligned with the boundaries of other a-locs; then a memory access `*a` can fetch portions of two or more a-locs. Similarly, an assignment to `*a` can overwrite portions of two or more a-locs. Such operations appear to forge new addresses. For instance, suppose that the address of a-loc `x` is `1000`, the address of a-loc `y` is `1004`, and the contents of a-loc `a` is `1001`. Then `*a` (as a 4-byte fetch) would retrieve 3 bytes of `x`'s value and 1 byte of `y`'s value.

This issue motivated the use of SIs because SIs are capable of representing certain non-convex sets of integers, and ranges (alone) are not. Suppose that the contents of `a` is the set {`1000`, `1004`}; then `*a` (as a 4-byte fetch) would retrieve all of `x` (and none of `y`) or all of `y` (and none of `x`). The range [1000, 1004] includes the addresses `1001`, `1002`, and `1003`, and hence *[1000, 1004] (as a 4-byte fetch) could result in a forged address. However, because VSA is based on SIs, {`1000`, `1004`} is represented exactly, as the SI 4[1000, 1004]. If VSA were based on range information rather than SIs, it would either have to try to track *segments* of (possible) contents of data objects, or treat such dereferences conservatively by returning $\top^{vs}$, thereby losing track of all information.

The Value-Set Abstract Domain. Value-sets form a lattice. Informal descriptions of a few 32-bit value-set operators are given below. (For a detailed description of the value-set domain, see Reps et al. [2006].)

—$(vs_1 \sqsubseteq^{\text{vs}} vs_2)$: Returns true if the value-set vs_1 is a subset of vs_2, false otherwise.

—$(vs_1 \sqcap^{\text{vs}} vs_2)$: Returns the meet (intersection) of value-sets vs_1 and vs_2.

—$(vs_1 \sqcup^{\text{vs}} vs_2)$: Returns the join (union) of value-sets vs_1 and vs_2.

—$(vs_1 \nabla^{\text{vs}} vs_2)$: Returns the value-set obtained by widening [Cousot and Cousot 1976] vs_1 with respect to vs_2, e.g., if $vs_1 = (\mathbf{4}[\mathbf{40}, \mathbf{44}])$ and $vs_2 = (\mathbf{4}[\mathbf{40}, \mathbf{48}])$, then $(vs_1 \nabla^{\text{vs}} vs_2) = (\mathbf{4}[\mathbf{40}, \mathbf{2^{31}} - \mathbf{4}])$. Note that the upper bound for the interval in the result is $\mathbf{2^{31}} - \mathbf{4}$ (and not $\mathbf{2^{31}} - \mathbf{1}$) because $\mathbf{2^{31}} - \mathbf{4}$ is the maximum positive value that is congruent to **40** modulo **4**.

—$(vs +^{\text{vs}} c)$: Returns the value-set obtained by adjusting all values in vs by the constant c, e.g., if $vs = (\mathbf{4}, \mathbf{4}[\mathbf{4}, \mathbf{12}])$ and $c = 12$, then $(vs +^{\text{vs}} c) = (\mathbf{16}, \mathbf{4}[\mathbf{16}, \mathbf{24}])$.

—$*(vs, s)$: Returns a pair of sets (F, P). F represents the set of "fully accessed" a-locs: it consists of the a-locs that are of size s and whose starting addresses are in vs. P represents the set of "partially accessed" a-locs: it consists of (i) a-locs whose starting addresses are in vs but are not of size s, and (ii) a-locs whose addresses are in vs but whose starting addresses and sizes do not meet the conditions to be in F. (This information is obtained using the a-loc layout map described in §2.2.)

—`RemoveLowerBounds(`*vs*`)`: Returns the value-set obtained by setting the lower

bound of each component SI to -2^{31}. For example, if $vs = (\mathbf{1}[\mathbf{0}, \mathbf{100}], \mathbf{1}[\mathbf{100}, \mathbf{200}])$, then `RemoveLowerBounds(`$vs$`)`$= (\mathbf{1}[-\mathbf{2^{31}}, \mathbf{100}], \mathbf{1}[-\mathbf{2^{31}}, \mathbf{200}])$.

—`RemoveUpperBounds(`vs`)`: Similar to `RemoveLowerBounds`, but sets the upper bound of each component to $2^{31} - 1$.

3.2 Abstract Environment (AbsEnv)

AbsEnv (for "abstract environment") is the abstract domain used during VSA to represent a set of concrete stores that arise at a given program point. This section formalizes AbsEnv.

Let Proc denote the set of memory-regions associated with procedures in the program; AllocMemRgn denote the set of memory-regions associated with heap-allocation sites; Global denote the memory-region associated with the global data area; and a-locs[R] denote the a-locs that belong to memory-region `R`. We work with the following basic domains:

$$\begin{aligned} \mathsf{MemRgn} &= \{\mathsf{Global}\} \cup \mathsf{Proc} \cup \mathsf{AllocMemRgn} \\ \mathsf{ValueSet} &= \mathsf{MemRgn} \rightarrow \mathsf{StridedInterval}_{\perp} \\ \mathsf{AlocEnv[R]} &= \mathsf{a\text{-}locs[R]} \rightarrow \mathsf{ValueSet} \\ \mathsf{Flag} &= \{\mathrm{CF}, \mathrm{ZF}, \mathrm{SF}, \mathrm{PF}, \mathrm{AF}, \mathrm{OF}\} \end{aligned}$$

Flag represents the set of x86 flags. An x86 flag is either set to True or False at runtime. To represent multiple possible Boolean values, we use the abstract domain Bool3:

$$\mathsf{Bool3} = \{\textsc{False}, \textsc{Maybe}, \textsc{True}\}.$$

In addition to the Booleans False and True, Bool3 has a third value, Maybe, which means "the value is unknown" (i.e., it may be False or it may be True). AbsEnv maps each region `R` to its corresponding AlocEnv[R], each register to a ValueSet, and each Flag to a Bool3:

$$\begin{aligned} \mathsf{AbsEnv} = \; & (\mathsf{register} \rightarrow \mathsf{ValueSet}) \\ & \times (\mathsf{Flag} \rightarrow \mathsf{Bool3}) \\ & \times (\{\mathsf{Global}\} \rightarrow \mathsf{AlocEnv[Global]}) \\ & \times (\mathsf{Proc} \rightarrow \mathsf{AlocEnv[Proc]}_{\perp}) \\ & \times (\mathsf{AllocMemRgn} \rightarrow \mathsf{AlocEnv[AllocMemRgn]}_{\perp}) \end{aligned}$$

In the above definitions, $\perp$ is used to denote a partial map. For instance, a ValueSet may not contain offsets in some memory-regions. Similarly, in AbsEnv, a procedure `P` whose activation record is not on the stack is mapped to $\perp$ rather than to a value in AlocEnv[`P`].

We use the following notational conventions:

—Given a memory a-loc or a register a-loc a and $ae \in \mathsf{AbsEnv}$, $ae[a]$ refers to the ValueSet for a-loc a in ae.

—Given $vs \in \mathsf{ValueSet}$ and $r \in \mathsf{MemRgn}$, $vs[r]$ refers to the strided interval for memory-region r in vs.

—Given $f \in \mathsf{Flag}$ and $ae \in \mathsf{AbsEnv}$, $ae[f]$ refers to the Bool3 for flag f in ae.

3.3 Representing Abstract Stores Efficiently

To represent the abstract store at each program point efficiently, we use applicative dictionaries, which provide a space-efficient representation of a collection of dictionary values when many of the dictionary values have nearly the same contents as other dictionary values in the collection [Reps et al. 1983; Myers 1984].

Applicative dictionaries can be implemented using applicative balanced trees, which are standard balanced trees on which all operations are carried out in the usual fashion, except that whenever one of the fields of an interior node M would normally be changed, a new node M' is created that duplicates M, and changes are made to the fields of M'. To be able to treat M' as the child of parent(M), it is necessary to change the appropriate child-field in parent(M), so a new node is created that duplicates parent(M), and so on, all the way to the root of the tree. Thus, new nodes are introduced for each of the original nodes along the path from M to the root of the tree.

Because an operation that restructures a standard balanced tree may modify all of the nodes on the path to the root anyway, and because a single operation on a standard balanced tree that has n nodes takes at most $O(\log n)$ steps, the same operation on an applicative balanced tree introduces at most $O(\log n)$ additional nodes and also takes at most $O(\log n)$ steps. The new tree resulting from the operation shares the entire structure of the original tree except for the nodes on a path from M' to the root, plus at most $O(\log n)$ other nodes that may be introduced to maintain the balance properties of the tree. In our implementation, the abstract stores from the VSA domain are implemented using applicative AVL trees [Myers 1984]. That is, each function or partial function in a component of AbsEnv is implemented with an applicative AVL tree.

3.4 Intraprocedural Value-Set Analysis

This subsection describes an intraprocedural version of VSA. For the time being, we consider programs that have a single procedure and no indirect jumps. To aid in explaining the algorithm, we adopt a C-like notation for program statements. We will discuss the following kinds of instructions, where `R1` and `R2` are two registers of the same size, c, c_1, and c_2 are explicit integer constants, and $\leq$ and $\geq$ represent signed comparisons:

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

Conditions of the two forms shown on the right are obtained from the instruction(s) that set condition codes used by branch instructions (see §3.4.2).

The analysis is performed on a control-flow graph (CFG) for the procedure. The CFG consists of one node per x86 instruction, and there is a directed edge $n_1 \rightarrow n_2$ between a pair of nodes $n1$ and $n2$ in the CFG if there is a flow of control from $n1$ to $n2$. The edges 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 labeled according to the outcome of the branch. For instance in the CFG for the program in Ex. 2.1, the edge `12`→`L1` is labeled `edx`<`5`, whereas the edge `12`→`13` is labeled `edx`≥`5`. Each CFG has two special nodes: (1) an `enter` node that represents the entry point of

the procedure, (2) an `exit` node that represents the exit point of the procedure.

Instruction	*AbstractTransformer*(*in*: AbsEnv): AbsEnv
$\texttt{R1} = \texttt{R2} + c$	Let $out := in$ and $vs_{\texttt{R2}} := in[\texttt{R2}]$ $out[\texttt{R1}] := vs_{\texttt{R2}} +^{\text{vs}} c$ **return** *out*

$*(\texttt{R1} + c_1) = \texttt{R2} + c_2$	Let $vs_{\texttt{R1}} := in[\texttt{R1}]$, $vs_{\texttt{R2}} := in[\texttt{R2}]$, $(F, P) = *(vs_{\texttt{R1}} +^{\text{vs}} c_1, s)$, and $out := in$
	Let *Proc* be the procedure containing the instruction
	if $(\lvert F\rvert = 1 \wedge \lvert P\rvert = 0 \wedge$ (F has no heap a-locs or a-locs of recursive procedures))
	then
	  $out[v] := vs_{\texttt{R2}} +^{\text{vs}} c_2$, where $v \in F$ // Strong update
	else
	  for each $v \in F$ **do**
	  $out[v] := out[v] \sqcup^{\text{vs}} (vs_{\texttt{R2}} +^{\text{vs}} c_2)$ // Weak update
	  end for
	end if
	for each $v \in P$ **do** // Set partially accessed a-locs to $\top^{vs}$
	  $out[v] := \top^{vs}$
	end for
	return *out*

$\texttt{R1} = *(\texttt{R2} + c_1) + c_2$	Let $vs_{\texttt{R2}} := in[\texttt{R2}]$, $(F, P) = *(vs_{\texttt{R2}} +^{\text{vs}} c_1, s)$ and $out := in$
	if $(\lvert P\rvert = 0)$ **then**
	  Let $vs_{rhs} := \bigsqcup^{\text{vs}}\{in[v] \mid v \in F\}$
	  $out[\texttt{R1}] := vs_{rhs} +^{\text{vs}} c_2$
	else
	  $out[\texttt{R1}] := \top^{vs}$
	end if
	return *out*

$\texttt{R1} \leq c$	Let $vs_c := ([-2^{31}, c], \top^{si}, \ldots, \top^{si})$ and $out := in$ $out[\texttt{R1}] := in[\texttt{R1}] \sqcap^{\text{vs}} vs_c$ **return** *out*
$\texttt{R1} \geq \texttt{R2}$	Let $vs_{\texttt{R1}} := in[\texttt{R1}]$ and $vs_{\texttt{R2}} := in[\texttt{R2}]$ Let $vs_{lb} :=$ `RemoveUpperBounds`$(vs_{\texttt{R2}})$ and $vs_{ub} :=$ `RemoveLowerBounds`$(vs_{\texttt{R1}})$ $out := in$ $out[R1] := vs_{R1} \sqcap^{\text{vs}} vs_{lb}$ $out[R2] := vs_{R2} \sqcap^{\text{vs}} vs_{ub}$ **return** *out*

Fig. 4. Abstract transformers for VSA. (In the second and third instruction forms, s represents the size of the dereference performed by the instruction.)

Each edge in the CFG is associated with an abstract transformer that captures the semantics of the instruction represented by the CFG edge. Each abstract transformer takes an $in \in$ AbsEnv and returns a new $out \in$ AbsEnv. Sample abstract transformers for various kinds of edges are listed in Fig. 4. Interesting cases in Fig. 4 are described below:

—Because each AR region of a procedure that may be called recursively—as well as each heap region—potentially represents more than one concrete data object, assignments to their a-locs must be modeled by weak updates, i.e., the new value-set must be joined with the existing one, rather than replacing it (see case two of Fig. 4).

—Furthermore, unaligned writes can modify parts of various a-locs (which could possibly create forged addresses). In case 2 of Fig. 4, such writes are treated

```
 1: decl worklist: Set of Node
 2:
 3: proc IntraProceduralVSA()
 4:    worklist := {enter}
 5:    absEnv_enter := Initial values of global a-locs and esp
 6:    while (worklist ≠ ∅) do
 7:       Select and remove a node n from worklist
 8:       m := Number of successors of node n
 9:       for i = 1 to m do
10:          succ := GetCFGSuccessor(n, i)
11:          edge_amc := AbstractTransformer(n → succ, absEnv_n)
12:          Propagate(succ, edge_amc)
13:       end for
14:    end while
15: end proc
16:
17: proc Propagate(n: Node, edge_amc: AbsEnv)
18:    old := absEnv_n
19:    new := old ⊔^ae edge_amc
20:    if (old ≠ new) then
21:       absEnv_n := new
22:       worklist := worklist ∪ {n}
23:    end if
24: end proc
```

Fig. 5. Intraprocedural VSA Algorithm.

safely by setting the values of all partially modified a-locs to $\top^{vs}$. Similarly, case 3 treats a load of a potentially forged address as a load of $\top^{vs}$. (Techniques for more precise handling of partial accesses to a-locs are discussed in §4.)

Given a CFG G for a procedure (without calls), the goal of intraprocedural VSA is to annotate each node n with $\text{absEnv}_n \in \mathsf{AbsEnv}$, where absEnv_n represents an over-approximation of the set of memory configurations that arise at node n over all possible runs of the program. The intraprocedural version of the VSA algorithm is given in Fig. 5. The value of $\text{absEnv}_{\texttt{enter}}$ consists of information about the initialized global variables and the initial value of the stack pointer (`esp`).

The AbsEnv abstract domain has very long ascending chains.[6] Hence, to ensure termination, widening needs to be performed. Widening needs to be carried out at at least one edge of every cycle in the CFG; however, the edge at which widening is performed can affect the accuracy of the analysis. To choose widening edges, our implementation of VSA uses techniques due to Bourdoncle [1993] (see Balakrishnan [2007, Ch. 7]).

Example 3.1. This example presents the results of intraprocedural VSA for the program in Ex. 2.1. For the program in Ex. 2.1, the AbsEnv for the entry node of `main` is $\{\texttt{esp} \mapsto (\bot, \mathbf{0}), \texttt{mem\_4} \mapsto (\mathbf{1}, \bot), \texttt{mem\_8} \mapsto (\mathbf{2}, \bot)\}$. Recall that instruction "`L1:mov [eax], ebx`" updates the `x` members of array `pts`. Instruction "`14: mov`

[6]The domain is of bounded height because strided intervals are based on 32-bit two's complement arithmetic. However, for a given executable, the bound is very large: each a-loc can have up to |MemRgn| SIs; hence the height is $(n \times |\mathsf{MemRgn}| \times 2^{32})$, where n is the total number of a-locs.

`eax, [edi]`" initializes the return value of `main` to `p[0].y`. The results of the VSA algorithm at instructions `L1`, `8`, and `14` are as follows:

Instruction `L1` and `8`		Instruction `14`	
`esp`	$\mapsto (\bot, -\mathbf{44})$	`esp`	$\mapsto (\bot, -\mathbf{44})$
`mem_4`	$\mapsto (\mathbf{1}, \bot)$	`mem_4`	$\mapsto (\mathbf{1}, \bot)$
`mem_8`	$\mapsto (\mathbf{2}, \bot)$	`mem_8`	$\mapsto (\mathbf{2}, \bot)$
`eax`	$\mapsto (\bot, \mathbf{8}[-\mathbf{40}, \mathbf{2^{31}} - \mathbf{8}])$	`eax`	$\mapsto (\bot, \mathbf{8}[-\mathbf{40}, \mathbf{2^{31}} - \mathbf{8}])$
`ebx`	$\mapsto (\mathbf{1}, \bot)$	`ebx`	$\mapsto (\mathbf{1}, \bot)$
`ecx`	$\mapsto (\mathbf{2}, \bot)$	`ecx`	$\mapsto (\mathbf{2}, \bot)$
`edx`	$\mapsto (\mathbf{1}[\mathbf{0}, \mathbf{4}], \bot)$	`edx`	$\mapsto (\mathbf{5}, \bot)$
`edi`	$\mapsto \top^{vs}$	`edi`	$\mapsto (\bot, -\mathbf{36})$
`var_44`	$\mapsto (\bot, -\mathbf{36})$	`var_44`	$\mapsto (\bot, -\mathbf{36})$

For instance, VSA recovers the following facts:

—At instruction `L1`, the set of possible values for `edx` is $\{0, 1, 2, 3, 4\}$. At instruction `14`, the only possible value for `edx` is 5. (Recall that `edx` corresponds to the loop variable `i` in the C program.)

—At instruction `L1`, `eax` holds the following set of addresses:

$$\{(\texttt{AR\_main}, -\mathbf{40}), (\texttt{AR\_main}, -\mathbf{32}), \ldots, (\texttt{AR\_main}, \mathbf{0}), \ldots, (\texttt{AR\_main}, \mathbf{2^{31}} - \mathbf{8})\}.$$

That is, at instruction `L1`, `eax` holds the addresses of the local a-locs `var_40`, `var_36`, `ret-addr`, and `FormalGuard`. (See Fig. 3(b) for the layout of `AR_main`.) Therefore, instruction `L1` possibly modifies `var_40`, `var_36`, `ret-addr`, and `FormalGuard`.

Similarly, at instruction `8`, `eax+4` refers to the following set of addresses:

$$\{(\texttt{AR\_main}, -\mathbf{36}), (\texttt{AR\_main}, -\mathbf{28}), \ldots, (\texttt{AR\_main}, \mathbf{4}), \ldots, (\texttt{AR\_main}, \mathbf{2^{31}} - \mathbf{4})\}.$$

Therefore, instruction `8` possibly modifies `var_36` and `FormalGuard`.

—At instruction `14`, the only possible value for `edi` is the address $(\texttt{AR\_main}, -\mathbf{36})$, which corresponds to the address of the local a-loc `var_36`.

The value-sets obtained by the analysis can be used to discover the data dependence that exists between instructions `8` and `14`. At instruction `8`, the set of possibly-modified a-locs is {`var_36`, `FormalGuard`}. At instruction `14`, the set of used a-locs is {`var_36`}. Reaching-definitions analysis based on this information reveals that instruction `14` is data dependent on instruction `8`.

Reaching-definitions analysis based on the information at instruction `L1` would also reveal that instruction `14` is also data dependent on instruction `L1`, which is spurious (i.e., a false positive), because the set of actual addresses accessed at instruction `L1` and instruction `14` are different. The reason for the spurious data dependence is that the Semi-Naïve algorithm, described in §2, recovers too coarse a set of a-locs. For instance, for the program in Ex. 2.1, the Semi-Naïve algorithm failed to recover any information about the array `pts`. §4 presents an improved a-loc-recovery algorithm that is capable of recovering information about arrays, fields of structs, etc., thereby reducing the number of spurious data dependences.

At instruction `L1`, the set of possibly-modified a-locs includes `ret-addr`, which is the a-loc for the return address. This is because the analysis was not able to determine a precise upper bound for `eax` at instruction `L1`, although register `edx` has a precise upper and lower bound at instruction `L1`. Note that, because `eax`

and edx are incremented in lock-step within the loop, the affine relation eax = (esp + edx × 8) + 4 holds at instruction L1. The implemented system identifies such affine relations and uses them to find precise upper or lower bounds for registers, such as eax, within a loop [Balakrishnan 2007, Ch. 7, Sect. 2].

■

3.4.1 *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 were to be performed directly. Some examples are given below.

XOR r1,r2, when r1 = r2 = r. The XOR instruction sets its first operand to the bitwise-exclusive-or (^) 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 r is set to the value-set $(\mathbf{0}[\mathbf{0},\mathbf{0}], \bot, \ldots)$.

TEST r1,r2, when r1 = r2 = r. 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 r has the form $(\mathbf{si}, \bot, \ldots)$:

$$\text{ZF} := \begin{cases} \textsc{True} & \text{if } \gamma(\mathbf{si}) = \{0\} \\ \textsc{False} & \text{if } \gamma(\mathbf{si}) \cap \{0\} = \emptyset \\ \textsc{Maybe} & \text{otherwise} \end{cases}$$

where 'γ' is the concretization function for the strided-interval domain (see §3.1).

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 $(\bot, \ldots, si_{k1}, si_{k2}, \ldots)$, and the value-set for b is $(\mathbf{0}[\mathbf{0},\mathbf{0}], \bot, \ldots)$. Then ZF is set to False.

3.4.2 *Predicates for Conditional Branch Instructions.* In x86 architectures, predicates used in high-level control constructs such as if, while, for, etc. are implemented using conditional branch instructions. A conditional branch instruction (say jxx TGT) evaluates a predicate involving the processor's flags and transfers control to the target instruction (TGT) if the predicate expression is True; otherwise, it transfers control to the next instruction. For instance, a jl instruction evaluates the conditional expression SF = 1, where SF is the sign flag. It is not clear from conditional expressions such as SF = 1 what the high-level predicate is.

To determine the high-level predicate, it is necessary to consider the instruction that sets the processor's flags before the conditional jump instruction is executed. In Ex. 2.1, i < 5 is compiled down to the x86 instruction sequence (cmp edx, 5; jl L1). The cmp operation sets the processor's flags to the result of computing the arithmetic expression edx − 5. Instruction "cmp edx, 5" sets SF to 1 iff (edx − 5 < 0), i.e., iff edx < 5. Because instruction jl is preceded by "cmp edx, 5" and jl transfers control to L1 iff SF = 1, we conclude that the instruction sequence (cmp edx, 5; jl L1) implements the high-level predicate edx < 5. High-level predicates

	cmp X, Y		sub X, Y		test X, Y	
	Flag Predicate	Predicate	Flag Predicate	Predicate	Flag Predicate	Predicate
	Unsigned Comparisons					
ja,jnbe	$\neg$CF $\wedge$ $\neg$ZF	X $>_u$ Y	$\neg$CF $\wedge$ $\neg$ZF	X$' \neq 0$	$\neg$ZF	X&Y $\neq 0$
jae,jnb,jnc	$\neg$CF	X $\geq_u$ Y	$\neg$CF	True	True	True
jb,jnae,jc	CF	X $<_u$ Y	CF	X$' \neq 0$	False	False
jbe,jna	CF $\vee$ ZF	X $\leq_u$ Y	CF $\vee$ ZF	True	ZF	X&Y $= 0$
je,jz	ZF	X $=$ Y	ZF	X$' = 0$	ZF	X&Y $= 0$
jne,jnz	$\neg$ZF	X $\neq$ Y	$\neg$ZF	X$' \neq 0$	$\neg$ZF	X&Y $\neq 0$
	Signed Comparisons					
jg,jnle	$\neg$ZF $\wedge$ (OF $\Leftrightarrow$ SF)	X $>$ Y	$\neg$ZF $\wedge$ (OF $\Leftrightarrow$ SF)	X$' > 0$	$\neg$ZF $\wedge$ $\neg$SF	(X&Y $\neq 0$) $\wedge$ (X $> 0 \vee$ Y > 0)
jge,jnl	OF $\Leftrightarrow$ SF	X $\geq$ Y	OF $\Leftrightarrow$ SF	True	$\neg$SF	(X $\geq 0 \vee$ Y ≥ 0)
jl,jnge	(OF $\oplus$ SF)	X $<$ Y	(OF $\oplus$ SF)	X$' < 0$	SF	(X $< 0 \wedge$ Y < 0)
jle,jng	ZF $\vee$ OF $\oplus$ SF	X $\leq$ Y	ZF $\vee$ (OF $\oplus$ SF)	True	ZF $\vee$ SF	(X&Y $= 0$) $\vee$ (X $< 0 \wedge$ Y < 0)
(Note: A $\oplus$ B $=$ ($\neg$A $\wedge$ B) $\vee$ (A $\wedge$ $\neg$B), & refers to the bitwise-and operation.)						

Fig. 6. High-level predicates for conditional jump instructions. (In column 5, X′ refers to the value of X after the instruction executes. Because test sets CF and OF to False, the flag predicates in column 6 have been simplified accordingly.)

for various instruction sequences involving conditional jump instructions are shown in Fig. 6.

3.5 Context-Insensitive Interprocedural Value-Set Analysis

Let us consider procedure calls, but ignore indirect jumps and indirect calls for now. The interprocedural algorithm is similar to the intraprocedural algorithm, but analyzes the supergraph of the executable.

Supergraph. In addition to the nodes used in an intraprocedural CFG, a supergraph has two nodes for every call-site: a call node and an end-call node. A supergraph for a program is obtained by first building CFGs for individual procedures and adding edges among call, end-call, enter, and exit nodes as follows:

—For every call-site `call P`, an edge is added from the CFG node for `call P` to the enter node of procedure `P`.

—For every procedure `P`, an edge is added from the exit node of `P` to the end-call node associated with every call to procedure `P`.

The call→enter and the exit→end-call edges are referred to as *linkage edges*. The abstract transformers for non-linkage edges in a supergraph are similar to the ones used in §3.4. The abstract transformers for the linkage edges are discussed in this section.

Example 3.2. We use the program shown in Fig. 8 to explain the interprocedural version of VSA. The program consists of two procedures, `main` and `initArray`. Procedure `main` has an array `pts` of `struct Point` objects, which is initialized by calling `initArray`. After initialization, `initArray` returns the value of `pts[0].y`.

The memory-regions and their layout are shown in Fig. 7. Note that all the local variables in `initArray` are mapped to registers in the disassembly: `i` is mapped to `edx`, `p` is mapped to `eax`, and `py` is mapped to `edi`. Therefore, `AR_initArray` only has the following three a-locs: the return address, formal parameter `arg_0`, and formal parameter `arg_4`. ■

Observation 3.3. *In our abstract memory model, we do not assume anything about the relative positions of the memory-regions. However, at a call, it is possible*

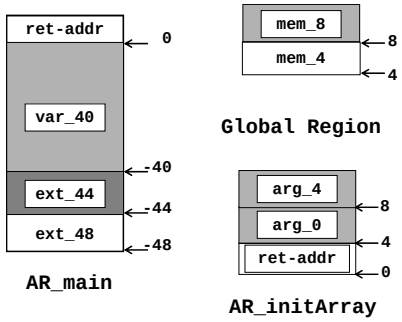


Fig. 7. Layout of the memory-regions for the program in Ex. 3.2. (`LocalGuard` and `FormalGuard` are not shown.)

```
typedef struct {
     int x,y;
} Point;

int a = 1, b = 2;

int initArray(
  struct Points pts[],
  int n) {
     int i, *py, *p;
     py = &pts[0].y;
     p  = &pts[0];
     for(i = 0; i < n; ++i) {
         p->x = a;
         p->y = b;
         p += 8;
     }
     return *py;
}

int main(){
     Point pts[5];
     return initArray(pts, 5);
}
```

(a)

```
      proc initArray    ;
  1   sub esp, 4        ;Allocate locals
  2   lea eax, [esp+16] ;t1 = &pts[0].y
  3   mov [esp+0], eax  ;py = t1
  4   mov ebx, [4]      ;ebx = a
  5   mov ecx, [8]      ;ecx = b
  6   mov edx, 0        ;i = 0
  7   lea eax, [esp+12] ;p = &pts[0]
 L1:  mov [eax], ebx    ;p->x = a
  8   mov [eax+4],ecx   ;p->y = b
  9   add eax, 8        ;p += 8
 10   inc edx           ;i++
 11   cmp edx,[esp+4]   ;
 12   jl L1             ;(i < n)?L1:exit loop
 13   mov edi, [esp+0]  ;t2 = py
 14   mov eax, [edi]    ;set return value (*t2)
 15   add esp, 12       ;Deallocate locals and
                        ;actuals
 16   retn              ;
                        ;
      proc main         ;
 17   sub esp, 40       ;Allocate locals
 18   push 5            ;2nd actual
 19   push esp          ;1st actual
 20   call initArray    ;
 21   add esp, 40       ;
 22   retn              ;
```

(b)

Fig. 8. (a) A C program that initializes an array of structs; (b) the corresponding x86 program.

to establish the relative positions of the caller's AR-region (`AR_C`) and the callee's AR-region (`AR_X`). Fig. 9 illustrates this idea. At runtime, `AR_C` and `AR_X` overlap on the stack just before a call is executed. Specifically, the abstract address $(\texttt{AR\_C}, -s)$ *in memory-region `AR_C` corresponds to the abstract address* $(\texttt{AR\_X}, \mathbf{4})$ *in memory-region `AR_X`. Therefore, the value of `esp` at a call refers to the abstract address* $(\texttt{AR\_C}, -s)$ *or* $(\texttt{AR\_X}, \mathbf{4})$*. This observation about the relative positions of `AR_C` and `AR_X` established at a call-site is used to develop the abstract transformers for the linkage edges.*

For instance, at instruction `20` in Ex. 3.2, $(\texttt{AR\_main}, -48)$ *corresponds to* $(\texttt{AR\_initArray}, 4)$*. Note that the observation about the relative positions of `AR_main` and `AR_initArray` at instruction `20` enables us to establish a correspondence be-*

tween the formal parameters `arg_0` *and* `arg_4` *of* `AR_initArray` *and the actual parameters* `ext_48` *and* `ext_44` *of* `AR_main`*, respectively. (See Fig. 7.) This correspondence between the actuals parameters of the caller and the formal parameters of the callee is used to initialize the formal parameters in the abstract transformer for a call→enter edge.*

■

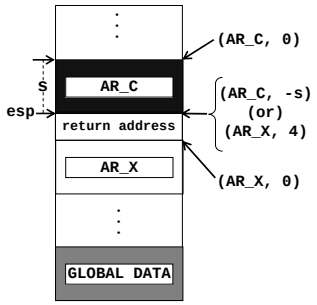


Fig. 9. Relative positions of the AR-regions of the caller (C) and callee (X) at a call.

3.5.1 *Abstract Transformer for a call→enter Edge.* The pseudo-code for the abstract transformer for a call→enter edge is shown in Fig. 10. Procedure *CallEnterTransformer* takes the current AbsEnv value at the call node as an argument and returns a new AbsEnv value for the call→enter edge. As a first step, the value-set of esp in the newly computed value is set to $(\bot, \ldots, \mathbf{0}, \ldots, \bot)$, where the **0** occurs in the slot for AR_X (line 4 in Fig. 10). This step corresponds to changing the current AR from that of AR_C to AR_X. After initializing esp, for every a-loc $a \in$ a-locs[AR_X], the corresponding set of a-locs in the AR_X is determined (line 8 of Fig. 10), and a new value-set for a (namely new_a) is computed (lines 6–15 of Fig. 10). (Note that line 8 of Fig. 10 is based on Obs. 3.3.) If procedure X is not recursive, the value-set for a in *out* is initialized to new_a (line 19 of Fig. 10). If procedure X is recursive, a weak update is performed (line 17 of Fig. 10). It is necessary to perform a weak update (rather than a strong update as at line 19 of Fig. 10) because the AR-region for a recursive procedure (say P) represents more than one concrete instance of P's activation record. Note that initialization of the a-locs of callee X (lines 5–20 of Fig. 10) has the effect of copying the actual parameters of caller C to the formal parameters of callee X.[7]

Example 3.4. In the fixpoint solution for the program in Ex. 3.2, the AbsEnv for the enter node of initArray is as follows:

[7]Note that when processing the other instructions of callee X that update the value of a formal parameter, we do not update the corresponding actual parameter of the caller, which is unsound. We do not update the value-set of the actual parameter simultaneously because we do not know relative positions of AR_C and AR_X at these instructions. The problem can be addressed by tracking the relative positions of the memory-regions at all instructions (and an experimental implementation that does so was carried out by J. Lim).

```
 1: proc CallEnterTransformer(in : AbsEnv): AbsEnv
 2:    Let C be the caller and X be the callee.
 3:    out := in
 4:    out[esp] := (⊥, ..., 0, ..., ⊥) // 0 occurs in the slot for AR_X
 5:    for each a-loc a ∈ a-locs[AR_X] do
 6:       Let S_a be the size of a-loc a.
 7:       // Find the corresponding a-locs in AR_C.
 8:       (F, P) := *(in[esp] +^vs offset(AR_X, a), S_a)
 9:       new_a := ⊥^vs
10:       if (P ≠ ∅) then
11:          new_a := ⊤^vs
12:       else
13:          vs_actuals := ⊔^vs{in[v] | v ∈ F}
14:          new_a := vs_actuals
15:       end if
16:       if X is recursive then
17:          out[a] := in[a] ⊔^vs new_a
18:       else
19:          out[a] := new_a
20:       end if
21:    end for
22:    return out
23: end proc
```

Fig. 10. Transformer for call→enter edge.

mem_4	$\mapsto (\mathbf{1}, \bot, \bot)$	eax	$\mapsto (\bot, -\mathbf{40}, \bot)$
mem_8	$\mapsto (\mathbf{2}, \bot, \bot)$	esp	$\mapsto (\bot, \bot, \mathbf{0})$
arg_0	$\mapsto (\bot, -\mathbf{40}, \bot)$	ext_48	$\mapsto (\bot, -\mathbf{40}, \bot)$
arg_4	$\mapsto (\mathbf{5}, \bot, \bot)$	ext_44	$\mapsto (\mathbf{5}, \bot, \bot)$

(The regions in the value-sets are listed in the following order: Global, AR_main, AR_initArray.) Note that the formal parameters arg_0 and arg_4 of initArray have been initialized to the value-sets of the corresponding actual parameters ext_48 and ext_44, respectively. ■

3.5.2 *Abstract Transformer for an exit→end-call Edge.* Unlike other abstract transformers, the transformer for an exit→end-call edge takes two AbsEnv values: (1) in_c, the AbsEnv value at the corresponding call node, and (2) in_x, the AbsEnv value at the exit node. The desired value for the exit→end-call edge is similar to in_x except for the value-sets of ebp, esp, and the a-locs of AR_C. The new $out \in$ AbsEnv is obtained by merging in_c and in_x, as shown in Fig. 11.

In standard code, the value of ebp at the exit node of a procedure is usually restored to the value of ebp at the call. Therefore, the value-set for ebp in the new *out* is obtained from the value-set for ebp at the call-site (line 5 of Fig. 11). The actual implementation of VSA checks the assumption that the value-set of ebp at the exit node has been restored to the value-set of ebp at the corresponding call node by comparing in_x[ebp] and in_c[ebp]. If in_x[ebp] is different from in_c[ebp], VSA issues a report to the user.

Obs. 3.3 about the relative positions of AR-regions AR_C and AR_X is used to determine the value-set for esp at the end-call node (lines 6–18 of Fig. 11). Recall that if esp holds the abstract address (AR_C, **s**) at the call, (AR_C, **s**) corre-

```
 1: proc MergeAtEndCall(in_c: AbsEnv, in_x: AbsEnv): AbsEnv
 2:    out := in_x
 3:    Let AR_C be the caller's memory-region.
 4:    Let AR_X be the callee's memory-region.
 5:    out[ebp] := in_c[ebp]
 6:    SI_c := in_c[esp][AR_C]
 7:    SI_x := in_x[esp][AR_X]
 8:    if (SI_x ≠ ⊥) then
 9:       VS'_esp := out[esp]
10:       VS'_esp[AR_C] := (SI_c +^si SI_x)
11:       if (AR_C ≠ AR_X) then VS'_esp[AR_X] := ⊥
12:       out[esp] := VS'_esp
13:       for each a-loc a ∈ a-locs[AR_X]\{FormalGuard, LocalGuard} do
14:          Update those a-locs in a-locs[AR_C] that correspond to a. (This step is similar to
             lines 5–20 of Fig. 10.)
15:       end for
16:    else
17:       out[esp] := in_x[esp]
18:    end if
19:    return out
20: end proc
```

Fig. 11. Abstract transformer for exit→end-call edge.

sponds to $(\texttt{AR\_X}, \mathbf{4})$. When the call is executed, the return address is pushed on the stack. Therefore, at the enter node of procedure `X`, `esp` holds the abstract address $(\texttt{AR\_C}, \mathbf{s} - \mathbf{4})$ or $(\texttt{AR\_X}, \mathbf{0})$. Consequently, if `esp` holds the abstract address $(\texttt{AR\_X}, \mathbf{t})$ at the exit node of procedure `X`,[8] the value-set of `esp` in the new AbsEnv value for the exit→end-call edge can be set to $(\texttt{AR\_C}, \mathbf{s} +^{\text{si}} \mathbf{t})$. For instance, at the call instruction 20 in Ex. 3.2, the value-set for `esp` is $(\bot, -\mathbf{48}, \bot)$. Therefore, the abstract address $(\texttt{AR\_main}, -\mathbf{48})$ corresponds to the abstract address $(\texttt{AR\_initArray}, \mathbf{4})$. Furthermore, at the exit node of procedure `initArray`, `esp` holds the abstract address $(\texttt{AR\_initArray}, \mathbf{8})$. Consequently, the value-set of `esp` at the end-call node is the abstract address $(\texttt{AR\_main}, -\mathbf{40})$. Note that this adjustment to `esp` corresponds to restoring the space allocated for actual parameters at the call-site of `AR_main`. Finally, the value-sets of the a-locs in a-locs[AR_C] are updated, which is similar to lines 5–20 of Fig. 10. If the value-set for `esp` at the exit node has no offsets in `AR_X` (the false branch of the condition at line 8 of Fig. 11), the value-set of `esp` for the exit→end-call edge is set to the value-set for `esp` at the exit node. (The condition at line 8 of Fig. 11 is usually false for procedures, such as `alloca`, that do not allocate a new activation record on the stack.)

3.5.3 *Interprocedural VSA algorithm.* The algorithm for interprocedural VSA is similar to the intraprocedural VSA algorithm given in Fig. 5 except that the *Propagate* procedure is replaced with the one shown in Fig. 12.

3.6 Indirect Jumps and Indirect Calls

The supergraph of the program will not be complete in the presence of indirect jumps and indirect calls. Consequently, the supergraph has to be augmented with

[8] We assume that the return address has been popped off the stack when the exit node is processed.

```
 1: proc Propagate(n: node, edge_amc: AbsEnv)
 2:    old := absEnv_n
 3:    if n is an end-call node then
 4:       Let c be the call node associated with n
 5:       edge_amc := MergeAtEndCall(edge_amc, absEnv_c(cs))
 6:    end if
 7:    new := old ⊔^ae edge_amc
 8:    if (old ≠ new) then
 9:       absEnv_n := new
10:       worklist := worklist ∪ {n}
11:    end if
12: end proc
```

Fig. 12. *Propagate* procedure for interprocedural VSA.

missing jump and call edges using abstract memory configurations determined by VSA. For instance, suppose that VSA is interpreting an indirect-jump instruction `J1:jmp` $[1000 + \texttt{eax} \times 4]$, and let the current abstract store at this instruction be $\{\texttt{eax} \mapsto (\mathbf{1}[\mathbf{0}, \mathbf{9}], \bot, \ldots, \bot)\}$. Edges need to be added from `J1` to the instructions whose addresses could be in memory locations {1000, 1004, ..., 1036}. If the addresses {1000, 1004, ..., 1036} refer to the read-only section of the program, then the addresses of the successors of `J1` can be read from the header of the executable. If not, the addresses of the successors of `J1` in locations {1000, 1004, ..., 1036} are determined from the current abstract store at `J1`. Due to possible imprecision in VSA, it could be the case that VSA reports that the locations {1000, 1004, ..., 1036} have all possible addresses. In such cases, VSA proceeds without recording any new edges. However, this could lead to an under-approximation of the value-sets at program points. Therefore, the analysis issues a report to the user whenever such decisions are made. We will refer to such instructions as *unsafe instructions*. Another issue with using the results of VSA is that an address identified as a successor of `J1` might not be the start of an instruction. Such addresses are ignored, and the situation is reported to the user.

When new edges are identified, instead of adding them right away, VSA defers the addition of new edges until a fixpoint is reached for the analysis of the current supergraph. After a fixpoint is reached, the new edges are added and VSA is restarted on the new supergraph. This process continues until no new edges are identified during VSA.

Indirect calls are handled similarly, with a few additional complications.

—A successor instruction identified by the method outlined above may be in the middle of a procedure. In such cases, VSA reports this to the user.

—The successor instruction may not be part of a procedure that was identified by IDAPro. This can be due to the limitations of IDAPro's procedure-finding algorithm: IDAPro does not identify procedures that are called exclusively via indirect calls. In such cases, VSA can invoke IDAPro's procedure-finding algorithm explicitly, to force a sequence of bytes from the executable to be decoded into a sequence of instructions and spliced into the IR for the program. (At present, this technique has not been incorporated in our implementation.)

3.7 Context-Sensitive Interprocedural Value-Set Analysis

The VSA algorithm discussed so far is context-insensitive, i.e., at each node in a procedure it does not maintain different abstract states for different calling contexts. Merging information from different calling contexts can result in a loss of precision. In this section, we discuss a context-sensitive VSA algorithm based on the call-strings approach [Sharir and Pnueli 1981]. The context-sensitive VSA algorithm distinguishes information from different calling contexts to a limited degree, thereby computing a tighter approximation of the set of reachable concrete states at every program point.

Call-Strings. The call-graph of a program is a labeled graph in which each node represents a procedure, each edge represents a call, and the label on an edge represents the call-site corresponding to the call represented by the edge. A call-string [Sharir and Pnueli 1981] is a sequence of call-sites $(s_1 s_2 \dots s_n)$ such that call-site s_1 belongs to the entry procedure, and there exists a path in the call-graph consisting of edges with labels s_1, s_2, …, s_n. CallString is the set of all call-strings for the executable. CallSites is the set of call-sites in the executable.

A call-string suffix of length k is either $(c_1 c_2 \dots c_k)$ or $(*c_1 c_2 \dots c_k)$, where c_1, c_2, …, $c_k \in$ CallSites. $(c_1 c_2 \dots c_k)$ represents the string of call-sites $c_1 c_2 \dots c_k$. $(*c_1 c_2 \dots c_k)$, which is referred to as a *saturated* call-string, represents the set $\{cs \in \mathsf{CallString} \mid cs = \pi c_1 c_2 \dots c_k, \pi \in \mathsf{CallString}, \text{and } |\pi| \geq 1\}$. $\mathsf{CallString}_k$ is the set of saturated call-strings of length k, plus non-saturated call-strings of length $\leq k$. Consider the call-graph shown in Fig. 13(a). The set $\mathsf{CallString}_2$ for this call-graph is $\{\epsilon, \mathtt{C}_1, \mathtt{C}_2, \mathtt{C}_1\mathtt{C}_3, \mathtt{C}_2\mathtt{C}_4, *\mathtt{C}_3\mathtt{C}_5, *\mathtt{C}_4\mathtt{C}_5, *\mathtt{C}_5\mathtt{C}_4\}$.

The following operations are defined for a call-string suffix:

—$cs \ll^{cs} c$: Let $cs \in \mathsf{CallString}_k$ and $c \in$ CallSites. $cs \ll^{cs} c$ returns a new call-string suffix $c' \in \mathsf{CallString}_k$ as follows:

$$c' = \begin{cases} (c_1 c_2 \dots c_i c) & \text{if } cs = (c_1 c_2 \dots c_i) \wedge (i < k) \\ (*c_2 c_3 \dots c_k c) & \text{if } cs = (c_1 c_2 \dots c_k) \end{cases}$$

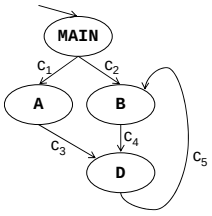
—$cs_1 \rightsquigarrow^{cs} cs_2$: Let $cs_1 \in \mathsf{CallString}_k$ and $cs_2 \in \mathsf{CallString}_k$. $(cs_1 \rightsquigarrow^{cs} cs_2)$ evaluates to TRUE if cs_1 leads to cs_2, i.e., if $\exists c \in$ CallSites such that $(cs_1 \ll^{cs} c) = cs_2$; otherwise, it evaluates to FALSE.

Context-Sensitive VSA Algorithm. The context-sensitive VSA algorithm [Balakrishnan 2007] associates each program point with an AbsMemConfig:

$$\boxed{\mathsf{AbsMemConfig} = (\mathsf{CallString}_k \rightarrow \mathsf{AbsEnv}_\perp)}$$

That is, at every program point, VSA maps each call-string to a different AbsEnv, thereby possibly distinguishing the information obtained from different call-sites to a limited extent.

3.7.1 *Memory-Region Status Map.* Recall from case 2 of Fig. 4 that, for an a-loc that belongs to the AR of a recursive procedure, it is only possible to perform a weak update during intraprocedural VSA. During context-sensitive VSA, on the other hand, it is possibly to perform a strong update in certain cases. For instance, we can perform a strong update for a-locs that belong to a recursive procedure, if recursion has not yet occurred in the given calling context. During VSA, all abstract



(a)

Call-string	Length	Memory-region status map	Comment
C_1C_3	2	MAIN ↦ NS, A ↦ NS, D ↦ NS	B is inaccessible
C_2C_4	2	MAIN ↦ NS, B ↦ NS, D ↦ NS,	A is inaccessible
$*C_5C_4$	2	MAIN ↦ NS, A ↦ NS, B ↦ S,D ↦ S	

(b)

Fig. 13. (a) Call-graph; (b) memory-region status map for different call-strings. (Key: NS: non-summary, S: summary; * refers to a saturated call-string.)

transformers are passed a *memory-region status map* that indicates which memory-regions, in the context of a given call-string *cs*, are summary memory-regions. Whereas the Global region is always non-summary and all malloc-regions are always summary, to decide whether a procedure P's memory-region is a summary memory-region, first call-string *cs* is traversed, and then the call graph is traversed, to see whether the runtime stack could contain multiple pending activation records for P. Fig. 13(b) shows the memory-region status map for different call-strings of length 2.

The memory-region status map provides one of two pieces of information used to identify when a strong update can be performed. In particular, an abstract transformer can perform a strong update if the operation modifies (a) a register, or (b) a non-array variable[9] in a non-summary memory-region.

3.8 Soundness of VSA

Soundness would mean that, for each instruction in the executable, value-set analysis would identify an AbsMemConfig that over-approximates the set of all possible concrete stores that a program reaches during execution for all possible inputs. This is a lofty goal; however, it is not clear that a tool that achieves this goal would have practical value. (It is achievable trivially, merely by setting all value-sets to $\top^{vs}$.) There are less lofty goals that do not meet this standard—but may result in a more practical system. In particular, we may not care if the system is sound, as long as it can provide warnings about the situations that arise during the analysis that threaten the soundness of the results. This is the path that we followed in our work.

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

—The program can read or write past the end of an AR. A report is generated at each point at which `LocalGuard` or `FormalGuard` could be read from or written to.

—The control-flow graph and call-graph may not identify all successors of indirect jumps and indirect calls. Report generation for such cases is discussed in §3.6.

[9]The Semi-Naïve algorithm described in §2 does not recover information about arrays. However, the a-loc-recovery algorithm described in §4 is capable of recovering information about arrays.

—A related situation is a jump to a code sequence concealed in the regular instruction stream; the alternative code sequence would decode as a legal code sequence when read out-of-registration with the instructions in which it is concealed. The analysis could detect this situation as an anomalous jump to an address that is in the code segment, but is not the start of an instruction.

—With self-modifying code, the control-flow graph and call-graph are not available for analysis. The analysis can detect the possibility that the program is self-modifying by identifying an anomalous jump or call to a modifiable location, or by a write to an address in the code region.

3.9 Dynamically Loaded Libraries (DLLs)

In addition to statically linked libraries, application programs also use dynamically loaded libraries (DLLs) to access common APIs. Unlike statically linked libraries, the code for a DLL is not included with the executable. Therefore, to be sound, CodeSurfer/x86 has to find and include the code for the DLLs during the analysis. It is relatively simple for CodeSurfer/x86 to deal with DLLs that are known at link time. Information about such DLLs is available in the executable's header, and therefore, CodeSurfer/x86 has to simply mimic the OS loader.

Executables may also load DLLs programmatically. A typical API sequence[10] used to load a DLL and invoke an API method in the newly loaded DLL is as follows: (1) invoke `LoadLibrary` with the name of the DLL as an argument, (2) obtain the address of the required API method by invoking `GetProcAddress` with the name of the API method as an argument, and (3) use the address obtained from `GetProcAddress` to make the API call. Information about such DLLs is not known until CodeSurfer/x86 analyzes the corresponding `LoadLibrary` call. Furthermore, the calls to APIs in DLLs that are loaded programmatically appear as indirect function calls in the executable. We deal with such DLLs in a way similar to how indirect jumps and indirect calls are handled. Whenever a `LoadLibrary` call is encountered, instead of loading the DLL right away, only the information about the new DLL is recorded. When VSA reaches a fixpoint, the newly discovered DLLs are loaded, and VSA is restarted. This process is repeated until no new DLLs are discovered. §5 describes this iteration loop in detail.

4. IMPROVED TECHNIQUES FOR DISCOVERING (PROXIES FOR) VARIABLES

In this section, we return to the issue that was discussed in §2.2—namely, how to identify variable-like entities ("a-locs") that can serve as proxies for the missing source-level variables in algorithms for further analysis of executables, such as VSA.

IDAPro's Semi-Naïve algorithm for identifying a-locs, described in §2.2, has certain limitations. IDAPro's algorithm only considers accesses to global variables that appear as "[*absolute-address*]", and accesses to local variables that appear as "[`esp` + *offset*]" or "[`ebp` − *offset*]" in the executable. It does not take into account accesses to elements of arrays and variables that are only accessed through pointers, and sometimes cannot take into account accesses to fields of structures, because these accesses are performed in ways that do not match any of the patterns that IDAPro considers. Therefore, it generally recovers only very coarse information

[10]The Windows API sequence is presented here. The techniques are not OS specific.

about arrays and structures. Moreover, this approach fails to provide any information about the fields of heap-allocated objects, which is crucial for understanding programs that manipulate the heap.

The aim of the work presented in this section is to improve the state of the art by using abstract interpretation [Cousot and Cousot 1977] to replace local analyses with ones that take a more comprehensive view of the operations performed by the program. We present an algorithm that combines Value-Set Analysis (VSA) as described in §3 and Aggregate Structure Identification (ASI) [Ramalingam et al. 1999], which is an algorithm that infers the substructure of aggregates used in a program based on how the program accesses them, to recover variables that are better than those recovered by IDAPro. As explained in §4.4, the combination of VSA and ASI allows us (a) to recover variables that are based on *indirect* accesses to memory, rather than just the explicit addresses and offsets that occur in the program, and (b) to identify structures, arrays, and nestings of structures and arrays. Moreover, when the variables that are recovered by our algorithm are used during VSA, the precision of VSA improves. This leads to an interesting abstraction-refinement scheme; improved precision during VSA causes an improvement in the quality of variables recovered by our algorithm, which, in turn, leads to improved precision in a subsequent round of VSA, and so on.

Our goal is to subdivide the memory-regions of the executable into variable-like entities (which we call *a-locs*, for "abstract locations"). These can then be used as variables in tools that analyze executables. Memory-regions are subdivided using the information about how the program accesses its data. The intuition behind this approach is that data-access patterns in the program provide clues about how data is laid out in memory. For instance, the fact that an instruction in the executable accesses a sequence of four bytes in memory-region `M` is an indication that the programmer (or the compiler) intended to have a four-byte-long variable or field at the corresponding offset in `M`. First, we present the problems in developing such an approach, and the insights behind our solution, which addresses those problems. Details are provided in §4.4.

4.1 The Problem of Indirect Memory Accesses

The *Semi-Naïve algorithm* described in §2.2 uses the access expressions of the forms "[*absolute-address*]", "[`esp` + *offset*]", and "[`ebp` −*offset*]" to recover a-locs. That approach produces poor results in the presence of indirect memory operands.

EXAMPLE 4.1. The program shown below initializes the two fields `x` and `y` of a local struct through the pointer `pp` and returns 0. `pp` is located at offset -12,[11] and struct `p` is located at offset -8 in the activation record of `main`. Address expression "`ebp-8`" refers to the address of `p`, and address expression "`ebp-12`" refers to the address of `pp`.

[11] Recall that we follow the convention that the value of `esp` (the stack pointer) at the beginning of a procedure marks the origin of the procedure's AR-region.

```
typedef struct {
   int x, y;
} Point;

int main(){
   Point p, *pp;
   pp = &p;
   pp->x = 1;
   pp->y = 2;
   return 0;
}
```

```
    proc main
1   mov ebp, esp
2   sub esp, 12
3   lea eax, [ebp-8]
4   mov [ebp-12], eax
5   mov [eax], 1
6   mov [eax+4], 2
7   mov eax, 0
8   add esp, 12
9   retn
```

Instruction `4` initializes the value of `pp`. (Instruction "`3 lea eax, [ebp-8]`" is equivalent to the assignment `eax := ebp-8`.) Instructions `5` and `6` update the fields of `p`. Observe that, in the executable, the fields of p are updated via `eax`, rather than via the pointer `pp` itself, which resides at address `ebp-12`. ■

In Ex. 4.1, `-8` and `-12` are the offsets relative to the frame pointer (i.e., `ebp`) that occur explicitly in the program. The Semi-Naïve algorithm would say that offsets `-12` through `-9` of the AR of `main` constitute one variable (say `var_12`), and offsets `-8` through `-1` of AR of `main` constitute another (say `var_8`). The Semi-Naïve algorithm correctly identifies the position and size of `pp`. However, it groups the two fields of `p` together into a single variable because it does not take into consideration the indirect memory operand `[eax+4]` in instruction `6`.

Typically, indirect operands are used to access arrays, fields of structures, fields of heap-allocated data, etc. Therefore, to recover a useful collection of variables from executables, one has to look beyond the explicitly occurring addresses and stack-frame offsets. Unlike the operands considered in the Semi-Naïve algorithm, local methods do not provide information about what an indirect memory operand accesses. For instance, an operand such as "[`ebp` − *offset*]" (usually) accesses a local variable. However, "`[eax + 4]`" may access a local variable, a global variable, a field of a heap-allocated data-structure, etc., depending upon what `eax` contains.

Obtaining information about what an indirect memory operand accesses is not straightforward. In this example, `eax` is initialized with the value of a register (minus a constant offset). In general, a register used in an indirect memory operand may be initialized with a value read from memory. In such cases, to determine the value of the register, it is necessary to know the contents of that memory location, and so on. Fortunately, Value-Set Analysis (VSA), described in §3, can provide such information.

4.2 The Problem of Granularity and Expressiveness

The granularity and expressiveness of recovered variables can affect the precision of analysis clients that use the recovered variables as the executable's data objects.

As a specific example of an analysis client, consider a data-dependence analyzer, which answers such questions as: *"Does the write to memory at instruction `L1` in Ex. 2.1 affect the read from memory at instruction `14`"*. Note that in Ex. 2.1 the write to memory at instruction `L1` does not affect the read from memory at instruction `14` because `L1` updates the `x` members of the elements of array `pts`,

while instruction 14 reads the y member of array element pts[0]. To simplify the discussion, assume that a data-dependence analyzer works as follows: (1) annotate each instruction with used, killed, and possibly-killed variables, and (2) compare the used variables of each instruction with killed or possibly-killed variables of every other instruction to determine data dependences.[12]

Consider three different partitions of the AR of main in Ex. 2.1:

VarSet$_1$. As shown in Fig. 3(c), the Semi-Naïve approach from §2.2 would say that the AR of main has three variables: var_44 (4 bytes), var_40 (4 bytes), and var_36 (36 bytes). The variables that are possibly killed at L1 are {var_40, var_36}, and the variable used at 14 is var_36. Therefore, the data-dependence analyzer reports that the write to memory at L1 might affect the read at 14. (This is sound, but imprecise.)

VarSet$_2$. As shown in Fig. 3(c), there are two variables for each element of array pts. The variables possibly killed at L1 are {pts[0].x, pts[1].x, pts[2].x, pts[3].x, pts[4].x}, and the variable used at instruction 14 is pts[0].y. Because these sets are disjoint, the data-dependence analyzer reports that the memory write at instruction L1 definitely does not affect the memory read at instruction 14.

VarSet$_3$. Suppose that the AR of main is partitioned into just three variables: (1) py, which represents the local variable py, (2) pts[?].x, which is a representative for the x members of the elements of array pts, and (3) pts[?].y, which is a representative for the y members of the elements of array pts. pts[?].x and pts[?].y are summary variables because they represent more than one concrete variable. The summary variable that is possibly killed at instruction L1 is pts[?].x and the summary variable that is used at instruction 14 is pts[?].y. These are disjoint; therefore, the data-dependence analyzer reports a definite answer, namely, that the write at L1 does not affect the read at 14.

Of the three alternatives presented above, VarSet$_3$ has several desirable features:

—It has a smaller number of variables than VarSet$_2$. When it is used as the set of variables in a data-dependence analyzer, it provides better results than VarSet$_1$.

—The variables in VarSet$_3$ are capable of representing a set of non-contiguous memory locations. For instance, pts[?].x represents the locations corresponding to pts[0].x, pts[1].x, ..., pts[4].x. The ability to represent non-contiguous sequences of memory locations is crucial for representing a specific field in an array of structures.

—The AR of main is only partitioned as much as necessary. In VarSet$_3$, only one summary variable represents the x members of the elements of array pts, while each member of each element of array pts is assigned a separate variable in VarSet$_2$.

[12]This method provides flow-insensitive data-dependence information; flow-sensitive data-dependence information can be obtained by performing a reaching-definitions analysis in terms of used, killed, and possibly-killed variables. This discussion is couched in terms of flow-insensitive data-dependence information solely to simplify the discussion; the same issues arise even if one uses flow-sensitive data-dependence information.

A good variable-recovery algorithm should partition a memory-region in such a way that the set of variables obtained from the partition has the desirable features of $\texttt{VarSet}_3$. When debugging information is available, this is a trivial task. However, debugging information is often not available. Data-access patterns in the program provide information that can serve as a substitute for debugging information. For instance, instruction `L1` accesses each of the four-byte sequences that start at offsets $\{-40, -32, \ldots, -8\}$ in the AR of main. The common difference of 8 between successive offsets is evidence that the offsets may represent the elements of an array. Moreover, instruction `L1` accesses every four bytes starting at these offsets. Consequently, the elements of the array are judged to be structures in which one of the fields is four bytes long.

4.3 Background: Aggregate Structure Identification (ASI)

Ramalingam et al. [1999] observe that there can be a loss of precision in the results that are computed by a static-analysis algorithm if it does not distinguish between accesses to different parts of the same aggregate (in Cobol programs). They developed the Aggregate Structure Identification (ASI) algorithm to distinguish among such accesses, and showed how the results of ASI can improve the results of dataflow analysis. This section briefly describes the ASI algorithm. (In §4.4, we show how to use the information gathered during VSA to harness ASI to the problem of identifying variable-like entities in executables.)

ASI [Ramalingam et al. 1999] is a unification-based, flow-insensitive algorithm to identify the structure of aggregates in a program (such as arrays, C structs, etc.). The algorithm ignores any type information known about 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 called *atoms*.

The data-access patterns in the program are specified to the ASI algorithm through a data-access constraint language (DAC). The syntax of DAC programs is shown in Fig. 14. There are two kinds of constructs in a DAC program: (1) `DataRef` is a reference to a set of bytes, and provides a means to specify how the data is accessed in the program; (2) `UnifyConstraint` provides a means to specify the flow of data in the program. Note that the direction of data flow is not considered in a `UnifyConstraint`. The justification for this is that a flow of data from one sequence of bytes to another is evidence that they should both have the same structure. ASI uses the constraints in the DAC program to find a coarsest refinement of the aggregates.

```
              Pgm   ::=  ϵ | UnifyConstraint Pgm
UnifyConstraint   ::=  DataRef ≈ DataRef ;
        DataRef   ::=  ProgVars | DataRef[UInt:UInt] | DataRef\UInt+
```

Fig. 14. Data-Access Constraint (DAC) language. `UInt` is the set of non-negative integers; $\texttt{UInt}_+$ is the set of positive integers; and `ProgVars` is the set of program variables.

There are three kinds of data references:

—A variable `P` ∈ `ProgVars` refers to all the bytes of variable `P`.

—`DataRef[`l`:`u`]` refers to bytes l through u in `DataRef`. For example, `P[8:11]` refers to bytes `8..11` of variable `P`.

—`DataRef\`n is interpreted as follows: `DataRef` is an array of n elements and `DataRef\`n refers to the bytes of an element of array `DataRef`. For example, `P[0:11]\3` refers to the sequences of bytes `P[0:3]`, `P[4:7]`, or `P[8:11]`.

Instead of going into the details of the ASI algorithm, we provide the intuition behind the algorithm by means of an example. Consider the source-code program shown in Ex. 2.1. The data-access constraints for the program are

```
pts[0:39]\5[0:3]  ≈  a[0:3];
pts[0:39]\5[4:7]  ≈  b[0:3];
return_main[0:3]  ≈  pts[4:7];
          i[0:3]  ≈  const_1[0:3];
          p[0:3]  ≈  const_2[0:3];
         py[0:3]  ≈  const_3[0:3];
```

The first constraint encodes the initialization of the `x` members, namely, `pts[i].x = a`. The DataRef `pts[0:39]\5[0:3]` refers to the bytes that correspond to the `x` members in array `pts`. The third constraint corresponds to the return statement; it represents the fact that the return value of `main` is assigned bytes `4..7` of `pts`, which correspond to `pts[0].y`. The constraints reflect the fact that the size of `Point` is `8` and that `x` and `y` are laid out next to each other.

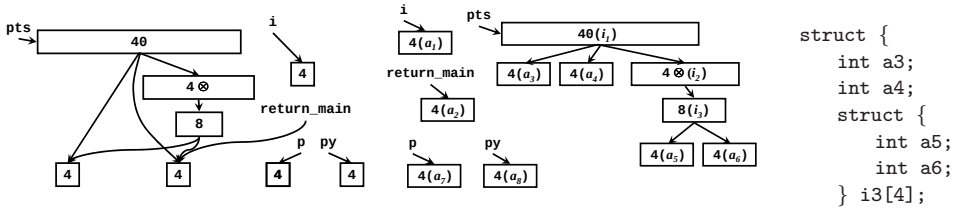


Fig. 15. (a) ASI DAG, (b) ASI tree, and (c) the struct recovered for the program in Ex. 2.1. (To avoid clutter, global variables are not shown.)

The result of the ASI *atomization* algorithm is a DAG that shows the structure of each aggregate as well as relationships among the atoms of aggregates. The DAG for Ex. 2.1 is shown in Fig. 15(a). An ASI DAG has the following properties:

—A node represents a set of bytes.

—A sequence of bytes that is accessed as an array in the program is represented by an *array* node. Array nodes are labeled with $\bigotimes$. The number in an array node represents the number of elements in the array. An array node has one child, and the DAG rooted at the child represents the structure of the array element. In Fig. 15(a), bytes `8..39` of array `pts` are identified as an array of four 8-byte elements. Each array element is a struct with two fields of 4 bytes each.

—A sequence of bytes that is accessed like a C struct in the program is represented by a *struct* node. The number in the struct node represents the length of the struct; the children of a struct node represent the fields of the struct. The order of the children in the DAG represent the order of the fields in the struct. In Fig. 15(a), bytes `0..39` of `pts` are identified as a struct with three fields: two 4-byte scalars and one 32-byte array.

—Nodes are shared if there is a flow of data in the program involving the corresponding sequence of bytes either directly or indirectly. In Fig. 15(a), the nodes for the sequences of bytes `return_main[0:3]` and `pts[4:7]` are shared because of the `return` statement in `main`. Similarly, the sequence of bytes that correspond to the `y` members of array `pts`, namely `pts[0:39]\5[4:7]`, share the same node because they are all assigned the same constant at the same instruction.

The ASI DAG is converted into an ASI tree by duplicating shared nodes. The atoms of an aggregate are the leaves of the corresponding ASI tree. Fig. 15(b) shows the ASI tree for Ex. 2.1. ASI has identified that `pts` has the structure shown in Fig. 15(c).

4.4 Recovering A-locs via Iteration

The atoms identified by ASI for Ex. 2.1 are close to the set of variables $\texttt{VarSet}_3$ that was discussed in §4.2. One might hope to apply ASI to an executable by treating each memory-region as an aggregate and determining the structure of each memory-region (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. When applying ASI to programs written in languages such as Cobol this is possible: the data-access patterns—in particular, the data-access patterns for array accesses—are apparent from the syntax of the Cobol constructs under consideration. Unfortunately, this is not the case for executables. For instance, the memory operand `[eax]` can either represent an access to a single variable or to the elements of an array.

Fortunately, the results of VSA provide information that can be used to generate suitable data-access constraints for an executable. A value-set is an over-approximation of a set of offsets in each memory-region. We use VSA results to interpret each indirect memory operand to obtain an over-approximation of the set of locations that the operand may access. Together with the information about the number of bytes accessed (which is available from the instruction), this provides the information needed to generate data-access constraints for the executable.

Some of the features of VSA that are useful in a-loc recovery are

—*VSA provides information about indirect memory operands:* For the program in Ex. 4.1, VSA determines that the value-set of `eax` at instruction `6` is $(\emptyset, \mathbf{0}[-\mathbf{8}, -\mathbf{8}])$, which means that `eax` must hold the offset -8 in the AR-region of `main`. Using this information, we can conclude that `[eax+4]` refers to offset -4 in the AR-region of `main`.

—*VSA provides data-access patterns:* For the program in Ex. 2.1, VSA determines that the value-set of `eax` at program point `L1` is $(\emptyset, \mathbf{8}[-\mathbf{40}, -\mathbf{8}])$, which means that `eax` may hold the offsets $\{-40, -32, \ldots, -8\}$ in the AR-region of `main`. (These offsets are the starting addresses of field `x` of elements of array `pts`.)

—*VSA tracks updates to memory:* This is important because, in general, the registers used in an indirect memory operand may be initialized with a value read from memory. If updates to memory are not tracked, we may neither have useful information for indirect memory operands nor useful data-access patterns for the executable.

Furthermore, when we use the atoms of ASI as a-locs in VSA, the results of VSA can improve. Consider the program in Ex. 4.1. Recall from §4.1 that the length of `var_8` is 8 bytes. Because value-sets are only capable of representing a set of 4-byte addresses and 4-byte values, VSA recovers no useful information for `var_8`: it merely reports that the value-set of `var_8` is $\top^{vs}$ (meaning any possible value or address). Applying ASI (using data-access patterns provided by VSA) results in the splitting of `var_8` into two 4-byte a-locs, namely, `var_8.0` and `var_8.4`. Because `var_8.0` and `var_8.4` are each four bytes long, VSA can now track the set of values or addresses in these a-locs. Specifically, VSA would determine that `var_8.0` (i.e., `p.x`) has the value `1` and `var_8.4` (i.e., `p.y`) has the value `2` at the end of `main`.

We can use the new VSA results to perform another round of ASI. If the value-sets computed by VSA are improved from the previous round, the next round of ASI may also improve. We can repeat this process as long as desired, or until the process converges (see §5).

Although not illustrated by Ex. 4.1, additional rounds of ASI and VSA can result in further improvements. For example, suppose that the program uses a chain of pointers to link `struct`s of different types, e.g., variable `ap` points to a `struct A`, which has a field `bp` that points to a `struct B`, which has a field `cp` that points to a `struct C`, and so on. Typically, the first round of VSA recovers the value of `ap`, which lets ASI discover the a-loc for `A.bp` (from the code compiled for `ap->bp`); the second round of VSA recovers the value of `ap->bp`, which lets ASI discover the a-loc for `B.cp` (from the code compiled for `ap->bp->cp`); etc.

To summarize, the algorithm for recovering a-locs is

(1) Run VSA using a-locs recovered by the Semi-Naïve approach
(2) Generate data-access patterns from the results of VSA
(3) Run ASI
(4) Run VSA
(5) Repeat steps 2, 3, and 4 until there are no improvements to the results of VSA.[13]

Because ASI is a unification-based algorithm, generating data-access constraints for certain kinds of instructions leads to undesirable results (see §4.8 for more details). Fortunately, it is not necessary to generate data-access constraints for all instructions in the program that contain memory-access expressions because VSA generates sound results for *any* collection of a-locs with which it is supplied. For these reasons, ASI is used only as a heuristic to find a-locs for VSA. (If VSA is supplied with very coarse a-locs, many a-locs will be found to have the value $\top^{vs}$

[13]Or, equivalently, until the set of a-locs discovered in step 3 is unchanged from the set previously discovered in step 3 (or step 1).

at most points; however, by refining the a-locs in use, more precise answers can generally be obtained.)

In short, our abstraction-refinement principles are as follows:

(1) VSA results are used to interpret memory-access expressions in the executable.
(2) ASI is used as a heuristic to determine the structure of each memory-region according to information recovered by VSA.
(3) Each ASI tree reflects the memory-access patterns in one memory-region, and the leaves of the ASI trees define the a-locs that are used for the next round of VSA.

ASI alone is not a replacement for VSA. That is, ASI cannot be applied to executables without the information that is obtained from VSA—namely value-sets.

In the rest of this section, we describe the interplay between VSA and ASI: (1) we show how value-sets are used to generate data-access constraints for input to ASI, and (2) how the atoms in the ASI trees are used as a-locs during the next round of VSA.

4.5 Generating Data-Access Constraints

This section describes the algorithm that generates ASI data-references for x86 operands. Three forms of x86 operands need to be considered: (1) register operands, (2) memory operands of form "[*register*]", and (3) memory operands of the form "[*base* + *index* × *scale* + *offset*]".

To prevent unwanted unification during ASI, we rename registers using live-ranges. For a register `r`, the ASI data-reference is $\texttt{r}_{lr}[0 : n-1]$, where lr is the live-range of the register at the given instruction and n is the size of the register (in bytes).

In the rest of the section, we describe the algorithm for memory operands. First, we consider indirect operands of the form [`r`]. To gain intuition about the algorithm, consider operand `[eax]` of instruction `L1` in Ex. 2.1. The value-set associated with `eax` is $(\emptyset, \mathbf{8}[-\mathbf{40}, -\mathbf{8}])$. The stride value of 8 and the interval $[-40, -8]$ in the AR of `main` provide evidence that `[eax]` is an access to the elements of an array of 8-byte elements in the range $[-40, -8]$ of the AR of `main`; an array access is generated for this operand.

Recall that a value-set is an n-tuple of strided intervals. The strided interval $s[l, u]$ in each component represents the offsets in the corresponding memory-region. Fig. 16 shows the pseudocode to convert offsets in a memory-region into an ASI reference. Procedure *SI2ASI* takes the name of a memory-region r, a strided interval $s[l, u]$, and *length* (the number of bytes accessed) as arguments. The *length* parameter is obtained from the instruction. For example, the *length* for [`eax`] is 4 because the instruction at `L1` in Ex. 2.1 is a four-byte data transfer. The algorithm returns a pair in which the first component is an ASI reference and the second component is a Boolean. The significance of the Boolean component is described later in this section. The algorithm works as follows: If $s[l, u]$ is a singleton (i.e., it represents just a single value, and thus $s = 0$ and $l = u$), then the ASI reference is the one that accesses offsets l to $l + \mathit{length} - 1$ in the aggregate associated with memory-region r. If $s[l, u]$ is not a singleton, then the offsets represented by $s[l, u]$ are treated as references to an array. The size of the array element is the stride s

Input: The name of a memory-region r, strided interval $s[l, u]$, number of bytes accessed *length*.
Output: A pair in which the first component is an ASI reference for the sequence of *length* bytes starting at offsets $s[l, u]$ in memory-region r and the second component is a Boolean that represents whether the ASI reference is an exact reference (true) or an approximate one (false). (‖ denotes string concatenation.)

```
proc SI2ASI(r: String, s[l, u]: StridedInterval, length: Integer)
  if s[l, u] is a singleton then
    return ⟨r ‖ "[l : l + length − 1]", true⟩
  else
    size := max(s, length)
    n := ⌊(u − l)/size⌋ + 1
    ref := r ‖ "[l : u + size − 1]\n[0 : length − 1]"
    return ⟨ref, (s ≥ length)⟩
  end if
end proc
```

Fig. 16. Algorithm to convert a given strided interval into an ASI reference.

whenever ($s \geq$ *length*). However, when ($s <$ *length*) an overlapping set of locations is accessed by the indirect memory operand. Because an overlapping set of locations cannot be represented using an ASI reference, the algorithm chooses *length* as the size of the array element. This is not a problem for the soundness of subsequent rounds of VSA because of refinement principle 2. The Boolean component of the pair denotes whether the algorithm generated an exact ASI reference or not. The number of elements in the array is $\lfloor (u - l)/size \rfloor + 1$.

For operands of the form [`r`], the set of ASI references is generated by invoking procedure *SI2ASI* shown in Fig. 16 for each non-empty memory-region in `r`'s value-set. For Ex. 2.1, the value-set associated with `eax` at `L1` is $(\emptyset, \mathbf{8}[-\mathbf{40}, -\mathbf{8}])$. Therefore, the set of ASI references is {AR_main[(-40):(-1)]\5[0:3]}.[14] There are no references to the `Global` region because the set of offsets in that region is empty.

The algorithm for converting indirect operands of the form [*base* + *index* × *scale* + *offset*] is given in Fig. 17. One typical use of indirect operands of the form [*base* + *index* × *scale* + *offset*] is to access two-dimensional arrays. Note that *scale* and *offset* are statically-known constants. Because abstract values are strided intervals, we can absorb *scale* and *offset* into *base* and *index*. Hence, without loss of generality, we only discuss memory operands of the form [*base*+*index*]. Assuming that the two-dimensional array is stored in row-major order, one of the registers (usually *base*) holds the starting addresses of the rows and the other register (usually *index*) holds the indices of the elements in the row. Fig. 17 shows the algorithm to generate an ASI reference, when the set of offsets in a memory-region is expressed as a sum of two strided intervals as in [*base*+*index*]. Note that we could have used procedure *SI2ASI* shown in Fig. 16 by computing the abstract sum ($+^{si}$) of the two strided intervals. However, doing so results in a loss of precision because strided intervals can only represent a single stride exactly, and this would prevent us from recovering the structure of two-dimensional arrays. (In some circumstances,

[14] Offsets in a `DataRef` cannot be negative. Negative offsets are used for clarity. Negative offsets are mapped to the range $[0, 2^{31} - 1]$; non-negative offsets are mapped to the range $[2^{31}, 2^{32} - 1]$.

Input: The name of a memory-region r, two strided intervals $s_1[l_1, u_1]$ and $s_2[l_2, u_2]$, number of bytes accessed *length*.
Output: An ASI reference for the sequence of *length* bytes starting at offsets $s_1[l_1, u_1] + s_2[l_2, u_2]$ in memory region r.

```
proc TwoSIsToASI(r: String, s1[l1,u1]: StridedInterval, s2[l2,u2]: StridedInterval, length: Integer)
  if (s1[l1,u1] or s2[l2,u2] is a singleton) then
    return SI2ASI(r, s1[l1,u1] +^si s2[l2,u2], length)
  end if
  if s1 ≥ (u2 − l2 + length) then
    baseSI := s1[l1,u1]
    indexSI := s2[l2,u2]
  else if s2 ≥ (u1 − l1 + length) then
    baseSI := s2[l2,u2]
    indexSI := s1[l1,u1]
  else
    return SI2ASI(r, s1[l1,u1] +^si s2[l2,u2], length)
  end if
  ⟨baseRef, __⟩ := SI2ASI(r,baseSI, stride(baseSI)) // SI2ASI always returns an exact reference here.
  return baseRef ‖ SI2ASI("", indexSI, length)
end proc
```

Fig. 17. Algorithm to convert the set of offsets represented by the sum of two strided intervals into an ASI reference.

our implementation of ASI can recover the structure of arrays of 3 and higher dimensions.)

Procedure *TwoSIsToASI* works as follows: First, it determines which of the two strided intervals is used as the *base* because it is not always apparent from the representation of the operand. The strided interval that is used as the *base* should have a stride that is greater than the length of the interval in the other strided interval. Once the roles of the strided intervals are established, the algorithm generates the ASI reference for *base* followed by the ASI reference for *index*. In some cases, the algorithm cannot establish either of the strided intervals as the base. In such cases, the algorithm computes the abstract sum ($+^{si}$) of the two strided intervals and invokes procedure *SI2ASI*.

Procedure *TwoSIsToASI* generates a richer set of ASI references than procedure *SI2ASI* shown in Fig. 16. For example, consider the indirect memory operand `[eax+ecx]` from a loop that traverses a two-dimensional array of type `char`[5][10]. Suppose that the value-set of `ecx` is $(\emptyset, 10[-50, -10])$, the value-set of `eax` is $(1[0, 9], \emptyset)$, and *length* is 1. For this example, the ASI reference that is generated is "`AR[-50:-1]\5[0:9]\10[0:0]`". That is, AR is accessed as an array of five 10-byte entities, and each 10-byte entity is accessed as an array of ten 1-byte entities. In contrast, if we performed $(\emptyset, 10[-50, -10]) +^{vs} (1[0, 9], \emptyset) = (\emptyset, 1[-50, -1])$ and applied *SI2ASI*, the ASI reference that would be generated is "`AR[-50:-1]\50[0:0]`"; i.e., AR is accessed as an array of fifty 1-byte entities.

4.6 Interpreting Indirect Memory References

This section describes a lookup algorithm that finds the set of a-locs accessed by a memory operand. The algorithm is used to interpret pointer-dereference operations during VSA. For instance, consider the instruction "`mov [eax], 10`". During VSA, the lookup algorithm is used to determine the a-locs accessed by `[eax]` and the value-sets for the a-locs are updated accordingly. In §3, the algorithm to determine the set of a-locs for a given value-set is trivial because each memory-region in §3 consists of a linear list of a-locs generated by the Semi-Naïve approach. However, after ASI is performed, the structure of each memory-region is an ASI tree.

Ramalingam et al. [1999] present a lookup algorithm to retrieve the set of atoms for an ASI expression. However, their lookup algorithm is not appropriate for use in VSA because the algorithm assumes that the only ASI expressions that can arise during lookup are the ones that were used during the atomization phase. Unfortunately, this is not the case during VSA, for the following reasons:

—ASI is used as a heuristic. As will be discussed in §4.8, some data-access patterns that arise during VSA should be ignored during ASI.

—The executable can access fields of structures that have not yet been broken down into atoms. For example, the initial round of ASI, which is based on data-access constraints generated using the Semi-Naïve approach, will not have performed atomization based on accesses on fields of structures. However, the first round of VSA may have to interpret such field accesses.

We will use the tree shown in Fig. 15(b) to describe the lookup algorithm. Every node in the tree is given a unique name (shown within parentheses). The following terms are used in describing the lookup algorithm:

—`NodeFrag` is a descriptor for a part of an ASI tree node and is denoted by a triple $\langle name, start, length \rangle$, where *name* is the name of the ASI tree node, *start* is the starting offset within the ASI tree node, and *length* is the length of the fragment.

—`NodeFragList` is an ordered list of `NodeFrag` descriptors, $[nd_1, nd_2, \ldots, nd_n]$. A `NodeFragList` represents a contiguous set of offsets in an aggregate. For example, $[\langle a_3, 2, 2\rangle, \langle a_4, 0, 2\rangle]$ represents the offsets `2..5` of node i_1; offsets `2..3` come from $\langle a_3, 2, 2\rangle$ and offsets `4..5` come from $\langle a_4, 0, 2\rangle$.

The lookup algorithm traverses the ASI tree, guided by the ASI reference for the given memory operand. First, the memory operand is converted into an ASI reference using the algorithm described in §4.5, and the resulting ASI reference is broken down into a sequence of ASI operations. The task of the lookup algorithm is to interpret the sequence of operations working left-to-right. There are three kinds of ASI operations: (1) `GetChildren`(*aloc*), (2) `GetRange(`*start*, *end*`)`, and (3) `GetArrayElements(`m`)`. For example, the list of ASI operations for "`pts[0:39]\10[0:1]`" is `[GetChildren(pts), GetRange(0,39), GetArrayElements(10), GetRange(0,1)]`. Each operation takes a `NodeFragList` as argument and returns a set of `NodeFragList` values. The operations are performed from left to right. The argument of each operation comes from the result of the operation that is immediately to its left. The a-locs that are accessed are all the a-locs in the final set of `NodeFrag` descriptors.

The `GetChildren(`*aloc*`)` operation returns a `NodeFragList` that contains `NodeFrag` descriptors corresponding to the children of the root node of the tree associated with the aggregate *aloc*.

`GetRange(`*start*, *end*`)` returns a `NodeFragList` that contains `NodeFrag` descriptors representing the nodes with offsets in the given range [*start* : *end*].

`GetArrayElements(`m`)` treats the given `NodeFragList` as an array of m elements and returns a set of `NodeFragList` lists. Each `NodeFragList` list represents an array element. There can be more than one `NodeFragList` for the array elements because an array can be split during the atomization phase and different parts of the array might be represented by different nodes.

The following examples illustrate traces of a few lookups.

EXAMPLE 4.2. Lookup `pts[0:3]`

$$\begin{array}{rc} & [\langle i_1, 0, 40\rangle] \\ \texttt{GetChildren(pts)} & \Downarrow \\ & [\langle a_3, 0, 4\rangle, \langle a_4, 0, 4\rangle, \langle i_2, 0, 32\rangle] \\ \texttt{GetRange(0,3)} & \Downarrow \\ & [\langle a_3, 0, 4\rangle] \end{array}$$

`GetChildren(pts)` returns the `NodeFragList` $[\langle a_3, 0, 4\rangle, \langle a_4, 0, 4\rangle, \langle i_2, 0, 32\rangle]$. Applying `GetRange(0,3)` returns $[\langle a_3, 0, 4\rangle]$ because that describes offsets `0..3` in the given `NodeFragList`. The a-loc that is accessed by `pts[0:3]` is a_3. ■

EXAMPLE 4.3. Lookup `pts[0:39]\5[0:3]`

Let us look at `GetArrayElements(5)` because the other operations are similar to Ex. 4.2. `GetArrayElements(5)` is applied to $[\langle a_3, 0, 4\rangle, \langle a_4, 0, 4\rangle, \langle i_2, 0, 32\rangle]$. The total length of the given `NodeFragList` is 40 and the number of required array elements is 5. Therefore, the size of the array element is 8. Intuitively, the operation unrolls the given `NodeFragList` and creates a `NodeFragList` for every unique n-byte sequence starting from the left, where n is the length of the array element. In this example, the unrolled `NodeFragList` is $[\langle a_3, 0, 4\rangle, \langle a_4, 0, 4\rangle, \langle a_5, 0, 4\rangle, \langle a_6, 0, 4\rangle, \ldots, \langle a_5, 0, 4\rangle, \langle a_6, 0, 4\rangle]$. The set of unique 8-byte `NodeFragLists` has two ordered lists: $\{[\langle a_3, 0, 4\rangle, \langle a_4, 0, 4\rangle], [\langle a_5, 0, 4\rangle, \langle a_6, 0, 4\rangle]\}$.

$$\begin{array}{rc} & [\langle i_1, 0, 40\rangle] \\ \texttt{GetChildren(pts)} & \Downarrow \\ & [\langle a_3, 0, 4\rangle, \langle a_4, 0, 4\rangle, \langle i_2, 0, 32\rangle] \\ \texttt{GetRange(0,39)} & \Downarrow \\ & [\langle a_3, 0, 4\rangle, \langle a_4, 0, 4\rangle, \langle i_2, 0, 32\rangle] \\ \texttt{GetArrayElements(5)} & \Downarrow \\ & [\langle a_3, 0, 4\rangle, \langle a_4, 0, 4\rangle], \\ & [\langle a_5, 0, 4\rangle, \langle a_6, 0, 4\rangle] \\ \texttt{GetRange(0,3)} & \Downarrow \\ & [\langle a_3, 0, 4\rangle], \\ & [\langle a_5, 0, 4\rangle] \end{array}$$

■

EXAMPLE 4.4. Lookup `pts[8:37]\5[0:5]`

This example shows a slightly complicated case of the `GetArrayElements` operation. Unrolling of $[\langle i_2, 0, 30\rangle]$ results in four distinct representations for 6-byte array

elements, namely, $[\langle a_5, 0, 4\rangle,\ \langle a_6, 0, 2\rangle]$, $[\langle a_6, 2, 2\rangle,\ \langle a_5, 0, 4\rangle]$, $[\langle a_6, 0, 4\rangle,\ \langle a_5, 0, 2\rangle]$, and $[\langle a_5, 2, 2\rangle,\ \langle a_6, 0, 4\rangle]$.

$$
\begin{array}{rc}
 & [\langle i_1, 0, 40\rangle] \\
\texttt{GetChildren(pts)} & \Downarrow \\
 & [\langle a_3, 0, 4\rangle, \langle a_4, 0, 4\rangle, \langle i_2, 0, 32\rangle] \\
\texttt{GetRange(8, 37)} & \Downarrow \\
 & [\langle i_2, 0, 30\rangle] \\
\texttt{GetArrayElements(5)} & \Downarrow \\
 & [\langle a_5, 0, 4\rangle, \langle a_6, 0, 2\rangle],\ [\langle a_6, 2, 2\rangle, \langle a_5, 0, 4\rangle], \\
 & [\langle a_6, 0, 4\rangle, \langle a_5, 0, 2\rangle],\ [\langle a_5, 2, 2\rangle, \langle a_6, 0, 4\rangle] \\
\texttt{GetRange(0, 5)} & \Downarrow \\
 & [\langle a_5, 0, 4\rangle, \langle a_6, 0, 2\rangle],\ [\langle a_6, 2, 2\rangle, \langle a_5, 0, 4\rangle], \\
 & [\langle a_6, 0, 4\rangle, \langle a_5, 0, 2\rangle],\ [\langle a_5, 2, 2\rangle, \langle a_6, 0, 4\rangle]
\end{array}
$$

■

Handling an access to a part of an a-loc. The abstract transformers for VSA as shown in Fig. 4 handle partial updates to a-locs (i.e., updates to *parts* of an a-loc) very imprecisely. For instance, the abstract transformer for "$*(\texttt{R1} + c_1) = \texttt{R2} + c_2$" in Fig. 4 sets the value-sets of all the partially accessed a-locs to $\top^{vs}$. Consider "`pts[0:1] = 0x10`".[15] The lookup operation for `pts[0:1]` returns $[\langle a_3, 0, 2\rangle]$, where $\langle a_3, 0, 2\rangle$ refers to the first two bytes of a_3. The abstract transformer from Fig. 4 "gives up" (because only part of a_3 is affected) and sets the value-set of a_3 to $\top^{vs}$, which would lead to imprecise results. Similarly, a memory read that only accesses a part of an a-loc is treated conservatively as a load of $\top^{vs}$ (cf. case 3 of Fig. 4). The abstract transformers for VSA are modified as outlined below to handle partial updates and partial reads more precisely.

The value-set domain [Reps et al. 2006] provides bit-wise operations such as bit-wise and ($\&^{vs}$), bit-wise or ($|^{vs}$), left shift ($\ll^{vs}$), right shift ($\gg^{vs}$), etc. We use these operations to adjust the value-set associated with an a-loc when a partial update has to be performed during VSA. Assuming that the underlying architecture is little-endian, the abstract transformer for "`pts[0:1] = 0x10`" updates the value-set associated with a_3 as follows:

$\mathsf{ValueSet}'(a_3) = (\mathsf{ValueSet}(a_3)\ \&^{vs}\ 0\text{xffff0000})\ |^{vs}\ (0\text{x10})$.

(A read that only accesses a part of an a-loc is handled in a similar manner.)

4.7 Hierarchical A-locs

The iteration of ASI and VSA can over-refine the memory-regions. For instance, suppose that the 4-byte a-loc a_3 in Fig. 15(b) used in some round i is partitioned into two 2-byte a-locs, namely, $a_3.0$, and $a_3.2$ in round $i+1$. This sort of over-refinement can affect the results of VSA; in general, because of the properties of strided-intervals, a 4-byte value-set reconstructed from two adjacent 2-byte a-locs can be less precise than if the information was retrieved from a 4-byte a-loc. For instance, suppose that at some instruction `S`, a_3 holds either 0x100000 or 0x110001. In round i, this information is exactly represented by the 4-byte strided interval 0x10001[0x100000, 0x110001] for a_3. On the other hand, the same set of numbers can only be over-approximated by two 2-byte strided intervals, namely,

[15] Numbers that start with "0x" are in C hexadecimal format.

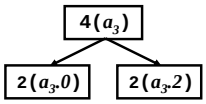


Fig. 18. Hierarchical a-locs.

1[0x0000, 0x0001] for a_3.0, and 0x1[0x10,0x11] for a_3.2 (for a little-endian machine). Consequently, if a 4-byte read of a_3 in round $i+1$ is handled by reconstituting a_3's value from a_3.0 and a_3.2, the result would be less precise:

$$\begin{aligned}\mathsf{ValueSet}(a_3) &= (\mathsf{ValueSet}(a_3.2) \ll^{vs} 16)|^{vs}\mathsf{ValueSet}(a_3.0)\\ &= \{\text{0x100000, 0x100001, 0x110000, 0x110001}\}\\ &\supset \{\text{0x100000, 0x110001}\}.\end{aligned}$$

We avoid the effects of over-refinement by keeping track of the value-sets for a-loc a_3 as well as a-locs a_3.0 and a_3.2 in round $i+1$. Whenever any of a_3, a_3.0, and a_3.2 is updated during round $i+1$, the overlapping a-locs are updated as well. For example, if a_3.0 is updated then the first two bytes of the value-set of a-loc a_3 are also updated (for a little-endian machine). For a 4-byte read of a_3, the value-set returned would be 0x10001[0x100000, 0x110001].

In general, if an a-loc a of length ≤ 4 gets partitioned into a sequence of a-locs $[a_1, a_2, \ldots, a_n]$ during some round of ASI, in the subsequent round of VSA, we use a as well as $\{a_1, a_2, \ldots, a_n\}$. We also remember the parent-child relationship between a and the a-locs in $\{a_1, a_2, \ldots, a_n\}$ so that we can update a whenever any of the a_i is updated during VSA and vice versa. In our example, the ASI tree used for round $i+1$ of VSA is identical to the tree in Fig. 15(b), except that the node corresponding to a_3 is replaced with the tree shown in Fig. 18.

One of the sources of over-refinement is the use of union types in the program. The use of hierarchical a-locs allows at least some degree of precision to be retained in the presence of unions.

4.8 Pragmatics

ASI takes into account the accesses and data transfers involving memory, and finds a partition of the memory-regions that is consistent with these transfers. However, from the standpoint of accuracy of VSA and its clients, it is not always beneficial to take into account all possible accesses:

—VSA might obtain a very conservative estimate for the value-set of a register (say `R`). For instance, the value-set for `R` could be $\top^{vs}$, meaning that register `R` can possibly hold all addresses and numbers. For a memory operand `[R]`, we do not want to generate ASI references that refer to each memory-region as an array of 1-byte elements.

—Some compilers initialize the local stack frame with a known value to aid in debugging uninitialized variables at runtime. For instance, some versions of the Microsoft Visual Studio compiler initialize all bytes of a local stack frame with the value 0xC. The compiler might do this initialization by using a `memcpy`. Generating ASI references that mimic `memcpy` would cause the memory-region associated with this procedure to be broken down into an array of 1-byte elements, which is not desirable.

To deal with such cases, some options are provided to tune the analysis:

—The user can supply an integer threshold. If the number of memory locations that are accessed by a memory operand is above the threshold, no ASI reference is generated.

—The user can supply a set of instructions for which ASI references should not be generated. One possible use of this option is to suppress `memcpy`-like instructions.

—The user can supply explicit references to be used during ASI.

4.9 Experiments

In this section, we present the results of our experiments, which were designed to answer the following questions:

(1) How do the a-locs identified by abstraction refinement compare with the program's debugging information? This provides insight into the usefulness of the a-locs recovered by our algorithm for a human analyst.

(2) How much more useful for static analysis are the a-locs recovered by an abstract-interpretation-based technique when compared to the a-locs recovered by purely local techniques?

In this section, we highlight the important results from the experiments; the experiments are presented in more detail in Balakrishnan and Reps [2007], Balakrishnan [2007, Ch. 5], and Reps and Balakrishnan [2008].

The experiments were carried out on a 32-bit desktop equipped with an Intel P4 3.0 GHz processor and 4 GB of physical memory, running Windows XP.

4.9.1 *Comparison of A-locs with Program Variables.* To measure the quality of the a-locs identified by the abstraction-refinement algorithm, we used a set of C++ benchmarks collected from Aigner and Hölzle [1996] and Pande and Ryder [1996]. These programs make heavy use of inheritance and virtual functions, and hence are a challenging set of examples for the algorithm. We compiled the set of programs using the Microsoft VC 6.0 compiler with debugging information, and ran the a-loc-recovery algorithm on the executables produced by the compiler until the results converged. After each round of ASI, for each program variable `v` present in the debugging information, we compared `v` with the structure identified by our algorithm (which did *not* use the debugging information).

On average, our technique is successful in identifying correctly over 88% of the local variables and over 89% of the fields of heap-allocated objects (and was 100% correct for fields of heap-allocated objects in over half of the examples). In contrast, the Semi-Naïve approach recovered 83% of the local variables, but 0% of the fields of heap-allocated objects.

In most of the programs, only one round of ASI was required to identify all the fields of heap-allocated data structures correctly. In some of the programs, however, it required more than one round to find all the fields of heap-allocated data-structures. Those programs that required more than one round of ASI-VSA iteration used a chain of pointers to link structs of different types, as discussed in §4.4. Most of the example programs do not have structures that are declared local to a procedure. Consequently, the Semi-Naïve approach identified a large fraction

Program	Procedures	Instructions	n	Time
src/vdd/dosioctl/krnldrvr	70	2824	3	21s
src/general/ioctl/sys	76	3504	3	37s
src/general/tracedrv/tracedrv	84	3719	3	1m
src/general/cancel/startio	96	3861	3	26s
src/general/cancel/sys	102	4045	3	26s
src/input/moufiltr	93	4175	3	4m
src/general/event/sys	99	4215	3	31s
src/input/kbfiltr	94	4228	3	3m
src/general/toaster/toastmon	123	6261	3	5m
src/storage/filters/diskperf	121	6584	3	7m
src/network/modem/fakemodem	142	8747	3	16m
src/storage/fdc/flpydisk	171	12752	3	31m
src/input/mouclass	192	13380	2	1h 51m
src/input/mouser	188	13989	3	40m
src/kernel/serenum	184	14123	3	38m
src/wdm/1394/driver/1394diag	171	23430	3	28m
src/wdm/1394/driver/1394vdev	173	23456	3	23m
mplayer2	172	14270	2	0h 11m
smss	481	43034	3	2h 8m
print	563	48233	3	0h 20m
doskey	567	48316	3	2h 4m
attrib	566	48785	3	0h 23m
routemon	674	55586	3	2h 28m
cat	688	57505	3	0h 54m
ls	712	60543	3	1h 10m

Table I. Windows Device Drivers (top) and Executables (bottom). (n is the number of VSA-ASI rounds.)

of the local variables correctly. However, when programs had structures that are local to a procedure, our approach identifies more local variables correctly.

4.9.2 *Usefulness of the A-locs for Static Analysis.* The aim of this experiment was to evaluate the quality of the variables and values discovered as a platform for performing additional static analysis. In particular, because resolution of indirect operands is a fundamental primitive that essentially any subsequent analysis would need, the experiment measured how well we can resolve indirect memory operands not based on global addresses or stack-frame offsets (e.g., accesses to arrays and heap-allocated data objects).

We ran several rounds of VSA on the collection of commonly used Windows executables and Windows device drivers listed in Tab. I, as well as the set of C++ benchmarks mentioned in §4.9.1. The executables for the Windows device-driver examples in Tab. I were obtained by compiling the driver source code along with the harness and OS environment model used in the SDV toolkit [Ball et al. 2006]. (See §6 for more details.) For the programs from §4.9.1 and the drivers in Tab. I, we ran VSA-ASI iteration until convergence. For the executables listed in the bottom third of Tab. I, we limited the number of VSA-ASI rounds to at most three. Round 1 of VSA performs its analysis using the a-locs recovered by the Semi-Naïve approach; all subsequent rounds of VSA use the a-locs recovered by the abstraction-refinement algorithm. After the first and final rounds of VSA, we classify each memory operand as follows:

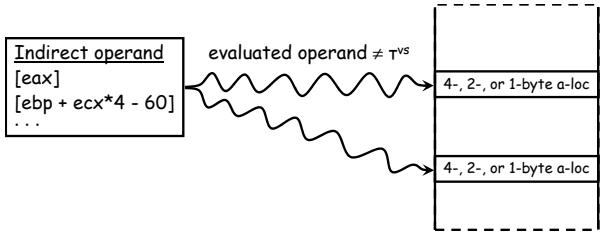


Fig. 19. Properties of a strongly-trackable memory operand.

—A memory operand is *strongly-trackable* (see Fig. 19) if
  —the lvalue evaluation of the operand does not yield $\top^{vs}$, and
  —each lvalue obtained refers to a 4-, 2-, or 1-byte a-loc.

—A memory operand is *weakly-trackable* if
  —the lvalue evaluation of the operand does not yield $\top^{vs}$, and
  —at least one of the lvalues obtained refers to a 4-, 2-, or 1-byte a-loc.

—Otherwise, the memory operand is *untrackable*; i.e., either
  —the lvalue evaluation of the operand yields $\top^{vs}$, or
  —all of the lvalues obtained refer to an a-loc whose size is greater than 4 bytes.

VSA tracks value-sets for a-locs whose size is less than or equal to 4 bytes, but treats a-locs greater than 4 bytes as having the value-set $\top^{vs}$. Therefore, untrackable memory operands are ones for which VSA provides no useful information at all, and strongly-trackable memory operands are ones for which VSA can provide useful information.

We refer to a memory operand that is used to read the contents of memory as a *use-operand*, and a memory operand that is used to update the contents of memory as a *kill-operand*. VSA can provide some useful information for a weakly-trackable kill-operand, but provides no useful information for a weakly-trackable use-operand. To understand why, first consider the kill-operand `[eax]` in "`mov [eax], 10`". If `[eax]` is weakly-trackable, then VSA may be able to update the value-set—to a value other than $\top^{vs}$—of those a-locs that are (i) accessed by `[eax]` and (ii) of size less than or equal to 4 bytes. (The value-sets for a-locs accessed by `[eax]` that are of size greater than 4 bytes already hold the value $\top^{vs}$.) In contrast, consider the use-operand `[eax]` in "`mov ebx, [eax]`"; if `[eax]` is weakly-trackable, then at least one of the a-locs accessed by `[eax]` holds the value $\top^{vs}$. In a `mov` instruction, the value-set of the destination operand (`ebx` in our example) is set to the join ($\sqcup^{vs}$) of the value-sets of the a-locs accessed by the source operand (`[eax]` in our example); consequently, the value-set of `ebx` would be set to $\top^{vs}$—which is the same as what happens when `[eax]` is untrackable.

We classified memory operands as either *direct* or *indirect*. A *direct* memory operand is a memory operand that uses a global address or stack-frame offset. An *indirect* memory operand is a memory operand that uses a non-stack-frame register (e.g., a memory operand that accesses an array or a heap-allocated data object).

Direct Memory Operands. For direct use-operands and direct kill-operands, both the Semi-Naïve approach and our abstract-interpretation-based a-loc-recovery

	Percentages of Trackable Memory Operands					
	Strongly-Trackable Indirect Uses (%)		Strongly-Trackable Indirect Kills (%)		Weakly-Trackable Indirect Kills (%)	
Test Suite	First	Final	First	Final	First	Final
C++ Examples	8	46	3	80	4	83
Windows Device Drivers	19	29	8	30	9	33
Windows Executables	2	6	6	19	6	22

Table II. Percentages of trackable memory operands in the first and final rounds. The numbers reported for each test suite are the geometric means of the percentages measured for that test suite.

algorithm perform equally well: for all three test suites, almost 100% of the direct uses and kills are strongly trackable.

Indirect Memory Operands. For indirect memory operands, the results are substantially better with the abstraction-interpretation-based method. Tab. II summarizes the results. (Note that the "Weakly-Trackable Indirect Kills" are a superset of the "Strongly-Trackable Indirect Kills".)

We were surprised to find that the Semi-Naïve approach was able to provide a small amount of useful information for indirect memory operands. On closer inspection, we found that these indirect memory operands access local or global variables that are also accessed directly elsewhere in the program. (In source-level terms, the variables are accessed both directly and via pointer indirection.) For instance, a local variable `v` of procedure `P` that is passed by reference to procedure `Q` will be accessed directly in `P` and indirectly in `Q`.

Our abstract-interpretation-based a-loc-recovery algorithm works well for the C++ examples, but the algorithm is not so successful for the Windows device-driver examples and the Windows executables. Several sources of imprecision in VSA prevent us from obtaining useful information at many of the indirect memory operands in those executables. One source of imprecision is widening [Cousot and Cousot 1977]. VSA uses a widening operator during abstract interpretation to accelerate fixpoint computation (see §3.4). Due to widening, VSA may fail to find non-trivial bounds for registers that occur in indirect memory operands that implement (source-level) array-access expressions; such indirect memory operands will be classified as untrackable.

The fact that the VSA domain is non-relational amplifies this problem. (To a limited extent, we overcome the lack of relational information by obtaining relations among the values of the x86 registers using an additional analysis called affine-relation analysis; see Balakrishnan [2007, Ch. 7] for details.) The widening problem is actually orthogonal to the issue of finding a suitable set of a-locs. Even if an a-loc-recovery algorithm were to recover all of the source-level variables exactly, imprecision due to widening would still be an issue.

In Balakrishnan [2007, Ch. 7] and Reps and Balakrishnan [2008], we described a technique, called GMOD-based merging, that increases the precision of abstract interpretation of procedure calls and also reduces the undesirable effects of widening. As shown in Tab. III, when GMOD-based merging was used for the Windows device-driver examples, the percentage of trackable memory operands in the final round improved dramatically: from 29%, 30%, and 33% to 81%, 85%, and 90%, respectively.

	Percentages of Trackable Memory Operands		
	Strongly-Trackable Indirect Uses (%)	Strongly-Trackable Indirect Kills (%)	Weakly-Trackable Indirect Kills (%)
Without GMOD-based merging	29	30	33
With GMOD-based merging	81	85	90

Table III. Percentages of trackable memory operands in the final round for the Windows device-driver examples.

These measurements show that the results of VSA are significantly better when a-locs identified using abstract-interpretation and abstraction-refinement are used in place of the a-locs identified by the Semi-Naïve algorithm, which uses purely local techniques.

5. ITERATIVE REFINEMENT IN CODESURFER/X86

This section describes the abstraction-refinement loop used in CodeSurfer/x86. The refinement loop runs repeated phases of ASI, ARA,[16] and VSA. The goal of the loop is not only to improve precision, but also to invoke the analysis components that allow CodeSurfer/x86 to overcome the lack of any initial information about a program's variables.

The order in which the analyses are applied is

$$\text{Dis ASI}_0 \text{ ARA VSA } (\text{ Dis}_{\text{DLL}} \text{ ASI ARA VSA })^n$$

where "Dis" is the disassembler phase (our implementation uses IDAPro [IDAPro]), which includes creating the initial IRs for DLLs known at link time; ASI_0 is a "minimal" ASI that uses the disassembler results as input, and implements the Semi-Naïve approach to identifying a-locs; Dis_{DLL} disassembles the DLLs discovered from arguments to `LoadLibrary` during the most recent phase of VSA, and creates initial IRs for them. $(\dots)^n$ denotes n repetitions of the parenthesized expression, where n can be controlled by the user.

The first round of VSA can uncover memory accesses that are not explicit in the program—e.g., due to operands that have forms other than "[*absolute-address*]", "[`esp` + *offset*]", and "[`ebp` − *offset*]"—which allows ASI to refine the a-locs for the next round of VSA, and may, in turn, produce more precise value-sets because it is based on a better set of a-locs. Similarly, subsequent rounds of VSA can uncover more memory accesses, and hence allow ASI to refine the a-locs. The refinement of a-locs cannot go on indefinitely because, in the worst case, an a-loc can only be partitioned into a sequence of 1-byte entities. However, in practice, the refinement process converges before the worst-case partitioning occurs.

When the data structures in a program are more complex, more rounds of the refinement loop will (in general) be required to analyze them. Roughly speaking, each round of the loop will resolve one layer of indirection in program data structures: ASI refines the a-locs that can be determined from previously-identified structures

[16] ARA refers to affine-relation analysis [Müller-Olm and Seidl 2005]. ARA is used in CodeSurfer/x86 to identify relationships among x86 machine registers. ARA is also performed over certain collections of registers, global a-locs, and local a-locs. The information from ARA is used to overcome the lack of relational information during VSA due to the non-relational nature of the VSA domain (see Lal et al. [2005] and Balakrishnan [2007, Ch. 7, Sect. 2]).

and memory accesses, and then VSA uses these a-locs to determine value sets whose contents may include pointers that ASI will use in the next round.

Determining the targets of library function calls and indirect jumps is closely related to determining the contents of memory locations, and also takes place in the refinement loop. A program with more levels of library function calls and/or indirect jumps will (in general) require more rounds of the refinement loop for full analysis than a program with fewer levels. The iteration process converges when the set of a-locs, and the set of targets for indirect function calls and indirect jumps does not change between successive rounds.

Most of the analyses in CodeSurfer/x86 are interdependent. The refinement loop is a relatively simple way to allow the results of one analysis to improve the results of the other analyses. The analyses influence one another in the following ways:
ASI → VSA : ASI results are useful to VSA in two important ways:

—The a-locs represent the collection of containers whose values will be tracked by VSA.

—The size and structure of a-locs are also important to VSA. Value sets only represent values up to 4 bytes (for a 32-bit machine). Therefore, if an a-loc is larger than 4 bytes and does not have any identifiable substructure, VSA cannot determine a value set other than "unknown" ($\top^{vs}$) to represent the values it might hold. Once ASI has subdivided such an a-loc into 1-, 2-, or 4-byte subcomponents, VSA is generally able to produce more precise value sets for the subcomponents.

VSA → ASI : ASI determines the location and structure of a-locs based on memory access patterns within the program. If a memory access is indirect, its destination must be determined before ASI can use it to identify an a-loc. VSA provides the value sets needed to compute those addresses.
ARA → VSA : If an affine relation is known to hold between two a-locs at a particular program point, VSA can make use of the relation to determine a more precise value set for one or both of the a-locs.
VSA → $\mathbf{Dis_{DLL}}$: The value sets refined by VSA may include the targets of indirect jumps and/or indirect function calls. Resolving these targets may result in new control-flow edges added to the CFG, new call edges added to the call-graph, and initial IRs created for the DLLs discovered from arguments to `LoadLibrary` during the most recent phase of VSA,
VSA and $\mathbf{Dis_{DLL}}$ → ARA : After new targets of indirect jumps and/or indirect function calls have been identified, or initial IRs have been created for newly discovered DLLs, ARA must be re-run so that this new information can be taken into account in the affine relations computed for the program's instructions.
ASI → ARA : ARA is also performed over certain collections of registers, global a-locs, and local a-locs. Because ASI refines the set of a-locs in each round, this ARA information must be recomputed for the refined set of a-locs.

6. CASE STUDY: ANALYZING WINDOWS DEVICE DRIVERS

A device driver is a program in the operating system that is responsible for managing a hardware device attached to the system. In Windows, a (kernel-level) device driver resides in the address space of the kernel, and runs at a high privilege level;

therefore, a bug in a device driver can cause the entire system to crash. The Windows kernel API [Oney 2003] requires a programmer to follow a complex set of rules: (1) a call to the functions *IoCallDriver* or *PoCallDriver* must occur only at a certain interrupt request level, (2) the function *IoCompleteRequest* should not be called twice with the same parameter, etc.

The device drivers running in a given Windows installation are one of the sources of instability in the Windows platforms: according to Swift et al. [2005], bugs in kernel-level device drivers cause 85% of the system crashes in Windows XP. Because of the complex nature of the Windows kernel API, the probability of introducing a bug when writing a device driver is high. Moreover, drivers are typically written by less-experienced or less-skilled programmers than those who wrote the Windows kernel itself.

Several approaches to improve the reliability of device drivers have been previously proposed [Ball et al. 2006; Ball and Rajamani 2001; Chou et al. 2001; Swift et al. 2005]. Swift et al. [2004; 2005] propose a runtime approach that works on executables; they isolate the device driver in a lightweight protection domain to reduce the possibility of whole-system crashes. Because their method is applied at runtime, it may not prevent all bugs from causing whole-system crashes. Other approaches [Ball et al. 2006; Ball and Rajamani 2000; 2001; Henzinger et al. 2002] are based on static program analysis of a device driver's source code. Ball et al. [2006; 2001] developed the Static Driver Verifier (SDV), a tool based on model checking to find bugs in device-driver source code. A kernel API usage rule is described as a finite-state machine (FSM), and SDV analyzes the source code for the driver to determine whether there is a path in the driver that violates the rule.

In our work, we extended the algorithms developed for CodeSurfer/x86 to create a static-analysis tool for checking properties of stripped Windows device-driver executables. With this tool, called Device-Driver Analyzer for x86 (DDA/x86), neither source code nor symbol-table/debugging information need be available (although DDA/x86 can use debugging information, such as Windows .pdb files, if it is available). Consequently, DDA/x86 can provide information that is useful in the common situation where one needs to install a device driver for which source code has not been furnished.

Microsoft has a program for signing Windows device drivers, called Windows Hardware Quality Lab (WHQL) testing. Device vendors submit driver executables to WHQL, which runs tests on different hardware platforms with different versions of Windows, reviews the results, and, if the driver passes the tests, creates a digitally signed certificate for use during installation that attests that Microsoft has performed some degree of testing. However, there is anecdotal evidence that device vendors have tried to cheat [WHQL 2004]. A tool like DDA/x86 could allow static analysis to play a role in such a certification process.

Even if you have a driver's source code (and can build an executable) and also have tools for examining executables equipped with symbol-table/debugging information, this would not address the effects of the optimizer. If you want to look for bugs in an optimized version, you would have a kind of "partially stripped" executable, due to the loss of debugging information caused by optimization. This is a situation where our techniques for analyzing stripped executables should be of assistance.

```
KeInitializeEvent(&event, NotificationEvent, FALSE);

IoSetCompletionRoutine(Irp,
                       (PIO_COMPLETION_ROUTINE) MouFilter_Complete,
                       &event,
                       TRUE,
                       TRUE,
                       TRUE); // No need for Cancel

status = IoCallDriver(devExt->TopOfStack, Irp);

if (STATUS_PENDING == status) {
    KeWaitForSingleObject(
        &event,
        Executive, // Waiting for reason of a driver
        KernelMode, // Waiting in kernel mode
        FALSE, // No alert
        NULL); // No timeout
}

if (NT_SUCCESS(status) && NT_SUCCESS(Irp->IoStatus.Status)) {
    //
    // As we are successfully now back from our start device
    // we can do work.
    //
    devExt->Started = TRUE;
    devExt->Removed = FALSE;
    devExt->SurpriseRemoved = FALSE;
}

//
// We must now complete the IRP, since we stopped it in the
// completion routine with MORE_PROCESSING_REQUIRED.
//
Irp->IoStatus.Status = status;
Irp->IoStatus.Information = 0;
IoCompleteRequest(Irp, IO_NO_INCREMENT);

break;
```

(a)

```
push 1
push 1
push 1
lea ecx, dword ptr [ebp + var_1C]
push ecx
push sub_1002270
mov edx, dword ptr [ebp + arg_4]
push edx
call dword ptr [_sdv_IoSetCompletionRoutine@24]
mov edx, dword ptr [ebp + arg_4]
mov eax, dword ptr [ebp + var_4]
mov ecx, dword ptr [eax + 8]
call dword ptr [@IofCallDriver@8]
mov dword ptr [ebp + var_20], eax
cmp dword ptr [ebp + var_20], 103h
jnz loc_10013E1
push 0
push 0
push 0
push 0
lea ecx, dword ptr [ebp + var_1C]
push ecx,
call dword ptr [_sdv_KeWaitForSingleObject@20]

loc_10013E1:
cmp dword ptr [ebp + var_20], 0
jl loc_1001405
mov edx, dword ptr [ebp + arg_4]
cmp dword ptr [edx + 18h], 0
jl loc_1001405
mov eax, dword ptr [ebp + var_4]
mov byte ptr [eax + 30h], 1
mov ecx, dword ptr [ebp + var_4]
mov byte ptr [ecx + 32h], 0
mov edx, dword ptr [ebp + var_4]
mov byte ptr [edx + 31h], 0

loc_1001405:
mov eax, dword ptr [ebp + arg_4]
mov ecx, dword ptr [ebp + var_20]
mov dword ptr [eax + 18h], ecx
mov edx, dword ptr [ebp + arg_4]
mov dword ptr [edx + 1Ch], 0
push 0
mov eax, dword ptr [ebp + arg_4]
push eax
call dword ptr [_sdv_IoCompleteRequest@8]
jmp loc_10014B0
```

(b)

Fig. 20. (a) SDV trace; (b) DDA/x86 trace. The three shaded areas in (b) correspond to those in (a).

A skeptic might question how well an analysis technique can perform on a stripped executable. §6.2 presents some quantitative results about how well the answers obtained by DDA/x86 compare to those obtained by SDV; here we will just give one example that illustrates the ability of DDA/x86 to provide information that is qualitatively comparable to the information obtained by SDV. Fig. 20 shows fragments of the witness traces reported by SDV (Fig. 20(a)) and DDA/x86 (Fig. 20(b)) for one of the examples in the test suite. Fig. 20 shows that in this case the tools report comparable information: the three shaded areas in Fig. 20(b) correspond to those in Fig. 20(a).

Although not illustrated in Fig. 20, because of the WYSINWYX phenomenon it is possible for DDA/x86 to provide higher-fidelity answers than tools for analyzing device-driver source code. In particular, compilation effects can be important if one is interested in better diagnoses of the causes of bugs, or in detecting security vulnerabilities. For instance, a Microsoft report about writing kernel-mode drivers

in C++ recommends examining ". . . the object code to be sure it matches your expectations, or at least will work correctly in the kernel environment" [WHDC 2007].

This section describes the design and implementation of DDA/x86, and presents a case study in which we used it to find problems in Windows device drivers by analyzing the drivers' stripped executables. The key idea that allows DDA/x86 to achieve a substantial measure of success was to combine VSA with the path-sensitive method of interpreting property automata from ESP [Das et al. 2002]. The resulting algorithm explores an over-approximation of the set of reachable states, and hence can verify correctness by determining that all error configurations are unreachable. The contributions of the work include

—DDA/x86 can analyze *stripped device-driver executables*, and thus provides a capability not found in previous tools for analyzing device drivers [Ball and Rajamani 2000; Henzinger et al. 2002].

—Our case study shows that this approach is viable. DDA/x86 was able to identify some known bugs (previously discovered by source-code-based analysis tools) along with useful error traces, while having a reasonably low false-positive rate: On a corpus of 17 device-driver executables, 10 were found to pass the *PendedCompletedRequest* rule (definitely no bug), 5 false positives were reported, and 2 were found to have real bugs—for which DDA/x86 supplied feasible error traces.

—We developed a novel, low-cost mechanism for instrumenting a dataflow-analysis algorithm to provide witness traces.

One of the challenges that we faced was to find ways of coping with the differences that arise when property checking is performed at the machine-code level, rather than on an IR created from source code. In particular, the domains of discourse—the alphabets of actions to which the automata respond—are different in the two situations. This issue is discussed in §6.2.

6.1 Property Checking in Executables using VSA

This section describes the extensions that we made to our IR-recovery algorithm to perform path-sensitive property checking. Fig. 21 shows an FSM that checks for violations of the memory-safety property "pointer `p` should not be dereferenced if its value is `NULL`". One approach to determining if there is a null-pointer dereference in the executable is to start from the initial state (UNSAFE) at the entry point of the executable, and find an over-approximation of the set of reachable states at each statement in the executable. This can be done by determining the states for the successors at each statement based on the transitions in the FSM that encodes the memory-safety property.

Another approach is to use abstract interpretation to determine the abstract memory configurations at each statement in the routine, and use the results to check the memory-safety property. For executables, we could use the information computed by the IR-recovery algorithms of CodeSurfer/x86. For instance, for each instruction I in an executable, VSA determines an over-approximation of the set of memory addresses and numeric values held in each register and variable when I executes.

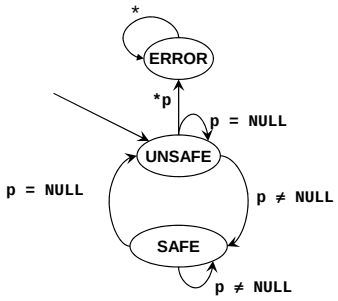


Fig. 21. An FSM that encodes the rule that pointer p should not be dereferenced if it is NULL.

Suppose that we have the results of VSA and want to use them to check the memory-safety property; the property can be checked as follows:

If the abstract set of addresses and numeric values computed for p *possibly contains* NULL *just before a statement, and the statement dereferences* p*, then the memory-safety property is potentially violated.*

Unfortunately, the approaches described above would result in a lot of false positives because they are not path-sensitive. To overcome the limitations of the two approaches described above, DDA/x86 follows Das et al. [2002] and Fischer et al. [2005], who showed how to obtain a degree of path-sensitivity by combining the propagation of automaton states with the propagation of abstract-state values during abstract interpretation. Let State denote the set of property-automaton states. The path-sensitive VSA algorithm [Balakrishnan 2007; Balakrishnan and Reps 2008] associates each program point with an $\mathsf{AbsMemConfig}^{\text{ps-cs}}$ value:

$$\begin{aligned}\mathsf{AbsMemConfig}^{\text{ps-cs}} &= \mathsf{CallString}_k \times \mathsf{State} \times \mathsf{State} \to \mathsf{AbsEnv}_\perp \\ &\simeq \mathsf{CallString}_k \to (\mathsf{State} \times \mathsf{State} \to \mathsf{AbsEnv}_\perp)\end{aligned}$$

In the pair of property-automaton states at a node n, the first component refers to the state of the property automaton at the enter node of the procedure to which node n belongs, and the second component refers to the current state of the property automaton at node n. If an AbsEnv entry for the pair $\langle cs, s_0, s_{cur}\rangle$ exists at node n, then n is possibly reachable from call-context suffix cs with the property automaton in state s_{cur} from a memory configuration at the enter node of the procedure in which the property automaton was in state s_0.

In addition to distinguishing AbsEnvs at a node based on the call-string suffix, the path-sensitive context-sensitive VSA algorithm also distinguishes AbsEnvs according to the states of the property automaton. Technically, the extension amounts to using reduced cardinal power [Cousot and Cousot 1979] of the edges in the transitive closure of the automation's transition relation and the original VSA domain; i.e., we perform context-sensitive interprocedural value-set analysis (§3.7), but use the domain $(\mathsf{State} \times \mathsf{State} \to \mathsf{AbsEnv}_\perp)$ in place of $\mathsf{AbsEnv}_\perp$ (see also Balakrishnan [2007] and Balakrishnan and Reps [2008]).

6.2 Experiments

This section presents a case study in which we used DDA/x86 to analyze the executables of Windows device drivers. The study was designed to test how well

different extensions of the VSA algorithm could detect problems in Windows device drivers by analyzing device-driver executables—without accessing source code, symbol-tables, or debugging information. In particular, if DDA/x86 were successful at finding the bugs that the Static Driver Verifier (SDV) [Ball et al. 2006; Ball and Rajamani 2001] tool finds in Windows device drivers, that would be powerful evidence that our approach is viable—i.e., that it will be possible to find previously undiscovered bugs in device drivers for which source code is not available, or for which compiler/optimizer effects make source-code analysis unsafe. We selected a subset of drivers from the Windows Driver Development Kit (DDK) [Windows DDK 2003] release 3790.1830 for the case study. For each driver, we obtained an executable by compiling the driver source code along with the harness and the OS environment model [Ball et al. 2006] of the SDV toolkit. (Thus, as in SDV and other source-code-analysis tools, the harness and OS environment models are analyzed; however, DDA/x86 analyzes the *machine code* that the compiler produces for the harness and the models. This creates certain difficulties, which are discussed below.)

A device driver is analogous to a library that exports a collection of subroutines. Each subroutine exported by a driver implements an action that needs to be performed when the OS makes an I/O request (on behalf of a user application or when a hardware-related event occurs). For instance, when a new device is attached to the system, the OS invokes the `AddDevice` routine provided by the device driver; when new data arrives on a network interface, the OS calls the `DeviceRead` routine provided by the driver; etc. For every I/O request, the OS creates a structure called the "I/O Request Packet (IRP)", which contains such information as the type of the I/O request, the parameters associated with the request, etc.; the OS then invokes the appropriate driver's dispatch routine. The dispatch routine performs the necessary actions, and returns a value that indicates the status of the request. For instance, if a driver successfully completes the I/O request, the driver's dispatch routine calls the *IoCompleteRequest* API function to notify the OS that the request has been completed, and returns the value `STATUS_SUCCESS`. Similarly, if the I/O request is not completed within the dispatch routine, the driver calls the *IoMarkPending* API function and returns `STATUS_PENDING`.

A harness in the SDV toolkit is C code that simulates the possible calls to the driver that could be made by the OS. An application generates requests, which the OS passes on to the device driver. Both levels are modeled by the harness. For the drivers used in our experiments, the harness defined in the SDV toolkit acts as a client that exercises all possible combinations of the dispatch routines that can occur in two successive calls to the driver. The harness that was used in our experiments calls the following driver routines (in the order given below):

(1) `DriverEntry`: initializes the driver's data structures and the global state.

(2) `AddDevice`: simulates the addition of a device to the system.

(3) The plug-and-play dispatch routine (called with an `IRP_MN_START_DEVICE` I/O request packet): simulates the starting of the device by the OS.

(4) Some dispatch routine, deferred procedure call, interrupt service routine, etc.: simulates various actions on the device.

(5) The plug-and-play dispatch routine (called with an `IRP_MN_REMOVE_DEVICE` I/O request packet): simulates the removal of the device by the OS.

(6) `Unload`: simulates the unloading of the driver by the OS.

The OS environment model in the SDV toolkit consists of a collection of functions (written in C) that conservatively model the API functions in the Windows DDK. The models are conservative in the sense that they simulate all possible behaviors of an API function. For instance, if an API function `Foo` returns the value 0 or 1 depending upon the input arguments, the model for `Foo` consists of a nondeterministic `if` statement that returns 0 in the true branch and 1 in the false branch. Modeling the API functions conservatively enables a static-analysis tool to explore all possible behaviors of the API.

Adapting the SDV Harness and OS Models. The harness and OS models obtained from the SDV toolkit are intended to be used by a particular source-level analyzer [Ball and Rajamani 2001] whose abstract domain is based on predicate abstraction [Graf and Saïdi 1997]. Such domains have limitations on their precision, and hence it is not necessary for SDV to have harnesses and OS models that are entirely faithful to the source-level semantics. In contrast, we needed a harness and OS models that could be compiled—and used in compiled form—with the various different abstract domains incorporated in DDA/x86. DDA/x86's domains also have limitations on their precision, but they are different than those of the domain used by SDV. Consequently, we had to make some changes to the harness and OS models obtained from SDV.

For instance, each driver has a device-extension structure that is used to maintain extended information about the state of each device managed by the driver. The number of fields and the type of each field in the device-extension structure is specific to a driver. However, in SDV's OS model, a single integer variable is used to represent the device-extension object. Therefore, in a driver executable built using SDV's models, when the driver writes to a field at offset o of the device-extension structure, it would appear as a write to the memory address that is offset o bytes from the memory address of the integer that represents the device-extension object.

SDV's OS models use a function named `SdvMakeChoice` to represent nondeterministic choice. However, the body of `SdvMakeChoice` contains just a single statement: "`return 0;`".[17] Consequently, instead of exploring all possible behaviors of an API function, DDA/x86 would explore only a subset of the behaviors of the API function. We had to modify SDV's OS environment model to avoid such problems.

Case Study. We chose the following "*PendedCompletedRequest*" rule for our case study:

> *A driver's dispatch routine should not return `STATUS_PENDING` on an I/O Request Packet (IRP) if it has called IoCompleteRequest on the IRP, unless it has also called IoMarkIrpPending.*

Fig. 22 shows the FSM for this rule.[18]

[17] According to T. Ball [2006], the C front end used by SDV treats `SdvMakeChoice` specially.

[18] According to the Windows DDK documentation, *IoMarkPending* has to be called before *Io-*

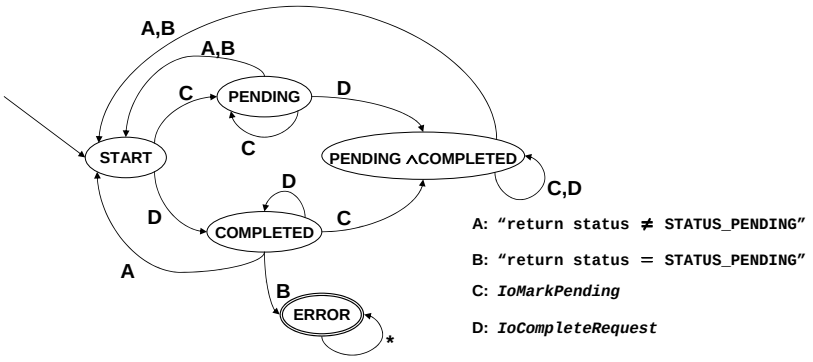


Fig. 22. Finite-state machine for the rule *PendedCompletedRequest*.

Config.	A-locs	Property Automaton
⊖	IDAPro-based algorithm	Fig. 22
◎	ASI-based algorithm	Fig. 22
★	ASI-based algorithm	Cross-product of the automata in Figs. 22 and 24

Table IV. Variants of the VSA algorithm used in the experiments.

We used the three different variants of the VSA algorithm listed in Tab. IV for our experiments; Tab. V presents the results. The experiments were carried out on a Dell Precision 490 Desktop, equipped with a 64-bit Intel Xeon 5160 3.0 GHz dual core processor and 16GB of physical memory, running Windows XP. (Although the machine has 16GB of physical memory, the size of the per-process virtual user-address space for a 32-bit application is limited to 4GB.) The column labeled "Result" indicates whether the VSA algorithm reported that there is some node n at which the `ERROR` state in the *PendedCompletedRequest* FSM is reachable, when one starts from the initial memory configuration at the entry node of the executable.

Configuration '⊖' uses an algorithm that is similar to the one used in IDAPro to recover variable-like entities. That algorithm does not provide variables of the right granularity and expressiveness, and therefore, not surprisingly, configuration '⊖' reports many false positives for all of the drivers.[19]

Configuration '◎', which uses only the *PendedCompletedRequest* FSM, also reports a lot of false positives. Fig. 23 shows an example that illustrates one of the reasons for the false positives in configuration '◎'. As shown in the right column of Fig. 23, the set of values for `status` at the return statement (`P3`) for the property-automaton state `COMPLETED` contains both `STATUS_PENDING` and `STATUS_SUCCESS`. Therefore, VSA reports that the dispatch routine possibly violates the *PendedCompletedRequest* rule. The problem is as follows: because the state of the *PendedCompletedRequest* automaton is the same after both branches of the `if` statement at `P1`

CompleteRequest; however, the FSM defined for the rule in SDV is the one shown in Fig. 22. We used the same FSM for our experiments.

[19] In this case, a false positive reports that the `ERROR` state is (possibly) reachable at some node n, when, in fact, it is never reachable. This is sound (for the reachability question), but imprecise.

			⊖		◎		★			
Driver	Procedures	Instructions	Result	Feasible Trace?	Result	Feasible Trace?	Result	Feasible Trace?	Time	Rounds
src/vdd/dosioctl/krnldrvr	70	2824	FP	-	√	-	√	-	14s	2
src/general/ioctl/sys	76	3504	FP	-	√	-	√	-	13s	2
src/general/tracedrv/tracedrv	84	3719	FP	-	√	-	√	-	16s	2
src/general/cancel/startio	96	3861	FP	-	√	-	√	-	12s	2
src/general/cancel/sys	102	4045	FP	-	√	-	√	-	10s	2
src/input/moufiltr	93	4175	×	No	×	No	×	Yes	3m 3s	5
src/general/event/sys	99	4215	FP	-	√	-	√	-	20s	2
src/input/kbfiltr	94	4228	×	No	×	No	×	Yes	2m 53s	5
src/general/toaster/toastmon	123	6261	FP	-	FP	-	√	-	4m 1s	3
src/storage/filters/diskperf	121	6584	FP	-	FP	-	√	-	3m 17s	3
src/network/modem/fakemodem	142	8747	FP	-	FP	-	√	-	11m 6s	3
src/storage/fdc/flpydisk	171	12752	FP	-	FP	-	FP	-	1h 6m	5
src/input/mouclass	192	13380	FP	-	FP	-	FP	-	40m 26s	5
src/input/mouser	188	13989	FP	-	FP	-	FP	-	1h 4m	5
src/kernel/serenum	184	14123	FP	-	FP	-	√	-	19m 41s	2
src/wdm/1394/driver/1394diag	171	23430	FP	-	FP	-	FP	-	1h33m	5
src/wdm/1394/driver/1394vdev	173	23456	FP	-	FP	-	FP	-	1h38m	5

Table V. Results of checking the *PendedCompletedRequest* rule in Windows device drivers. (√: passes rule; ×: a real bug found; FP: false positive.) See Tab. IV for an explanation of ⊖, ◎, and ★. (For the examples that pass the rule, "Rounds" represents the number of VSA-ASI rounds required to prove the absence of the bug; for the other examples, the maximum number of rounds was set to 5.)

```
int dispatch_routine(...) {
  int status, c = 0;
  ⋮
  status = STATUS_PENDING;
  P1:if(...) {
    status = STATUS_SUCCESS;
    c = 1;
  }
  P2:
  ⋮
  if(c == 1) {
    IoCompleteRequest(...)
  }
  P3:  return status;
}
```

Information at P3 with the FSM shown in Fig. 22
START:
c ↦ {0, 1}
status ↦ {STATUS_SUCCESS, STATUS_PENDING}
COMPLETED:
c ↦ {0, 1}
status ↦ {STATUS_SUCCESS, STATUS_PENDING}

Information at P3 with the FSM shown in Fig. 24
ST_PENDING:
c ↦ {0}
status ↦ {STATUS_PENDING}
ST_NOT_PENDING:
c ↦ {1}
status ↦ {STATUS_SUCCESS}

Fig. 23. An example illustrating false positives in device-driver analysis.

are analyzed, VSA merges the information from both of the branches, and therefore the correlation between `c` and `status` is lost after the statement at `P2`.

Fig. 24 shows an FSM that enables VSA to maintain the correlation between `c` and `status`. Basically, the FSM changes the abstraction in use, and enables VSA to distinguish paths in the executable based on the contents of the variable `status`. We refer to a variable (such as `status` in Fig. 24) that is used to keep track of the current status of the I/O request in a dispatch routine as the *status-variable*. To be able to use the FSM in Fig. 24 for analyzing an executable, it is necessary to determine the status-variable for each procedure. However, because debugging information is usually not available, we use the following heuristic to identify the status-variable for each procedure in the executable:

By convention, `eax` *holds the return value in the x86 architecture. Therefore, the a-loc (if any) that is used to initialize the value of* `eax` *just before returning from the dispatch routine is considered to be the status-variable.*

Configuration '★' uses the automaton obtained by combining the *PendedCompletedRequest* FSM and the FSM shown in Fig. 24 (instantiated using the above heuristic) using a cross-product construction. As shown in Tab. V, for configuration '★', the number of false positives is substantially reduced.

It required substantial manual effort to find an abstraction that had sufficient fidelity to reduce the number of false positives reported by DDA/x86. To create a practical tool, it would be important to automate the process of refining the abstraction based on the property to be checked. The model-checking community has developed many techniques that could be applicable, although the discussion above shows that the definition of a suitable refinement can be quite subtle.

As a point of comparison, SDV also found the bugs in both "moufiltr" and "kbfiltr", but had no false positives in any of the examples. However, one should not leap to the conclusion that machine-code-analysis tools are necessarily inferior to source-code-analysis tools.

—The basic capabilities are different: DDA/x86 can analyze stripped device-driver executables, which goes beyond the capabilities of SDV.

—The analysis techniques used in SDV and in DDA/x86 are incomparable: SDV uses predicate-abstraction-based abstractions [Graf and Saïdi 1997], plus abstraction refinement; DDA/x86 uses a combined numeric-plus-pointer analysis (VSA), together with a different kind of abstraction refinement (iteration of ASI and VSA). Thus, there may be examples for which DDA/x86 outperforms SDV.

Moreover, SDV is a multiple man-year effort, with a professional team at Microsoft devoted to its development. In contrast, the prototype DDA/x86 was created in only a few man-months (although multiple man-years went into building the underlying CodeSurfer/x86 infrastructure).

Property Automata for the Analysis of Machine Code. Property automata for the analysis of machine code differ from the automata used for source-level analysis. In particular, the domain of discourse—the alphabet of actions to which an automaton responds—is different when property checking is performed at the machine-code level, rather than on an IR created from source code.

In some cases, it is possible to recognize a source-level action based on information available in the recovered IR. For instance, a source-code procedure call with actual parameters is usually implemented as a sequence of instructions that evaluate the actuals, followed by a `call` instruction to transfer control to the starting address of the procedure. The IR-recovery algorithms used in CodeSurfer/x86 will identify the call along with its arguments.

In other cases, a source-level action is not identifiable. One contributing factor is that a source-level action can correspond to a sequence of instructions. Moreover, the instruction sequences for two source-level actions could be interleaved. We did not have a systematic way to cope with such problems except to rewrite the automaton of interest based on instruction-level actions.

Fortunately, most of the instruction-level actions that need to be tracked boil down to memory accesses/updates. Because VSA is precise enough to interpret many memory accesses (§4.9), it is possible for DDA/x86 to perform property checking using the extended version of VSA sketched in §6.1 [Balakrishnan 2007; Balakrishnan and Reps 2008]. In our somewhat limited experience, we found that

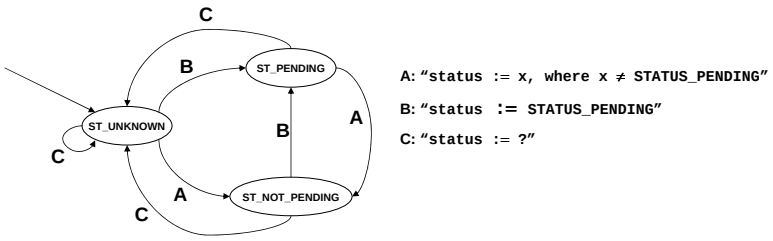


Fig. 24. Finite-state machine that tracks the contents of the variable `status`.

for many property automata it is possible to rewrite them based on memory accesses/updates so that they can be used for the analysis of executables.

Finding a Witness Trace. If the VSA algorithm reports that the `ERROR` state in the property automaton is reachable, it is useful to find a sequence of instructions that shows how the property automaton can be driven to `ERROR`. Rather than extending the VSA implementation to generate and manage explicitly the information required for reporting witness traces, we exploited the fact that the standard algorithms for solving reachability problems in pushdown systems (PDSs) [Bouajjani et al. 1997; Finkel et al. 1997] provide a witness-trace capability to show how a given (reachable) configuration is reachable.

The algorithm sketched in §6.1 was augmented to emit the rules of a PDS on-the-fly. The PDS constructed is equivalent to a PDS that would be obtained by a cross-product of the property automaton and a PDS that models the interprocedural control-flow graph, except that, by emitting the PDS on-the-fly as VSA variant '★' is run, the cross-product PDS is pruned according to what the VSA algorithm and the property automaton both agree on as being reachable. The PDS is constructed as follows:

PDS rules	Control flow modeled
$\langle q, [n_0, s]\rangle \hookrightarrow \langle q, [n_1, s']\rangle$	Intraprocedural CFG edge from node n_0 in state s to node n_1 in state s'
$\langle q, [c, s]\rangle \hookrightarrow \langle q, [enter_{\mathtt{P}}, s][r, s']\rangle$ $\langle q_{[x_{\mathtt{P}}, s']}, [r, s']\rangle \hookrightarrow \langle q, [r, s']\rangle$	Call to procedure `P` from c in state s that returns to r in state s'.
$\langle q, [x_{\mathtt{P}}, s']\rangle \hookrightarrow \langle q_{[x_{\mathtt{P}}, s']}, \epsilon\rangle$	Return from `P` at exit node $x_{\mathtt{P}}$ in state s'

In our case, to obtain a witness trace, we merely use the witness trace returned by the PDS reachability algorithm to determine if a PDS configuration $\langle q, [n, \mathtt{ERROR}]\rangle$—where n is a node in the interprocedural CFG—is reachable from the configuration $\langle q, \mathtt{enter}_{\mathtt{main}}\rangle$.

Because the PDS used for reachability queries is based on the results of VSA, which computes an *over-approximation* of the set of reachable concrete memory states, the witness traces provided by the reachability algorithm may be infeasible. In our experiments, only for configuration '★' were the witness traces for `kbfiltr` and `moufiltr` feasible. (Feasibility was checked by hand.)

This approach is not specific to VSA; it can be applied to essentially any worklist-based dataflow-analysis algorithm when it is extended with a property automaton, and provides a conceptually low-cost mechanism for augmenting such algorithms to provide witness traces.

7. RELATED WORK

To confine the scope of the paper, we have not discussed several additional techniques that are used in CodeSurfer/x86:

—The use of affine relations [Müller-Olm and Seidl 2005] over registers to obtain more precise value-sets for registers used in a loop. In particular, if VSA identifies constraints on the value of a register that is used as a loop-index variable, affine relations over registers can be used to propagate these constraints to other registers used in the loop (see Lal et al. [2005] and Balakrishnan [2007, Ch. 7, Sect. 2]).

—The use of affine relations over registers, global a-locs, and local a-locs to find more precise value-sets for registers, global a-locs, and local a-locs used in a loop.

—A technique, called GMOD-based merging ([Balakrishnan 2007, Ch. 7] and [Reps and Balakrishnan 2008]), that increases the precision of abstract interpretation of procedure calls.

—An abstraction of heap-allocated storage, called the *recency abstraction* [Balakrishnan and Reps 2006]. This involves using more than one memory-region per call-site on `malloc`, and overcomes some of the imprecision that arises due to the need to perform weak updates—i.e., accumulate information via join—on fields of summary malloc-regions.

Information About Memory Accesses in Executables. There is an extensive body of work on techniques to obtain information from executables by means of static analysis, including [Amme et al. 2000; Backes 2004; Bergeron et al. 2001; Bergeron et al. 1999; Cifuentes and Fraboulet 1997a; 1997b; Cifuentes et al. 1998; Debray et al. 1998; Guo et al. 2005; Larus and Schnarr 1995; Mycroft 1999]. However, previous work on analyzing memory accesses in 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. VSA does a much better job than previous work because it tracks the integer-valued and address-valued quantities that the program's data objects can hold; in particular, VSA tracks the values of data objects other than just the hardware registers, and thus is not forced to give up all precision when a load from memory is encountered.

The work that is most closely related to VSA is the alias-analysis algorithm for executables proposed by Debray et al. [1998]. The basic goal of the algorithm proposed by Debray et al. is similar to that of VSA: for them, it is to find an over-approximation of the set of values that each *register* can hold at each program point; for us, it is to find an over-approximation of the set of values that each (abstract) data object can hold at each program point, where data objects include *memory locations* in addition to registers. In their analysis, a set of addresses is approximated by a set of congruence values: they keep track of only the low-order bits of addresses. However, unlike VSA, their algorithm does not make any effort to track values that are not in registers. Consequently, they lose a great deal of precision whenever there is a load from memory.

The two other pieces of work that are closely related to VSA are the algorithm for data-dependence analysis of assembly code of Amme et al. [2000] and the algorithm for pointer analysis on a low-level intermediate representation of Guo et al. [2005].

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 [Wilson and Lam 1995] to achieve context-sensitivity. The transfer functions are parameterized by "unknown initial values" (UIVs); however, it is not clear whether the algorithm accounts for the possibility of called procedures corrupting the memory locations that the UIVs represent.

The xGCC tool [Backes 2004] 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 $\mathbf{4}[\mathbf{1020}, \mathbf{1028}]$, 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.

Static analysis of machine code is also a key component of tools to bound the worst-case execution time (WCET) of programs [Wilhelm et al. 2008]. The execution time of a program depends upon multiple factors, including the time required for each instruction, the number of times an instruction executes, the structure of the pipeline in the architecture, the number of cache misses at each instruction, etc. To bound the WCET of a program as precisely as possible, it is necessary to gather information about the possible execution behaviors of the program. To analyze data-cache behavior—in particular, to determine how a given memory access in an instruction can change the state of the cache (which is used to determine

whether an access is always a cache hit or could be a cache miss)—an analyzer needs information about what concrete addresses could be read from or written to.

Ferdinand et al. [2001] present a value-analysis algorithm to determine the contents of registers and memory locations. Because the algorithm of Ferdinand et al. needs concrete addresses to be able to track the state of the data cache, their problem is mostly incompatible with the techniques that we use in the VSA algorithm (§3)—i.e., tracking the values of a-locs (§2.2 and §4) in terms of an abstract model of memory (§2.1). As discussed in §2.1, the use of concrete memory addresses instead of a-locs can sometimes be problematic for accesses to local variables.

In Ferdinand et al. [2001], the abstract domain for representing a set of concrete addresses is based on intervals. As discussed above, an abstract arithmetic based solely on intervals does not provide enough information to check for non-aligned accesses, which is why VSA is based on strided intervals (see §3.1 and [Balakrishnan and Reps 2004; Reps et al. 2006]). Since 2006 [Ferdinand 2009], the aiT tool [aiT] has used an abstract domain similar to our strided-interval domain [Grewe 2008].

Xu et al. [2000; 2001] created a system that used theorem-proving techniques to analyze executables in the absence of symbol-table and/or debugging information. The goal of their system was to establish whether or not certain memory-safety properties held in SPARC executables. Similarly, there has been other work based on logic to deal with self-modifying code [Cai et al. 2007], embedded code pointers [Ni and Shao 2006], aliases [Brumley and Newsome 2006], and stack-based control abstractions [Feng et al. 2006].

Decompilation. Past work on decompiling assembly code to a high-level language [Cifuentes et al. 1998; Chang et al. 2006] is also peripherally 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, the work of Cifuentes et al. [1998] primarily concentrates on recovery of (a) expressions from instruction sequences, and (b) control flow. We believe that decompilers would benefit from the memory-access-analysis method described in this paper, which can be performed prior to decompilation proper, to recover information about numeric values, address values, physical types, and definite links from objects to virtual-function tables [Balakrishnan and Reps 2006]. By providing methods that expose a rich source of information about the way data is laid out and accessed in executables, our work raises the bar on what should be expected from a future best-of-breed decompilation tool.

Analysis of Source Code. Dor et al. [2003] present a static-analysis technique—implemented for programs written in C—whose aim is to identify string-manipulation errors, such as potential buffer overruns. In their work, they use a flow-insensitive pointer analysis followed by a linear-relation analysis [Cousot and Halbwachs 1978] to identify potential buffer overruns in string-manipulation operations. Rugina and Rinard [2005] have also used a combination of pointer and numeric analysis to determine information about a program's memory accesses.

There are several reasons why these algorithms were not suitable for the problem that we faced. In our work, we are interested in discovering fine-grained information about the structure of memory-regions. As already discussed in §3.1, it is important for the analysis to discover alignment and stride information so that it can interpret indirect-addressing operations that implement field-access operations in an array

of structs or pointer-dereferencing operations. Because we need to represent non-convex sets of numbers, linear-relation analysis is not appropriate. For this reason, the numeric component of VSA is based on strided intervals, which are capable of representing certain non-convex sets of integers.

Our analysis *combines* pointer analysis with numeric analysis, whereas the analyses of Rugina and Rinard [2005] and Dor et al. [2003] use two separate phases: pointer analysis *followed by* numeric analysis. An advantage of combining the two analyses is that information about numeric values can lead to improved tracking of pointers, and pointer information can lead to improved tracking of numeric values. In our context, this kind of positive interaction is important for discovering alignment and stride information (cf. §3.1). Moreover, additional benefits can accrue to clients of VSA; for instance, it can happen that extra precision will allow VSA to identify that a strong update, rather than a weak update, is possible (i.e., an update can be treated as a kill rather than as a possible kill; cf. case two of Fig. 4). The advantages of combining pointer analysis with numeric analysis have been studied by Pioli and Hind [1999]. In the context studied by Pioli and Hind, combining the two analyses only improves precision. As discussed at the beginning of §3, a combined analysis is essential because numeric and address-dereference operations are inextricably intertwined in even simple instructions, such as one that loads a local variable into a register: `mov eax,[ebp-12]`.

Analysis in the Presence of Additional Information. Several platforms have been created for manipulating executables in the presence of additional information, such as source code and debugging information, including ATOM [Srivastava and Eustace 1994], EEL [Larus and Schnarr 1995], Phoenix [Phoenix], and Vulcan [Srivastava et al. 2001]. Several people have also developed techniques to analyze executables in the presence of such additional information [Bergeron et al. 2001; Bergeron et al. 1999; Rival 2003]. Analysis techniques that assume access to such information are limited by the fact that it must not be relied on when dealing with programs such as viruses, worms, and mobile code (even if such information is present).

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 [Ramalingam et al. 1999]. A similar algorithm was devised by Eidorff et al. [1999] 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 [Mycroft 1999]. 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 information about interprocedural side-effects, strides, and sizes of arrays. Furthermore, Mycroft excludes from consideration programs in which addresses of local variables are taken. This is a problematic restriction 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. It should be possible to make use of Mycroft's techniques in conjunction with those used in CodeSurfer/x86. In particular, some of the issues discussed above could be addressed using information obtained by the techniques described in this paper.

Miné [2006] describes a combined data-value and points-to analysis that, at each program point, partitions the variables in the program into a collection of cells according to how they are accessed, and computes an over-approximation of the values in these cells. Miné's algorithm is similar in flavor to the VSA-ASI iteration scheme in that Miné finds his own variable-like quantities for static analysis. However, Miné's partitioning algorithm is still based on the set of variables in the program (which our algorithm assumes will not be available). His implementation does not support analysis of programs that use heap-allocated storage. Moreover, his techniques are not able to infer from loop-access patterns—as ASI can—that an unstructured cell (e.g., `unsigned char z[32]` has internal array substructures, (e.g., `int y[8];` or `struct {int a[3]; int b;} x[2];`).

In Miné's work, cells correspond to variables. The algorithm assumes that each variable is disjoint and is not aware of the relative positions of the variables. Instead, his algorithm issues an alarm whenever an indirect access goes beyond the end of a variable. Because our abstraction of memory is in terms of memory-regions (which can be thought of as cells for entire activation records), we are able to interpret an out-of-bound access precisely in most cases. For instance, suppose that two integers `a` and `b` are laid out next to each other. Consider the sequence of C statements "`p = &a; *(p+1) = 10;`". For the access `*(p+1)`, Miné's implementation issues an out-of-bounds access alarm, whereas we are able to identify that it is a write to variable `b`. (Such out-of-bounds accesses occur commonly during VSA because the a-loc-recovery algorithm can split a single source-level variable into more than one a-loc, e.g., array `pts` in Ex. 2.1.)

DDA/x86. DDA/x86 is the first known application of program analysis/verification techniques to stripped industrial executables. Among other techniques, it combines the IR-recovery algorithms from CodeSurfer/x86 [Balakrishnan 2007; Balakrishnan and Reps 2004; 2007] with the path-sensitive method of interpreting property automata from ESP [Das et al. 2002].

A number of algorithms have been proposed in the past for verifying properties of programs when source code is available [Ball et al. 2006; Ball and Rajamani

2001; Blanchet et al. 2003; Das et al. 2002; Fischer et al. 2005; Henzinger et al. 2002]. Among these techniques, SDV [Ball et al. 2006; Ball and Rajamani 2001] and ESP [Das et al. 2002] are closely related to DDA/x86. SDV builds a Boolean representation of the program using predicate abstraction; it reports a possible property violation if an error state is reachable in the Boolean model. In contrast, DDA/x86 uses value-set analysis [Balakrishnan and Reps 2004; Balakrishnan 2007] (along with property simulation) to over-approximate the set of reachable states; it reports a possible property violation if the error state is reachable at any instruction in the executable. To eliminate spurious error traces, SDV uses counter-example-guided abstraction refinement, whereas DDA/x86 leverages path sensitivity obtained by combining property simulation and abstract interpretation. In this respect, DDA/x86 is more closely related to ESP—in fact, the algorithm in §6.1 was inspired by ESP. However, unlike ESP, DDA/x86 provides a witness trace for a possible bug, as described in §6.2. Moreover, DDA/x86 uses a different kind of abstraction refinement (see §4 and §5).

Although combining the propagation of property-automaton states and abstract interpretation provides a degree of path sensitivity, it was not always sufficient to eliminate all of the false positives for the examples in our test suite. Therefore, we also distinguished paths based on the abstract state (using the automaton shown in Fig. 24) in addition to distinguishing paths based on property-automaton states. While the results of our experiments are encouraging, it required a lot of manual effort to reduce the number of false positives: spurious error traces were examined by hand, and the automaton in Fig. 24 was introduced to refine the abstraction in use. For DDA/x86 to be usable on a day-to-day basis, it would be important to automate the process of reducing the number of false positives. Several techniques have been proposed to reduce the number of false positives in abstract interpretation, including trace partitioning [Mauborgne and Rival 2005], qualified dataflow analysis [Holley and Rosen 1981], and the refinement techniques of Fischer et al. [2005] and Dhurjati et al. [2006]. All of these techniques are potentially applicable in DDA/x86.

Shared Data Structures. The use of shared data structures to reduce the space required for program analysis has a long history; it includes applicative shared dictionaries [Myers 1984; Reps et al. 1983], shared set representations [Pugh 1988], and binary decision diagrams [Bryant 1986; Burch et al. 1990]. Recent work that discusses efficient representations of data structures for program analysis includes Blanchet et al. [2003] and Manevich et al. [2002].

8. CONCLUSIONS

In recent years, the topic of improving programmer productivity and software reliability has become one of the main focal points of programming-language and compiler research. However, most analysis efforts have focused on programs for which source code is available; the problem of analyzing executables has received much less attention. The methods presented in this paper help to fill that gap.

The main focus of this paper is on algorithms that recover intermediate representations (IRs) from an executable that are similar to the ones that would be obtained by a compiler if we had started from source code. Just as the IRs created by a compiler form the backbone of tools for analyzing source code, the IRs

recovered using our algorithms form the backbone of tools for performing further analysis of executables. Moreover, because the IRs recovered by our algorithms are similar to the IRs created by a compiler, it is also possible to leverage techniques from source-code analysis to the analysis of executables—making adaptations as needed.

There are multiple challenges when the goal is to recover suitable IRs from an executable. In this paper, we outlined the challenges and presented our solutions that address those challenges. §2 presented an abstract memory model for analyzing executables, and introduced variable-like entities, referred to as a-locs, which serve as proxies for the program's actual variables. §3 presented various algorithms to obtain information about memory accesses in an executable. §4 presented an improved a-loc recovery algorithm. §5 presented an abstraction-refinement algorithm, which iteratively improves both the set of a-locs in use, as well as the precision of the results obtained via the algorithms presented in §3. Not all of our techniques could be presented in this paper; references to enhancements and variations are given in several places (e.g., see footnote 4 and the list at the beginning of §7).

Overall, the techniques that we developed are reasonably successful at providing a foundation for performing a variety of analyses on executables. In particular, the experiments reported on in §4.9 showed that there was substantial agreement between the a-locs discovered for an executable and the variables of the original source-level program. With the additional techniques presented elsewhere [Balakrishnan 2007, Ch. 7], we were able to recover information that is useful for static analysis at over 80% of the indirect memory accesses in an executable.

Moreover, our techniques opened up new opportunities for analyzing executables. Prior to our work, several analysis problems on executables had not been addressed using principled static-analysis techniques—only *ad-hoc* solutions had been proposed. For instance, Cifuentes and Fraboulet [1997b] give an algorithm to identify an intraprocedural slice of an executable by following the program's use-def chains. However, their algorithm makes no attempt to track values that are not in registers, and hence cuts short the slice when a load from memory is encountered. In contrast, we used our analyses to create CodeSurfer/x86 [Balakrishnan et al. 2005], the first program-slicing tool for executables that can help with understanding dependences across memory updates and memory accesses.

As described in §6, we were able to extend CodeSurfer/x86 to create DDA/x86 [Balakrishnan and Reps 2008], which represents the first automatic program-verification tool for stripped executables. DDA/x86 allows one to check that a stripped executable conforms to an API-usage rule specified as a finite-state machine. The experiments reported on in §6.2 showed that DDA/x86 was able to verify the absence of bugs for the majority of our test cases. In the test cases that had real bugs, it was able to find a useful counter-example sequence in the executable.

The CodeSurfer/x86 platform has been used for a number of other applications as well, including extracting file formats from executables [Lim et al. 2006] and determining summaries for library functions [Gopan and Reps 2006]. It has also been used by other researchers to identify the propagation mechanisms and payloads of worms [Brown et al. 2007].

Despite these successes, there is room for improvement. When implementing

the abstract transformers for CodeSurfer/x86's various static-analysis components (VSA, ASI, ARA, etc.), it was a major headache to maintain consistency among the various abstract semantics. Both the size of the x86 instruction set and the complexity of the abstract domains involved contributed to the problem. Furthermore, to port CodeSurfer/x86 to a new instruction set, for each abstract semantics it would have been necessary to hand-code new abstract transformers for the new set of instructions. Overall, to support m abstract domains and n instruction sets (each of size is, to simplify matters), the amount of work involved is $m \times n \times is$.

To address this problem, Lim and Reps [2008] developed the Transformer Specification Language (TSL) system. With TSL, one specifies (i) the *concrete* semantics of each instruction set (using an ML-like language to write an interpreter for each instruction set), along with (ii) a description of each abstract domain. From these inputs, the TSL system generates consistent *abstract* transformers for each abstract domain automatically. Therefore, instead of writing $m \times n \times is$ transformers, the TSL user need only perform $m + n \times is$ work: he must provide $n \times is$ concrete transformers to specify the concrete semantics of n instruction sets, and also write the specifications of m abstract domains. Consequently, TSL considerably reduces the effort required to create multiple versions—for different instruction sets—of a system, like CodeSurfer, that contains multiple analysis components.

Another area in which there is room for improvement concerns the nature of the VSA domain. Unlike abstract domains such as the polyhedral domain [Cousot and Halbwachs 1978], the VSA domain does not track inter-variable relationships. One of the main issues that we faced in our work is the loss of precision due to the non-relational nature of the VSA domain. We overcame some of the precision loss by (i) using information from auxiliary analyses, such as affine-relation analysis [Müller-Olm and Seidl 2005; Lal et al. 2005] and GMOD analysis [Cooper and Kennedy 1988; Reps and Balakrishnan 2008], and (ii) splitting abstract states at each program point based on an automaton (see §6.1). However, in all of these cases it required a lot of manual effort to identify the right combination of analyses and partitioning of the VSA states to achieve the desired level of precision. It would be useful to automate the process of tuning the analyzer based on the analysis problem at hand. Abstraction-refinement techniques, such as those of Henzinger et al. [2004], Fischer et al. [2005], and Dhurjati et al. [2006], have been successfully used in source-code-analysis tools. We believe that CodeSurfer/x86 would be even more useful if such abstraction-refinement techniques are combined with the VSA algorithm and the other analyses already incorporated in CodeSurfer/x86.

Acknowledgments. We are grateful to our collaborators at Wisconsin—J. Lim, A. Lal, N. Kidd, and D. Gopan—and at GrammaTech, Inc.—T. Teitelbaum, S. Yong, R. Gruian, D. Melski, and C.-H. Chen—for their many contributions to the project. We are also grateful to M. Sagiv, R. Wilhelm, and S. Jha for several helpful discussions about the work.

REFERENCES

Aigner, G. and Hölzle, U. 1996. Eliminating virtual function calls in C++ programs. In *Proc. European Conf. on Obj.-Oriented Prog.*

aiT. aiT Worst-Case Execution Time Analyzer. http://www.AbsInt.com/aiT.

Amme, W., Braun, P., Zehendner, E., and Thomasset, F. 2000. Data dependence analysis of assembly code. *Int. Journal of Parallel Programming*.

Backes, W. 2004. Programmanalyse des xrtl zwischencodes. Ph.D. thesis, Universitaet des Saarlandes. (In German.).

Bala, V., Duesterwald, E., and Banerjia, S. 2000. Dynamo: A transparent runtime optimization system. In *Proc. Conf. on Prog. Lang. Design and Implementation*.

Balakrishnan, G. 2007. WYSINWYX: What You See Is Not What You eXecute. Ph.D. thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI. Tech. Rep. 1603.

Balakrishnan, G., Gruian, R., Reps, T., and Teitelbaum, T. 2005. Codesurfer/x86 – A platform for analyzing x86 executables, (tool demonstration paper). In *Proc. Int. Conf. on Compiler Construction*.

Balakrishnan, G. and Reps, T. 2004. Analyzing memory accesses in x86 executables. In *Proc. Int. Conf. on Compiler Construction*. 5–23.

Balakrishnan, G. and Reps, T. 2006. Recency-abstraction for heap-allocated storage. In *Proc. Static Analysis Symp.*

Balakrishnan, G. and Reps, T. 2007. DIVINE: DIscovering Variables IN Executables. In *Proc. Verif., Model Checking, and Abs. Interp.*

Balakrishnan, G. and Reps, T. 2008. Analyzing stripped device-driver executables. In *Proc. Tools and Algs. for the Construct. and Anal. of Syst.*

Balakrishnan, G., Reps, T., Kidd, N., Lal, A., Lim, J., Melski, D., Gruian, R., Yong, S., Chen, C.-H., and Teitelbaum, T. 2005. Model checking x86 executables with CodeSurfer/x86 and WPDS++. In *Proc. Computer Aided Verif.*

Balakrishnan, G., Reps, T., Melski, D., and Teitelbaum, T. 2007. WYSINWYX: What You See Is Not What You eXecute. In *Proc. IFIP Working Conf. on Verified Software: Theories, Tools, Experiments*.

Ball, T. 2006. Personal communication.

Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S., and Ustuner, A. 2006. Thorough static analysis of device drivers. In *Proc. European Conf. on Computer Systems*.

Ball, T. and Rajamani, S. 2000. Bebop: A symbolic model checker for Boolean programs. In *Proc. Spin Workshop*. Lec. Notes in Comp. Sci., vol. 1885. 113–130.

Ball, T. and Rajamani, S. 2001. The SLAM toolkit. In *Proc. Computer Aided Verif.*

Bergeron, J., Debbabi, M., Desharnais, J., Erhioui, M., Lavoie, Y., and Tawbi, N. 2001. Static detection of malicious code in executable programs. *Int. J. of Req. Eng.*.

Bergeron, J., Debbabi, M., Erhioui, M., and Ktari, B. 1999. Static analysis of binary code to isolate malicious behaviors. In *Int. Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises*.

Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. 2003. A static analyzer for large safety-critical software. In *Proc. Conf. on Prog. Lang. Design and Implementation*. 196–207.

Bouajjani, A., Esparza, J., and Maler, O. 1997. Reachability analysis of pushdown automata: Application to model checking. In *Proc. CONCUR*.

Bourdoncle, F. 1993. Efficient chaotic iteration strategies with widenings. In *Int. Conf. on Formal Methods in Prog. and their Appl.* LNCS. Springer-Verlag.

Brown, R., Khazan, R., and Zhivich, M. 2007. AWE: Improving software analysis through modular integration of static and dynamic analyses. In *PASTE*.

Brumley, D. and Newsome, J. 2006. Alias analysis for assembly. Tech. Rep. CMU-CS-06-180, School of Comp. Sci., Carnegie Mellon University, Pittsburgh, PA. Dec.

Bryant, R. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp. C-35,* 6 (Aug.), 677–691.

Burch, J., Clarke, E., McMillan, K., Dill, D., and Hwang, L. 1990. Symbolic model checking: 10^{20} states and beyond. In *Proc. Symp. on Logic in Comp. Sci.* 428–439.

Bush, W., Pincus, J., and Sielaff, D. 2000. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience 30*, 775–802.

Cai, H., Shao, Z., and Vaynberg, A. 2007. Certified self-modifying code. In *Proc. Conf. on Prog. Lang. Design and Implementation*. ACM Press, 66–77.

Chandra, S. and Reps, T. 1999. Physical type checking for C. In *Proc. Prog. Analysis for Softw. Tools and Eng.* 66–75.

Chang, B.-Y., Harren, M., and Necula, G. 2006. Analysis of low-level code using cooperating decompilers. In *Proc. Static Analysis Symp.*

Chen, H. and Wagner, D. 2002. MOPS: An infrastructure for examining security properties of software. In *Conf. on Comp. and Commun. Sec.* 235–244.

Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. 2001. An empirical study of operating systems errors. In *Proc. Symp. on Op. Syst. Principles*.

Christodorescu, M., Goh, W.-H., and Kidd, N. 2005. String analysis for x86 binaries. In *Prog. Analysis for Softw. Tools and Eng.*

Cifuentes, C. and Fraboulet, A. 1997a. Interprocedural data flow recovery of high-level language code from assembly. Tech. Rep. 421, Univ. Queensland.

Cifuentes, C. and Fraboulet, A. 1997b. Intraprocedural static slicing of binary executables. In *Proc. Int. Conf. on Software Maintenance*. 188–195.

Cifuentes, C., Simon, D., and Fraboulet, A. 1998. Assembly to high-level language translation. In *Proc. Int. Conf. on Software Maintenance*. 228–237.

Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2000. Counterexample-guided abstraction refinement. In *Proc. Computer Aided Verif.* 154–169.

CodeSonar. CodeSonar, GrammaTech, Inc. www.grammatech.com/products/codesonar.

CodeSurfer. CodeSurfer, GrammaTech, Inc. www.grammatech.com/products/codesurfer.

Cooper, K. and Kennedy, K. 1988. Interprocedural side-effect analysis in linear time. In *Proc. Conf. on Prog. Lang. Design and Implementation*. 57–66.

Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Robby, and Zheng, H. 2000. Bandera: Extracting finite-state models from Java source code. In *Proc. Int. Conf. on Softw. Eng.* 439–448.

Cousot, P. and Cousot, R. 1976. Static determination of dynamic properties of programs. In *Proc. 2nd. Int. Symp on Programming*. Paris.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proc. Symp. on Princ. of Prog. Lang.* 238–252.

Cousot, P. and Cousot, R. 1979. Systematic design of program analysis frameworks. In *Proc. Symp. on Princ. of Prog. Lang.* 269–282.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear constraints among variables of a program. In *Proc. Symp. on Princ. of Prog. Lang.* 84–96.

Cova, M., Felmetsger, V., Banks, G., and Vigna, G. 2006. Static detection of vulnerabilities in x86 executables. In *Proc. Annual Comp. Sec. Applications Conf.*

Coverity. Coverity Prevent. www.coverity.com/html/coverity-prevent-static-analysis.html.

Das, M., Lerner, S., and Seigle, M. 2002. ESP: Path-sensitive program verification in polynomial time. In *Proc. Conf. on Prog. Lang. Design and Implementation*. ACM Press, New York, NY, 57–68.

De Sutter, B., De Bus, B., De Bosschere, K., Keyngnaert, P., and Demoen, B. 2000. On the static analysis of indirect control transfers in binaries. In *Proc. Int. Conf. on Parallel and Dist. Processing Techniques and Applications*.

Debray, S., Linn, C., Andrews, G., and Schwarz, B. 2004. Stack analysis of x86 executables. www.cs.arizona.edu/∼debray/Publications/stack-analysis.pdf.

Debray, S., Muth, R., and Weippert, M. 1998. Alias analysis of executable code. In *Proc. Symp. on Princ. of Prog. Lang.* 12–24.

Dhurjati, D., Das, M., and Yang, Y. 2006. Path-sensitive dataflow analysis with iterative refinement. In *Proc. Static Analysis Symp.* 425–442.

DMCA §1201. §1201. Circumvention of Copyright Protection Systems. www.copyright.gov/title17/92chap12.html#1201.

Dor, N., Rodeh, M., and Sagiv, M. 2003. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proc. Conf. on Prog. Lang. Design and Implementation*. 155–167.

Eidorff, P., Henglein, F., Mossin, C., Niss, H., Sørensen, M., and Tofte, M. 1999. Anno Domini: From type theory to year 2000 conversion tool. In *Proc. Symp. on Princ. of Prog. Lang.* 1–14.

Emmerik, M. V. 2007. Static single assignment for decompilation. Ph.D. thesis, School of Inf. Tech. and Elec. Eng., Univ. of Queensland, Brisbane, AU.

Engler, D., Chelf, B., Chou, A., and Hallem, S. 2000. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. Op. Syst. Design and Impl.* 1–16.

Feng, X., Shao, Z., Vaynberg, A., Xiang, S., and Ni, Z. 2006. Modular verification of assembly code with stack-based control abstractions. In *Proc. Conf. on Prog. Lang. Design and Implementation*. 401–414.

Ferdinand, C. 2009. Personal communication.

Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., and Wilhelm, R. 2001. Reliable and precise WCET determination for a real-life processor. In *Proc. of the First Int. Workshop on Embedded Software (EMSOFT)*. 469–485.

Finkel, A., B.Willems, and Wolper, P. 1997. A direct symbolic approach to model checking pushdown systems. *Elec. Notes in Theor. Comp. Sci. 9*.

Fischer, J., Jhala, R., and Majumdar, R. 2005. Joining dataflow with predicates. In *Proc. Found. of Softw. Eng.*

Gopan, D. and Reps, T. 2006. Lookahead widening. In *Proc. Computer Aided Verif.*

Gopan, D. and Reps, T. 2007. Low-level library analysis and summarization. In *Proc. Computer Aided Verif.*

Graf, S. and Saïdi, H. 1997. Construction of abstract state graphs with PVS. In *Proc. Computer Aided Verif.* Lec. Notes in Comp. Sci., vol. 1254. 72–83.

Grewe, D. 2008. Static congruence analysis of binaries. Bachelors thesis, Univ. des Saarlandes.

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

Havelund, K. and Pressburger, T. 2000. Model checking Java programs using Java PathFinder. *Softw. Tools for Tech. Transfer 2,* 4.

Henzinger, T., Jhala, R., Majumdar, R., and McMillan, K. L. 2004. Abstractions from proofs. In *Proc. Symp. on Princ. of Prog. Lang.* 232–244.

Henzinger, T., Jhala, R., Majumdar, R., and Sutre, G. 2002. Lazy abstraction. In *Proc. Symp. on Princ. of Prog. Lang.* 58–70.

Hind, M. 2001. Pointer analysis: Haven't we solved this problem yet? In *Proc. Prog. Analysis for Softw. Tools and Eng.*

Holley, L. and Rosen, B. 1981. Qualified data flow problems. *Trans. on Softw. Eng. 7,* 1, 60–78.

Howard, M. 2002. Some bad news and some good news. MSDN, Microsoft Corp., msdn2.microsoft.com/en-us/library/ms972826.aspx.

IDAPro. IDAPro disassembler. www.hex-rays.com/idapro/.

Kiriansky, V., Bruening, D., and Amarasinghe, S. 2002. Secure execution via program shepherding. In *Proc. USENIX Sec. Symp.*

Kiss, A., J. Jász, G. L., and Gyimóthy, T. 2003. Interprocedural static slicing of binary executables. In *Proc. Int. Workshop on Source Code Analysis and Manip.*

Kruegel, C., Kirda, E., Mutz, D., Robertson, W., and Vigna, G. 2005. Automating mimicry attacks using static binary analysis. In *Proc. USENIX Sec. Symp.*

Kurshan, R. 1994. *Computer-Aided Verification of Coordinating Processes*. Princeton Univ. Press.

Lal, A., Reps, T., and Balakrishnan, G. 2005. Extended weighted pushdown systems. In *Proc. Computer Aided Verif.*

Larus, J. and Schnarr, E. 1995. EEL: Machine-independent executable editing. In *Proc. Conf. on Prog. Lang. Design and Implementation*. 291–300.

Lim, J. and Reps, T. 2008. A system for generating static analyzers for machine instructions. In *Proc. Int. Conf. on Compiler Construction.*

Lim, J., Reps, T., and Liblit, B. 2006. Extracting output formats from executables. In *Proc. Working Conf. on Rev. Eng.*

Manevich, R., Ramalingam, G., Field, J., Goyal, D., and Sagiv, M. 2002. Compactly representing first-order structures for static analysis. In *Proc. Static Analysis Symp.* 196–212.

Mauborgne, L. and Rival, X. 2005. Trace partitioning in abstract interpretation based static analyzers. In *Proc. European Symp. on Programming.*

Miné, A. 2006. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. Conf. on Languages, Compilers, and Tools for Embedded Systems.*

Müller-Olm, M. and Seidl, H. 2005. Analysis of modular arithmetic. In *Proc. European Symp. on Programming.*

Mycroft, A. 1999. Type-based decompilation. In *Proc. European Symp. on Programming.*

Myers, E. 1984. Efficient applicative data types. In *Proc. Symp. on Princ. of Prog. Lang.*

Ni, Z. and Shao, Z. 2006. Certified assembly programming with embedded code pointers. In *Proc. Symp. on Princ. of Prog. Lang.* 320–333.

Nita, M., Grossman, D., and Chambers, C. 2008. A theory of platform-dependent low-level software. In *Proc. Symp. on Princ. of Prog. Lang.*

O'Callahan, R. and Jackson, D. 1997. Lackwit: A program understanding tool based on type inference. In *Proc. Int. Conf. on Softw. Eng.*

Oney, W. 2003. *Programming the Microsoft Windows Driver Model*. Microsoft Press.

Pande, H. and Ryder, B. 1996. Data-flow-based virtual function resolution. In *Proc. Static Analysis Symp.* 238–254.

Phoenix. Phoenix software optimization and analysis framework. http://connect.microsoft.com/phoenix.

Pioli, A. and Hind, M. 1999. Combining interprocedural pointer analysis and conditional constant propagation. Tech. Rep. RC 21532(96749), IBM T.J. Watson Research Center. Mar.

PREfast 2004. PREfast with driver-specific rules. www.microsoft.com/whdc/archive/PREfast-drv.mspx.

Pugh, W. 1988. Incremental computation and the incremental evaluation of functional programs. Ph.D. thesis, Cornell University.

Ramalingam, G., Field, J., and Tip, F. 1999. Aggregate structure identification and its application to program analysis. In *Proc. Symp. on Princ. of Prog. Lang.*

Regehr, J., Reid, A., and Webb, K. 2005. Eliminating stack overflow by abstract interpretation. In *ACM Trans. on Embedded Comp. Systs.* 751–778.

Reps, T. and Balakrishnan, G. 2008. Improved memory-access analysis for x86 executables. In *Proc. Int. Conf. on Compiler Construction.*

Reps, T., Balakrishnan, G., and Lim, J. 2006. Intermediate-representation recovery from low-level code. In *Proc. Part. Eval. and Semantics-Based Prog. Manip.*

Reps, T., Balakrishnan, G., Lim, J., and Teitelbaum, T. 2005. A next-generation platform for analyzing executables. In *Proc. Asian Symp. on Prog. Lang. and Systems.*

Reps, T., Teitelbaum, T., and Demers, A. 1983. Incremental context-dependent analysis for language-based editors. *Trans. on Prog. Lang. and Syst. 5,* 3 (July), 449–477.

Rival, X. 2003. Abstract interpretation based certification of assembly code. In *Proc. Verif., Model Checking, and Abs. Interp.*

Rugina, R. and Rinard, M. 2005. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *Trans. on Prog. Lang. and Syst. 27,* 2, 185–235.

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

Siff, M. and Reps, T. 1996. Program generalization for software reuse: From C to C++. In *Proc. Found. of Softw. Eng.*

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

Srivastava, A. and Eustace, A. 1994. ATOM - A system for building customized program analysis tools. In *Proc. Conf. on Prog. Lang. Design and Implementation*.

Swift, M., Annamalai, M., Bershad, B., and Levy, H. 2004. Recovering device drivers. In *Proc. Symp. on Op. Syst. Design and Impl.*

Swift, M., Bershad, B., and Levy, H. 2005. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst. 23,* 1.

van Deursen, A. and Moonen, L. 1998. Type inference for COBOL systems. In *Proc. Working Conf. on Rev. Eng.*

Wagner, D., Foster, J., Brewer, E., and Aiken, A. 2000. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. Network and Dist. Syst. Security.*

Wall, D. 1992. Systems for late code modification. In *Code Generation – Concepts, Tools, Techniques*, R. Giegerich and S. Graham, Eds. Springer-Verlag.

WHDC 2007. C++ for kernel mode drivers: Pros and cons. www.microsoft.com/whdc/driver/kernel/KMcode.mspx.

WHQL 2004. Defrauding the WHQL driver certification process. blogs.msdn.com/oldnewthing/archive/2004/03/05/84469.aspx.

Wikipedia: Enforceability. Software license agreement: Enforceability (in the United States). en.wikipedia.org/wiki/Software_license_agreement#Enforceability, Sept. 19, 2009.

Wikipedia: Shrink-Wrap and Click-Wrap Licenses. Software license agreement: Shrink-wrap and click-wrap licenses. en.wikipedia.org/wiki/Software_license_agreement#Shrink-wrap_and_click-wrap_licenses, Sept. 19, 2009.

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst. 7,* 3, 1–53.

Wilson, R. and Lam, M. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proc. Conf. on Prog. Lang. Design and Implementation*. 1–12.

Windows DDK 2003. Windows Server 2003 DDK. www.microsoft.com/whdc/devtools/ddk.

Xu, Z., Miller, B., and Reps, T. 2000. Safety checking of machine code. In *Proc. Conf. on Prog. Lang. Design and Implementation*.

Xu, Z., Miller, B., and Reps, T. 2001. Typestate checking of machine code. In *Proc. European Symp. on Programming*.

Zhang, J., Zhao, R., and Pang, J. 2007. Parameter and return-value analysis of binary executables. In *Comp. Softw. and Applications Conf.*