Improved Memory-Access Analysis for x86 Executables*

Thomas Reps[1,2] and Gogul Balakrishnan[**,3]

[1] University of Wisconsin
[2] GrammaTech, Inc.
[3] NEC Laboratories America, Inc.
reps@cs.wisc.edu, bgogul@nec-labs.com

Abstract. 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. It is relatively easy to track the effects of an instruction operand that refers to a global address (i.e., an access to a global variable) or that uses a stack-frame offset (i.e., an access to a local scalar variable via the frame pointer or stack pointer). In our work, our algorithms are able to provide useful information for close to 100% of such "direct" uses and defs.

It is much harder for a static-analysis algorithm to track the effects of an instruction operand that uses a non-stack-frame register. These "indirect" uses and defs correspond to accesses to an array or a dynamically allocated memory object. In one study, our approach recovered useful information for only 29% of indirect uses and 33% of indirect defs. However, using the technique described in this paper, the algorithm recovered useful information for 81% of indirect uses and 90% of indirect defs.

1 Introduction

Research carried out during the last decade by our research group [64, 65, 6, 56, 55, 7, 8, 36, 4, 49, 9] as well as by others [48, 22, 33, 14, 2, 31, 13, 44, 32, 3, 54, 37, 21, 46, 28, 19, 16, 34, 66] has developed the foundations for performing static analysis at the machine-code level. The machine-code-analysis problem comes in two versions: (i) with symbol-table/debugging information (unstripped executables), and (ii) without symbol-table/debugging information (stripped executables). Many tools address both versions of the problem, 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 [6]. 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 [48], and (ii) analyze control flow (sometimes by

* Supported by NSF under grants CCF-0540955 and CCF-0524051 and by AFRL under contract FA8750-06-C-0249.
** Work performed while at the University of Wisconsin.

applying local heuristics to try to resolve indirect calls and indirect jumps, but otherwise ignoring them).

The work presented in [6] provided a way to apply the tools of abstract interpretation [27] to the problem of analyzing stripped executables, and we followed this up with other techniques to complement and enhance the approach [56, 47, 55, 7, 8, 4, 9]. 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 [5].

It is relatively easy to track the effects of an instruction operand that refers to a global address (i.e., an access to a global variable) or that uses a stack-frame offset (i.e., an access to a local scalar variable via the frame pointer or stack pointer). In our work, our algorithms are able to provide useful information for close to 100% of such "direct" uses and defs.

It is much harder for a static-analysis algorithm to track the effects of an instruction operand that uses a non-stack-frame register. These "indirect" uses and defs correspond to accesses to an array or a dynamically allocated memory object. This paper describes a technique that had an important impact on the precision obtained for indirect uses and defs in CodeSurfer/x86. As we describe in §5, in one validation study on a collection of stripped device-driver executables, the algorithm reported in [8] recovered useful information for only 29% of indirect uses and 33% of indirect defs. However, using the improved technique described in this paper, the algorithm recovered useful information for 81% of indirect uses and 90% of indirect defs.

The remainder of the paper is organized as follows: §2 motivates the work by describing some of the advantages of analyzing machine code. §3 explains some of the ideas used in CodeSurfer/x86 for recovering intermediate representations (IRs). §4 describes an extension that we made to CodeSurfer/x86's IR-recovery algorithm, which had an important impact on precision. §5 presents experimental results that measure the gain in precision. §6 discusses related work.

2 The Case for Analyzing Machine Code

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 [38, 60, 35, 26, 17, 12, 20, 39, 30]. In these tools, static analysis is used to determine a conservative answer to the question "Can the program reach a bad state?"[4] Some of this work has already been transitioned to commercial products for source-code analysis [17, 11, 29, 23].

[4] 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 [27].

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 source code is available, source-code analysis has certain drawbacks [40, 62]. The reason is that computers do not execute source code; they execute *machine-code* programs that are generated from source code. The transformation that takes place between high-level source code and low-level machine code can cause there to be subtle but important differences between what a programmer intended and what is actually executed by the processor. Consequently, analyses that are performed on source code can fail to detect certain bugs and vulnerabilities.

For instance, during the Windows security push in 2002, the Microsoft C++ .NET compiler was found to introduce a vulnerability in the machine code for the following code fragment [40]:

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

Assume that the program has temporarily stored the user's password—in clear text—in a dynamically allocated buffer pointed to by the pointer variable `password`. To minimize the lifetime of the password, which is sensitive information, the code fragment shown above zeroes-out the buffer pointed to by `password` before returning it to the freelist. Unfortunately, the compiler's useless-code-elimination algorithm reasoned that the program never uses the values written by the call on `memset`, and therefore the call on `memset` could be removed—thereby leaving sensitive information exposed in the freelist.

Such a vulnerability is invisible in the source code; it can only be detected by examining the low-level code emitted by the optimizing compiler. Elsewhere [56, 10, 4], we have called this the WYSINWYX phenomenon (**W**hat **Y**ou **S**ee **I**s **N**ot **W**hat **Y**ou e**X**ecute).

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[5] 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 [61]. 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 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.[6] 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,

---

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

[6] We follow the conventions of abstract interpretation [27], 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.

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.

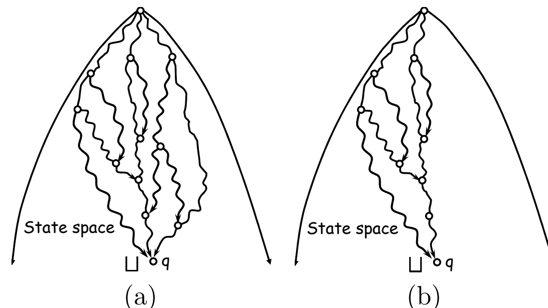


Fig. 1. Source-code analysis, which must account for all possible choices made by the compiler, must summarize more paths (see (a)) than machine-code analysis (see (b)). Because the latter can focus on fewer paths, it can yield more precise results.

Although it is possible to create source-code tools that strive to have greater fidelity to the program that is actually executed—examples include [18, 51]—in the limit, the tool would have to incorporate 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 [36].
- 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 [24] or do not push the analysis beyond sites of inlined assembly [52]. 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).

3 CodeSurfer/x86: A Platform for Recovering IRs from Stripped Executables

Given a stripped executable as input, CodeSurfer/x86 [5] recovers IRs that are similar to those that would be available had one started from source code. This section explains some of the ideas used in the IR-recovery algorithms of CodeSurfer/x86 [4, 6, 8].

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 [41]. 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 [33], data dependences [2, 22], targets of indirect calls [31], values of strings [21], bounds on stack height [54], and values of parameters and return values [66]. 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.

Variable and Type Discovery. 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). When performing source-code analysis, the programmer-defined variables provide us with the compartments for tracking data manipulations. When debugging information is absent, an executable's data objects are not easily identifiable. Consider, for instance, an access on a source-code variable `x` in some source-code statement. At the machine-code level, an access on `x` is performed 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. The variable and type-discovery phase of CodeSurfer/x86 [8, 4] recovers information about variables that are allocated globally, locally (i.e., on the stack), and dynamically (i.e., from the freelist). The recovered variables, called *a-locs* (for "abstract locations") are the basic variables used in CodeSurfer/x86's value-set-analysis (VSA) algorithm [6, 8, 4].

To accomplish this task, CodeSurfer/x86 makes use of a number of analyses, and the sequence of analyses performed is itself iterated [4, 8]. On each round, CodeSurfer/x86 uses VSA to identify an over-approximation of the memory accesses performed at each instruction. Subsequently, the results of VSA are used to perform aggregate structure identification (ASI) [53], which identifies commonalities among accesses to an aggregate data value, to refine the current set of a-locs. The new set of a-locs are used to perform another round of VSA. If the over-approximation of memory accesses computed by VSA improves from the previous round, the a-locs computed by the subsequent round of ASI may also improve. This process is repeated as long as desired, or until the process

converges. By this means, CodeSurfer/x86 bootstraps its way to a set of a-locs that serve as proxies for the program's original variables.

4 GMOD-Based Merge Function

This section describes one of the extensions that we made to our IR-recovery algorithm that had an important impact on precision. The context-sensitive VSA algorithm associates each program point with an AbsMemConfig:

$$\mathsf{AbsMemConfig} = (\mathsf{CallString_k} \rightarrow \mathsf{AbsEnv}_\perp)$$

where an AbsEnv value [4, 8] maps each a-loc and register to an over-approximation of its set of possible values (referred to as a *value-set*), and a $\mathsf{CallString_k}$ value is an abstraction of the structure of the run-time stack.[7] The context-sensitive VSA algorithm characterizes a set of concrete states by a set of calling contexts in which those states can arise.

Fig. 2 shows the context-sensitive VSA algorithm, which is based on a worklist. For the time being, consider the statements that are underlined as being absent. The entries in the worklist are $\langle \mathsf{CallString_k}, \mathsf{Node} \rangle$ pairs, and each entry represents the calling contexts of the corresponding node that have not yet been explored. The algorithm selects an entry from the worklist, executes the abstract transformer for each edge out of the node, and propagates the information to all the successors of the node.

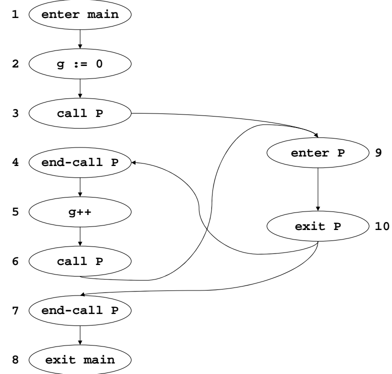


Fig. 3. Example showing the need for a GMOD-based merge function.

For all nodes, including an end-call[8] node, the algorithm combines the result of the abstract transformer with the old abstract state at the successor. Although *Propagate* computes a sound AbsEnv value for an end-call node, it may not always be precise. Consider the interprocedural CFG (ICFG) shown in Fig. 3. In any concrete execution, the only possible value for g at node 4 is 0. However, context-insensitive VSA (i.e., VSA with call-strings of length 0) computes the range $[-2^{31}, 2^{31}-1]$ for

[7] Let CallSites denote the set of call-sites in an executable. The executable's *call graph* is a labeled multi-graph in which each node represents a procedure, and each edge (labeled with a call-site in the calling procedure) represents a call. A call-string in the call graph is a finite-length path $(c_1 \ldots c_n)$ such that c_1 is a call-site in the entry procedure. CallString is the set of all call-strings.

A call-string suffix of length k [59] is either $(c_1 \ldots c_k) \in \mathsf{CallString}$, or $(*c_1 \ldots c_k)$, where $c_1, \ldots, c_k \in \mathsf{CallSites}$; the latter, referred to as a *saturated* call-string, represents the set of call-strings $\{cs \mid cs \in \mathsf{CallString}, cs = \pi c_1 \ldots c_k, \text{ and } |\pi| \geq 1\}$. $\mathsf{CallString_k}$ is the set of saturated call-strings of length k, plus non-saturated call-strings of length $\leq k$.

[8] An end-call node represents the return site for a call node.

```
1: decl worklist: set of ⟨CallString_k, Node⟩
2:
3: proc ContextSensitiveVSA()
4:     worklist := {⟨∅, enter⟩}
5:     absEnv_enter := Initial values of global a-locs and esp
6:     while (worklist ≠ ∅) do
7:         while (worklist ≠ ∅) do
8:             Remove a pair ⟨cs, n⟩ from worklist
9:             m := Number of successors of node n
10:            for i = 1 to m do
11:                succ := GetSuccessor(n, i)
12:                edge_ae := AbstractTransformer(n → succ, absMemConfig_n[cs])
13:                cs_set := GetCSSuccs(cs, n, succ)
14:                for (each succ_cs ∈ cs_set) do
15:                    Propagate(succ_cs, succ, edge_ae)
16:                end for
17:            end for
18:        end while
19:        GMOD' := ComputeGMOD()
20:        if (GMOD' ≠ GMOD) then
21:            for each call-site c ∈ CallSites and cs ∈ CallString_k do
22:                if in_c[cs] ≠ ⊥ then worklist := worklist ∪ {⟨cs, c⟩}
23:            end for
24:            GMOD := GMOD'
25:        end if
26:    end while
27: end proc
28:
29: proc GetCSSuccs(pred_cs: CallString_k, pred: Node, succ: Node): set of CallString_k
30:     result := ∅
31:     if (pred is an exit node and succ is an end-call node) then
32:         Let c be the call node associated with succ
33:         for each succ_cs in absMemConfig_c do
34:             if (pred_cs ⇝^cs succ_cs) then
35:                 result := result ∪ {succ_cs}
36:             end if
37:         end for
38:     else if (succ is a call node) then
39:         result := {(pred_cs ≪^cs c)}
40:     else
41:         result := {pred_cs}
42:     end if
43:     return result
44: end proc
45:
46: proc Propagate(cs: CallString_k, n: Node, edge_ae: AbsEnv)
47:     old := absMemConfig_n[cs]
48:     if n is an end-call node and round > 0 then
49:         Let c be the call node associated with n
50:         edge_ae := GMODMergeAtEndCall(edge_ae, absMemConfig_c[cs])
51:     end if
52:     new := old ⊔^ae edge_ae
53:     if (old ≠ new) then
54:         absMemConfig_n[cs] := new
55:         worklist := worklist ∪ {⟨cs, n⟩}
56:     end if
57: end proc
```

Fig. 2. Context-sensitive VSA algorithm with GMOD-based merge function.

g at node 4. In context-insensitive VSA, the call-return structure of the ICFG is ignored (i.e., the ICFG is considered to be an ordinary graph). Note that 6→9 is a back-edge, and hence is a suitable location for widening to be performed [15].

Consider the path $\pi = (6, 9, 10, 4)$. Although π is an invalid execution path, context-insensitive VSA explores π. The effects of statement `g++` at node `5` and the results of widening at `6`→`9` are propagated to node `4`, and consequently, the range computed for `g` at node `4` by context-insensitive VSA is $[-2^{31}, 2^{31} - 1]$ (due to widening and wrap-around in 32-bit arithmetic). One possible solution to the problem is to increase the length of call-strings. However, it is impractical to increase the length of call-strings beyond a small value. Therefore, increasing the call-string length is not a complete solution to the problem.

1: **proc** *GMODMergeAtEndCall*(in_c: AbsEnv, in_x: AbsEnv): AbsEnv
2: $in'_c := SetAlocsToTop(in_c, \texttt{GMOD}[\texttt{X}])$
3: $in'_x := SetAlocsToTop(in_x, U - \texttt{GMOD}[\texttt{X}])$
4: $out := in'_c \sqcap^{\text{ae}} in'_x$
5: **return** out
6: **end proc**

Fig. 4. GMOD-based merge function. `GMOD[X]` represents the set of a-locs modified (directly or transitively) by procedure `X`, and U is the universal set of a-locs.)

Suppose that we modify *Propagate* in Fig. 4 by adding line [50], which invokes procedure *GMODMergeAtEndCall*. *GMODMergeAtEndCall* takes two AbsEnv values: (1) in_c, the AbsEnv value at the corresponding call node, and (2) in_x, the AbsEnv value at the corresponding exit node. Let `C` and `X` be the procedures containing the call and exit nodes, respectively. *SetAlocsToTop*(*ae*, *AlocSet*) returns the AbsEnv value $ae[a \mapsto \top^{vs} \mid a \in AlocSet]$. Operation $ae_1 \sqcap^{\text{ae}} ae_2$ yields a new AbsEnv value in which the set of values for each a-loc (register) is the meet of the value-sets for the corresponding a-loc (register) in ae_1 and ae_2.

In the earlier implementation of *Propagate* (i.e., when *Propagate* does not call *GMODMergeAtEndCall* on line [50]), the value-sets of all a-locs in in_x are propagated to the end-call node. In contrast, when *Propagate* does call *GMODMergeAtEndCall*, only the value-sets of a-locs that are modified (directly or transitively) in procedure `X` are propagated from in_x to the AbsEnv value at the end-call node. The value-sets for other a-locs are obtained from in_c. Because procedure `P` does not modify global variable `g`, using *GMODMergeAtEndCall* during context-insensitive VSA results in better information at nodes `4` and `7`; at node `4` the range for `g` is $[0, 0]$, and at node `7` the range for `g` is $[1, 1]$.

The actual implementation [4, Ch. 7] of *GMODMergeAtEndCall* is slightly more complicated. In addition to combining the information from the call-site and the exit node, it performs the following operations:

- At the exit node, the stack pointer `esp` points to the activation record of callee `X`. The value of `esp` in the AbsEnv value returned by *GMODMergeAtEndCall* is adjusted to point to the activation record of caller `C` .
- The value of the frame pointer `ebp` is set to the value of `ebp` in in_c. This change corresponds to the common situation in which the value of `ebp` at the exit node of a procedure is usually restored to the value of `ebp` at the call-site. (This is one of the aspects of the executable that VSA checks; a report is issued to the user if the behavior does not conform to what is expected.)
- The values of those a-locs that go out of scope, such as the local variables of callee `X`, are set to a special invalid abstract address.

The procedure shown in Fig. 4 uses GMOD information [25]; i.e., for each procedure P in the executable, information is required about the set of a-locs that P could possibly modify (directly or transitively). To perform GMOD analysis, information is required for each instruction about the set of a-locs that the instruction could possibly modify (i.e., IMOD information [25]). However, complete information about the a-locs accessed by each instruction is not available until the end of VSA. As discussed in §3, CodeSurfer/x86 makes use of a number of analyses, and the sequence of analyses performed is itself iterated. At the end of each round of VSA, GMOD information is computed for use during the next round of VSA (see lines [19]–[25] in Fig. 2); i.e., the GMOD sets for use during VSA round i are computed using the VSA results from round $i-1$. For the initial round of VSA ($i = 0$), *GMODMergeAtEndCall* is not used.[9] For each subsequent round, procedure *GMODMergeAtEndCall* is used as the merge function.

The process mentioned above may not be sound in the presence of indirect jump and indirect calls. In addition to determining an over-approximation of the set of states at each program point, VSA also determines the targets of indirect jumps and indirect calls. For pragmatic reasons, if VSA determines that the target address of an indirect jump or indirect call is $\top^{vs}$, it does not add any new edges.[10] Consequently, in the presence of indirect jumps and indirect calls, the ICFG used during round $i-1$ of VSA can be different from the ICFG used during round i. Therefore, for round i of VSA, it may not be sound to use the GMOD sets computed using the VSA results from round $i-1$. To ensure that the VSA results computed by round i are sound with respect to the current ICFG, the context-sensitive VSA algorithm of Fig. 2 does not terminate until the GMOD sets are consistent with the VSA results (see lines [19]–[25] in Fig. 2): when VSA reaches a fix-point in round i, the GMOD sets are recomputed using the current VSA results (GMOD′ on line [19] in Fig. 2) and compared against the current GMOD sets; if they are equal, then the VSA results are sound, and VSA terminates; otherwise, all call-sites $c \in$ CallSites are added to the worklist (line [22]) and VSA is resumed with the new worklist (line [5]). (For each call-site c, only those call-strings that have a non-$\bot$ AbsEnv at c are added to the worklist (line [22] in Fig. 2).) Even though VSA is restarted from a non-$\bot$ state by *reinitializing* the worklist (line [22]), VSA is guaranteed to converge because *Propagate accumulates* values at each program point using join ($\sqcup$); see line [52].

5 Experiments

This section describes a study that we carried out to measure the gain in precision that was obtained via the technique presented in §4. The study measured certain characteristics of the variables and values discovered by IR-recovery. The characteristics that we measured provide information about how good the recovered information would be as a starting point for some client tool that needs to perform additional static analysis on the executable. In particular, because

[9] Alternatively, *GMODMergeAtEndCall* could be called with GMOD[X] $= U$.

[10] A report is issued so that the user will be aware of the situation.

resolution of indirect operands is a fundamental primitive that essentially any subsequent analysis would need, we were particularly interested in how well our techniques could resolve indirect memory operands that use a non-stack-frame register (e.g., accesses to arrays and heap-allocated data objects).

Driver	Procedures	Instructions	Running time (seconds)	
			No GMOD	With GMOD
src/vdd/dosioctl/krnldrvr	70	284	34	25
src/general/ioctl/sys	76	2824	63	58
src/general/tracedrv/tracedrv	84	3719	122	45
src/general/cancel/startio	96	3861	44	32
src/general/cancel/sys	102	4045	43	33
src/input/moufiltr	93	4175	369	427
src/general/event/sys	99	4215	53	61
src/input/kbfiltr	94	4228	370	404
src/general/toaster/toastmon	123	6261	576	871
src/storage/filters/diskperf	121	6584	647	809
src/network/modem/fakemodem	142	8747	1410	2149
src/storage/fdc/flpydisk	171	12752	2883	5336
src/input/mouclass	192	13380	10484	13380
src/input/mouser	188	13989	4031	8917
src/kernel/serenum	184	14123	3777	9126
src/wdm/1394/driver/1394diag	171	23430	3149	12161
src/wdm/1394/driver/1394vdev	173	23456	2461	10912

Table 1. Running times for VSA with and without the GMOD-based merge function. (For the drivers listed above in **boldface**, round-by-round details of the percentages of strongly-trackable indirect operands are given in Fig. 7.)

To evaluate the effect of using the GMOD-based merge function on the precision of value-set analysis, we selected seventeen device drivers from the Windows Driver Development Kit [63] release 3790.1830; see Tab. 1. The executable for each device driver was obtained by compiling the driver source code along with the harness and OS environment model used in the SDV toolkit [11] (see [9] for more details). The resulting executable was then stripped; i.e., symbol-table and debugging information was removed.

We analyzed each executable using two versions of VSA: (1) VSA without the GMOD-based merge function (as sketched at the beginning of §4), and (2) VSA with the GMOD-based merge function shown in Fig. 4. For the experiments, we used 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.)

Except for the difference in the merge function, all other parameters, such as the lengths of call-strings, the number of rounds of VSA-ASI iteration, etc., were the same for both versions. We ran VSA-ASI iteration until convergence,

and then, based on the results of the final round of each run, we classified the memory operands in the executable into *strongly-trackable*, *weakly-trackable*, and *untrackable* operands:

- A memory operand is *strongly-trackable* (see Fig. 5) if
  - the lvalue evaluation of the operand does not yield $\top^{vs}$, and
  - each lvalue obtained refers to a 4-, 2-, or 1-byte (inferred) variable.
- 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 (inferred) variable.
- 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 (inferred) variable 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 $\top^{vs}$ [6, 55, 4]. 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 ($\sqcup^{\text{vs}}$) 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.

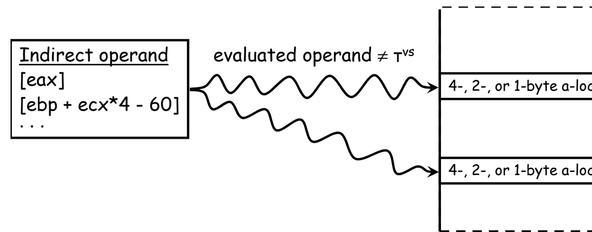


Fig. 5. Properties of a strongly-trackable memory operand.

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.

Category	Geometric Mean (for the final round)		
	Strongly-trackable indirect uses	Strongly-trackable indirect kills	Weakly-trackable indirect kills
Without GMOD-based merge function	29%	30%	33%
With GMOD-based merge function	81%	85%	90%

Table 2. Percentages of trackable memory operands in the final round.

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 versions perform equally well: the percentages of strongly-trackable direct use-operands and both strongly-trackable and weakly-trackable direct kill-operands are 100% for almost all of the drivers [4, §7.5.1].

Indirect Memory Operands. Tab. 2 summarizes the results for indirect operands. As shown in Tab. 2, when the technique described in §4 is used, the percentages of trackable indirect memory operands in the final round improve dramatically. (Note that the "Weakly-trackable indirect kills" are a superset of the "Strongly-trackable indirect kills".)

Fig. 6 shows the effects, on a per-application basis, of using the GMOD-based merge function on the percentages of strongly-trackable indirect use-operands, strongly-trackable indirect kill-operands, and weakly-trackable indirect kill-operands.

For the six Windows device drivers listed in **boldface** in Tab. 1, the graphs in Fig. 7 show the percentages of strongly-trackable indirect operands in different rounds for the two versions. The graphs show the positive interaction that exists between VSA and ASI: the percentages of strongly-trackable indirect operands increase with each round for both versions. However, for the VSA algorithm without the GMOD-based merge function, the improvements in the percentages of strongly-trackable indirect operands peter out after the third round because the value-sets computed for the a-locs are not as precise as the value-sets computed by the VSA algorithm with the GMOD-based merge function.

Columns 4 and 5 of Tab. 1 show the times taken for the two versions of VSA. The running times are comparable for smaller programs. However, for larger programs, the VSA algorithm with the GMOD-based merge function runs slower by a factor of 2 to 5. We believe that the slowdown is due to the increased precision during VSA obtained using the GMOD-based merge function. We use applicative AVL trees [50] to represent abstract stores. In our representation, if the value-set of a-loc a is $\top^{vs}$, meaning that a could hold any possible address or value, the AVL tree for the abstract store has no entry for a (and abstract operations on such values are performed quickly). When a technique improves the precision of VSA, there will be more a-locs whose value-set is not $\top^{vs}$; consequently, there will be more entries in the AVL trees for the abstract stores, and each abstract operation on the abstract store takes more time.

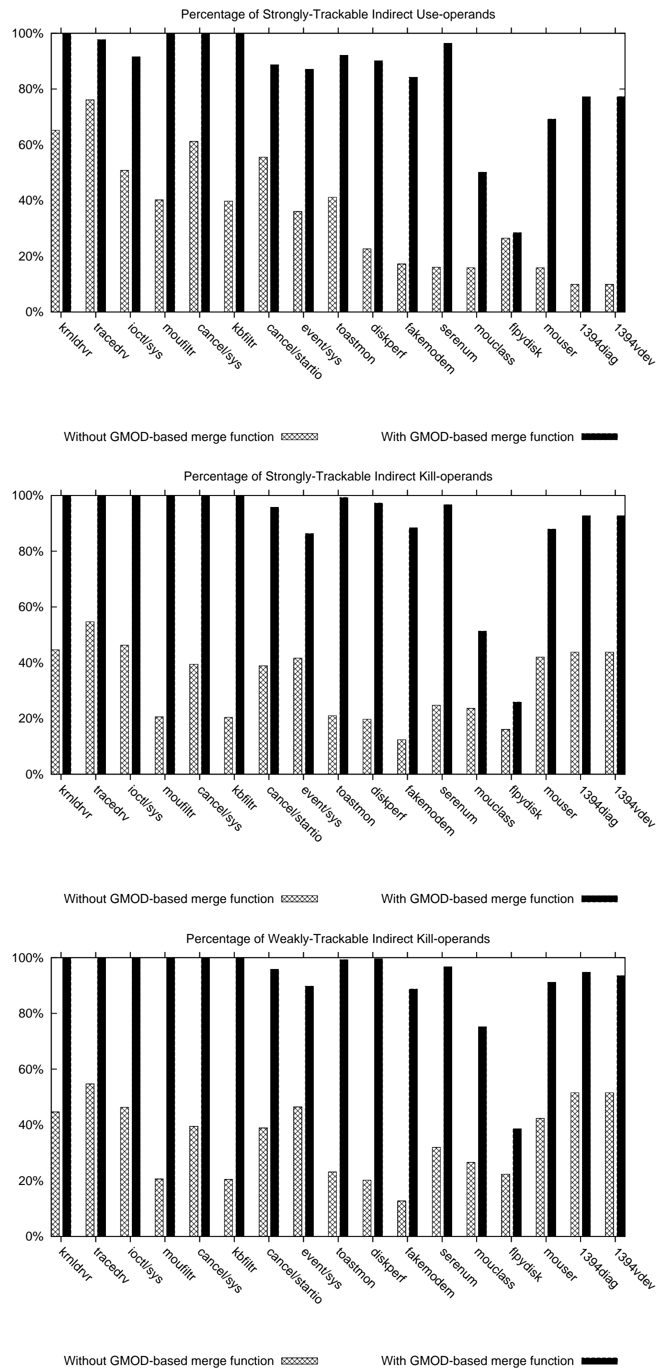
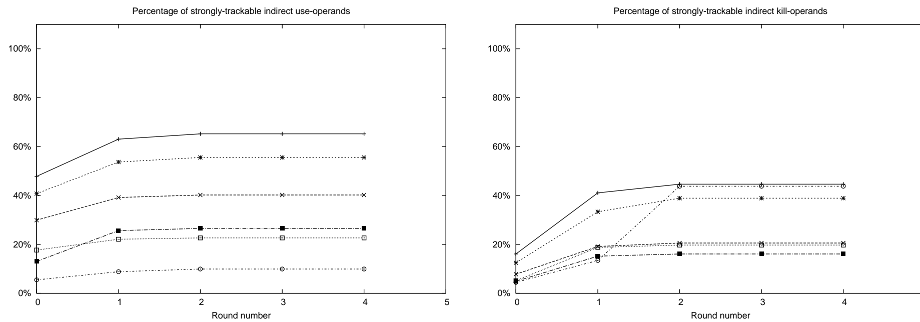
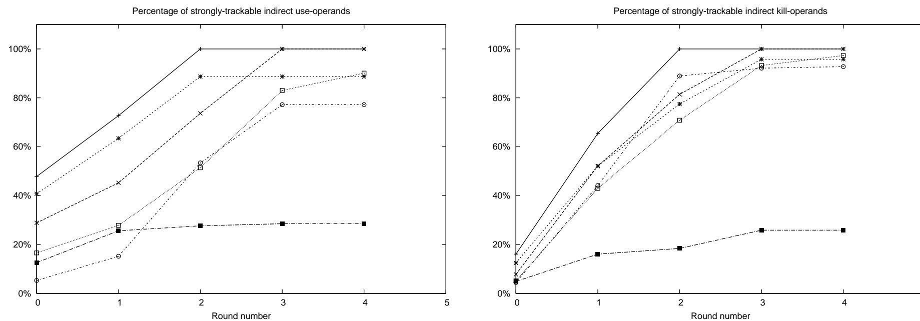


Fig. 6. Effects of using the GMOD-based merge function on the percentages of strongly-trackable indirect use-operands, strongly-trackable indirect kill-operands, and weakly-trackable indirect kill-operands.



(a) Percentages for the VSA algorithm *without* the GMOD-based merge function.



(b) Percentages for the VSA algorithm *with* the GMOD-based merge function.

Fig. 7. Percentage of strongly-trackable indirect operands in different rounds (for the six device drivers listed in **boldface** in Tab. 1).

6 Related Work

A large amount of related work has already been mentioned in the body of the paper. This section discusses a few additional issues.

Knoop and Steffen [45] introduced the use of merge functions in interprocedural dataflow analysis as a way to handle local variables at procedure returns. At a call site at which procedure P calls procedure Q, the local variables of P are modeled as if the current incarnations of P's locals are stored in locations that are inaccessible to Q and to procedures transitively called by Q—consequently, the contents of P's locals cannot be affected by the call to Q. To create the abstract state at the end-call node in P, the merge function integrates the stored abstract values for P's locals into the abstract state returned by Q. This idea is used in many other papers on interprocedural dataflow analysis, including [58, 42, 47, 1], as well as several systems (e.g., [57, 43]).

Note that this model agrees with programming languages like Java, where it is not possible to have pointers to local variables (i.e., pointers into the stack). For machine-code programs, as well as programs written in languages such as C and C++ (where the address-of operator (`&`) allows the address of a local variable

to be obtained), if P passes the address of a local to Q, it is possible for Q (or a procedure transitively called from Q) to affect a local of P by making an indirect assignment through the address. Conventional interprocedural dataflow-analysis algorithms address this issue by (i) performing several preliminary analyses (e.g., first points-to analysis, which is used to determine IMOD information [25] for individual statements, and then GMOD analysis [25]), and (ii) using the GMOD-analysis results to create sound transformers for the primary interprocedural dataflow analysis of interest.

The approach taken in the algorithm from §4 is similar, except that because VSA is not only the primary interprocedural dataflow analysis of interest but is also used to obtain points-to information, VSA and GMOD analysis are iterated.

References

1. R. Alur and P. Madhusudan. Adding nesting structure to words. In *DLT*, 2006.
2. W. Amme, P. Braun, E. Zehendner, and F. Thomasset. Data dependence analysis of assembly code. *IJPP*, 2000.
3. W. Backes. *Programmanalyse des XRTL Zwischencodes*. PhD thesis, Universitaet des Saarlandes, 2004. (In German.).
4. G. Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, C.S. Dept., Univ. of Wisconsin, Madison, WI, August 2007. TR-1603.
5. G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86 – A platform for analyzing x86 executables, (tool demonstration paper). In *CC*, 2005.
6. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, 2004.
7. G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *SAS*, 2006.
8. G. Balakrishnan and T. Reps. DIVINE: DIscovering Variables IN Executables. In *VMCAI*, 2007.
9. G. Balakrishnan and T. Reps. Analyzing stripped device-driver executables. In *TACAS*, 2008.
10. G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *VSTTE*, 2007.
11. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S.K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys*, 2006.
12. T. Ball and S.K. Rajamani. The SLAM toolkit. In *CAV*, 2001.
13. J. Bergeron, M. Debbabi, J. Desharnais, M.M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. *IJRE*, 2001.
14. J. Bergeron, M. Debbabi, M.M. Erhioui, and B. Ktari. Static analysis of binary code to isolate malicious behaviors. In *WETICE*, 1999.
15. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Int. Conf. on Formal Methods in Prog. and their Appl.*, 1993.
16. D. Brumley and J. Newsome. Alias analysis for assembly. CMU-CS-06-180, School of Comp. Sci., Carnegie Mellon University, Pittsburgh, PA, December 2006.
17. W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30:775–802, 2000.
18. S. Chandra and T. Reps. Physical type checking for C. In *PASTE*, 1999.

19. B.-Y. Chang, M. Harren, and G.C. Necula. Analysis of low-level code using cooperating decompilers. In *SAS*, 2006.
20. H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *CCCS*, 2002.
21. M. Christodorescu, W.-H. Goh, and N. Kidd. String analysis for x86 binaries. In *PASTE*, 2005.
22. C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *ICSM*, 1997.
23. CodeSonar, GrammaTech, Inc., http://www.grammatech.com/products/codesonar.
24. CodeSurfer, GrammaTech, Inc., http://www.grammatech.com/products/codesurfer.
25. K.D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI*, 1988.
26. J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE*, 2000.
27. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL*, 1977.
28. M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *ACSAC*, 2006.
29. Coverity Prevent. http://www.coverity.com/products/prevent_analysis_engine.html.
30. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, 2002.
31. B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In *PDPTA*, 2000.
32. S.K. Debray, C. Linn, G.R. Andrews, and B. Schwarz. Stack analysis of x86 executables. www.cs.arizona.edu/∼debray/Publications/stack-analysis.pdf, 2004.
33. S.K. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *POPL*, 1998.
34. M.J. Van Emmerik. *Static Single Assignment for Decompilation*. PhD thesis, School of Inf. Tech. and Elec. Eng., Univ. of Queensland, Brisbane, AU, May 2007.
35. D.R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.
36. D. Gopan and T. Reps. Low-level library analysis and summarization. In *CAV*, 2007.
37. B. Guo, M.J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D.I. August. Practical and accurate low-level pointer analysis. In *CGO*, 2005.
38. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *STTT*, 2(4), 2000.
39. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
40. M. Howard. Some bad news and some good news, October 2002. MSDN, Microsoft Corp., http://msdn2.microsoft.com/en-us/library/ms972826.aspx.
41. IDAPro disassembler, http://www.datarescue.com/idabase/.
42. B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *SAS*, 2004.
43. N. Kidd, T. Reps, D. Melski, and A. Lal. WPDS++: A C++ library for weighted pushdown systems, 2004. http://www.cs.wisc.edu/wpis/wpds++/.
44. Á. Kiss, G. Lehotai J. Jász, and T. Gyimóthy. Interprocedural static slicing of binary executables. In *SCAM*, 2003.

45. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC*, 1992.
46. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *USENIX Sec. Symp.*, 2005.
47. A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *CAV*, 2005.
48. J.R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *PLDI*, 1995.
49. J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In *CC*, 2008.
50. E.W. Myers. Efficient applicative data types. In *POPL*, 1984.
51. M. Nita, D. Grossman, and C. Chambers. A theory of platform-dependent low-level software. In *POPL*, 2008.
52. PREfast with driver-specific rules, October 2004. WHDC, Microsoft Corp., http://www.microsoft.com/whdc/devtools/tools/PREfast-drv.mspx.
53. G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *POPL*, 1999.
54. J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. In *Trans. on Embedded Comp. Systs.*, 2005.
55. T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *PEPM*, 2006.
56. T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum. A next-generation platform for analyzing executables. In *APLAS*, 2005.
57. S. Schwoon. Moped system. http://www.fmi.uni-stuttgart.de/szs/tools/moped/.
58. H. Seidl and C. Fecht. Interprocedural analyses: A comparison. *JLP*, 2000.
59. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
60. D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, February 2000.
61. D.W. Wall. Systems for late code modification. In R. Giegerich and S.L. Graham, editors, *Code Generation – Concepts, Tools, Techniques*. Springer-Verlag, 1992.
62. C++ for kernel mode drivers: Pros and cons, February 2007. WHDC web site, http://www.microsoft.com/whdc/driver/kernel/KMcode.mspx.
63. http://www.microsoft.com/whdc/devtools/ddk/default.mspx.
64. Z. Xu, B. Miller, and T. Reps. Safety checking of machine code. In *PLDI*, 2000.
65. Z. Xu, B. Miller, and T. Reps. Typestate checking of machine code. In *ESOP*, 2001.
66. J. Zhang, R. Zhao, and J. Pang. Parameter and return-value analysis of binary executables. In *COMPSAC*, 2007.