WYSINWYX: What You See Is Not What You eXecute

G. Balakrishnan[1], T. Reps[1,2], D. Melski[2], and T. Teitelbaum[2]

[1] Comp. Sci. Dept., University of Wisconsin; {bgogul,reps}@cs.wisc.edu
[2] GrammaTech, Inc.; {melski,tt}@grammatech.com

Abstract. What You See Is Not What You eXecute: computers do not execute source-code programs; they execute machine-code programs that are generated from source code. Not only can the WYSINWYX phenomenon create a mismatch between what a programmer intends and what is actually executed by the processor, it can cause analyses that are performed on source code to fail to detect certain bugs and vulnerabilities. This issue arises regardless of whether one's favorite approach to assuring that programs behave as desired is based on theorem proving, model checking, or abstract interpretation.

1 Introduction

Recent research in programming languages, software engineering, and computer security has led to new kinds of tools for analyzing code for bugs and security vulnerabilities [23, 39, 18, 12, 8, 4, 9, 25, 15]. In these tools, static analysis is used to determine a conservative answer to the question "Can the program reach a bad state?"[3] However, these tools all focus on analyzing *source code* written in a high-level language, which has certain drawbacks. In particular, there can be a mismatch between what a programmer intends and what is actually executed by the processor. Consequently, analyses that are performed on source code can fail to detect certain bugs and vulnerabilities due to the WYSINWYX phenomenon: "What You See Is Not What You eXecute". The following source-code fragment, taken from a login program, illustrates the issue [27]:

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

The login program temporarily stores the user's password—in clear text—in a dynamically allocated buffer pointed to by the pointer variable `password`. To minimize the lifetime of the password, which is sensitive information, the code fragment shown above zeroes-out the buffer pointed to by `password` before returning it to the heap. Unfortunately, a compiler that performs useless-code elimination may reason that the program never uses the values written by the call on `memset`, and therefore the call on memset can be removed—thereby leaving sensitive information exposed in the heap. This is not just hypothetical; a similar vulnerability was discovered during the Windows security push in 2002 [27]. This vulnerability is invisible in the source code; it can only be detected by examining the low-level code emitted by the optimizing compiler.

The WYSINWYX phenomenon is not restricted to the presence or absence of procedure calls; on the contrary, it is pervasive:

- Bugs and security vulnerabilities can exist because of a myriad of platform-specific details due to features (and idiosyncrasies) of compilers and optimizers, including

[3] 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 [13].

  - memory-layout details, such as (i) the positions (i.e., offsets) of variables in the runtime stack's activation records, and (ii) padding between structure fields.
  - register usage
  - execution order (e.g., of actual parameters)
  - optimizations performed
  - artifacts of compiler bugs

  Access to such information can be crucial; for instance, many security exploits depend on platform-specific features, such as the structure of activation records. Vulnerabilities can escape notice when a tool does not have information about adjacency relationships among variables.
- Analyses based on source code[4] typically make (unchecked) assumptions, e.g., that the program is ANSI-C compliant. This often means that an analysis does not account for behaviors that are allowed by the compiler (e.g., arithmetic is performed on pointers that are subsequently used for indirect function calls; pointers move off the ends of arrays and are subsequently dereferenced; etc.)
- Programs typically make extensive use of libraries, including dynamically linked libraries (DLLs), which may not be available in source-code form. Typically, analyses are performed using code stubs that model the effects of library calls. Because these are created by hand they are likely to contain errors, which may cause an analysis to return incorrect results.
- Programs are sometimes modified subsequent to compilation, e.g., to perform optimizations or insert instrumentation code [40]. (They may also be modified to insert malicious code.) Such modifications are not visible to tools that analyze source.
- The source code may have been written in more than one language. This complicates the life of designers of tools that analyze source code because multiple languages must be supported, each with its own quirks.
- Even if the source code is primarily written in one high-level language, it may contain inlined assembly code in selected places. Source-level tools typically either skip over inlined assembly code [11] or do not push the analysis beyond sites of inlined assembly code [1].

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

- Source-level tools are only applicable when source is available, which limits their usefulness in security applications (e.g., to analyzing code from open-source projects).
- Even if source code is available, a substantial amount of information is hidden from analyses that start from source code, which can cause bugs, security vulnerabilities, and malicious behavior to be invisible to such tools. Moreover, a source-code tool that strives to have greater fidelity to the program that is actually executed would have to duplicate all of the choices made by the compiler and optimizer; such an approach is doomed to failure.

The issue of whether source code is the appropriate level for verifying program properties is one that should concern all who are interested in assuring that programs

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

behave as desired. The issues discussed above arise regardless of whether one's favorite approach is based on theorem proving, model checking, or abstract interpretation.

The remainder of the paper is organized as follows: §2 presents some examples that show why analysis of an executable can provide more accurate information than a source-level analysis. §3 discusses different approaches to analyzing executables. §4 describes our work on CodeSurfer/x86, as an example of how it is possible to analyze executables in the absence of source code.

2 Advantages of Analyzing Executables

The example presented in §1 showed that an overzealous optimizer can cause there to be a mismatch between what a programmer intends and what is actually executed by the processor. Additional examples of this sort have been discussed by Boehm [5]. He points out that when threads are implemented as a library (e.g., for use in languages such as C and C++, where threads are not part of the language specification), compiler transformations that are reasonable in the absence of threads can cause multi-threaded code to fail—or exhibit unexpected behavior—for subtle reasons that are not visible to tools that analyze source code.

A second class of examples for which analysis of an executable can provide more accurate information than a source-level analysis arises because, for many programming languages, certain behaviors are left unspecified by the semantics. In such cases, a source-level analysis must account for all possible behaviors, whereas an analysis of an executable generally only has to deal with *one* possible behavior—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 of the abstract descriptors that result from analyzing each permutation of the actuals. In contrast, an analysis of an executable only needs to analyze the particular sequence of instructions that lead up to the call.

A second example in this class involves pointer arithmetic and an indirect call:

```
int (*f)(void);
int diff = (char*)&f2 - (char*)&f1; // The offset between f1 and f2
f = &f1;
f = (int (*)())((char*)f + diff); // f now points to f2
(*f)(); // indirect call;
```

Existing source-level analyses (that we know of) are ill-prepared to handle the above code. The conventional assumption is that arithmetic on function pointers leads to undefined behavior, so source-level analyses either (a) assume that the indirect function call might call any function, or (b) ignore the arithmetic operations and assume that the indirect function call calls `f1` (on the assumption that the code is ANSI-C compliant). In contrast, the analysis described by Balakrishnan and Reps [3] correctly identifies `f2` as the invoked function. Furthermore, the analysis can detect when arithmetic on addresses creates an address that does not point to the beginning of a function; the use of such an address to perform a function "call" is likely to be a bug (or else a very subtle, deliberately introduced security vulnerability).

A third example related to unspecified behavior is shown in Fig. 1. The C code on the left uses an uninitialized variable (which triggers a compiler warning, but compiles successfully). A source-code analyzer must assume that `local` can have any value, and therefore the value of `v` in `main` is either 1 or 2. The assembly listings on the right show how the C code could be compiled, including two variants for the prolog of function `callee`. The Microsoft compiler (cl) uses the second variant, which includes the following strength reduction:

> *The instruction `sub esp,4` that allocates space for `local` is replaced by a `push` instruction of an arbitrary register (in this case, `ecx`).*

In contrast to an analysis based on source code, an analysis of an executable can determine that this optimization results in `local` being initialized to 5, and therefore `v` in `main` can only have the value 1.

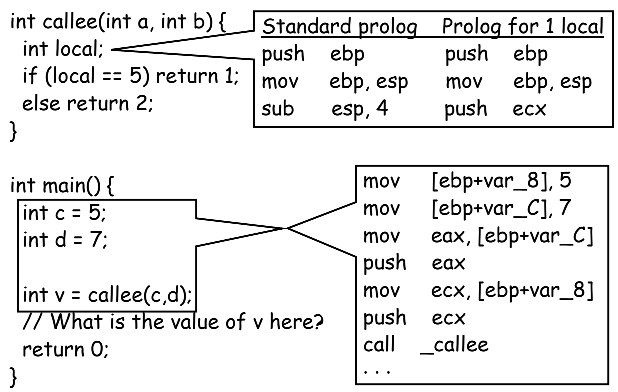

Fig. 1. Example of unexpected behavior due to compiler optimization. The box at the top right shows two variants of code generated by an optimizing compiler for the prolog of `callee`. Analysis of the second of these reveals that the variable `local` necessarily contains the value 5.

A fourth example related to unspecified behavior involves a function call that passes fewer arguments than the procedure expects as parameters. (Many compilers accept such (unsafe) code as an easy way to implement functions that take a variable number of parameters.) With most compilers, this effectively means that the call-site passes some parts of one or more local variables of the calling procedure as the remaining parameters (and, in effect, these are passed by reference—an assignment to such a parameter in the callee will overwrite the value of the corresponding local in the caller.) An analysis that works on executables can be created that is capable of determining what the extra parameters are [3], whereas a source-level analysis must either make a cruder over-approximation or an unsound under-approximation.

3 Approaches to Analyzing Executables

The examples in §2 illustrate some of the advantages of analyzing executables instead of source code: an executable contains the actual instructions that will be executed, and hence reveals more accurate information about the behaviors that might occur during execution; an analysis can incorporate platform-specific details, including memory layout, register usage, execution order, optimizations, and artifacts of compiler bugs.

Moreover, many of the issues that arise when analyzing source code disappear when analyzing executables:

– The entire program can be analyzed—including libraries that are linked to the program. Because library code can be analyzed directly, it is not necessary to rely on potentially unsound models of library functions.
– If an executable has been modified subsequent to compilation, such modifications are visible to the analysis tool.
– Source code does not have to be available.
– Even if the source code was written in more than one language, a tool that analyzes executables only needs to support one language.
– Instructions inserted because of inlined assembly directives in the source code are visible, and do not need to be treated any differently than other instructions.

The challenge is to build tools that can benefit from these advantages to provide a level of precision that would not otherwise be possible.

One dichotomy for classifying approaches is whether the tool assumes that information is available in addition to the executable itself—such as the source code, symbol-table information, and debugging information. For instance, the aim of *translation validation* [32, 31] is to verify that compilation does not change the semantics of a program. A translation-validation system receives the source code and target code as input, and attempts to verify that the target code is a correct implementation (i.e., a refinement) of the source code. Rival [35] presents an analysis that uses abstract interpretation to check whether the assembly code produced by a compiler possesses the same safety properties as the original source code. The analysis assumes that both source code and debugging information is available. First, the source code and the assembly code of the program are analyzed. Next, the debugging information is used to map the results of assembly-code analysis back to the source code. If the results for the corresponding program points in the source code and the assembly code are compatible, then the assembly code possesses the same safety properties as the source code.

4 Analyzing Executables in the Absence of Source Code

For the past few years, we have been working to create a platform to support the analysis of executables in the absence of source code. The goal of the work is to extend static vulnerability-analysis techniques to work directly on stripped executables. We have developed a prototype tool set for analyzing x86 executables. The members of the tool set are: *CodeSurfer/x86*, *WPDS++*, and the *Path Inspector*. Fig. 2 shows how the components of CodeSurfer/x86 fit together.

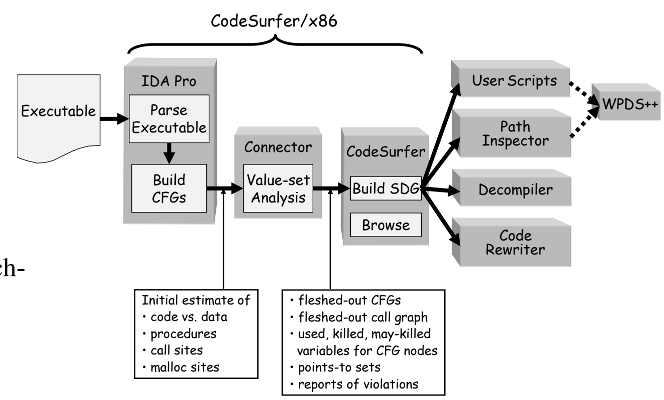

Fig. 2. Organization of CodeSurfer/x86 and companion tools.

Recovering IRs from x86 executables. To be able to apply analysis techniques like the ones used in [23, 39, 18, 12, 8, 4, 9, 25, 15], one already encounters a challenging program-analysis problem. From the perspective of the model-checking community, one would consider the problem to be that of "model extraction": one needs to extract a suitable *model* from the executable. From the perspective of the compiler community, one would consider the problem to be "IR recovery": one needs to recover *intermediate representations* from the executable that are similar to those that would be available had one started from source code.

To solve the IR-recovery problem, several obstacles must be overcome:

- For many kinds of potentially malicious programs, symbol-table and debugging information is entirely absent. Even if it is present, it cannot be relied upon.
- To understand memory-access operations, it is necessary to determine the set of addresses accessed by each operation. This is difficult because
  - While some memory operations use explicit memory addresses in the instruction (easy), others use indirect addressing via address expressions (difficult).
  - Arithmetic on addresses is pervasive. For instance, even when the value of a local variable is loaded from its slot in an activation record, address arithmetic is performed.
  - There is no notion of type at the hardware level, so address values cannot be distinguished from integer values.

To recover IRs from x86 executables, CodeSurfer/x86 makes use of both IDAPro [28], a disassembly toolkit, and GrammaTech's CodeSurfer system [11], a toolkit for building program-analysis and inspection tools.

An x86 executable is first disassembled using IDAPro. In addition to the disassembly listing, IDAPro also provides access to the following information: (1) procedure boundaries, (2) calls to library functions, and (3) statically known memory addresses and offsets. IDAPro provides access to its internal resources via an API that allows users to create plug-ins to be executed by IDAPro. We created a plug-in to IDAPro, called the Connector, that creates data structures to represent the information that it obtains from IDAPro. The IDAPro/Connector combination is also able to create the same data structures for dynamically linked libraries, and to link them into the data structures that represent the program itself. This infrastructure permits whole-program analysis to be carried out—including analysis of the code for all library functions that are called.

Using the data structures in the Connector, we implemented a static-analysis algorithm called *value-set analysis* (VSA) [3]. VSA does not assume the presence of symbol-table or debugging information. Hence, as a first step, a set of data objects called a-locs (for "abstract locations") is determined based on the static memory addresses and offsets provided by IDAPro. VSA is a combined numeric and pointer-analysis algorithm that determines an over-approximation of the set of numeric values and addresses (or *value-set*) that each a-loc holds at each program point.[5] A key feature of VSA is that it tracks integer-valued and address-valued quantities simultaneously. This is crucial for analyzing executables because numeric values and addresses are indistinguishable at execution time.

[5] VSA is a flow-sensitive, interprocedural dataflow-analysis algorithm that uses the "call-strings" approach [38] to obtain a degree of context sensitivity.

IDAPro does not identify the targets of all indirect jumps and indirect calls, and therefore the call graph and control-flow graphs that it constructs are not complete. However, the information computed during VSA can be used to augment the call graph and control-flow graphs on-the-fly to account for indirect jumps and indirect calls.

VSA also checks whether the executable conforms to a "standard" compilation model—i.e., a runtime stack is maintained; activation records are pushed onto the stack on procedure entry and popped from the stack on procedure exit; a procedure does not modify the return address on stack; the program's instructions occupy a fixed area of memory, are not self-modifying, and are separate from the program's data. If it cannot be confirmed that the executable conforms to the model, then the IR is possibly incorrect. For example, the call-graph can be incorrect if a procedure modifies the return address on the stack. Consequently, VSA issues an error report whenever it finds a possible violation of the standard compilation model; these represent possible memory-safety violations. The analyst can go over these reports and determine whether they are false alarms or real violations.

Once VSA completes, the value-sets for the a-locs at each program point are used to determine each point's sets of used, killed, and possibly-killed a-locs; these are emitted in a format that is suitable for input to CodeSurfer. CodeSurfer then builds a collection of IRs, consisting of abstract-syntax trees, control-flow graphs (CFGs), a call graph, a system dependence graph (SDG) [26], VSA results, the sets of used, killed, and possibly killed a-locs at each instruction, and information about the structure and layout of global memory, activation records, and dynamically allocated storage. CodeSurfer supports both a graphical user interface (GUI) and an API (as well as a scripting language) to provide access to these structures.

Model-checking facilities. For model checking, the CodeSurfer/x86 IRs are used to build a *weighted pushdown system* (WPDS) [7, 33, 34, 30] that models possible program behaviors. Weighted pushdown systems generalize a model-checking technology known as *pushdown systems* (PDSs) [6, 19], which have been used for software model checking in the Moped [37, 36] and MOPS [9] systems. Compared to ordinary (unweighted) PDSs, WPDSs are capable of representing more powerful kinds of abstractions of runtime states [34, 30], and hence go beyond the capabilities of PDSs. For instance, the use of WPDSs provides a way to address certain kinds of security-related queries that cannot be answered by MOPS.

WPDS++ [29] is a library that implements the symbolic reachability algorithms from [34, 30] on weighted pushdown systems. We follow the standard approach of using a pushdown system (PDS) to model the interprocedural control-flow graph (one of CodeSurfer/x86's IRs). The stack symbols correspond to program locations; there is only a single PDS state; and PDS rules encode control flow as follows:

Rule	Control flow modeled
$q\langle u\rangle \hookrightarrow q\langle v\rangle$	Intraprocedural CFG edge $u \rightarrow v$
$q\langle c\rangle \hookrightarrow q\langle entry_P\ r\rangle$	Call to P from c that returns to r
$q\langle x\rangle \hookrightarrow q\langle\rangle$	Return from a procedure at exit node x

In a configuration of the PDS, the symbol at the top of the stack corresponds to the current program location, and the rest of the stack holds return-site locations—this allows the PDS to model the behavior of the program's runtime execution stack.

An encoding of the interprocedural control-flow as a pushdown system is sufficient for answering queries about reachable control states (as the Path Inspector does; see below): the reachability algorithms of WPDS++ can determine if an undesirable PDS configuration is reachable. However, WPDS++ also supports *weighted* PDSs, which are PDSs in which each rule is weighted with an element of a (user-defined) semiring. The use of weights allows WPDS++ to perform interprocedural dataflow analysis by using the semiring's *extend* operator to compute weights for sequences of rule firings and using the semiring's *combine* operator to take the meet of weights generated by different paths [34, 30]. (When the weights on rules are conservative abstract data transformers, an over-approximation to the set of reachable concrete configurations is obtained, which means that counterexamples reported by WPDS++ may actually be infeasible.)

The advantage of answering reachability queries on WPDSs over conventional dataflow-analysis methods is that the latter merge together the values for all states associated with the same program point, regardless of the states' calling context. With WPDSs, queries can be posed with respect to a regular language of stack configurations [7, 33, 34, 30]. (Conventional merged dataflow information can also be obtained [34].)

The Path Inspector provides a user interface for automating safety queries that are only concerned with the possible control configurations that an executable can reach. It uses an automaton-based approach to model checking: the query is specified as a finite automaton that captures forbidden sequences of program locations. This "query automaton" is combined with the program model (a WPDS) using a cross-product construction, and the reachability algorithms of WPDS++ are used to determine if an error configuration is reachable. If an error configuration is reachable, then *witnesses* (see [34]) can be used to produce a program path that drives the query automaton to an error state.

The Path Inspector includes a GUI for instantiating many common reachability queries [17], and for displaying counterexample paths in the disassembly listing.[6] In the current implementation, transitions in the query automaton are triggered by program points that the user specifies either manually, or using result sets from CodeSurfer queries. Future versions of the Path Inspector will support more sophisticated queries in which transitions are triggered by matching an AST pattern against a program location, and query states can be instantiated based on pattern bindings.

Related work. 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.

[6] We assume that source code is not available, but the techniques extend naturally if it is: one can treat the executable code as just another IR in the collection of IRs obtainable from source code. The mapping of information back to the source code would be similar to what C source-code tools already have to perform because of the use of the C preprocessor (although the kind of issues that arise when debugging optimized code [24, 42, 14] complicate matters).

The basic goal of the algorithm proposed by Debray et al. [16] 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.

Cifuentes and Fraboulet [10] give an algorithm to identify an intraprocedural slice of an executable by following the program's use-def chains. However, their algorithm also makes no attempt to track values that are not in registers, and hence cuts short the slice when a load from memory is encountered.

The two pieces of work that are most closely related to VSA are the algorithm for data-dependence analysis of assembly code of Amme et al. [2] and the algorithm for pointer analysis on a low-level intermediate representation of Guo et al. [22]. 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. [22] 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 [41] to achieve context-sensitivity. The transfer functions are parameterized by "unknown initial values" (UIVs); however, it is not clear whether the the algorithm accounts for the possibility of called procedures corrupting the memory locations that the UIVs represent.

Challenges for the future. There are a number of challenging problems for which additional research is needed. Most of these are similar to the challenges one faces when analyzing source code:

– efficiency and scalability of analysis algorithms, including how to create summary transformers for procedures
– accounting for non-local transfers of control (e.g., `setjmp`/`longjmp` and C++ exception handling)
– analysis of variable-argument functions
– analysis of multi-threaded code
– analysis of heap-allocated data structures

As with source-code analysis, it would be useful to develop specialized analyses for particular kinds of data or particular programming idioms, including

– how strings are used in the program
– the "macro-level" effects of loops that perform array operations (e.g., that an array-initialization loop initializes all elements of an array [21])
– the effects of loops that perform sentinel search
– analysis of self-modifying code [20]

References

1. PREfast with driver-specific rules, October 2004. Windows Hardware and Driver Central (WHDC) web site, http://www.microsoft.com/whdc/devtools/tools/PREfast-drv.mspx.

2. W. Amme, P. Braun, E. Zehendner, and F. Thomasset. Data dependence analysis of assembly code. *Int. J. Parallel Proc.*, 2000.
3. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Comp. Construct.*, pages 5–23, 2004.
4. T. Ball and S.K. Rajamani. The SLAM toolkit. In *Computer Aided Verif.*, volume 2102 of *Lec. Notes in Comp. Sci.*, pages 260–264, 2001.
5. H.-J. Boehm. Threads cannot be implemented as a library. In *PLDI*, pages 261–268, 2005.
6. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proc. CONCUR*, 1997.
7. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, pages 62–73, 2003.
8. W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software–Practice&Experience*, 30:775–802, 2000.
9. H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Conf. on Comp. and Commun. Sec.*, pages 235–244, November 2002.
10. C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *Int. Conf. on Softw. Maint.*, pages 188–195, 1997.
11. CodeSurfer, GrammaTech, Inc., http://www.grammatech.com/products/codesurfer/.
12. 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.
13. 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.
14. D.S. Coutant, S. Meloy, and M. Ruscetta. DOC: A practical approach to source-level debugging of globally optimized code. In *PLDI*, 1988.
15. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, 2002.
16. S.K. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *POPL*, 1998.
17. M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, 1999.
18. D.R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Op. Syst. Design and Impl.*, pages 1–16, 2000.
19. A. Finkel, B.Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Elec. Notes in Theor. Comp. Sci.*, 9, 1997.
20. R. Gerth. Formal verification of self modifying code. In *Proc. Int. Conf. for Young Computer Scientists*, pages 305–313, 1991.
21. D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, 2005.
22. B. Guo, M.J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D.I. August. Practical and accurate low-level pointer analysis. In *3nd Int. Symp. on Code Gen. and Opt.*, 2005.
23. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Softw. Tools for Tech. Transfer*, 2(4), 2000.
24. J.L. Hennessy. Symbolic debugging of optimized code. *Trans. on Prog. Lang. and Syst.*, 4(3):323–344, 1982.
25. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
26. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Trans. on Prog. Lang. and Syst.*, 12(1):26–60, January 1990.
27. M. Howard. Some bad news and some good news. *MSDN*, October 2002. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure10102002.asp.

28. IDAPro disassembler, http://www.datarescue.com/idabase/.
29. 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++/.
30. A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *CAV*, 2005.
31. G. Necula. Translation validation for an optimizing compiler. In *PLDI*, 2000.
32. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, 1998.
33. T. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *SAS*, 2003.
34. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. of Comp. Prog.* To appear.
35. X. Rival. Abstract interpretation based certification of assembly code. In *VMCAI*, 2003.
36. S. Schwoon. Moped system. "http://www.fmi.uni-stuttgart.de/szs/tools/moped/".
37. S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technical Univ. of Munich, Munich, Germany, July 2002.
38. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
39. D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Dist. Syst. Security*, February 2000.
40. D.W. Wall. Systems for late code modification. In R. Giegerich and S.L. Graham, editors, *Code Generation – Concepts, Tools, Techniques*, pages 275–293. Springer-Verlag, 1992.
41. R.P. Wilson and M.S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, pages 1–12, 1995.
42. P.T. Zellweger. *Interactive Source-Level Debugging of Optimized Programs*. PhD thesis, Univ. of California, Berkeley, 1984.