CodeSurfer/x86—A Platform for Analyzing x86 Executables ⋆

Gogul Balakrishnan[1], Radu Gruian[2], Thomas Reps[1,2], and Tim Teitelbaum[2]

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

Abstract. CodeSurfer/x86 is a prototype system for analyzing x86 executables. It uses a static-analysis algorithm called *value-set analysis* (VSA) to recover intermediate representations that are similar to those that a compiler creates for a program written in a high-level language. A major challenge in building an analysis tool for executables is in providing useful information about operations involving memory. This is difficult when symbol-table and debugging information is absent or untrusted. CodeSurfer/x86 overcomes these challenges to provide an analyst with a powerful and flexible platform for investigating the properties and behaviors of potentially malicious code (such as COTS components, plugins, mobile code, worms, Trojans, and virus-infected code) using (i) CodeSurfer/x86's GUI, (ii) CodeSurfer/x86's scripting language, which provides access to all of the intermediate representations that CodeSurfer/x86 builds for the executable, and (iii) GrammaTech's Path Inspector, which is a tool that uses a sophisticated pattern-matching engine to answer questions about the flow of execution in a program.

1 Introduction

In recent years, there has been a growing need for tools that analyze executables. Computer-security issues provide one motivation: one would like to ensure that third-party applications do not perform malicious operations, and in this context it is important for analysts to be able to decipher the behavior of Trojans, worms, and virus-infected code. Static analysis provides techniques that can help with such problems; however, there are several obstacles that must be overcome:

– For potentially malicious programs, symbol-table and debugging information is either entirely absent, or cannot be relied upon if present.
– Instructions that perform memory operations use explicit memory addresses and indirect addressing, which complicates the task of understanding the overall behavior of the code.

Several others [3, 2, 10, 5, 12] have proposed algorithms for statically analyzing executables. However, existing tools assume the presence of symbol-table and/or debugging information, or ignore instructions with memory operands altogether, or assume that an instruction with memory operands can write-to/read-from any part of memory. None of these solutions are satisfactory in terms of understanding how an x86 executable works. Recently, Balakrishnan and Reps developed a static-analysis algorithm, called *value-set analysis* (VSA),

⋆ Supported by Air Force (AFRL/Rome) SBIR contracts F30602-01-{C-0112, C-0051}, ONR contracts N00014-{02-C-0188, 03-C-0502, 01-1-0708, 01-1-0796}, and NSF grant CCR-9986308.

to recover information about the contents of memory locations and how they are manipulated by an executable [1]. By combining VSA with facilities provided by the IDAPro and CodeSurfer toolkits, we have created CodeSurfer/x86, a prototype tool for browsing, inspecting, and analyzing x86 executables. From an x86 executable, CodeSurfer/x86 recovers an intermediate representation that is similar to what would be created by a compiler for a program written in a high-level language. In this document, we emphasize the facilities of CodeSurfer/x86 that provide an analyst with a powerful and flexible platform for investigating the properties and behaviors of an x86 executable.

Because CodeSurfer/x86 works on the actual executable code that is run on the machine, it automatically takes into account platform-specific aspects of the code, such as the positions (i.e., offsets) of variables in the run-time stack's activation records. This is a key ability, because many security exploits depend on platform-specific features, such as the structure of activation records. In this sense, CodeSurfer/x86 is a "higher fidelity" tool than most tools that analyze source code.

2 CodeSurfer/x86

CodeSurfer/x86 is the outcome of a joint project between the Univ. of Wisconsin and GrammaTech, Inc. CodeSurfer/x86 makes use of both IDAPro [9], a disassembly toolkit, and GrammaTech's CodeSurfer system [4], a toolkit for building program-analysis and inspection tools. Fig. 1 shows the various components of CodeSurfer/x86. This section sketches how these components are combined in CodeSurfer/x86.

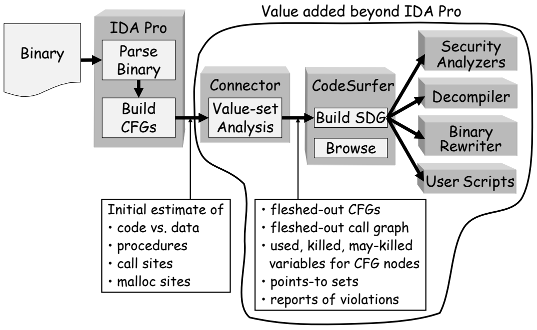

Fig. 1. Organization of CodeSurfer/x86.

An x86 executable is first disassembled using IDAPro. In addition to the disassembly listing and control-flow graphs, IDAPro also provides access to the following information: (1) procedure boundaries, (2) calls to library functions (identified using an algorithm called the Fast Library Identification and Recognition Technology (FLIRT) [7]), 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 obtained 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.*

Based on the data structures in the Connector, we implemented a static analysis algorithm called *value-set analysis* (VSA) [1]. VSA does not assume the presence of symbol-table and debugging information.[3] 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 or addresses that each a-loc holds at each program point. The set of addresses and numeric values is referred to as a *value-set*. 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 in an executable.

Note that 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. In fact, the relationship between VSA and the preliminary IRs created by IDAPro is similar to the relationship between a points-to-analysis algorithm in a C compiler and the preliminary IRs created by the C compiler's front end. In both cases, the preliminary IRs are fleshed out during the course of analysis.

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 takes in this information and builds a collection of IRs, consisting of abstract-syntax trees, control-flow graphs (CFGs), a call graph, and a system dependence graph (SDG). An SDG consists of a set of program dependence graphs (PDGs), one for each procedure in the program. A vertex in a PDG corresponds to a construct in the program, such as a statement or instruction, a call to a procedure, an actual parameter of a call, or a formal parameter of a procedure. The edges correspond to data and control dependences between the vertices [6]. The PDGs are connected together with interprocedural edges that represent control dependences between procedure calls and entries, and data dependences between actual parameters and formal parameters/return values.

Dependence graphs are invaluable for many applications, because they highlight chains of dependent instructions that may be widely scattered through the program. For example, given a instruction, it is often useful to know its *data-dependence predecessors* (instructions that write to locations read by that instruction) and its *control-dependence predecessors* (control points that may affect whether a given instruction gets executed). Similarly, it may be useful to

[3] Although VSA does not need debugging/symbol-table information, in principle, it would be possible to extend VSA to use such information.

know for a given instruction its *data-dependence successors* (instructions that read locations written by that instruction) and *control-dependence* successors (instructions whose execution depends on the decision made at a given control point).

3 CodeSurfer/x86 Facilities

As described in the Sect. 2, given an executable as input, CodeSurfer/x86 builds a collection of IRs for it. In addition to building the IRs, CodeSurfer/x86 also checks whether the executable conforms to a "standard" compilation model—i.e., a runtime stack is maintained; activation records (ARs) 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 the 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 will be incorrect if a procedure modifies the return address on the stack. Consequently, CodeSurfer/x86 issues error reports if it finds one or more violations of the "standard" compilation model. The analyst can go over these reports and determine whether they are false alarms or real violations.

CodeSurfer's GUI supports browsing ("surfing") of an SDG, along with a variety of operations for making queries about the SDG—such as slicing [8] and chopping [11].[4] The GUI allows a user to navigate through the assembly code using these dependences in a manner analogous to navigating the World Wide Web. CodeSurfer's API provides a programmatic interface to these operations, as well as to lower-level information, such as the individual nodes and edges of the program's SDG, call graph, and control-flow graph, and a node's sets of used, killed, and possibly-killed a-locs. By writing programs that traverse CodeSurfer's IRs to implement additional program analyses, the API can be used to extend CodeSurfer's capabilities.

CodeSurfer/x86 can be used in conjunction with GrammaTech's Path Inspector, which is a tool that uses a sophisticated pattern-matching engine to answer questions about the flow of execution in a program. The Path Inspector checks *sequencing properties* of events in a program, which—in the context of security analysis, for example—can be used to answer such questions as "Is it possible for the program to bypass the authentication routine?" (which indicates that the program may contain a trapdoor).

With the Path Inspector, such questions are posed as questions about the existence of problematic event sequences; after checking the query, if a problematic path exists, it is displayed in the Path Explorer tool. This lists all of the program points that may occur along the problematic path. These items are

[4] A backward slice of a program with respect to a set of program points S is the set of all program points that might affect the computations performed at S; a forward slice with respect to S is the set of all program points that might be affected by the computations performed at members of S [8]. A program chop between a set of source program points S and a set of target program points T shows how S can affect the points in T [11]. Chopping is a key operation in information-flow analysis.

linked to the disassembly; the analyst can navigate from a point in the path to the corresponding assembly-code element. In addition, the Path Inspector allows the analyst to step forward and backward through the path, while simultaneously stepping through the assembly code. (The code-stepping operations are similar to the single-stepping operations in a traditional debugger.)

References

1. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Comp. Construct.*, pages 5–23, 2004.
2. C. Cifuentes and A. Fraboulet. Interprocedural data flow recovery of high-level language code from assembly. Technical Report 421, Univ. Queensland, 1997.
3. C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation. In *Int. Conf. on Softw. Maint.*, pages 228–237, 1998.
4. CodeSurfer, GrammaTech, Inc., http://www.grammatech.com/products/codesurfer/.
5. S.K. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Princ. of Prog. Lang.*, pages 12–24, 1998.
6. J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *Trans. on Prog. Lang. and Syst.*, 3(9):319–349, 1987.
7. Fast library identification and recognition technology, DataRescue sa/nv, Liège, Belgium, http://www.datarescue.com/idabase/flirt.htm.
8. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Trans. on Prog. Lang. and Syst.*, 12(1):26–60, January 1990.
9. IDAPro disassembler, http://www.datarescue.com/idabase/.
10. A. Mycroft. Type-based decompilation. In *European Symp. on Programming*, 1999.
11. T. Reps and G. Rosay. Precise interprocedural chopping. In *Found. of Softw. Eng.*, 1995.
12. X. Rival. Abstract interpretation based certification of assembly code. In *Int. Conf. on Verif., Model Checking, and Abs. Int.*, 2003.