Model Checking x86 Executables with CodeSurfer/x86 and WPDS++

G. Balakrishnan[1], T. Reps[1,2], N. Kidd[1], A. Lal[1], J. Lim[1],
D. Melski[2], R. Gruian[2], S. Yong[2], C.-H. Chen, and T. Teitelbaum[2]

[1] Comp. Sci. Dept., University of Wisconsin; {bgogul,reps,kidd,akash,junghee}@cs.wisc.edu
[2] GrammaTech, Inc.; {melski,radu,suan,chi-hua,tt}@grammatech.com

Abstract. This paper presents a toolset for model checking x86 executables. The members of the toolset are *CodeSurfer/x86*, *WPDS++*, and the *Path Inspector*. CodeSurfer/x86 is used to extract a model from an executable in the form of a *weighted pushdown system*. WPDS++ is a library for answering generalized reachability queries on weighted pushdown systems. The Path Inspector is a software model checker built on top of CodeSurfer and WPDS++ that supports safety queries about the program's possible control configurations.

1 Introduction

This paper presents a toolset for model checking x86 executables. The toolset builds on (i) recent advances in static analysis of program executables [1], and (ii) new techniques for software model checking and dataflow analysis [14, 10]. In our approach, CodeSurfer/x86 is used to extract a model from an x86 executable, and the reachability algorithms of the WPDS++ library [9] are used to check properties of the model. The Path Inspector is a software model checker that automates this process for safety queries involving the program's possible control configurations (but not the data state). The tools are capable of answering more queries than are currently supported by the Path Inspector (and involve data state); we illustrate this by describing two custom analyses that analyze an executable's use of the run-time stack.

Our work has three distinguishing features:

- The program model is extracted from the executable code that is run on the machine. This means that it automatically takes into account platform-specific aspects of the code, such as memory-layout details (i.e., offsets of variables in the run-time stack's activation records and padding between fields of a struct), register usage, execution order, optimizations, and artifacts of compiler bugs. Such information is hidden from tools that work on intermediate representations (IRs) that are built directly from the source code.
- The entire program is analyzed—including libraries that are linked to the program.
- The IR-construction and model-extraction processes do not assume that they have access to symbol-table or debugging information.

Because of the first two properties, our approach provides a "higher fidelity" tool than most software model checkers that analyze source code. This can be important for certain kinds of analysis; 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.

Although the present toolset is targeted to x86 executables, the techniques used [1, 14, 10] are language-independent and could be applied to other types of executables.

The remainder of the paper is organized as follows: §2 sketches the methods used in CodeSurfer/x86 for IR recovery. §3 gives an overview of the model-checking facilities that the toolset provides. §4 discusses related work.

2 Recovering Intermediate Representations from x86 Executables

To recover IRs from x86 executables, CodeSurfer/x86 makes use of both IDAPro [8], 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.

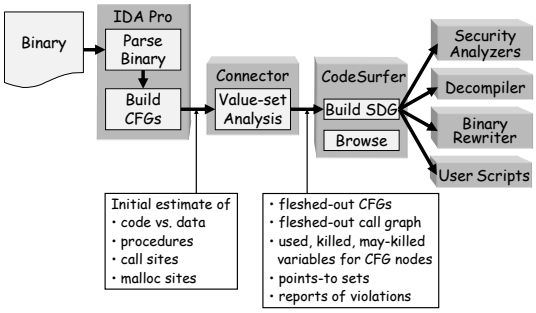

Fig. 1. Organization of CodeSurfer/x86.

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 using an algorithm called the Fast Library Identification and Recognition Technology (FLIRT) [6], 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) [1]. 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. 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.

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 (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 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, and a system dependence graph (SDG).

3 Model-Checking Facilities

For model checking, the CodeSurfer/x86 IRs are used to build a weighted pushdown system (WPDS) that models possible program behaviors. WPDS++ [9] is a library that implements the symbolic reachability algorithms from [14] on *weighted pushdown systems*. We follow the standard convention 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

Given a configuration of the PDS, the top stack symbol corresponds to the current program location, and the rest of the stack holds return-site locations—much like a standard run-time execution stack.

This encoding of the interprocedural CFG as a pushdown system is sufficient for answering queries about reachable control states (as the Path Inspector does; see §3.2): the reachability algorithms of WPDS++ can determine if an undesirable PDS configuration is reachable [2]. However, WPDS++ also supports *weighted* PDSs. These 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. (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.)

3.1 Stack-Qualified Dataflow Queries

The CodeSurfer/x86 IRs are a rich source of opportunities to check properties of interest using WPDS++. For instance, WPDS++ has been used to implement an illegal-stack-manipulation check: for each node n in procedure P, this checks whether the net change in stack height is the same along all paths from $entry_P$ to n that have perfectly matched calls and returns (i.e., along "same-level valid paths"). In this analysis, a weight

is a function that represents a stack-height change. For instance, `push ecx` and `sub esp,4` both have the weight $\lambda height.height - 4$. Extend is (the reversal of) function composition; combine performs a meet of stack-height-change functions. (The analysis is similar to linear constant propagation [15].) When a memory access performed relative to r's activation record (AR) is out-of-bounds, stack-height-change values can be used to identify which a-locs could be accessed in ARs of other procedures.

VSA is an interprocedural dataflow-analysis algorithm that uses the "call-strings" approach [16] to obtain a degree of context sensitivity. Each dataflow fact is tagged with a call-stack suffix (or *call-string*) to form (call-string, dataflow-fact) pairs; the call-string is used at the exit node of each procedure to determine to which call site a (call-string, dataflow-fact) pair should be propagated. The call-strings that arise at a given node n provide an opportunity to perform stack-qualified dataflow queries [14] using WPDS++. CodeSurfer/x86 identifies induction-variable relationships by using the affine-relation domain of Müller-Olm and Seidl [12] as a weight domain. A *post** query builds an automaton that is then used to find the affine relations that hold in a given calling context—given by call-string *cs*—by querying the *post**-automaton with respect to a regular language constructed from *cs* and the program's call graph.

3.2 The Path Inspector

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 [14]) 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 [5], and for displaying counterexample paths in the disassembly listing.[3] 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. Future versions will also eliminate (many) infeasible counterexamples by using transition weights to represent abstract data transformers (similar to those used for interprocedural dataflow analysis).

4 Related Work

Several others have proposed techniques to obtain information from executables by means of static analysis (see [1] for references). However, previous techniques deal with memory accesses very conservatively; e.g., 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

[3] 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 is similar to what C source-code tools already have to perform because of the use of the C preprocessor.

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. This is a fundamental issue; the absence of such information places severe limitations on what previously developed tools can be applied to.

Christodorescu and Jha used model-checking techniques to detect malicious code variants [3]. Given a sample of malicious code, they extract a parameterized state machine that will accept variants of the code. They use CodeSurfer/x86 to extract a model of each procedure of the program, and determine potential matches between the program's code and fragments of the malicious code. Their technique is intraprocedural, and does not analyze data state.

Other groups have used run-time program monitoring and checkpointing to perform a systematic search of a program's dynamic state space [7, 11, 13]. Like our approach, this allows for model checking properties of the low-level code that is actually run on the machine. However, because the dynamic state space can be unbounded, these approaches cannot perform an exhaustive search. In contrast, we use static analysis to perform a (conservative) exhaustive search of an abstract state space.

References

1. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Comp. Construct.*, Lec. Notes in Comp. Sci., pages 5–23. Springer-Verlag, 2004.
2. H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Symp. on Network and Distributed Systems Security*, 2004.
3. M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *USENIX Security Symposium*,, 2003.
4. CodeSurfer, GrammaTech, Inc., http://www.grammatech.com/products/codesurfer/.
5. M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Int. Conf. on Softw. Eng.*, 1999.
6. Fast library identification and recognition technology, DataRescue sa/nv, Liège, Belgium, http://www.datarescue.com/idabase/flirt.htm.
7. P. Godefroid. Model checking for programming languages using VeriSoft. In ACM, editor, *Princ. of Prog. Lang.*, pages 174–186. ACM Press, 1997.
8. IDAPro disassembler, http://www.datarescue.com/idabase/.
9. N. Kidd, T. Reps, D. Melski, and A. Lal. WPDS++: A C++ library for weighted pushdown systems. Univ. of Wisconsin, 2004.
10. A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *Computer Aided Verif.*, 2005.
11. P. Leven, T. Mehler, and S. Edelkamp. Directed error detection in C++ with the assembly-level model checker StEAM. In *Spin Workshop*, 2004.
12. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *ESOP*, 2005.
13. M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A pragmatic approach to model checking real code. In *Op. Syst. Design and Impl.*, 2002.
14. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. of Comp. Prog.*, 2005. To appear.
15. M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comp. Sci.*, 167:131–170, 1996.
16. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1981.