Practical Analysis of Stripped Binary Code

Laune C. Harris and Barton P. Miller
Computer Sciences Department
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706-1685 USA
{lharris,bart}@cs.wisc.edu

1. Abstract

The executable binary code is the authoritative source of information about program content and behavior. The compile, link, and optimize steps can cause a program's detailed execution behavior to differ substantially from its source code. Binary code analysis is used to provide information about a program's content and structure, and is therefore a foundation of many applications, including binary modification, binary translation, binary matching, performance profiling, debugging, extraction of parameters for performance modeling, and computer security and forensics. Ideally, binary analysis should produce information about the content of the program's code (instructions, basic blocks, functions, and modules), structure (control and data flow), and data structures (global and stack variables). The quality and availability of this information affects applications that rely on binary analysis.

This paper addresses the problem of using static binary analysis to extract structural information from stripped binary code. Stripped binaries are executables that lack information about the locations, sizes and layout of functions and objects. This information is typically contained in compiler generated symbol tables. Analysts, however, are often confronted with stripped binaries. Commercial software is usually stripped to deter reverse engineering and unlisenced use. Malicious code is stripped to resist analysis. System libraries and utilities are often stripped to reduce disk space requirements. Occassionally, even available symbol tables need to be ignored because they contain incorrect or misleading information.

This structural information consists of both inter- and intra-procedural control flow in addition to the start addresses of functions, function ranges, basic blocks, entry and exit points. Our approach to extracting structural information builds on control flow extraction and function identification techniques that have been employed in previous work. In particular our work extends techniques used by LEEL and RAD which use breadth first static call graph and control flow graph traversal to discover and classify code. RAD additionally uses pattern matching against standard function preambles to discover functions in sections of the code space that are not statically reachable.

Previous approaches are incapable of discovering and classifying code belonging to functions that are never statically referenced and have no recognizable preamble. In addition they treat functions as range code ranges, and due optimizations in modern compilers, and hand written assembly often found in libraries, this view does not always reflect the structure of production programs.

This paper does the following:

- Presents a framework for stripped code analysis. This framework has a modular design with replaceable and interchangeble components making it generic (independent of operating system, file format, and machine architecture)
- Describes a control flow based model for representing functions. This model is able to properly describe previously unconventional code constructs that are becoming increasingly common in modern. These constructs include non-contiguous code ranges and multiple entry point functions. Our function model can represent both modified (rewritten or instrumented) and unmodified code.
- Documents our experiences dealing with peculiarites of production code.
- Presents evaluation of our analysis on hundreds of programs. Our evaluation compares the our results to source code and symbol tables.

Our design consists of three main components: a *file format reader*, an *instruction decoder*, and a *code parser*. The file format reader extracts relevant file information including the locations of the code and data segments, any available symbol table information, and the address of the program's entry point. Typical formats include ELF9,

COFF, PE, and XCOFF. The file format reader also handles extraction of information from external debugging files. The scanner decodes byte streams (usually from the program's code segment) and produces architecture specific instruction sequences. The parser reconstructs the program's control flow and is built on top of the scanner and file format reader. The parser produces the following output: the program's call graph, control flow graph, and function information (symbols). The parser interacts with the file format reader and with the scanner through a generic assembly language interface. The assembly language interface exports a machine independent instruction representation to the parser. The scanner and file format reader are pluggable and interchangeable components enabling our techniques to function on broad range of platforms by changing the combination of scanner and file format reader. Currently, our implementation has scanners supporting the IA32, Power, and AMD64 architectures, and file format readers supporting ELF, COFF, XCOFF, and PE.

Our code analyzer uses a multiple entry control flow graph model that treats all code connected by intra-procedural control flow as part of the same function. This model was chosen for its ability to accurately represent structures in modern code (multiple entry points, non-contiguous code ranges) and for its ability to represent pre- and post modified code without alteration.