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 unlicensed use. Malicious code is stripped to resist analysis. System libraries and utilities are often stripped to reduce disk space requirements. Occasionally, 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 builds on techniques used by LEEL[8] and RAD[4] which both use breadth first static call graph and control flow graph traversal to discover and classify code. Beginning at a program's entry point both RAD and LEEL traverse the static call graph to find function entry points. Recursive traversal disassembly is used to create intra-procedural control flow graphs and discover each functions code. RAD additionally uses pattern matching against standard function preambles to discover functions in sections of the code space that are not reachable by static call instructions in previously seen code. For functions without static references or recognizable preambles, RAD uses an optimistic identification strategy: treat unanalyzed byte sequences as code unless consistency checks indicate otherwise.

This paper makes the following contributions:

- Augments the binary parsing methods used by LEEL and RAD with an extended function model that more realistically describes the structure of modern code and additional assurance checks to boost the confidence in the analysis.
- Presents tests and evaluation results on a large set of programs.
- Reports our experiences dealing with peculiarities of production code.

One of the first tasks in designing a binary parser that identifies functions is defining an appropriate function model. Previous function models were designed to represent code that complies with traditional code conventions; for example, functions must have single entry points and contiguous code ranges. 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 unconventional structures that are increasingly common in modern code. Our model can also, without alteration, represent code both before and after modification such as instrumentation.

Our experience has shown that while optimistic function identification is a valuable strategy, it needs to be supported by strong assurance checks to reduce the number of false positives (data or junk bytes interpreted as code). To improve our confidence in optimistic identification we incorporate some new assurance checks including techniques used by Orso et. al[2] to verify the disassembly correctness of obfuscated code.

Our analysis techniques were implemented in Dyninst [1], a run-time binary modification tool is used in a wide variety of research and commercial environments. Since Dyninst is multi-platform (operates on multiple operating systems and architectures) we created a generic framework for stripped code analysis. Our framework has a modular design with replaceable and interchangeable components making it generic (independent of operating systems, file format, and machine architecture). The design consists of four main components: a *file format reader*, an *instruction decoder*, an *abstract assembly language interface* 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 ELF [3], PE [4], and XCOFF [7]. The file format reader also handles extraction of information from external debugging files. The instruction decoder transforms byte streams (usually from the program's code segment) into 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 the abstract assembly language interface. The abstract assembly language interface exports a machine independent instruction representation to the code parser. The instruction decoder and file format reader are pluggable and interchangeable components enabling our techniques to function on broad range of platforms by varying 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.

To evaluate our implementation we analyzed stripped versions of a large set of test binaries. These tests, along with feedback from Dyninst users gave us insight into the issues involved in implementing robust stripped code analysis to meet the needs of a general purpose binary editor.

References

[1] B. Buck and J.K. Hollingsworth, "An API for Runtime Code Patching", International Journal of High Performance Computing Applications 14, 4, pp. 317-329, Winter 2000.

[2] C. Cruegel, W. Robertson, F. Valeur, and G. Vigna, ''Static Disassembly of Obfuscated Binaries,'' Proc. 13th. USENIX Security Symposium, pp. 255-270, pp. August 2004

[3] Executable and linking format, http://www.skyfree.org/linux/references/ELF_Format.pdf

[4] Microsoft Portable Executable and Common Object File Format Specification, http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

[5] M. Prasad and T. Chiueh, "A Binary Rewriting Defense Against Stack-based Buffer Overflow Attacks", USENIX Annual Technical Conference, June 2003, pp. 211--224

[6] B. Schwarz, S. K. Debray, and G. R. Andrews, "Disassembly of executable code revisited", Proc. IEEE Ninth Working Conference on Reverse Engineering, Richmond, October 2002.

[7] XCOFF File Format, http://www.unet.univie.ac.at/aix/files/aixfiles/XCOFF.htm

[8] L. Xun, "A linux executable editing library", Masters Dissertation, National University of Singapore, 1999. http://www.geocities.com/fasterlu/leel.htm