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. Introduction

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[3,12,22,31], binary translation[5,29], binary matching[30], performance profiling[13,16,18], debugging, extraction of parameters for performance modeling, computer security[7,8] and forensics[23,26]. 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[31] and RAD[22], which both use breadth first static call graph and control flow graph traversal to discover and classify code. Both RAD and LEEL begin at a program's entry point and traverse the static call graph to find function entry points. 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.

This paper makes the following contributions:

- Augments existing binary parsing methods with an extended function model that more realistically describes the structure of modern code and additional assurance checks to boost confidence in the analysis.
- Presents tests and evaluation results on a large set of real 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 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 intraprocedural control flow as part of the same function. This model was chosen for its ability to accurately represent unconventional structures that are becoming increasingly common in modern code. Our model can also, without alteration, represent code both before and after modification such as instrumentation.

Our analysis techniques were implemented in Dyninst [3], a run-time binary modification tool 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 independent of the operating system, file format, and machine architecture.

The evaluation of our implementation is split into three categories:

- Comparison of our parsing of stripped binaries with the information provided by the compiler's symbol table.

- Comparison of our parsing of stripped binaries with other tools.
- How well we can instrument a stripped binary based on our parse results.

Our comparisons against *source* code reveal that, in general, the functions that we recover correspond well to the source. However, in some cases compiler optimizations, such as dead code removal and inlining, can cause significant differences between the binary analysis results and the source.

Compiler generated symbol tables provide a (sometimes reasonable) approximate account of the number of functions in a binary and their locations. Initial results from automated assessments on roughly 500 test programs indicate that we recover, on average, better than 95% of the functions in stripped executables.

We compared our analysis to IDAPro [10], a popular commercial disassembler. Our parser usually recovers more function information than IDAPro, and in several cases, substantially more.

Instrumentation tests use our analysis results to insert entry and exit instrumentation into the functions in our test programs. The assessment rules were simple: abnormal termination indicates failure and normal termination indicates success. We were able to successfully instrument better than 87% of the stripped test programs and better than 99% of the unstripped ones. Note that most of the failures are due to difficulties in instrumenting the code rather than limitations of the analysis.

Our test experiences, along with feedback from Dyninst users, gave us insights into the technical issues involved in implementing robust stripped code analysis to meet the needs of a general purpose binary editor. We report some of these insights.

2. Related Work

This paper, and the related work we discuss focus on reconstruction of structural information using static binary analysis. There are several categories of tools that perform static binary analysis to obtain structural information about binary code including disassemblers, binary rewriters, decompilers and binary translators. Since the tools in each category tend to have similar requirements or employ similar techniques, we partition our discussion of related work according to application category.

While static analysis is commonly used, some tools use an alternative method called dynamic code analysis where structural information such inter-procedural control flow and call graph is determined at run time. Dynamo[1], DynamoRio[2], DIOTA[15], Self-Propelled Instrumentation [18], and PIN[24] are program optimization tools that use dynamic code analysis. Other tools, IDTrace [21], for example, use both static and dynamic analysis. The dynamic phase is used to correct errors resulting from static analysis and to analyze code not detectable statically.

Disassemblers

Disassemblers convert binary code to symbolic assembly language representations. A primary function of disassemblers is to visually present these assembly language representations of binary code to human analysts. To aid the analyst by organizing information, some disassemblers present control flow information or attempt to partition programs into functions. IDA-Pro [10], arguably the best commercial disassembly tool, does both. IDA-Pro uses a depth first call graph traversal to determine function start addresses and intra-procedural control flow analysis to determine function ranges. IDA-Pro, however, does not provide support for non contiguous functions and does not identify functions that are only reached via indirect calls (both constructs are common in production code). An another approach used by some tools is to partition a binary into functions by recognizing function prologues [20]. This method has the advantage of being simple and fast. However, it often creates a coarse grained or inaccurate function structure if functions have non-standard prologues or if there is data interspersed in the code.

Producing correct disassembly is often challenging due variable length instruction sets, data mixed with code, indirect control flow, and deliberate code obfuscation. Scharwz et al combined linear sweep disassembly and recursive traversal to create a hybrid disassembly algorithm[27]. Their work is based on the idea that if both linear sweep and recursive traversal disassembly agree then a disassembly is likely to be correct. Orso, et. al developed a disassembly method for use on obfuscated code [20] in response to a static code obfuscator developed by Linn and Debray [14]. Their method uses CFG checks to verify disassembly correctness; e.g., branches to the middle of an instruction indicate an error. These disassembly checks used by both methods form the core of a set of safety checks we use for our speculative function discovery.

Binary Modification

Binary modifiers are tools that rewrite code either statically or at run time. Major uses of binary modification include post compilation optimization, instrumentation, and profiling. Most binary modification tools rely on the presence of symbol table and debugging information. Some tools, however, are able to reconstruct some structural information and therefore operate on stripped binary code. EEL operates on SPARC binaries[12]. The two main disadvantages of the EEL approach are that it is not well suited to variable length instruction set architectures and that it does not identify functions that are reachable only through an indirect control transfer. The first of these disadvantages is addressed by LEEL [31], an EEL-based binary editing tool designed for Linux on Intel's IA32 architecture. Beginning at the program's entry point, LEEL walks the static call flow graph to build the set of call targets that correspond to function starting points. Recursive disassembly at these points finds the code blocks in each function and establishes the function's size. For gaps in the code space that are caused by the presence of functions only reachable through indirect control flow or by data bytes, LEEL provides two options. The first option is conservative: do not analyze. The second option is more aggressive: assume that the first byte of a gap is the starting address of a function. These two options present undesirable extremes. The first, while safer for binary modification, provides low code coverage in binaries where a high proportion of functions are reached only though indirect calls and in binaries where indirect calls form bridges to large sections of the call graph. The approach taken by RAD [22] is similar to LEEL's. RAD is a binary editor that defends against stack based buffer overflow attacks by instrumenting function entry and exit points. RAD achieves better code coverage than LEEL's conservative option by matching byte sequences in gaps to known function prologues. RAD's approach will not locate functions that have no static references and have no known prologues. Both RAD and LEEL use a conventional view of a function: i.e., a self-contained sequence of code with a single entry point, at least one exit point, and laid out in a contiguous area of memory. This view, while generally applicable, is not adequate for all applications. In binary code, functions sometimes have multiple entry points and are spread across non-contiguous areas of the executable. Our work is designed to extend the approaches used by LEEL and RAD to improve code coverage and to properly handle unconventional function structures.

LEEL and RAD are able to operate on stand-alone executable code: i.e. no debugging information is needed. Some other tools rely on the presence of symbol tables, relocation tables, and other debugging information to perform similar analyses (code discovery, CFG creation). Examples include Etch [25] and PLTO[28], which all require relocation information.

3 Design

The compilation, optimization and linking steps often create binaries that are structurally different from the corresponding source code. Therefore one of the first tasks in designing a binary parser that identifies functions is defining an appropriate function model. Section 3.1 presents the model that we use. The remaining parts of this section describe our implementation.

3.1. Function Models

The function model determines the degree to which the analysis output reflects the structure of the binary. For our implementation, we chose a multiple-entry control flow graph model. Control flow graph models treat functions as sets of basic blocks that form a control flow graphs. CFG models are able to represent functions with non contiguous code; basic blocks can be located anywhere in a program's address space. Figure 1 shows three control flow models: single entry, shared code, and multiple entry. The single entry model is the simplest of the three CFG models presented above. Each function in this model has a has one entry block which dominates all other blocks. The shared code model is a variant of the single entry model that allows basic blocks to belong to multiple functions. The multiple entry model treats all code connected by intra-procedural control flow as part of the same function.

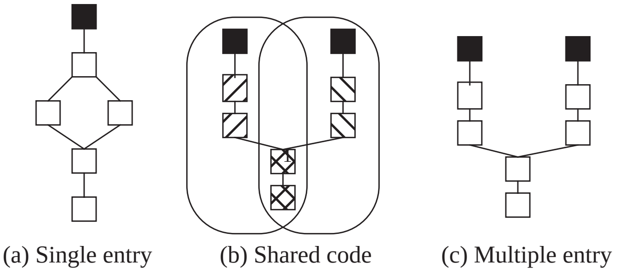
(a) Single entry (b) Shared code (c) Multiple entry

Figure 1: Control Flow Models

Alternative models include the Prologue/Prologue and Prologue/Exit models where functions are the regions between known prologues or exits, and the Symbol Table model which defines a function as a named start address and size pair. These models were rejected since they do not naturally describe binary level functions that have non contiguous code ranges, multiple entry points, or shared code blocks. In addition, the prologue models are vulnerable to the presence of data in the code space and to unrecognizable or missing prologues.

3.2. Implementation

Our parser was designed to be modular, with replaceable and interchangeable components. These components fall into three categories: file format reader, instruction decoder, and code parser. In this section, we describe the components along with our generic assembly language interface that ties them together.

The Dyninst system also includes components for dynamic code generator and process control, but these are outside the scope of this paper.

File format reader

Our file format readers extract information from file headers that describe the layout of the executable and contain the addresses and sizes of interesting points and sections including the locations of text, data, symbol tables and the program's entry point. In addition, the file format readers use heuristics determine the address of the function main. Most programs, after executing initial start-up code, transfer control to a function commonly called main. It is often difficult, however, to use static control flow analysis to locate main. Locating main using heuristics, while not strictly necessary, gives two benefits. First, labeling the function main is useful for tools that expect to find it. Secondly, the program's call graph is rooted at main. Finding this root early tends to reduce the parser's reliance on speculative function discovery techniques.

We provide format readers for ELF [6], COFF [19], XCOFF [32], and PE [19].

Instruction decoder

The instruction decoder plays two roles. One is a disassemblers; it interprets byte streams as machine instructions. The other role is to extract semantic information from the instructions and implement the architecture dependent routines that are accessed through the generic assembly language interface.

We currently provide stripped code parsers for the Intel IA32, AMD 64, and IBM Power architectures.

Generic Assembly Language interface

The generic assembly language interface exports a uniform assembly language to the parser independent of the underlying architecture. The AAL's object model allows description of generic operations. Each instruction is treated as an object that exports the following methods and operators: isCondBranch, isUncondBranch, isIndirBranch, isCall, isReturn, getBranchTarget, getMultiBranchTargets, isPrologue, isIllegal, --, ++, and size. All parsing operations that reference machine instructions use this interface.

Parser

Control flow discovery and function identification is done by the parser which accesses the program's instruction stream via the generic assembly language interface. The generic assembly language interface enables us to use a single parsing algorithm for multiple platforms. In the best case, correct symbol tables and debugging information will be available and we use symbol start addresses as analysis points. In the worst case, however, the program's entry point (obtained by the file format reader) guarantees a reliable starting point for analysis.

This initial set of starting addresses is used as input for our parsing algorithm. Breadth-first call graph traversal and recursive disassembly finds all code statically reachable from the reachable from the initial set of addresses. Valid call targets addresses are named as functions then disassembled to build control flow graphs. Additional call targets discovered during CFG creation are added to the list of call targets for analysis. CFG conflicts and illegal instructions cause rejection of a function. When the algorithm terminates, there may be gaps that are not analyzed remaining in the text space due to the presence of data, alignment padding, or functions that are never statically referenced in previously visited code. We use two speculative gap completion phases to analyze these gaps.

The first phase searches each gap for known function prologues. At each potential code address, the parser creates an instruction and calls isPrologue. An architecture-specific method in the decoder checks for instruction sequences commonly used to implement prologues on each platform. For example, a common prologue sequence on IA32 is:

```
push ebp
mov esp, ebp
```

If isPrologue returns true, the corresponding address is assumed to be the start of a function, and we invoke the core algorithm to build the sub section of the program's call graph rooted at this function.

On completion of the first speculative discovery phase completion phase, gaps may still be present in the text space. These gaps may be due to functions that are called indirectly and have no known prologue. We disassemble these gaps using Orso et al's speculative completion method. Their method is to build control flow graphs starting at each potential code address in a gap. CFG conflicts prune the set of control flow graphs. For example, if branches in a CFG target the apparent middle of another instruction then that CFG is rejected. In addition, we add the requirement that during this phase CFGs must contain at least one non-terminal control transfer instruction. This requirement reduces the number of false positives and boosts confidence in the functions discovered.

Two issues need to be addressed to effectively follow function local control transfers on production code. The first is the analysis of indirect branches; i.e branches that have targets determined at run time. Indirect branches are commonly used to implement switch statements. Cifuentes and Van Emmeriks describe a machine and compiler independent method of using program slicing and expression substitution to statically recover the targets of indirect branches that use jump tables[4]. In practice, however, simple machine and compiler dependent methods have proved effective at recovering jump table values. Our approach is to backtrack along the control paths leading to the indirect branch to discover the instructions that set up the jump table access. These instructions give the base location and number of entries in a jump table.

The second issue is the identification of exit points. An exit point that is identified as an intraprocedural transfer will cause overestimation of function sizes. In addition to return instruction detection, the instruction decoder implements platform specific tail-call detection. Also known non-returning call instructions (for example calls to exit or abort) are considered to be exit points. If a transfer is not identified as an exit, it is assumed to be intra-procedural.

Our implementation was tested with the following compilers: Microsoft's Visual C++, GNU's gcc, IBM's xlc, and Intel's icc.

4. Evaluation

Proper evaluation of a stripped binary analysis tool is a challenging process. Human inspection and verification is useful for small sets of trivial test programs. For larger sets of programs or larger, more sophisticated programs, however, human inspection is inappropriate. For our evaluation we used automated comparisons against compiler generated symbols tables and automated instrumentation tests. We also compared our function recovery rate to that of IDAPro, a popular commercial disassembly tool.

Our evaluations were done primarily on Linux and Microsoft Windows. For evaluations requiring large numbers of binaries, we used programs obtained from our department's standard `bin` directory (a very large collection of programs).

Comparison vs. symbol tables

We filtered the output from objdump to display unique function addresses for each of 519 programs. We then compared the number of functions we recovered to that reported by objdump. Our results on the test binaries show a 95.6% average recovery rate.

Comparison vs. IDAPro

Table 1 shows a comparison of our function recover rate against IDAPro. The results show that in general our stripped code analysis identifies more functions than IDAPro. For the firefox binary, inspection suggests that IDAPro is incorrectly identifying thousands of two instruction sequences as functions. (they appear to be part of dispatch tables).

	Stripped		Unstripped	
Passed	454	87.5%	517	99.6%
Failed	65	12.5%	2	0.4%

Table 2: Success Rate of Instrumentation Tests

Instrumentation tests

Using Dyninst we inserted function entry and exit point instrumentation into 519 test binaries. Success means the instrumented binary runs to completion without failure or hanging. Results from these tests are report in Table 2.

5. Experiences

In this section, we describe our experiences with real but unusual code patterns.

Binary	Platform	IDA		Dyninst
		Number	Percent	Number
aim	window	75	100.0%	75
alara	linux	431	65.9%	654
bash	linux	1,539	92.0%	1,655
bubba	linux	53	96.4%	55
calc	windows	168	99.4%	169
eon	linux	616	53.2%	1,157
firefox*	windows	34,064	116.0%	29,372
gimp	linux	3,889	91.8%	4,237
kwrite	linux	7	77.8%	9
notepad	windows	84	98.8%	85
paradyn	linux	4,506	35.7%	12,617
SecureCRT	windows	4,139	97.8%	4,233
vcross	linux	52	62.7%	83
X	linux	3,991	92.7%	4,307

Table 1: Functions found using Dyninst and IDAPro

"Percent" measures the percentage of functions found by IDAPro compared to the total found by Dyninst.

Pseudo call instructions

Not all call instructions are valid interprocedural transfers. Calls that target the instruction immediately following the call are often used by position independent code to obtain the program counter. The targets of these instructions must not be used as function entries.

The targets of some call instructions are filled in a load time. This means that the target value obtained by static analysis of the binary is incorrect. In some cases this value is obvious (target address 0, for example). In others the target address is a junk value. We detect the first case when we encounter calls, and rely on CFG and disassembly checks to guard against the second case.

Exception handling code: unreachable blocks

C++ exceptions creates code blocks that appear to be unreachable. Currently, our most reliable solution to this problem is to extract exception information from compiler generated exception tables.

Missing "main"

Binaries without an explicit main function illustrate the need for speculative function discovery. Analysis of the kwrite binary (part of the KDE tools) revealed that the executable contained no main function. main was located in a dynamically loaded library and accessed through the Procedure Linkage Table. None of the functions in the stripped kwrite binary were statically reachable from the program's entry point.

False positives

Speculative code discovery analyses data bytes. Data bytes that are interpreted as terminal control flow instructions often give the appearance of single-basic-block functions. For example, data bytes might be interpreted as the following instruction sequence:

```
mov reg, mem
mov reg, mem
ret
```

To eliminate the occurrence of these false positives we require that functions discovered during the speculative discovery phase have two or more control flow instructions. This restriction, while improving reliability, reduces coverage by excluding legitimate functions that fit this pattern. We are currently evaluating alternatives to this strategy. One potential direction is to use simple semantic analysis to determine whether a single block function discovered during speculative discovery should be accepted or rejected.

References

[1] V. Bala, E. Duesterwald and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System", *ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, June 2000.

[2] Bruening, D., Garnett, T., Amarasinghe, S. "An Infrastructure for Adaptive Dynamic Optimization". *First Annual International Symposium on Code Generation and Optimization*, March 2003.

[3] 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.

[4] C. Cifuentes and M. Van Emmerik, "Recovery of Jump Case Statements from Binary Code", *7th International Workshop on Program Comprehension*, Washington, DC, May 1999.

[5] C. Cifuentes, M. Van Emmerik, and N. Ramsey, "The Design of a Resourceable and Retargetable Binary Translator", *Sixth Working Conference on Reverse Engineering*, Atlanta, October 1999.

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

[7] J.T. Giffin, S. Jha, and B.P. Miller, "Detecting Manipulated Remote Call Streams", *11th USENIX Security Symposium*, San Francisco, California, August 2002

[8] J.T. Giffin, S. Jha, and B.P. Miller, "Efficient Context-Sensitive Intrusion Detection", *11th Network and Distributed System Security Symposium*, San Diego, California, February 2004

[9] HI-PVM, http://www.parasys.co.uk/

[10] IDAPro, http://www.datarescue.com/idabase/overview.htm.

[11] C. Cruegel, W. Robertson, F. Valeur, and G. Vigna, "Static Disassembly of Obfuscated Binaries", *13th USENIX Security Symposium*, August 2004

[12] J.R. Larus and E. Schnarr, "Eel: Machine-independent executable editing", *SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, June 1995.

[13] J.R. Larus and T. Bal, "Rewriting Executable Files to Measure Program Behavior", *Software-Practice and Experience* **4**, 2, February 1994.

[14] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly", *10th ACM Conference on Computer Communications and Security (CCS),* October 2003.

[15] J. Maebe, M. Ronsse, K. De Bosschere, "DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications", *WBT-2002: Workshop on Binary Translation*, Charlottesville, Virginia, September 2002.

[16] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam and T. Newhall, "The Paradyn Parallel Performance Measurement Tool", *IEEE Computer* **28**, 11, November 1995, pp. 37-46.

[17] A.V. Mirgorodskiy and B.P. Miller, "CrossWalk: A Tool for Performance Profiling Across the User-Kernel Boundary", *International Conference on Parallel Computing (ParCo)*, Dresden, Germany, September 2003.

[18] A.V. Mirgorodskiy and B.P. Miller, "Autonomous Analysis of Interactive Systems with Self-Propelled Instrumentation", *Multimedia Computing and Networking Conference*, San Jose, California, January 2005.

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

[20] A. Orso, M.J. Harrold, and G. Vigna, "MASSA: Mobile Agents Security through Static/Dynamic Analysis", *First ICSE Workshop on Software Engineering and Mobility (WSEM 2001)*, Toronto, Canada, April 2001.

[21] J. Pierce, J. and T. Mudge, "IDtrace - A Tracing Tool for i486 Simulation", University of Michigan Tech. Report CSE-TR-203-94. 1994.

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

[23] Project Fenris: http://lcamtuf.coredump.cx/fenris/whatis.shtml

[24] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation", *Programming Language Design and Implementation (PLDI),* Chicago, Illinois, June 2005.

[25] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Chen, and B. Bershad. Instrumentation and Optimization of Win32/Intel Executables Using Etch. *USENIX Windows NT Workshop*, August 1997.

[26] K. Rozinov, "Reverse Code Engineering: An In-Depth Analysis of the Bagle Virus", Bell Labs - Government Communication Laboratory - Internet Research, August 2004.

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

[28] B. Schwarz, S. Debray, and G. Andrews. "PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture". *2001 Workshop on Binary Translation (WBT-2001)"*, Barcelona, Spain, Sept. 2001.

[29] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S.G. Robinson, "Binary Translation", *Digital Tech. Journal* **4**, 4, 1992

[30] Z. Wang, K. Pierce, S. McFarling, "BMAT- A Binary Matching Tool", *Feedback-Directed Optimization (FDO2),* Haifa, Israel, November 1999.

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

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