Disassembly of Executable Code Revisited[*]

Benjamin Schwarz Saumya Debray Gregory Andrews
Department of Computer Science
University of Arizona
Tucson, AZ 85721
{bschwarz, debray, greg}@cs.arizona.edu

Abstract

Machine code disassembly routines form a fundamental component of software systems that statically analyze or modify executable programs. The task of disassembly is complicated by indirect jumps and the presence of non-executable data—jump tables, alignment bytes, etc.—in the instruction stream. Existing disassembly algorithms are not always able to cope successfully with executable files containing such features and fail silently—i.e., produce incorrect disassemblies without any indication that the results they are producing are incorrect. This can be a serious problem, since it can compromise the correctness of a binary rewriting tool. In this paper we examine two commonly-used disassembly algorithms and illustrate their shortcomings. We propose a hybrid approach that performs better than these algorithms in the sense that it is able to detect situations where the disassembly may be incorrect and limit the extent of such disassembly errors. Experimental results indicate that the algorithm is quite effective: the amount of code flagged as incurring disassembly errors is usually quite small.

1 Introduction

There has been a significant amount of attention focused on binary rewriting and link-time code optimization in recent years [5, 6, 15, 17, 19]. A fundamental requirement of any software system that aims to statically analyze or modify an executable program is accurate disassembly of its machine code instructions. The task of recovering these instructions is often complicated by the presence of non-executable data—jump tables, alignment bytes, etc.—in the instruction stream. This poses a chicken-and-egg problem: we cannot identify the instructions without knowing what is data, and vice versa. The fact that link-time binary modification tools have to be prepared to deal with hand-coded assembly routines, e.g., due to statically linked libraries, complicates the problem further because it means that we cannot always assume that the code follows familiar source-level conventions (e.g., that a function has a single entry point) or uses recognizable compiler idioms.

The presence of variable-length instructions—commonly found in CISC architectures such as the widely used Intel x86—results in an additional degree of complexity, and renders simple heuristics for extracting instruction sequences ineffective. In this paper we examine techniques currently used for disassembly, discuss their drawbacks, and introduce an improved method for the extraction of instructions from a statically-linked binary that contains relocation information. Our algorithm is capable of identifying jump tables embedded within the text segment, offset tables for position independent code (PIC) sequences, and data inserted for alignment purposes, e.g., to align loop headers. Most importantly, it is able to avoid some disassembly errors that can occur when using existing disassembly techniques.

We have implemented our approach in PLTO, a post-link-time optimizer for the Intel x86 architecture. Experimental results indicate that our algorithm is able to cope with statically linked executables containing highly optimized hand-coded assembly code with a high degree of precision, identifying potential disassembly problems rather than failing silently and limiting the extent of such problems to a small portion of the input executables.

2 Preliminaries

2.1 Relocation Information

Linkers are capable of producing relocation tables at each stage during the linking process. By default, the final executables do not contain relocation information because it is not needed by the loader to re-map the program. However, many binary rewriting frameworks that carry out translation or optimization utilize such information. The tables are used to identify the bit-sequences in the executable that correspond to addresses of the program. A single

[*]This work was supported in part by the National Science Foundation under grants CCR-0073394, EIA-0080123, and CCR-0113633.

entry in the table usually contains: (*i*) a section offset, (*ii*) a bit that specifies whether the relocation is PC-relative or absolute, and (*iii*) the width (typically the size of an address on the architecture) of the relocation.

Systems that analyze and transform machine code programs use this information in much the same way that linkers do. After the code has been moved around, references to addresses have changed, and they need to be updated to reflect their new position in the executable. Without knowledge about the locations of address, a binary modification system has to be fairly conservative in the kinds of code transformations it is able to effect. The remainder of this paper assumes that relocation tables are available in the executable. We do not feel this is unnecessarily onerous: a user who is sufficiently concerned about performance to use a link-time optimizer seems likely to be willing to invoke the compiler with the additional flags needed to retain relocation information. Other binary rewriting systems, notably OM [19] and Atom [18], have the same requirement, and most linkers are capable of producing these tables.

2.2 Position-Independent Code

Many compilers can be instructed to emit code that does not rely on being bound to any particular position in the program's address space. These code sequences are often referred to as *position-independent code* (PIC). In particular, PIC sequences do not contain any relocatable addresses embedded in the instructions. This property enables the code to work regardless of its memory location at runtime. Furthermore, PIC does not need to be patched by the loader, enabling it to be mapped as read-only data—which is useful for shared code such as dynamically linked libraries [14].

When a compiler is emitting position-independent code it typically creates jump tables that are also position-independent. These tables are usually embedded in the text segment of the executable and consist of a sequence of offsets rather than virtual addresses. A jump that uses the offset table first loads a nearby address,[1] then uses this to index into the table and retrieve an offset. The offset is added to the address that was previously loaded and then used in an indirect jump to reach the desired destination. The problems posed by position-independent jump tables are three-fold: (*i*) the offset tables, which are really no different than data, appear in the instruction stream; (*ii*) the code sequences that perform the indirect jumps are often complicated and may not adhere to a single pattern that is easily recognizable; and (*iii*) it is entirely possible that an offset table does not contain relocation entries. Taken together, these properties make the task of disassembling PIC sequences involving jump tables more difficult than standard code.

3 Two Methods for Instruction Disassembly

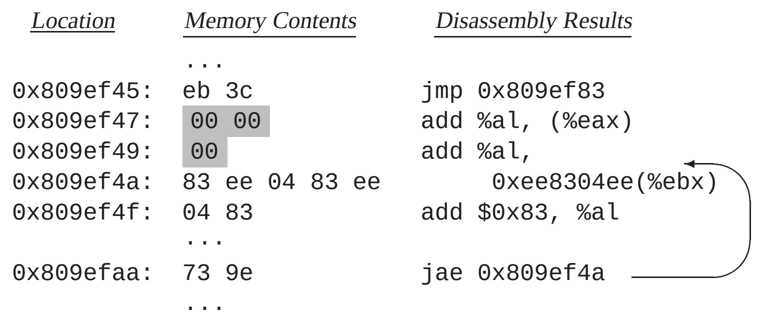
3.1 Linear Sweep

A straightforward approach to disassembly is to decode everything appearing in sections of the executable that are typically reserved for machine code. This method is used by programs such as the GNU utility *objdump* [9] as well as by link-time optimizers such as *alto* [15], OM [19], and Spike [6]. Its main advantage is simplicity. However, it has the disadvantage that any data that is embedded in the instruction stream is misinterpreted as code and disassembled. Only under special circumstances (such as when an invalid opcode is decoded) can these situations be discovered.

The problem is illustrated by the code fragment shown in Figure 1, taken from the machine code for the function `strrchr` in the standard C library (libc) under RedHat Linux on a Pentium III processor. Starting at address `0x809ef47`, three `NULL` bytes of data (`0x00`, shown highlighted) were inserted to push the loop header at address `0x809ef4a` forward, presumably for alignment purposes. The NULL bytes and subsequent instructions are misinterpreted by the utility *objdump*, as it uses the scheme described above to decode instructions. By inspection, we can figure out that the jump at address `0x809efaa` targets the middle of what *objdump* belives to be an instruction. In addition, the instructions it decoded are rather suspicious in their current context (the `add` at address `0x809ef49` references an absolute memory location that does not even appear in the scope of executable!). The instruction sequence is clearly invalid, but the linear sweep algorithm is unable to discern data from code.

The problem in this case arises because on the Intel x86 architecture, a NULL byte can be a valid opcode; it would not have arisen if the programmer had used `nop` instructions to force alignment. However, the larger point illustrated by this example remains valid: Data embedded in the text segment can be misidentified as code by the linear sweep algorithm, and this can cause disassembly errors in some or all of the remainder of the instruction stream.

[1]On the Intel x86 this is done using a '`call 0`' instruction followed by a '`pop %eax`' instruction, which has the effect of storing the latter instruction's address into register `%eax`.



```
    Location        Memory Contents        Disassembly Results
                    ...
0x809ef45:     eb 3c                  jmp 0x809ef83
0x809ef47:     00 00                  add %al, (%eax)
0x809ef49:     00                     add %al,
0x809ef4a:     83 ee 04 83 ee              0xee8304ee(%ebx)
0x809ef4f:     04 83                  add $0x83, %al
               ...
0x809efaa:     73 9e                  jae 0x809ef4a
               ...
```

Figure 1: An Example of Disassembly Problems using Linear Sweep

3.2 Recursive Traversal

The problem with the linear sweep algorithm, illustrated by the example in Figure 1, is that it does not take into account the control flow behavior of the program: in particular, the `jmp` instruction immediately before the three NULL bytes inserted for alignment. As a result, it is unable to discern that these alignment bytes are not reachable during execution, and mistakenly interprets them as executable code. An obvious fix would be to take into account the control flow behavior of the program being disassembled in order to determine what to disassemble. Intuitively, whenever we encounter a branch instruction during disassembly, we determine the possible control flow successors of that instruction, i.e., addresses where execution could continue, and proceed with disassembly at those addresses (e.g., for a conditional branch instruction we would consider the branch target and the fall-through address).

Variations on this basic approach to disassembly, which we term *recursive traversal*, are used by a number of binary translation and optimization systems [3, 20]. A virtue of the algorithm is its simplicity and effectiveness in avoiding disassembly of data. The basic algorithm for recursive traversal is:

```
proc Disassemble(Addr, instrList)
{
  if (Addr has already been visited)
    return;
  do {
    instr = DecodeInstr(Addr);
    Addr.visited = true;
    add instr to instrList;
    if (instr is a branch or function call) {
      T = set of possible control flow successors of instr;
      for each (target ∈ T) {
        Disassemble(target, instrList);
      }
    }
    else Addr += instr.length;   /* addr of next instruction */
  } while Addr is a valid instruction address;
}
```

Each executable contains an entry point, which is usually specified in the program header. The routine `Disassemble()` is initially invoked with this entry point. Under the assumption that we are able to identify all possible control flow successors of each branch and function call operation in the program, this ensures that any instruction that is reachable from the program entry is correctly disassembled.

This method is able to handle the code fragment shown in Figure 1. Upon decoding the jump instruction at address `0x809ef45`, disassembly continues at address `0x809ef83`, the (only) control flow successor for this instruction. Eventually the instruction at address `0x809efaa` is reached by a path from this point, and this in turn causes disassembly to proceed from the instruction at `0x809ef4a`. The three `NULL` bytes are never disassembled, since they are not reachable by any execution path through the program.

```
  Location      Memory Contents            Disassembly Results
                  ...
0x80b1d8b:  8d 84 c0 95 1d 0b 08    lea  0x80b1d95(%eax,%eax,8),%eax
0x80b1d92:  ff e0                   jmp  *%eax
0x80b1d94:  8d                      lea
0x80b1d95:  74 26 00                     0x0(%esi,1),%esi
0x80b1d98:  8b 06                   mov  (%esi),%eax
0x80b1d9a:  13 02                   adc  (%edx),%eax
0x80b1d9c:  89 07                   mov  %eax,(%edi)
                  ...
```

Figure 2: An Example of Disassembly Problems using Recursive Traversal

The key assumption in this algorithm is that we can identify all possible control flow successors of each control transfer operation in the program. This may not always be straightforward in the case of indirect jumps. For jump tables appearing in the text segment, this poses a correctness issue: any imprecision in determining the size of such a jump table will result either in a failure to disassemble some reachable code (if the table size is overestimated) or erroneous disassembly of data (if its size is underestimated). The problem is complicated by the fact that the structure of the code generated for *switch* statements can differ widely from one instance of a *switch* to another, even for a specific compiler and target architecture.

Existing proposals for identifying the targets of indirect jumps usually resort to nontrivial program analyses such as program slicing [4] or constant propagation [8]. We need a control flow graph for the function in order to carry out such analyses. Unfortunately, the construction of a control flow graph for a function before all of its instructions have been disassembled does not seem straightforward.[2] Instead, we resort to a simpler technique based on relocation information. When disassembling the code for a function f, let R_f be the set of relocatable text segment addresses a such that a lies between the start address for f and the start address of the function following f, and let J_f be the set of addresses a such that $a \in R_f$ and location a itself contains a relocatable text segment address. Intuitively, we expect an indirect jump to an address a be implemented by loading a (which must be a text segment address, under the assumption that all code is in the text segment) into a register r and then jumping indirectly through r, and in this case the address a has to be relocatable; the set R_f consists of all such addresses that lie within the function f, and hence might be possible targets for an indirect jump in f. The set J_f specifies those elements of R_f that are jump table entries, i.e., which do not contain code and hence cannot be the target of a jump. The set of possible targets of an indirect jump within f is then taken to be the set of addresses $R_f - J_f$.

This approach seems plausible, in that it uses a conservative over-estimate of the set of possible targets of each indirect jump, which means that every address that could in fact be a target of the jump is considered and all reachable code is disassembled. The problem is that we may also consider addresses that are not in fact targets. This can produce incorrect disassembly results, as illustrated by an example from a C library routine under RedHat Linux called `__mpn_add_n`, shown in Figure 2.

In the Intel x86 instruction set, an `lea` ("load effective address") instruction of the form '`lea` $baseAddr(r_0, r_1, m), r_2$' has the effect

$$r_2 \leftarrow baseAddr + \mathsf{contentsOf}(r_0) + m \times \mathsf{contentsOf}(r_1).$$

The `lea` instruction at address `0x80b1d8b` in Figure 2 therefore computes an address into register `%eax` whose value depends on the contents of `%eax` before this instruction. An inspection of the hand-coded assembly routine for this function reveals that a loop begins at address at `0x80b1d98`, and the address computed by this `lea` instruction

[2] Accurate identification of the possible targets of an indirect jump through a jump table can be difficult even if we assume that a control flow graph is available, since we cannot in general count on the jump in a program being accompanied by a bounds check that would enable us to identify the extent of the jump table. Such checks may be excised from hand-crafted assembly code by a careful programmer who is aware of specific invariants that hold in the program; an aggressive optimizing compiler may be able to elide the check based on program analyses to identify the range of values for a variable [10] or using optimizations analogous to the elimination of array bounds checks [11, 16]. We may also encounter indirect jumps that don't involve a jump table and hence don't have a bounds check.

is somewhere in the middle of this loop; exactly where is determined by the contents of `%eax`.[3] It turns out that this register always takes on a value that results in a valid instruction address being computed. However, during a static examination of the instruction stream during disasembly, we cannot guarantee that $\mathtt{contentsOf}(\mathtt{\%eax}) \neq 0$, since such guarantees in general require nontrivial analyses such as constant propagation or program slicing, which in turn require the control flow graph for the function, which is not available during disassembly. Since the address `0x80b1d95` appears as a relocatable text segment address within the function, and this location does not itself contain a relocatable text segment address, it is considered as a possible target of the indirect jump at location `0x80b1d92` during recursive traversal disassembly (this corresponds to the possibility that register `%eax` could have the value 0 when this instruction is executed). As a result, we continue disassembling the input starting at location `0x80b1d95`. The problem is that this address is in the middle of an instruction, i.e., recursive traversal produces an incorrect disassembly in this case.

4 An Improved Algorithm

The linear sweep and recursive traversal disassembly algorithms discussed in the previous section have complementary strengths and weaknesses. The former does not rely on the precise identification of targets of indirect jumps for correct disassembly, but it has trouble coping with data embedded in the instruction stream; the latter is able to decode around data embedded in the text segment, but it may have problems with indirect jumps if their targets cannot be precisely identified. This section discusses how these two algorithms can be combined to exploit the strengths of each.

4.1 Extending the Linear Sweep Algorithm

The simple linear sweep algorithm discussed in Section 3.1 has the disadvantage that any data appearing in the text segment causes disassembly errors. In particular, this means that this algorithm cannot deal with jump tables embedded in the text segment. In this section we discuss how the linear sweep algorithm can be extended to handle jump tables embedded in the instruction stream.

As mentioned in Section 2.1, we assume that relocation information is available in the file being disassembled. We can take advantage of such information to identify jump tables embedded in the text segment (note that jump tables in the data segment do not pose a problem: our primary goal here is to identify the extent of jump tables in the text segment so that we can avoid misinterpreting them as code). Each address a_i appearing in a jump table embedded in the text segment has the following properties:

(i) the memory locations containing a_i are marked relocatable; and

(ii) the address a_i itself points into the text segment.

These properties, while necessary for jump table entries, may not be sufficient: depending on the architecture, relocatable addresses, possibly pointing into the text segment, may also appear as immediate operands in an instruction. However, the instruction sets of typical modern architectures impose an (architecture-specific) upper bound K_{max} on the number of such immediate operands that can appear adjacent to each other in an instruction (e.g., for the Intel x86 architecture, $K_{max} = 2$). Thus, if the text segment contains n adjacent relocatable addresses each of which point into the text segment ($n > K_{max}$), at most the first K_{max} of these may be part of an instruction; the remaining $n - K_{max}$ addresses must be data. We can use this information to modify the linear sweep algorithm so that, during disassembly, it goes around any such data blocks identified in the text segment. Of course, this does not resolve the status of the first K_{max} entries in the sequence, i.e., determine whether they are part of the jump table or immediate operands of an instruction. We will return to this point shortly.

A crucial property of this approach is that it allows us to identify the end of a jump table that appears in the text segment. The text segment therefore becomes divided into "chunks" of code separated by jump tables. Each chunk starts either at the entry point of a function or at the end of the previous jump table. We use the simple linear sweep algorithm of Section 3.1 to disassemble each such chunk, then examine the last instruction in the disassembled chunk. Suppose that the last instruction contains m addresses ($0 \leq m \leq K_{max}$) as immediate operands appearing at the end of the instruction. Then we know that of the n contiguous relocatable addresses appearing at the end of that chunk, m addresses are part of instructions and the remaining $n - m$ addresses constitute jump table entries. The resulting algorithm is as follows:

[3]The instruction '`lea 0x0(%esi,1),%esi`' at address `0x80b1d94` serves as a 4-byte no-op whose purpose is to align the first instruction in the loop on an 8-byte boundary.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase®
Smarter legal research.