Stony Brook University

OFFICIAL COPY

The official electronic file of this thesis or dissertation is maintained by the University Libraries on behalf of The Graduate School at Stony Brook University.

© All Rights Reserved by Author.

Static Disassembly of Stripped Binaries

A Thesis Presented
by
Arvind Ayyangar

to
The Graduate School
in Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in
Computer Science
Stony Brook University

August 2010

Stony Brook University

The Graduate School

Arvind Ayyangar

We, the thesis committee for the above candidate for
the degree of Master of Science,
hereby recommend acceptance of this thesis.

Dr. R. C. Sekar, (Advisor)
Professor, Computer Science Department

Dr. Robert Johnson, (Chairman)
Assistant Professor, Computer Science Department

Dr. C. R. Ramakrishnan, (Committee Member)
Associate Professor, Computer Science Department

This thesis is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the thesis

Static Disassembly of Stripped Binaries

by

Arvind Ayyangar

Master of Science

in

Computer Science

Stony Brook University

2010

Disassembly of binaries plays an important role in computer security. Tools for binary analysis and reverse engineering rely heavily on static disassembly. Current disassemblers are not able to reliably disassemble executables or libraries that contain data (or junk bytes) in the midst of code, or make extensive use of indirect jumps or calls. These features can cause these tools to fail silently, thus making them inappropriate for applications that critically depend on correct disassembly, e.g., binary instrumentation. An incorrectly disassembled binary can lead to incorrect instrumentation, which can in turn cause the instrumented program to fail, or more generally, exhibit differences in behavior from the original binary. In this thesis, we analyze existing disassembly approaches, their shortcomings, and propose a technique to overcome these shortcomings. We investigate the use of static data flow analysis and

type analysis to overcome the many challenges posed by disassembly of commercial off-the-shelf software binaries.

To my parents,

my sister Hema,

and my wonderful neice Tarika.

Contents

List of Tables viii

List of Figures ix

Acknowledgments x

1 Introduction 1

1.1 Disassembly Methods 4

1.1.1 Static Disassembly 4

1.1.2 Dynamic Disassembly 5

1.2 Disassembly Algorithms: 5

1.2.1 Previous Implementations 7

1.2.2 Self-repairing disassembly 8

1.3 Contributions 8

1.4 Organization 9

2 Background 10

2.1 Stack Layout 10

2.2 GCC calling conventions 11

2.3 The ELF File Format 12

2.4 Dynamic Linking and Procedure Linkage Tables 13

2.5 Static Single Assignment 14

3 Design 16

3.1 Overview 16

3.1.1 What we do *not* assume 16
3.1.2 What we assume 18
3.2 Approach Overview 18
3.3 Function Identification 18
3.4 Control Flow Graph 19
3.5 Disassembly of Library Functions 20
3.6 Function pointers and functions with known signatures 21
3.7 Intra-procedural Data Flow Analysis 22
3.8 Generating Function Summaries 23
3.9 Type Analysis 25
3.10 Putting them together 29

4 Implementation 31
4.1 Overview 31
4.2 Control Flow Graphs 31
4.3 Disassembly of Library Functions 32
4.4 Function Summaries 33
4.5 Intra-procedural Analysis 34
4.6 Type Analysis and SSA 35

5 Evaluation 36
5.1 Preliminaries 36
5.1.1 Validation 36
5.1.2 Environment 37
5.1.3 Programs used for testing 38
5.2 Performance Results 38

Bibliography 40

List of Tables

1 Analysis of disassembler on “ls” binary. 38
2 Analysis of disassembler on other binaries. 39

List of Figures

1 Linear Disassembly 6
2 Control flow in a typical ELF binary 21
3 Abstract domain 23
4 Function summary 25
5 SSA representation of a program 29

Acknowledgments

> *Any intelligent fool can make things bigger and more complex... It takes a touch of genius - and a lot of courage to move in the opposite direction.* — Albert Einstein.

It has been a wonderful experience working on my Master's thesis and the System Security Lab. I would first like to thank my parents and my friends for their affection and support. I am heartily thankful to my advisor, Prof. R. Sekar, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject. He has been a great source of inspiration and has taught me the importance of sincerity and dedication. I would also like to thank Prof. Rob Johnson and Prof C. Ramakrishnan for taking their time to be on my thesis committee.

I would like to thank my friends: Pancham, Saurabh, Nitin, Amol, Vishal and Sayali who motivated me to pursue my Masters. I would also like to thank my frinds at Stony Brook: Anupama, Raveesh, Saathyaki, Praveen and Kalpit for the many lighter moments that were a welcome relief from the humdrum of graduate life. I would also like to thank my friends Ashwin, Pradeep, Sanny and Anupama for being there to give me the courage to pursue my goals.

CHAPTER 1

Introduction

With the existence of a large number of vulnerability-related bugs in current software, there is a need for tools that can carry out analysis and transformations to identify and/or mitigate these security vulnerabilities. These tools prevent buffer overflow attacks [12], format string vulnerabilities [23], examine security properties of software [9], etc. Existing tools typically assume the availability of the source code of the software, which is not always realistic. Hence there is a need for tools that can directly analyze and/or transform binaries.

Disassembly is the process of conversion of machine code from the binary image into assembly format [13]. The disassembler may not provide source code information like names of variables and functions since most of this information may not be available. Stripped binaries are binaries which lack information regarding the locations, offsets, sizes and layout of functions as well as objects. Typically, all this information is stored in a symbol table which is generated by the compiler, but is removed before distribution.

Although there is no performance improvement to be had by stripping a binary it could be done for various reasons. Commercial code is stripped to make it difficult to reverse engineer proprietary algorithms; system libraries are stripped to reduce the size on disk; and malware is stripped to obfuscate it and

thus complicate analysis. The general assumption across these applications is that the absence of symbol tables makes analysis of binaries more difficult.

An executable binary can be analyzed either statically or dynamically. Static binary analysis involves the following processes: reading the binary from the disk, interpreting the headers, analyzing the executable and data sections to obtain high level information. Static binary analysis is relatively difficult due to a number of issues as described below.

Problems with static disassembly:

There are many challenges associated with static binary analysis. The most important ones are discussed below.

- *Missing Information about function boundaries and object layouts:* Functions sizes, offset and layout are usually stored in the compiler generated symbol tables. If the symbol tables are absent as in stripped binaries, it is difficult to determine the entry and exit points of functions. Compiler optimizations like tail-call optimization where the function exit point is a jump to the entry point to another function make analyzing functions even more difficult.

- *Indirect Function calls:* Indirect functions are functions which are passed as arguments to other functions and stored in some data structure so that they can be invoked when certain events occur. A very common example is the *atexit()* function, which takes a pointer to a function that is to be invoked before termination of the application. Other examples include virtual method calls in C++ and event handlers in GUIs or event-driven applications.
  Objects may be stored in the stack, data or heap segments. The objects may contain pointers to functions and hence they may need to be analyzed to determine function entry points.

- *Variable Instruction sizes:* On CISC machines (e.g. x86 processors), the size of an instruction can vary from one byte to many bytes. The task

of disassembly would have been simpler and less error prone had the sizes of all instructions been the same. The x86 instruction set has 219 single byte opcodes out of the possible 255 values, and hence there is a high probability that any alignment or data byte can be successfully disassembled as an instruction.

- *Data between code and Alignment bytes:* Alignment bytes are garbage bytes placed between two functions or objects so that they are aligned at four, eight or sixteen byte boundaries as required by the architecture. Alignment bytes are neither data nor code. The compiler could also generate data between the code for a number of reasons. All these bytes poses disassembly problems since it is difficult to distinguish statically between code, data and alignment bytes.

- *Jump tables and Virtual Tables:* Jump tables are usually generated as a result of a switch statement or a n-conditional jump in the higher level program. Jump tables are indexed jumps; the table is stored in a read-only section of the binary and an index register decides the addresses in the table from where the execution has to resume.
  Virtual function tables are a mechanism to support run-time function binding. Virtual tables or vtables are stored in the object (usually at the beginning of the object) and are pointers to a table in the read-only section of the executable. The functions in the vtable are called using appropriate indexes as in jump tables. The difficulty with jump tables and vtables is that it is difficult to determine the size and location of these tables statically which results in incomplete disassembly of binaries.

- *Function with side effects:* Certain functions could affect the behavior of their callee by modifying the return address or performing de-allocation on the stack and changing the values of stack and base pointer registers in a way that could affect the next address to be decoded in the caller function. Thus it is necessary to analyze the behavior of functions to determine how they can affect the overall execution of the program.

Dynamic binary analysis, as the name suggests, analyzes the binary dynamically at runtime. Dynamic analysis is simpler to perform owing to the following factors: the start of the instruction is known accurately and it is easier to distinguish between code and data. However, dynamic analysis has a low coverage since it only analyzes code which is being executed. Moreover, in applications that require transformation (also called "instrumentation") of binaries, a dynamic approach leads to high overheads, typically slowing down the program by an order of magnitude for many applications [22].

Static binary analysis is therefore desirable for most instrumentation applications. Such applications include binary taint tracking [22], performance profiling [15, 17], memory error detection [19], etc. The reliability of these applications depends entirely on the correctness of static binary analysis.

IDApro [5] is the most popular disassembler used for reverse engineering and static analysis. IDAPro uses a depth-first call-graph traversal to determine function start addresses. The disassembler can only identify functions that are called directly with hundred percent accuracy. For the rest, it applies heuristics like looking for standard function prologue patterns as explained in Section 3.1.1. Thus, even though IDAPro has high static disassembly coverage it cannot be used by rewriting and instrumentation tools since these tools cannot tolerate occasional errors in disassembly output.

1.1 Disassembly Methods

1.1.1 Static Disassembly

Static disassembler disassembles the binary statically by reading the binary from a file and interpreting the headers and section contents. Since all the work is done statically, there are no runtime performance overheads in this approach. The output of a static disassembler when used by tools like profilers and binary rewriters gives better performance.

A very common example of a static disassembler is the GNU *objdump* tool.

1.1.2 Dynamic Disassembly

In dynamic disassembly, the disassembler interacts with the software that is to be disassembled. As the software executes, each instruction is disassembled just before it is executed.

The main advantage of this approach is that data can be easily distinguished from code since the disassembler disassembles only those instructions that are going to be executed. Since instructions are decoded when they are being executed, dynamic disassemblers can be used with self-modifying code.

The biggest disadvantage of dynamic disassemblers is performance. Since control has to be transferred to the disassembler before every instruction is executed, there is a considerable slowdown in the application runtime. Dynamic disassemblers also have a low coverage since only the paths which are executed are disassembled.
Dynamic disassembly is employed in many binary analysis and instrumentation tools today, including Pin [20] and valgrind [19].

1.2 Disassembly Algorithms:

There are two basic techniques for disassembly:

- **Linear Sweep:** This is the most straightforward and simple approach to disassembly. Examples of such a disassembler is the GNU disassembler, *objdump*. Disassembly starts from the entry point which is obtained in the header of the binary (e_entry field in ELF format binaries used on moat UNIX operating systems). Each successive instruction is disassembled from the next location, which is obtained by adding the length of the current instruction to the start address of the instruction. The

```
804964c:    55                push   %ebp
804964d:    89 e5             mov    %esp,%ebp
804964f:    53                push   %ebx
8049650:    83 ec 04          sub    \$0x4,%esp
8049653:    eb 04             jmp    0x8049658
8049655:    e6 02 04                 <junk>
8049658:    be 05 00 00 00    mov    \$0x5,%esi
```

(a) Dynamic Disassembly Output

```
804964c:    55                push   %ebp
804964d:    89 e5             mov    %esp,%ebp
804964f:    53                push   %ebx
8049650:    83 ec 04          sub    \$0x4,%esp
8049653:    eb 04             jmp    0x8049658
8049655:    e6 02             out 0x2, al
8049657:    04 be             add al, 0xbe
8049659:    05 00 00 00 12    add eax,0x12000
```

(b) Linear Disassembly Output

Figure 1: Linear Disassembly

basic disadvantage of this technique is that it cannot distinguish data from code. Any data embedded in the code is erroneously disassembled.

Figure 1 shows an example log which contains the actual disassembled output using a runtime disassembler and the output of objdump. As seen from the output of the runtime disassembler, there are some junk bytes stored after the jump instruction. The jump target is 0x4 bytes after the current instruction, which is 0x8049658. The junk bytes could be data or simply some alignment bytes. When disassembled with a linear disassembler, the output is as shown in Fig (b). After decoding the two byte jump instruction at address 0x8049653, the disassembler continues decoding at address 0x8049655 which is probably not code. Thus, the actual target is disassembled incorrectly and the output seems like a jump to the middle of the instruction.

- **Recursive Traversal :** Recursive traversal algorithm has some advantages over linear sweep since it takes into consideration the control flow in the binary. Thus it does not misinterpret data as code. When a jump instruction is decoded, the disassembler continues disassembly from the jump target instead of blindly disassembling the next instruction. The key problem in this approach arises in the presence of indirect control flow transfer. Code that is reachable only via such transfers will not be disassembled by a vanilla recursive disassembly algorithm.

1.2.1 Previous Implementations

The simplest and most commonly used static disassembler is *objdump* [7]. Other approaches based on the recursive traversal fix some of the bugs in the linear disassembler [11]. As discussed, recursive disassembler cannot disassemble instructions when the target of the control transfer is unknown, in which *speculative disassembly* [14] is used which does a linear sweep to analyze unreachable code. Other speculative disassemblers like [10] make certain assumptions to continue the disassembly process which they confirm later in order to accept the disassembly output. Prasad et al [21] uses a hybrid approach for disassembly. They use a combination of well know techniques, viz. recursive traversal and linear sweep and complement them with compiler-dependent pattern matching heuristics. They assume that the prologue of functions are rigid, and do a pattern matching for prologues to determine function entry points. Kruegal et al [14] use a control flow based mechanism and statistical techniques to disassemble binaries which have been shown to be robust even in the presence of some degree of obfuscation. His tools focuses on disassembling binaries which have been obfuscated by tools developed by Linn and Debray [16]. More recently, Nanda et al [18] developed a robust disassembly technique based on a quasi-static approach which is suitable for binary re-writing. In this approach, most parts of the binary are disassembled statically, while a small part which cannot be statically disassembled is instrumented at runtime. Bird performs speculative disassembly, a confidence score is accumulated on the possibility of an unreachable byte being an instruction. At the end, the instruction is considered valid if its confidence score is above a certain threshold. QEMU [8] is a processor emulator that uses a dynamic translator to convert instructions in an emulated program into the host's instruction set. Such conversion can be considered as one form of binary disassembly and instrumentation. Dyninst [2] applies static disassembly to Win32/X86 binary rewriting and optimization. However, it requires full debugging information to guarantee the safety of instrumentation.

Our disassembler does not make assumptions about the presence of symbol tables. We also do not make assumptions about the patterns of prologue and

epilogue in the binaries. However, since our disassembler is aimed at disassembling COTS binaries, we assume that the binary is not obfuscated. The assumptions made by out disassembler is detailed in Section 3.1.1.

1.2.2 Self-repairing disassembly

The problems in analyzing a binary statically lead to a number of disassembly errors. One may conclude that an incorrectly decoded byte would cause all bytes decoded following that byte to be decoded incorrectly; ultimately, one is bound to encounter an invalid opcode at which point the error would be identified. However, this is not true in practice. On the IA, the instruction set is encoded such that the process of disassembly happens to have the self-repairing [16] property, where-in the output of the disassembler synchronizes to the correct disassembly after a few incorrectly disassembled instructions. This makes it difficult to identify disassembly errors except by comparing with correct disassembly output.

1.3 Contributions

In this thesis we make the following contributions

- Jump Table Identification
  Jump tables are stored in the data or read-only data segments. They contain a sequence of jump targets which are used using a register as an index into the table. These targets are not called directly and hence a vanilla recursive traversal disassembler will not detect these targets.
  Jump tables are identified by keeping track of the read-only data section and monitoring jumps through this section. Once such jumps are determined, the table is analyzed for possible jump targets and those targets are disassembled.

- Library calls side effect analysis
  Library functions often take pointers to functions as arguments, and

these functions are called from the library. If the library functions are not disassembled and analyzed, these function entry points will be missed. Calls to library functions occur via the Procedure Linkage Table (PLT). On encountering such calls, the corresponding library is determined, the function is disassembled and analyzed for side effects.

- Abstract analysis
  Indirect function calls are called using a register or memory as an argument instead of an immediate value. Abstract analysis aims to determine static estimates of the values of registers, local memory and the parameters on the stack. The abstract values help identify targets of indirect function calls and jumps.

- Type Analysis
  Type Analysis aims to determine the types of registers and memory locations. Type analysis helps determine function pointers which could not have been determined otherwise by abstract analysis due to aliasing effects.

1.4 Organization

The thesis is organized as follows. Chapter 1 gives an introduction to disassembly and explain various problems and approaches related to disassembly. Chapter 2 covers some background information about file formats, calling conventions and static single assignment. Chapter 3 gives an overview of our design. Chapter 4 explains some implementation details and Chapter 5 presents some evaluation results of our disassembler.

CHAPTER 2

Background

2.1 Stack Layout

The stack on the x86 architecture is referenced using the *esp* register. On the x86 machine, the convention is that stack grows in the downward direction. Thus, the *push* instruction decrements the stack pointer while the *pop* instruction increments. The stack pointer always points to the last valid memory in the stack segment. The pseudo code for *push* and *pop* will make things more clear.

```
push %reg =
{
%esp <-- %esp - 4
(%esp) <-- %reg
}

pop %reg =
{
%reg <-- (%esp)
%esp <-- %esp + 4
}
```

2.2 GCC calling conventions

There is a need to standardize the application binary interface (ABI), without which it will be impossible for code generated by different compilers to work with each other. Unfortunately, in the Linux world, there is often no official standards relating to the ABI. Instead, a *de facto* standard is defined by the GCC compiler developers due to the compilers predominance in the UNIX platform.

The function making a call is called the *caller* and function actually called is called the *callee*. The caller and the callee agree on a certain machine state when the control is transfered to the callee. The transfer and return of the function occurs through the *call* and *ret* instructions.

On 32-bit, x86 based Linux machines, the following ABI requirements apply: At the function entry point, which is the first instruction executed after the call, the states of the registers expected is

- The instruction pointer (eip) points to the first instruction of the callee
- The value at the stack pointer, i.e. (%esp) contains the return address.
- The arguments are present at %esp+4, %esp+8,....,%esp+4*n, where n is the number of arguments.

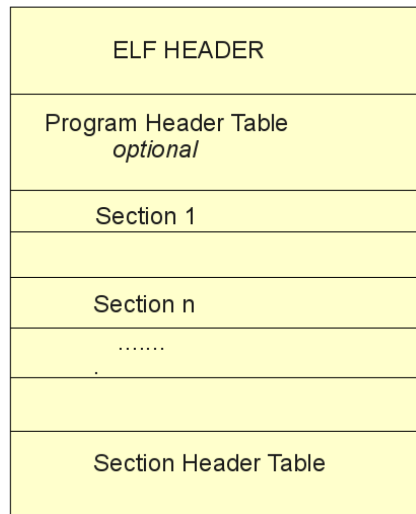
After the execution of the *ret* statement, the caller expects the following machine state

- The instruction pointer points to the next instruction in the caller to be executed, which is the return address
- The return value of the function is stored in register *eax*
- The contents of registers *ebp, esi, edi* and *ebx* are the same as that at the point of call.

The registers are broadly classified as caller saved and callee saved registers. Registers *ebp, esi, edi* and *ebx* are caller saved. If the callee requires the use of these registers, it would have to save the original values and restore before return.

2.3 The ELF File Format

ELF (Executable and Linkable Format) is the object file, executable program, shared object and core file format for Linux and many UNIX operating systems [3]. An ELF file contains an ELF header at the beginning of the file. The size of the ELF header is fixed and it contains information about the program header table and section header table. These values are zero if they are not



present.

The program header table, which is optional, tells how to create a process image. The section header table contains an array of *Elf32_Shdr* structures, which contain information about the various sections in the file. There can be any number of sections in the executable. Some of the common sections present in a typical binary are *.bss, .data, .dynamic, .debug, .got, .fini., .hash, .interp, .rodata* and *.text.*

2.4 Dynamic Linking and Procedure Linkage Tables

Libraries are a collection of subroutines or classes which can be shared between different applications. Shared libraries do not have a fixed load address and hence require the dynamic loader to intervene at run-time.

The Global Offset Table (GOT) and the Procedure Linkage Table (PLT) play a central role in dynamic linking of shared libraries. The PLT adds a level of indirection for function calls. The PLT also permits "lazy evaluation", that is, not resolving procedure addresses until they're called for the first time.

A typical PLT looks as below. Every PLT entry, except the first one which is special, belongs to a single function. When a function in the plt is called, the first jump instruction gets the address of the function from the GOT table. Each PLT function has an entry in the GOT table, which has been initialized by the dynamic linker. The initialized address contains the address of the second instruction, which is the *push* instruction of the corresponding PLT entry. The control is then transfered to the first PLT entry (at address 0x804967c in the example below) which eventually transfers control to the dynamic linker. The dynamic linker then determines the entry point of the function, and also stores this value in the GOT entry (the first jump location) of the corresponding PLT entry. Thus, all future references to that function do not go through the dynamic linker. Instead they behave like a single indirect jump.

```
0804967c <abort@plt-0x10>:
 804967c:       ff 35 f8 3f 06 08       pushl  0x8063ff8
 8049682:       ff 25 fc 3f 06 08       jmp    *0x8063ffc
 8049688:       00 00                   add    %al,(%eax)

0804968c <abort@plt>:
 804968c:       ff 25 00 40 06 08       jmp    *0x8064000
```

```
 8049692:       68 00 00 00 00          push   $0x0
 8049697:       e9 e0 ff ff ff          jmp    804967c <_init+0x30>

0804969c <__errno_location@plt>:
 804969c:       ff 25 04 40 06 08       jmp    *0x8064004
 80496a2:       68 08 00 00 00          push   $0x8
 80496a7:       e9 d0 ff ff ff          jmp    804967c <_init+0x30>
```

On the x86 architecture, the PLT section is read-only while the GOT is a read-write section. This is because the PLT is not being modified, but that GOT is being modified to store the address of the function resolved.

2.5 Static Single Assignment

Static single assignment, often abbreviated as SSA is a representation in compiler design where assignment to a variable can occur only once. Every assignment to a variable creates a new instance of the variable, which is usually represented by the variable name followed by a subscript. If a variable is assigned a value represented by the ϕ symbol then it is just a marker to represent that the LHS can be assigned any value from the group represented by ϕ.

The SSA representation for

```
x = 5
y = z + x
if y > 10 goto L1
x = 7
L1: z = rand()
y = x * z
```

is

x0 = 5

y0 = z0 + x0

x1 = 7

z1 = rand()

x1 = 7

x2 = ϕ (x0, x1)

y1 = x2 * z1

SSA helps in performing a number of compiler optimizations like constant propogation, dead code elimination, register allocation, etc..

CHAPTER 3

Design

3.1 Overview

Our disassembler is intended to disassemble binaries whose file format is supported by Binary File Descriptor (BFD) [1]. BFD is an object file library which permits applications to use same routines to process object files regardless of their format. It uses generic structures to manage information. It then translates data into generic form when reading files, and out of the generic form when writing files. BFD is normally built as a part of the GNU binutils package.

A BFD target is a file that has been loaded using libbfd, part of the GNU binutils distribution. BFD targets have many advantages: the target architecture is automatically detected and elements of the object file structure such as sections and symbols are available. Use of BFD targets is recommended when the target is supported by libbfd.

3.1.1 What we do *not* assume

- **Presence of Symbol Tables:** Our disassembler does not assume the presence of symbol tables. Symbol tables are typically created by the compiler and contain information about the location, name and sizes of

function and variables. The contents of the symbol table can be displayed using the *nm* command.

- **Function Prologue and Epilogue patterns:** Function prologue is the first few lines of assembly instructions present at the start of the function which allocates the stack frame and initializes certain registers (the ebp on x86). Similarly, function epilogue appears at the end of the function which is responsible for restoring the register values as expected by caller and also de-allocates the stack frame. The prologue and epiloque are fairly rigid and appear in the same form in almost all functions. Typically, binaries compiled with a gcc compiler have their prologue as

```
push %ebp
mov %esp, %ebp
sub X, %esp
```

  and epilogue as

```
pop %ebp
ret
```

  Even though the prologue and epilogue are same throughout all functions, different compilers can have different prologues and epilogues or compiler options (e.g. changing the x86 prologue in gcc). Moreover, certain optimizations (e.g. the *–fomit-frame-pointer* option in *gcc*) will change prologues and epilogues.

- **Bytes following a *call* instruction:** The disassembler also does not assume the bytes after a call instruction as the start of a valid instruction. When a call instruction is encountered, as per the recursive traversal algorithm, the called instruction is disassembled and analyzed to check if the functions returns or the return address is being modified.

For example, when a call to *abort@plt* is encountered, our analysis will say that the function does not return and hence we do not disassemble the instruction following the call. The application could store some data bytes after the call to *abort()*, and thus disassembling those bytes would result in interpreting data as code.

3.1.2 What we assume

- **Code Obfuscation:** We assume that the binary is *not* obfuscated. Obfuscation can arbitrarily complicate the disassembly problem with no general solution at all. This assumption is rarely violated for COTS binaries.

3.2 Approach Overview

Our disassembler starts disassembly from the first byte in the executable sections and applies the recursive traversal algorithm. This disassembles the binaries accurately by differentiating between code and non-code. Indirect jumps and calls are resolved by performing static analysis on the control flow graph to determine possible jump/call targets.

3.3 Function Identification

Our goal is to apply binary transformation at a function level. Sometimes it is required to apply transformations to only certain functions. Consider the application of a profiling tool where the rewriting tool adds binary code to the start and end of functions to collect timing information. In such cases it is necessary to have the notion of functions in binaries.

To recover the notion of functions, we start by identifying blocks in binary code. A block (or a basic block) is the basic unit of rewriting. It consists of a sequence of instructions, and has a single entry point and a single exit point.

There are no jumps to the middle of a basic block or jumps from the middle of a basic block to elsewhere. A function consists of a sequence of basic blocks that can be reached by a call instruction outside the function; and one or more exit points which either terminate in a call, jump or a return statement. Functions can also be called using the jump instruction (tail call optimization). Identifying functions called only via jump statements is tricky. We identify the target of a jump statement as a function entry point only if the target of the jump instruction is less than the address of the entry point of the current function being disassembled. Our observation shows that in most cases this is true. This does not affect correctness of disassembly since we disassemble instructions correctly except that the target will not be considered as a function entry point but will be a part of the preceding function. However, if there is some other instruction which does a *call* to that location, we mark it as a function entry point.

Functions could possible have multiple entry points. In this case, we disassemble each entry point as if it is a different function. This may lead to duplication of disassembly efforts, but does not affect correctness of disassembly.

3.4 Control Flow Graph

A control flow graph (CFG) is a representation of all code paths that might be traversed through a program during its execution. Control flow graphs are a by-product of disassembly which are used in further analysis of the binary.

CFGs need to be constructed in order to support various analyses that we perform on binaries. In our approach, disassembly, CFG construction and analysis are interleaved to a certain extent.

3.5 Disassembly of Library Functions

Libraries are binaries which contain re-usable code. Libraries can be either static libraries or dynamic libraries. Dynamic libraries are more popular because of a number of advantage such as low memory footprint and modularity. Calls to dynamic libraries go through a compiler generated table called the Procedure Linkage Table (PLT), whereas calls to static libraries appear like any other local function call, and the body of the function is added into the binary.

Library functions often take pointers to functions in an executable as arguments and the functions are executed at the callee site. Without examining such libraries entry points to those functions are missed resulting in incomplete disassembly. For example, consider the call to *_libc_start_main@plt*, which is the first function called from an ELF executable in Linux systems. This function takes the address of *main()* as an argument and *main()* is called from libc rather than the application. Hence, ignoring the disassembly of *_libc_start_main()* will result in incomplete disassembly as we will never reach *main()*, which is the entry function of the application that makes calls to all other functions in the binary.

The dissembler reads different sections of an executable and initially applies the disassembly algorithm only to the executable segment. On encountering a call to the plt section, the disassembler determines the name of the function and the library in which the function is defined. The list of libraries in which the symbol is to be searched is determined from the DT_NEEDED list in the binary. The DT_NEEDED list is the list of libraries that need to be loaded by the loader before the program is executed. Once the library is determined and read, the offset of the function inside the library is determined from the symbol table and the function is disassembled recursively.

Figure 2 shows a graphical representation of how the calls to *main()* happen in a typical ELF binary.

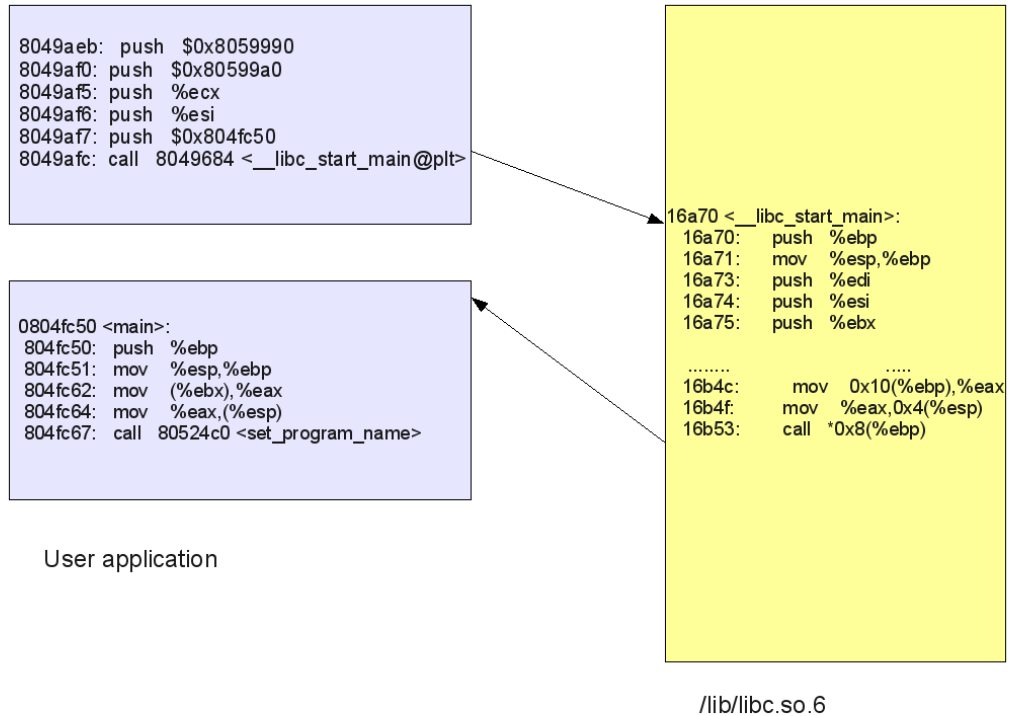


Figure 2: Control flow in a typical ELF binary

3.6 Function pointers and functions with known signatures

Most COTS applications make use of libraries. Many times, the source code of these libraries is available or the signature of the function is known. In such cases, we can analyze the parameters passed to the functions by checking its prototype and conclude that certain parameters are function pointers.

For example, consider a call to the function *atexit()*. The *atexit()* function registers a function to be called during normal process termination. Performing static analysis on the caller and callee does not reveal the fact that the parameter passed to *atexit()* is a function pointer. However, if we assume that

we know the signature of the library function, especially the libc functions like *atexit()* and *sigaction()*, we can disassemble additional functions which will never be called directly by the program.

3.7 Intra-procedural Data Flow Analysis

The recursive traversal and the above analysis cannot achieve 100% accurate disassembly, either, because it is difficult to contruct a complete control flow graph in the presence of indirect branch instructions such as *jmp *r/call *r* or *jmp *m32/call *m32*, where *r* is a machine register and *m32* is a memory location. One solution is to perform additional data flow analysis to determine the possible values of registers and memory locations. Once we determine the locations to which registers and memory can point, it is possible to unveil information about jump targets and function pointers when these registers and memory regions are used as operands to indirect control flow instructions.

Intra-procedural analysis determines the abstract values of registers and memory locations at the end of each function. Memory locations could be in the form of local variables or parameters. The analysis thus reasons about the parameters passed to functions and the values of local variables. We recover information about the registers and memory locations by using an abstract interpretation on an abstract domain. By modeling the value of the stack pointer register ESP at the entry of each function as "BaseSP" (base of activation record), we can track integer-values and stack-pointer based addresses uniformly in instructions.

The domain for abstract interpretation is show in Figure 3. Points in the domain are represented by $X + l$. X is a symbolic representation for the initial value of a variable or register at the entry point of the function. The integer l denotes the displacement of the value from the initial value. The value of X is zero if it has not been initialized by the *caller*.

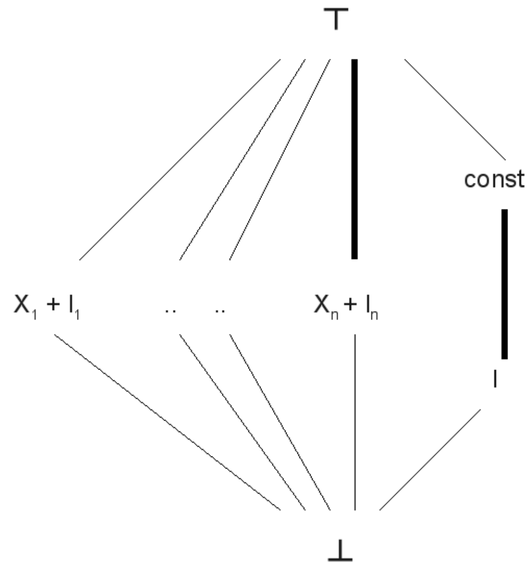


Figure 3: Abstract domain

The information obtained from intra-procedural analysis combined with the information obtained from type analysis (Section 3.9) can be used to identify additional branch targets and functions. Type analysis when applied to a function will designate certain registers and memory locations to be pointers to functions. We can then use the abstract values obtained from intra-procedural analysis to identify the content of these registers and memory locations, thus yielding entry points to functions.

3.8 Generating Function Summaries

The aim of generating function summaries is to recover information about parameters passed to functions and and its use at the *callee* site. The function summary also helps decide the next instruction to be disassembled after the call depending on modifications to the return address.

Functions in C take a set of parameters as input and produce an output in the form of a return value. The parameters to a function could be pointers to other functions. These pointers could be used by the *callee* to call other functions. The *callee* could also update the return address on the stack so that the function returns to a different address other than the byte after the call instruction. Functions return values to the caller in the *eax* register. The returned value could be used by the caller as operands to indirect control flow instructions.

Consider the example below.

```
push 0x805000
push ecx
call <foo>
jmp *0x8057200(,eax,4)


foo>:
  call *0xc(ebp)
  mov 0x8(ebp), eax
  sub eax, 2
  leave
  ret
```

In the example above, function *foo* takes two arguments, a pointer to a function and an integer. The pointer to the function is used to call the function from the callee site, i.e from foo. The integer argument is used to compute the return value. The return value is then used by the caller for indirect control flow.

Typically, the function *foo* could be called from different locations by different callers. The disassembler would disassemble the function only once

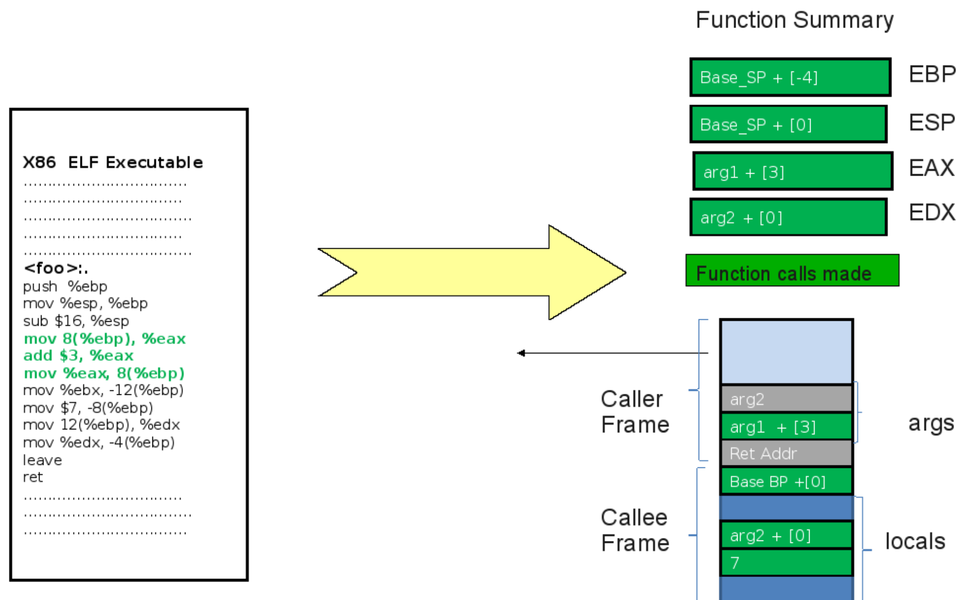


Figure 4: Function summary

and generate a summary for the function. The summary would contain information about how the function *foo* would affect the registers and parameters of the caller. It also stores information about the function calls made by *foo* using the parameters as operands and the updates to the return address, if any, by the *callee*.

Figure 4 shows a graphical representation of the information stored in function summaries.

3.9 Type Analysis

The goal of type analysis is to discover code pointers. Note that in assembly code, code pointers are essentially indistinguishable from integer constants. We are using a type inference technique on binaries to type registers and

memory contents. Any constant that is inferred to have the type of a code pointer will be taken as the possible starting point for further disassembly.

Use of type inference implies an iterative approach for disassembly: in the first iteration, only code reachable without the use of function pointers is disassembled. At this point, type inference can be used to identify some code pointers. By disassembling such code, we extend the CFG for the program, and we apply type inference to the new CFG to discover additional code pointers, if any. The whole process is repeated until no new code pointers are identified.

An alternative to type inference would be a dataflow analysis aimed at identifying the values stored in registers used in indirect control-flow transfer instructions. Traditional data flow analysis could be used for this purpose. However, in binaries, due to extensive aliasing and the absence of information about object boundaries and types will significantly impair the precision of such analysis. For instance, consider an integer value is loaded into a register X, stored into memory (say, in a field of a `struct`), and is then subsequently loaded back into another register Y and used as a function pointer. Due to intervening memory updates, the analysis will not be able to determine with certainty that the original value loaded into X is the same as the one that will be in register Y at the point of indirect control-flow transfer. For this reason, we have chosen to focus on identifying *possible code pointers* rather than guaranteeing that every constant identified by the analysis be a code pointer. Focusing on possible code pointers (rather than definite code pointers) provides the added benefit that a sound analysis can guarantee discovery of all reachable code, thus making fully static instrumentation possible. The downside, of course, is that we will need additional techniques to verify if the identified code pointers are in fact code pointers. Currently, we are relying on heuristics for this purpose, and a fully satisfactory solution is left as future work.

The above discussion suggests that we could potentially use a reaching definition analysis to discover function pointers. In particular, if we can keep track of all possible constant values that flow into a register operand of an indirect control-flow instruction, then we can uncover possible code pointers.

Rather than posing the problem as a flow-sensitive analysis, we have chosen to cast it as a flow-insensitive type analysis problem. We believe that the way in which function pointers are stored and manipulated in programs is better modeled using a type inference approach. For instance, function pointers are often stored in certain fields of a structure, and a pointer to this structure is passed around. Subsequently, this pointer is dereferenced at the offset corresponding to the function pointer, and the result used as the target of a function call. We can easily associate types with such structs, as well as pointers to these structs. (Naturally, in binaries, we will not know the precise boundaries of structs, but the pointer arithmetic used will point us to the relevant offsets within structs.)

So far, we have defined this type inference for only the very basic case of code that makes no function calls. This analysis is defined over an SSA representation derived from code. An SSA representation is needed since binary code may use the same register to store different types of values. An SSA representation treats each assignment of such a register as representing a unique variable, thus avoiding type confusions that may result from such reuse of registers. To describe the analysis, consider the following language:

Prog → FnBody*
FnBody → Lbl:Stmt+
Stmt → x:=c |
    x:=y+c |
    x:=ϕ(y,z) |
    x:= *y |
    *y:= x |
    call *x

Each of these statements induces type constraints as follows. By solving these constraints, we can assign a type to each variable and constant in the program.

jmp *x $\Rightarrow$ x $\in$ fptr

x := c $\Rightarrow$ c $\subseteq$ T(x)
x: = y+c, T(x) = ptr(z) $\Rightarrow$ T(y) $\subseteq$ ptr(x_1*x_2*x_3*...x_{c-1}*z)
x:=*y $\Rightarrow$ T(y)=ptr(z), z $\subseteq$ T(x)
x:=y $\Rightarrow$ T(x)=ptr(z), y $\subseteq$ z
x:=ϕ(y,z) $\Rightarrow$ y $\subseteq$ x, z $\subseteq$ x

Example. Consider the assembly program below.

```
foo>:
push %ebp
mov %esp, %ebp
mov 805000, %eax
call *%eax
leave
ret
```

The control flow graph and its SSA representation is shown in Figure 5. Applying the inference rules bottom-up to the SSA representation, we can make the following conclusions.

jmp *pc1 $\Rightarrow$ pc1 $\in$ fptr
pc1 = eax0 $\Rightarrow$ eax0 $\subseteq$ T(x) $\Rightarrow$ eax0 $\subseteq$ fptr
eax0 = 0x805000 $\Rightarrow$ 0x805000 $\subseteq$ T(eax0) $\Rightarrow$ 0x805000 $\subseteq$ fptr

We thus conclude that 0x805000 is an entry point to a function and start disassembly at that location.

We have implemented the SSA representation of the program. We are currently working on the implementation of the inference rules to determine pointers to functions in binaries.

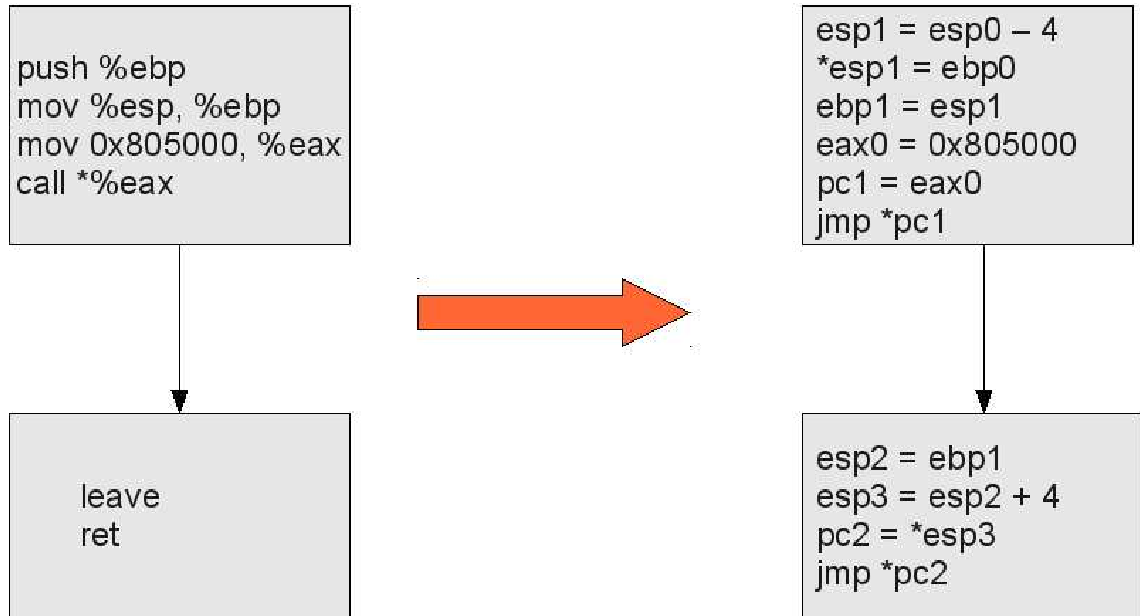


Figure 5: SSA representation of a program

3.10 Putting them together

The disassembler starts by disassembling the first byte of the section as a function entry point and continues disassembly by applying the recursive traversal algorithm. Control flow graph is created as and when control transfer instructions are disassembled. Whenever a library call is encountered, the corresponding function in the library is also disassembled. If the signature of the function being disassembled is known, information about parameters and register values are applied to discover additional functions.

As instructions are decoded, the abstract values of the operands (registers and/or memory locations) and the shadow stack is updated accordingly. These updates are used in generating function summaries, which can then be applied to update the abstract values at the caller site. Updates to return address are also monitored during this time.

The control flow graph thus created is passed as input to the type analysis system. The type analysis module converts assembly instructions to SSA representation by doing a top-down pass on the graph. We then aim to apply the type inference rules described in section 3.9 to discover additional code pointers.

CHAPTER 4

Implementation

4.1 Overview

The disassembler has been developed in C and has approximately 5KLOC of code, which includes the testing tools and the disassembler. We use the XED [6] library for decoding of instructions. We also use the bfd [1] utility to read and interpret the binaries.

4.2 Control Flow Graphs

Control Flow Graphs are constructed in memory and later dumped into a file *cfg.txt* to ease debugging. The data structure for storing CFG's is shown below. Each node in the Cfg has a list of successor's stored in the *succ* field and a list of predecessor or parent nodes stored in the *pred* field.

```
typedef struct _cfg {
        Block bblock;
        Successor *succ;
        Predecessor *pred ;
        int flag;
        unsigned usedId[NUMBER_OF_REG+2];
}Cfg;
```

```
typedef struct successor_blocks {
         struct _cfg *block;
         struct successor_blocks *next;
}Successor;
#define Successor Predecessor
typedef struct basic_block {
         bfd_vma head;
         bfd_vma tail;
}Block;
```

The predecessor field is required to do a bottom-up traversal of the CFG, which is required for type-analysis, and the list of successor nodes is required for parsing the CFG top-bottom which is required for abstract analysis.

The *add_edge()* function is used to add an edge between two basic blocks. This function updates the successor and the predecessor list of the basic blocks. The *split_basic_block()* function is used to split the basic blocks into two. If a jump instruction jumps to the middle of an existing basic block, then the block has to be divided into two, and at the same time, the successor and predecessor list has to be updated.

4.3 Disassembly of Library Functions

The list of dependent libraries are read from the ELF file using a shell script *get_dt_needed.sh* which dumps the list in a temporary file */tmp/dt_needed*. When a library call is encountered, the data structure for the library symbol table is built. We use a *lazy* approach and construct the data structure only to the point where the required library and symbol is found.
The data structure storing the library and symbol information is shown below:

```
typedef struct external_libs_and_symbols{
```

```
    struct external_libs_and_symbols *next;
    char *lname;
    asymbol **sorted_syms;
    asymbol **dynsyms;
    asymbol **syms;
    asymbol *synthsyms;
    long symcount;
    long dynsymcount ;
    long sorted_symcount ;
    long synthcount;
    bfd *abfd;
}ExtInfo;
```

4.4 Function Summaries

The disassembler maintains a *shadow* stack during disassembly of the binary. This stack is useful in analyzing libraries and local function calls.

The *shadow* stack is maintained as a singly linked list. On decoding a *push*, *pop* or a *mov* instruction to the stack, the linked list is updated accordingly with the source operands of these instructions. When a memory reference to the stack is decoded as the source operand of an instruction, the abstract values from the stack are used to update the other data structures. The data structure used to represent the stack is shown below.

```
typedef struct operandList {
    int flag;
    union {
        xed_int32_t imm;
        xed_reg_enum_t reg;
    };
    struct operandList *next;
}pushList;
```

4.5 Intra-procedural Analysis

At any point during the execution of the program, a register contain only one value. The value, however, defers depending on the context and execution flow. Hence, we maintain the values of register in a single dimensional array.

Static analysis and disassembly is performed hand-in-hand. Also, disassembly is performed depth-first, where a control transfer instruction in the current basic block is disassembled before that of its predecessor. Hence, at a *join* node, all the possible values of the register may not be known since it may not have been disassembled. Consider a simple assembly program as below.

```
100: mov eax, (10)
102: cmp eax, 0x1
104: ja 111
106: mov ebx, 200
108: jmp 113
111: mov ebx, 300
113: call *ebx
```

In the above example, when instruction 113 is disassembled, instruction 106 (or 111) will not be decoded, and the static analysis value of register *ebx* will be 300 (or 200), and the function at that address will be decoded. Thus, while disassembling instruction at address 113, all values of register *ebx* are not known.

Later, when the basic block at 106 (or 111) is disassembled, the control transfer to instruction 113 will not be followed since the instruction at that address has already been disassembled, and thus the function at address 200 (or 300) will not be disassembled (unless there is another direct call to that function). To avoid this, we perform a second pass which covers all possible execution paths in the CFG and thus discovering more function pointers.

4.6 Type Analysis and SSA

The control flow graph contains information of the basic blocks in the form of machine instructions. The machine dependent assembly instructions in the control flow graph is then converted to a machine independent intermediate SSA representation.

Each basic block contains a representation of the SSA form. While creating the SSA representation from the CFG, we have to make sure that the SSA representation of all the predecessors is computed before the successors. Hence we maintain a flag variable in the block structure to keep track of this. Special care has to be taken in case of cycles in the graph.

In our SSA representation, each register is denoted by a single character. Access to memory addresses is handled in a slightly different way. The start of the data segment is denoted using *D*. A write to a memory location at an offset of 100 from the start of the data segment is denoted as:

```
v_1 = D + 100
*v_1 = R
```

where R is the register value being assigned.
Similarly, local variables is denoted by u and parameter are denoted by w. Accessing a local variable *0x8(esp)* is denoted as

```
u = s_1 + 8
R = *u
```

where R is the register to which the local variable is being assigned. Parameters are assigned values similarly.

CHAPTER 5

Evaluation

5.1 Preliminaries

In this section, we present the set of programs the disassembler was tested with, the configuration parameters and the general environment setup for testing. We also discuss how the results were validated.

5.1.1 Validation

As we have seen in the previous chapters, it is difficult to design a disassembler because of a variety of reasons. It is also equally difficult to validate the output of a disassembler, especially because of the self repairing property. Hence it is necessary to devise a set of tools to perform validation of the disassembler.

The output of the disassembler cannot be compared with any other disassembler since there is no static disassembler that can disassemble all binaries accurately. Hence, to validate the output of the disassembler we can use one of the following methods.

- Compare with compiler generated object files: The compiler converts the source code to various intermediate stages before converting it into machine code. The assembly output of the compiler can then be used to be compared with the output of the disassembler.

However, this is not practical because of a number of reasons. Firstly, the object files or compiler generated assembly files do not have relocation and address information. Jumps and calls occur using symbols in the string table. Thus, automation of the process is difficult and becomes impossible while working with executables which have no symbol information. Secondly, when multiple files are linked together to generate a single final executable, the object files are created for each file. Thus, the assembly output to be compared will be distributed in multiple files.

- Pass disassembled output to assembler and compare binaries: The disassembler output is in the form of assembly instructions. If these assembly instructions are given as input to an assembler to generate the final executable, then this newly generated binary can be compared with the original binary. The comparison can be in the form of a *diff* or by validating the outputs for a well defined input set.

- Dynamic disassembly: The dynamic disassembler is very accurate as it disassembles each instruction before it is executed. However, the dynamic disassembler disassembles only those instructions that are executed.

We validate the output of our static disassembler by comparing it with the output of a dynamic disassembler. We have developed a tool based on Pin [20]. Since the coverage of the dynamic disassembler is low, we take output from multiple runs of the dynamic disassembler for testing.

5.1.2 Environment

The disassembler was tested on an Intel Core 2 Duo 2.1 GHz, 2 GB RAM with Ubuntu 9.10, Karmic Koala. The disassembler was built with gcc-4.4.1 and glibc 2.10.

5.1.3 Programs used for testing

To evaluate the disassembler, we tested it on the most widely used utilities in Linux, core-utils. Core utils [4] contain a bunch of utility tools like *ls, cat, diff, chmod* etc.,which are widely used. All programs are written in C and make use of function pointers and indirect jumps.

5.2 Performance Results

The table below shows the performance results in the disassembly of the *ls* binary. The percentage gaps and disassembly at various stages is shown as well.

Analysis	%GAPS	%Reachable code not disassembled
Recursive traversal	97.26	85.25
Disassembly of libraries for side effects	25.8	13.799
Jump table analysis	12.94	0.9396
Function pointer analysis	12.0004	0

Table 1: Analysis of disassembler on “ls” binary.

As seen from the table, better disassembly was achieved by performing various analysis and techniques. The final result still contains some code which has not been disassembled. These are valid gaps which correspond to alignment bytes and functions which have never been called.

To validate that the gaps are indeed valid, we used a number of techniques. We first executed the application using the dynamic disassembler. The application was executed with all the options so that close to hundred percent coverage was obtained. Then, these logs were put together to get close to 100% disassembly coverage. We then compared these logs to check if there were any *hits* in the areas we discovered as *gaps*.

As a second step to gain more confidence, we performed a second parse over the disassembled output to check if any of the addresses discovered as *gaps* is being used as an operand in the disassembled instructions. This is because often addresses of functions are passed on the stack using *push* or *mov* instructions, and the called function may use these functions as pointers. On detecting such instructions, we throw a warning to the user so that he can manually validate the address or take any further action. We also scan the *rodata*, *bss* and *data* section for references to these addresses.

Below are the disassembly results of some other binaries.

Application	%GAPS	%Reachable code not disassembled
chroot	14.80	0
chmod	12.70	0
cat	7.66	0
pdftops	2.84	0
dhclient	19.08	4

Table 2: Analysis of disassembler on other binaries.

Bibliography

[1] Binutils bfd. http://sourceware.org/binutils/docs/bfd/index.html.

[2] Dyninst: An application program interface (api) for runtime code generation. http://www.dyninst.org/.

[3] Executable and linkable format.

[4] Gnu core utilities. http://www.gnu.org/software/coreutils/.

[5] Idapro disassembler. http://datarescue.com/idabase/.

[6] Xed: X86 encoder decoder. http://www.pintool.org/docs/24110/Xed/html.

[7] Free software foundation, gnu binary utilities, March 2002. http://gnu.org/software.

[8] F. Bellard. Qemu, a fast and portable dynamic translator. In Proc of USENIX 2005 Annual Technical Conference, FREENIX Track, pp 41-46, 2005.

[9] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. Technical Report: CSD-02-1197, 2002.

[10] Cifuentes, M. V. Emmerik, D. Ung, and T. Waddington. Premliminary experience with the use of the uqbt binary translation framework. Proceedings on the workshop on Binary Translation, 1999.

[11] C. Cifuentes and K. Gough. Decompilation of binary programs. 1995.

[12] C. C. et al. Stackguard: Automatic adaptive detectionand prevention of buffer-overflow attacks. Proceedings of the 7th USENIX Security Symposium San Antonio, Texas, January 1998.

[13] W. Hsieh, D. Engler, and G. Back. Reverse-engineering instruction encodings. In USENIX Annual Technical Conference, pages 133-146, June 2001.

[14] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. 13th USENIX Security Symposium, 2004.

[15] J. Larus and T. Bal. Rewriting executable files to measure program behaviour. Software Practise and Experience 4,2, February 1994.

[16] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disasembly. 10th ACM conference on Computer and Communications Security (CCS), October 2003.

[17] 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, : 37-46. Special issue on performance evaluation tools for parallel and distributed computer systems, November 1995.

[18] S. Nanada, W. Li, L.-C. Lam, and T. cker Chiueh. Bird: Binary interpretation using runtime disassembly. Proceedings of the International Symposium on Code Generation and Optimization, 2006.

[19] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, June 2007.

[20] H. Patil, R. Cohn, M. Charney, R. Kapoor, and A. Karunanidhi. Representative portions of large intel itanium programs with dynamic instrumentation. Proceedings of the 37th Annual IEEE/ACM international Symposium on Microarchitecture, 2004.

[21] M. Prasad and T. Chiueh. A binary reriting defense against stack based buffer overflow attacks. Procesings of the General Track: USENIX Anual Technical Conference, 2003.

[22] P. Saxena, R. Sekar, and V. Puranik. Efficient binary instruementation with applications to taint tracking. ACM/IEEE International Symposium on Code Generation and Optimization (CGO), April 2008.

[23] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium, May 11, 2001.