2019

"Regsym" - dataflow analysis of linear x86 code

Grant Foudree
Iowa State University

"Regsym" - dataflow analysis of linear x86 code

by

Grant Reade Foudree

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering (Secure and Reliable Computing)

Iowa State University

Ames, Iowa

2019

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to take this opportunity to express my gratitude to everyone involved in my graduate studies and this thesis. I am truly thankful to have friends and family who have continuously extended their steadfast support, encouraging me to strive forward even if it was one step at at time. I would also like to extend thanks to my committee members for their guidance and flexibility throughout my research. Dr. Kothari, in particular, has been instrumental in helping me throughout my entire graduate journey, encouraging me to go forth and pursue research. It is an understatement to say I could not have done this without him.

ABSTRACT

Software analysis of binary code is an important and challenging area of ongoing research. Due to the significant complexity, tasks such as accurate function identification and control flow graph recovery remain open problems in both industry and academia. The ability to reliably analyze binary code is quite valuable to several industries, ranging from cybersecurity to software verification. While several tools exist to disassemble and model control flow in binary programs, none have a reliable and robust way of modeling dataflow. To address this gap, this thesis will present a new technique to trace register dataflow across program slices on the x86 architecture with a tool called "RegSym".

CHAPTER 1. GENERAL INTRODUCTION

1.1 Overview

In this section, several concepts of binary analysis and x86 architecture specifics will be described in order to provide the reader with a healthy amount of background information on the topic.

1.1.1 Applications of Dataflow Analysis

1.1.1.1 Reverse Engineering & Software Verification

Reverse engineering binary code has several applications such as verifying the correctness of a region of code and determining what a program does when executed. While several solutions exist to model key elements of reverse engineering, such as CFG extraction, having the ability to view dataflow information would significantly help an analyst when examining a program. For example, given a binary program containing a proprietary encryption routine, an analyst wants to crack or verify the security of the algorithm. By tracing the dataflow of the plaintext and key, one can obtain a better understanding of how the routine operates.

1.1.1.2 Cybersecurity

Determining whether software is vulnerable to exploitation is not an easy task, however there are several techniques that can be used to help mitigate and detect software vulnerabilities. One example is taint analysis, which is the process of "tainting" inputs to a program and all the subsequent regions in order to highlight the areas of the program that can be influenced by external inputs [28]. This allows a developer to determine which regions of the program can potentially be controlled by an attacker and guard against this appropriately. Typically this is done via "dynamic taint analysis" which involves running binary code that has been instrumented in order to record the tainted regions of memory during runtime. In order to do this from a static analysis standpoint, the ability to model dataflow is essential so that inputs can be traced throughout the program, and the taint propagated.

1.1.2 x86 Architecture

The majority of personal computers and servers today run on the x86 architecture which was designed by Intel. While other architectures are popular, such as ARM in the mobile environment, x86 was chosen for this research due to its complexity and prevalence in the computer market. The details of the x86 architecture that are salient to this research will be described in the following subsections.

1.1.2.1 Instructions

x86 is a complex instruction set computing (CISC) architecture with a large set of instructions that are encoded with a variable-width. Unfortunately it is hard to determine the exact number of instructions, as this changes from model to model and lots are undocumented [11], however according to one source, 1044 instructions were counted [7]. Since a lot of these instructions are complex, per the CISC architecture, some can perform several mini-operations in one instruction, reducing code size. On the other hand, this complicates analysis as not only are there 1000+ instructions that must be interpreted and modeled correctly, but a large portion of them are non-trivial instructions to process. For example, the `VINSERTF32x4` vector instruction is implemented as such [8]:

```
(KL, VL) = (8, 256), (16, 512)
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] ← SRC2[127:0]
        1: TMP_DEST[255:128] ← SRC2[127:0]
    ESAC.
FI;
```

```
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0]←SRC2[127:0]
        01: TMP_DEST[255:128]←SRC2[127:0]
        10: TMP_DEST[383:256]←SRC2[127:0]
        11: TMP_DEST[511:384]←SRC2[127:0]
    ESAC.
FI;
FOR j←0 TO KL-1
    i←j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i]←TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*
                THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Due to this overwhelming complexity and large instruction set, binary analysis tools that target the x86 architecture almost always leverage the use of an intermediate representation (IR) that abstracts these details from the analysis software.

1.1.2.2 Registers

In x86_64, the main registers consist of 14 general purpose registers (RAX, RBX, RCX, RDX, RDI, RSI, R8-R15), a stack pointer and base pointer (RSP, RBP) and an instruction pointer register (RIP). Other architectures, such as PowerPC, contain significantly more general purpose registers (r0-r32) [5]. The "R" prefixed registers (R{AX,BX,CX,...}) stand for 64-bit wide registers, however these registers can be accessed in lower bit modes such as 32, 16, and 8 bit mode by changing the name. To access the register in 32-bit mode, "R" is changed to "E" (EAX). 16-bit mode access removes the prefix (AX) and 8-bit mode can be accessed via AH, which is the upper 8 bits of AX, or AL for the lower 8 bits. Note that zeroing out RAX has the effect of setting EAX, AX, AH, and AL to zero as well as these are simply accessing the same register, just at different bit-widths.

CPU architectures that have a higher register count can make things such as dataflow analysis easier in functions that utilize a large amount of variables. Having lots of registers means that the values can be stored in registers as opposed to pushing/popping values to the stack when the registers get full. This concept presents a slight challenge in x86 due to the lower register count.

1.1.2.3 Calling Convention

In order to preserve application binary interface (ABI) compatibility amongst binary programs, a standard for how parameters and return values are exchanged during function calls is necessary. On x86, there are several different calling conventions available depending on the operating system and processor mode (32-bit vs 64-bit). In this thesis we will focus on the Linux operating system operating in 64-bit mode.

When a function is invoked in 64-bit Linux, which uses the System V AMD64 ABI, parameters are passed via registers in the following order: RDI, RSI, RDX, RCX, R8, R9, XMM0-XMM7 [15]. When a function returns, the value passed back is available in the RAX register. This is important because in order to trace the dataflow of function parameters, we must know which registers they are being placed in.

Since this order must be consistent to allow interoperability with system libraries, it can be useful for identifying potential functions as well as extracting their parameters. For the purpose of dataflow analysis, we can determine the order of parameters based on which register they are received in. For example, we know that a parameter in the RDI register is parameter 1, and a value in RSI would be the second parameter and so on.

The following example demonstrates a call to `printf()` with 5 parameters:

```
int main() {
        printf("1: %c 2: %d 3: %lu 4: %s", 'a', 5, 10, "test");
}
```

The disassembly of the compiled C code above reads:

```
mov r8d, str.test                       // String: "test"
mov ecx, 0xa
mov edx, 5
mov esi, 0x61
mov edi, str.1:__c_2:__d_3:__lu_4:__s // String: "1: %c 2: %d 3: %lu 4: %s"
mov eax, 0
call sym.imp.printf
```

With regard to program flow, once a function is invoked via a call instruction, the address of the next instruction is pushed to the stack and the target is jumped to. Upon the callee's exit, this return address is popped off the stack and jumped to. This information is important to perform things like stack-unwinding in a debugger.

1.1.3 Analyzing Binary Code

Substantial challenges exist when analyzing binary code, especially for complex architectures such as x86. In this section, several common obstacles will be addressed.

1.1.3.1 Creation of a Binary Executable

When compiling source code into an executable binary, the steps are roughly outlined in Figure 1.1.

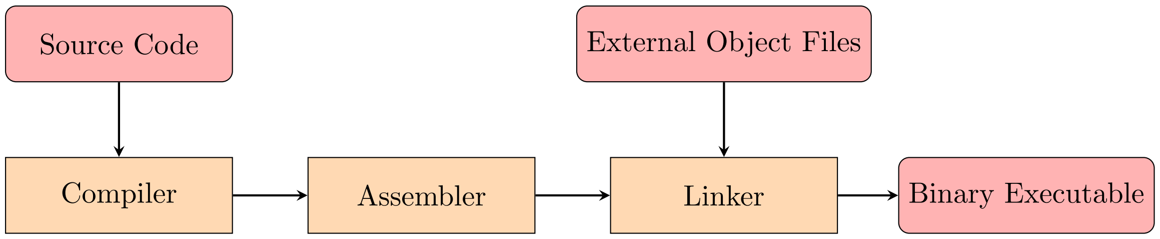


Figure 1.1: Compilation Stages

In the compiler phase, the input source code is parsed and fed through many complex algorithms to translate the code into primitive assembly instructions. The assembler then takes these instructions and encodes them into opcodes which the processor can understand, and outputs this in the form of an object file. Finally, the linker takes 1 or more object files and combines them together, patching up the addresses to produce an executable binary output.

Between the compiler phase and assembler phase, lots of high level information that makes the code human readable is lost. This is due to the fact that the processor does not need to be aware of several details, such as type information, and therefore it is removed for performance and size reasons. Unfortunately, this makes reverse engineering quite challenging because the concept of variables, types, and even functions and loops can be completely removed from the generated binary file. Since compiling is a "lossy" operation, it can be very difficult, if not impossible, to reconstruct the high-level source code from a given binary. Things like compiler optimizations and intentional code obfuscation techniques exacerbate the problem further.

To demonstrate the loss of type information, examine the following example of the compiling a simple C `struct` through each step of the pipeline.

```c
typedef struct ex {
        char first;
        unsigned int second;
} _ex;
struct ex foo(struct ex a) {
        a.second = 44;
        return a;
}
```

Compiling the C source into the LLVM IR still retains the name of the `struct`, but we have already lost the explicit information that we are accessing the `struct ex.second` element and have to deduce this from the offset instead of a name.

```llvm
%struct.ex = type { i8, i32 }

define dso_local i64 @foo(i64) #0 {
  %2 = alloca %struct.ex, align 4
  %3 = alloca %struct.ex, align 4
  %4 = bitcast %struct.ex* %3 to i64*
  store i64 %0, i64* %4, align 4
  %5 = getelementptr inbounds %struct.ex, %struct.ex* %3, i32 0, i32 1
  store i32 44, i32* %5, align 4
  %6 = bitcast %struct.ex* %2 to i8*
  %7 = bitcast %struct.ex* %3 to i8*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* align 4 %6, i8* align 4 %7, i64 8, i1 false)
  %8 = bitcast %struct.ex* %2 to i64*
  %9 = load i64, i64* %8, align 4
  ret i64 %9
}
```

After the IR is done being compiled, assembled, and linked, we get the following code:

```
0000000000000000 <foo>:
    mov    QWORD PTR [rsp-0x10],rdi
    mov    DWORD PTR [rsp-0xc],0x2c
    mov    rdi,QWORD PTR [rsp-0x10]
    mov    QWORD PTR [rsp-0x8],rdi
    mov    rax,QWORD PTR [rsp-0x8]
    ret
```

At this stage, we have lost all variable names, types, and even the concept of a `struct`. Since the processor operates on bits this makes sense, however from an analyst's point of view it makes things more difficult.

1.1.3.2 Control Flow Graph Recovery

One of the main challenges of binary analysis is the correct control flow graph (CFG) recovery of a program. For many reasons outlined below, CFG recovery is not always accurate. Concepts like dynamically-computed branch and jump targets present issues for CFGs [17]. For example, if there is a program that has a function pointer and assigns it to the address of a function from a loaded shared-object or DLL, it is difficult to statically determine the jump target. In the code shown in Listing 1, there is call to an address in the RDX register, but it is not trivially apparent that this is a call to the `double cos(double x)` function because the jump is to an address returned from `dlsym()`.

There are several other issues that complicate CFG recovery such as virtual function tables, relocated functions in position-independent executables, jump tables, and exception handlers. Discrepancies and errors in the CFG have compounding consequences for analysis tools. For example, incorrectly identifying branch conditions and basic blocks presents the issue of incorporating (or excluding) instructions into a program slice which can drastically change its dataflow.

Constructing an accurate CFG is an essential step for many binary analysis techniques.

Listing 1 Dynamically-computed jump

```
#include <dlfcn.h>
int main() {
        double (*cosine)(double);
        void *handle = dlopen("libm.so.6",

        *(void**)(&cosine) =
            dlsym(handle, "cos");
        (*cosine)(2.0);
        dlclose(handle);
}
```

```
call    401050 <dlsym@plt>
mov     QWORD PTR [rbx],rax
mov     rdx,QWORD PTR [rsp]
mov     rax,QWORD PTR [rip+0xe9e]
movq    xmm0,rax
call    rdx
```

1.1.3.3 Function Identification

In addition to control flow graph extraction, function identification with 100% accuracy is another challenging task of binary analysis [6], [26], [3]. Incorrectly identifying function boundaries might exclude (or include) extra bytes that are not taken into consideration which is problematic for several reasons. In the instance of dataflow analysis, several instructions might be missing (or added) which have an effect on the dataflow of that function.

For ELF binaries that have not been stripped, function identification can easily be achieved by reading the symbol table inside the ELF file. This table resides in the `.symtab` and `.strtab` sections of the ELF binary and contains the name of the function, the start address and the size. Below you can observe that the main function is at 0x401126 and is 57 bytes.

```
Symbol table '.symtab' contains 85 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
...
    80: 0000000000401040    47 FUNC    GLOBAL DEFAULT   13 _start
    81: 0000000000404024     0 NOTYPE  GLOBAL DEFAULT   24 __bss_start
    82: 0000000000401126    57 FUNC    GLOBAL DEFAULT   13 main
    83: 0000000000404028     0 OBJECT  GLOBAL HIDDEN    23 __TMC_END__
    84: 0000000000401000     0 FUNC    GLOBAL HIDDEN    11 _init
```

However with stripped ELF files, we lose explicit function information as shown below.

```
Symbol table '.dynsym' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
```

Stripped binaries are common as the symbol tables in are not necessary for program execution and exist to aid in debugging. Furthermore, stripping binaries reduces their size which is a common use-case for removing this information - especially in resource constrained environments.

With the loss of these explicit boundaries, several attempts have been made to develop a robust technique for identifying function boundaries. Challenges such as functions with multiple entrypoints, shared code, and tail calls are all common obstacles that make the identification of function boundaries difficult [12]. A common idea is to look fingerprint functions by searching for the creation and destruction of a stack frame and label those addresses as a function's boundaries. While this might work for a significant number of functions, it is common for the compiler to omit the stack frame for performance, or via *–fomit-frame-pointer*, and even use a region called the ”red zone” as a replacement for the stack when less space is needed [22]. Therefore, this sort of technique will often result in poor accuracy.

1.1.3.4 Generating Disassembly

In order to perform analysis of binary code, it is necessary to decode the raw opcodes by generating a disassembly of the file. To accomplish this, there are two main techniques, linear disassembly and recursive disassembly. The first technique, linear disassembly, is a very simplistic approach in which a disassembler starts in the *.code* section of a binary and iterates over the entire region, translating the bytes into assembly instructions [25]. The linear disassembly stops when an illegal instruction is hit which causes problems when data is mixed with code, as well as other issues.

Linear disassembly is straightforward on architectures that implement fixed-width instructions, or instructions that have opcodes of the same length, such as PowerPC [14] and MIPS [5]. On x86, however, instructions are variable-length encoded so binning a series of opcodes into 32-bit buckets and decoding each as an instruction will not work. As a result, linear disassemblers must be careful to properly determine the length of the opcode so that it can be decoded properly as failure to do so will have a cascading effect on subsequent instructions being decoded properly. This is challenging on x86 due to the CISC architecture where a large set of instructions exist, all of which need to be decoded properly.

Data that exists in the code region can cause problems for linear disassemblers because they interpret every byte they see as an opcode and do not make an attempt to differentiate data from code. For example, given the following assembly code in listing 2, we can observe that the linear disassembler Objdump [13], incorrectly identifies a word as an opcode at offset 9 in the disassembly. Because a word was inserted that decoded to a valid opcode, Objdump assumed it was an instruction.

Listing 2 Trick Assembly

```
.global main
.intel_syntax noprefix
main:
        mov rax, 0
        jmp over
        .word 0x03eb
over:
        mov rbx, 0
```

Given the large number of opcodes for the x86 architecture, there is a high probability that a "bad byte" will correspond to a valid opcode and be misinterpreted. This amplifies the problem as the disassembler loses track of the current alignment for what bytes it is interpreting as

Listing 3 Objdump Output

```
0000000000000000 <main>:
   0:         48 c7 c0 00 00 00 00          mov    rax,0x0
   7:         eb 02                         jmp    b <over>
   9:         eb 03                         jmp    e <over+0x3>
000000000000000b <over>:
   b:         48 c7 c3 00 00 00 00          mov    rbx,0x0
```

opcodes in addition to outputting incorrect disassembly. A ”domino” effect is observed as subsequent instructions are decoded improperly. In the example below, there is a jump over a data byte (`call L1 + 1`) to the instruction `pop rax` in listing 4. The problem arises when Objdump interprets the `.byte 0xe9` line as an instruction and starts decoding it and the subsequent 4 bytes as a valid opcode which is incorrect. As a consequence, Objdump has lost alignment and continues to misidentify instructions such as the `or DWORD PTR [rax-0x3d],edx` in listing 5, which does not exist. It is noteworthy that even though the disassembly reported by Objdump is incorrect, the processor will still execute the valid assembly code. This is important because tricks outline above can be used by malware and obsfucators to hide what is actually being executed as well as confuse analysis tools.

Listing 4 Assembly Source

```
main:
        call L1 + 1              // Jump to "pop rax"
L1:
        .byte 0xe9               // Skipped
        pop rax                  // Get program counter (RIP) into RAX
        add rax, 9               // Add 9 which is location of "nop"
        push rax
        ret                      // "return" to computed address
        .byte 0xe9               // Jumped over
        nop                      // Target
```

The other popular disassembly technique is recursive dissassembly which is more robust, and addresses some of the issues with linear disassembly outlined above. Instead of treating each

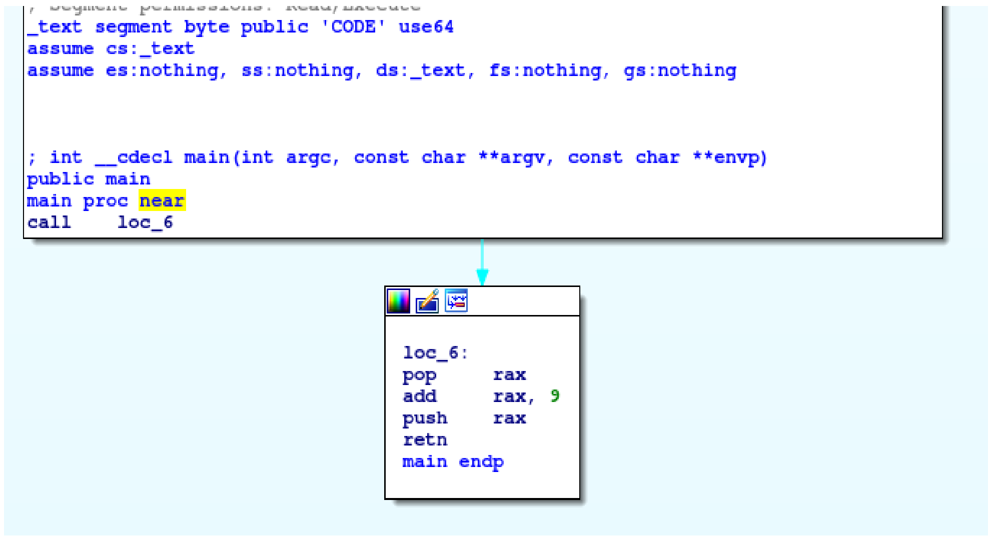
Listing 5 Objdump output

```
0000000000000000 <main>:
    0:          e8 01 00 00 00                call    6 <L1+0x1>

0000000000000005 <L1>:
    5:          e9 58 48 83 c0                jmp     ffffffffc0834862
    a:          09 50 c3                      or      DWORD PTR [rax-0x3d],edx
    d:          e9                            .byte 0xe9
    e:          90                            nop
```

sequential byte as a valid opcode, recursive disassemblers perform a basic control flow analysis, interpreting which bytes are actually reached during execution. As the disassembler iterates, jumps and branches are taken into consideration to discover new regions of code to disassemble. In the example above, it is clear that the `call L1 + 1` instruction jumps over the `.byte 0xe9` data byte and there are no other regions in the program that execute this byte.

IDA Pro [27] is a popular recursive disassembler and is the industry standard for disassembling and reverse engineering binaries. Using the same example above, in Figure 1.2, IDA Pro is able to properly disassemble the code due to its recursive approach, excluding the invalid data bytes inserted in the code region.



```
; Segment permissions: Read/Execute
_text segment byte public 'CODE' use64
assume cs:_text
assume es:nothing, ss:nothing, ds:_text, fs:nothing, gs:nothing


; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near
call    loc_6
```

```
loc_6:
pop     rax
add     rax, 9
push    rax
retn
main endp
```

Figure 1.2: IDA Pro Dissassembly

1.1.4 Intermediate Representations

As mentioned in the x86 architecture section, the use of an IR greatly aids in binary analysis of complex architectures due to the ability to simplify instructions to basic operations. In addition to simplifying CISC architectures, using an IR provides architecture independence for the higher level algorithms to operate on. As a result, porting tools to other architectures is quite easy as the low-level details have been abstracted via the IR.

Many different flavors of IRs exist. Some of the popular ones are VEX, which is used by Valgrind [34], PCode, which is used by Ghidra [2], and LLVM IR [18]. In this work, the VEX IR will be used with the Python bindings, PyVEX [29]. This was chosen due to its simplicity, ease of use, and correctness during testing when compared to other IR frameworks.

To demonstrate the value of an IR, we can look at the following example. The instruction `VPABSW xmm1, xmm2/m128` in the x86 ISA "computes the absolute value of 16-bit integers in xmm2/m128 and stores the unsigned result in xmm1" [1]. The instruction logic is described as:

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j←0 TO KL-1
    i←j * 8
    IF k1[j] OR *no writemask*
        THEN
            Unsigned DEST[i+7:i]←ABS(SRC[i+7:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*
                        ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

When "lifted" into the VEX IR, we are given the following representation:

```
IRSB {
   t0:Ity_V128 t1:Ity_V128 t2:Ity_V128 t3:Ity_I64 t4:Ity_I64 t5:Ity_I64 t6:Ity_I64 t7:Ity_I64
    ↪   t8:Ity_I64 t9:Ity_I64 t10:Ity_I64 t11:Ity_I64 t12:Ity_I64 t13:Ity_I64 t14:Ity_I64
    ↪   t15:Ity_I64 t16:Ity_I64 t17:Ity_I64 t18:Ity_I64 t19:Ity_I64 t20:Ity_I64 t21:Ity_I64
    ↪   t22:Ity_I64 t23:Ity_I64 t24:Ity_I64 t25:Ity_I64

   00 | ------ IMark(0x40000, 5, 0) ------
   01 | t1 = GET:V128(xmm2)
   02 | t15 = V128HIto64(t1)
   03 | t16 = V128to64(t1)
   04 | t8 = SarN16x4(t16,0x0f)
   05 | t17 = Not64(t8)
   06 | t7 = Sub16x4(0x0000000000000000,t16)
   07 | t19 = And64(t7,t8)
   08 | t20 = And64(t16,t17)
   09 | t18 = Or64(t20,t19)
   10 | t13 = SarN16x4(t15,0x0f)
   11 | t21 = Not64(t13)
   12 | t12 = Sub16x4(0x0000000000000000,t15)
   13 | t23 = And64(t12,t13)
   14 | t24 = And64(t15,t21)
   15 | t22 = Or64(t24,t23)
   16 | t2 = 64HLtoV128(t22,t18)
   17 | PUT(xmm1) = t2
   18 | PUT(272) = 0
   NEXT: PUT(rip) = 0x0000000000040005; Ijk_Boring
}
```

As one can observe, the VPABSW instruction is not terribly difficult to interpret and model by itself, however modeling each instruction of x86 ISA is quite a task seeing as there are plenty of instructions just as complex, if not worse. By using the VEX IR, we can simplify these instructions into primitive operations (such as AND, OR, NOT, ADD, MUL, etc...) that can be easily modeled and interpreted by analysis algorithms.

1.2 Problem Statement

As demonstrated above, binary analysis is quite challenging due to the low-level nature of binary executables and numerous obstacles regarding function and CFG recovery. While significant research has gone into CFG and function recovery, dataflow extraction for x86 binaries seems to have made less progress. Having the ability to trace the dataflow of a register throughout a section of linear code provides value to an analyst and has applications to cybersecurity and software verification as mentioned above. The objective of this research is to extract register dataflow graphs from x86 binaries with a high degree of accuracy.

In order to maintain a reasonable scope, the research in this thesis assumes that functions, disassembly and CFGs have been correctly recovered prior to generating the dataflow graphs as these are substantially difficult problems on their own.

CHAPTER 2. REVIEW OF LITERATURE AND TOOLS

2.1 Introduction

Several papers have been written on the topic of CFG and function recovery in x86 binaries, but there appears to be a lack of work in the area of dataflow analysis. Due to the difficulty of the subject at hand, this is somewhat to be expected. Listed below are tools and papers related to the topic of binary dataflow analysis.

Several tools perform a variant of this thesis's objective by doing things like dynamic taint analysis, however the majority perform analysis dynamically which is likely due to the fact that program flow cannot be determined entirely from static analysis alone due to complications such as computed jumps. This research differentiates itself by remaining strictly in the static analysis domain. The benefit of static analysis is the entire space of the program can be explored with reasonable effort. Dynamic analysis has the substantial limitation of having to drive the target program to its various states in order to analyse it which can be quite difficult due to the explosion of paths in large programs and the challenge of deducing what input is required to unlock a new state. Another advantage of static analysis is the fact that the target binary code does not have to be run which is ideal for tasks such as analyzing malware.

2.2 Tools

2.2.1 Angr

One project, Angr [30], has done some work to construct data dependency graphs (DDGs) and value flow graphs (VFGs) from binaries, as well as do program slicing. They have applied their framework in several papers and projects such as a symbolic execution fuzzer [31] and DARPA's Cyber Grand Challenge. Unfortunately in my evaluation I found Angr to be quite unreliable and would crash often on trivial examples. Upon searching the internet for guidance, it became clear that many other people experienced the same issues.

2.2.2 Miasm

Another popular tool is Miasm which is a reverse engineering framework that translates binary code into an IR, performs symbolic execution and emulation, and has some brief applications to dataflow analysis [21].

2.2.3 Dyninst

Popular in both academia and industry, Dyninst is a framework for binary analysis and instrumentation [23]. The project is comprised of several smaller projects, such as InstructionAPI, SymTabAPI, PatchAPI, and ParseAPI, which combined allow them to decode binaries into a higher-level abstraction in order to perform analysis and dynamic patching. There is also a subproject called DataflowAPI but appeared to be rather limited from a dataflow analysis standpoint. The subproject provided APIs to analyze stack heights, do program slicing, and determine register liveness, but did not have a means of generating register dataflow graphs.

2.2.4 Avast Retdec

As mentioned in the IR section, it is common for binary analysis tools to ”lift” binary code into an IR. The LLVM IR [19] is quite popular due to the complexity it can support, along with the analysis and transform ”passes” that have already been written for the IR. These passes allow variable simplification, loop identification, memory dependence analysis, loop unrolling, and much more [20]. The Clang compiler transforms code, such as C, into LLVM IR which is then fed to LLVM which generates binary code. This means, in theory, if you are to ”lift” a binary into correct LLVM IR, you can then re-create binary code by passing it through LLVM.

Avast has created a tool called Retdec which is a decompiler for several architectures, emitting C and LLVM IR code from binary input files [4]. In addition to this, the Retdec toolset contains several utilities to extract information about binaries such as their CFG, ELF metadata, and C++ class hierarchies. Unfortunately during testing, it was realized that Retdec had about a 70-80% accuracy when decompiling binaries into valid LLVM code that could be recompiled. For complex binaries, it was found that even when the LLVM IR would compile back into a binary, the generated binary did not run correctly or with the same functionality as the lifted binary. This was disappointing as LLVM IR is a great platform to do code analysis on.

2.3 Papers

2.3.1 InputTracer

The paper presents a tool called InputTracer which uses dynamic taint analysis to analyze the dataflow of a program with respect to its inputs [16]. The authors state that dataflow analysis has typically been performed manually, underlining the need for a tool to automate this and make the task more manageable. While the paper focuses on a similar goal of dataflow analysis in x86 binaries, their strategy differs in the way they approach the problem by performing a dynamic analysis of binary code. This master's thesis focuses on leveraging static analysis which has the advantage of being able to model arbitrary regions of a binary as opposed to only the ones that are touched by inputs during dynamic taint analysis.

2.3.2 Static Analysis of x86 Executables

This lengthy PhD dissertation goes into detail of binary analysis techniques for the x86 architecture. Specifically, it presents a new approach of augmenting CFG recovery by leveraging static dataflow analysis concurrently to generate a more correct CFG with respect to computed jumps. As part of this, Johannes created a tool *Jakstab*, that incorporates several dataflow subtopics such as constant propagation, forward expression substitution, and live variable analysis.

2.4 Summary

The current state of research and industry with regard to x86 dataflow analysis leaves much to be desired, especially with regard to accuracy. While some tools come close to the objective of this thesis's research, none have been found to directly implement it.

CHAPTER 3. METHODS AND PROCEDURES

3.1 Introduction

This section will describe the environment and steps taken to extract dataflow graphs from x86 binaries.

3.1.1 Tools & Software Used

3.1.1.1 Radare

Radare is an open-source, reverse engineering framework that supports a large variety of CPU architectures [24]. The disassembly and CFG extraction engine is quite accurate, and the project has APIs for multiple languages making it an attractive tool to use for this research.

Prior to selecting Radare for this research, the CFGs generated by the tool were carefully evaluated to ensure that they were correct. To accomplish this, the XINU [33] operating system was compiled for x86, ARM and MIPS and the CFGs were generated for many of the functions. Then, using the program analysis platform Atlas [10], CFGs were generated for the C source code and compared to the CFGs generated by Radare, checking for consistency. In all cases, except for several jump tables on the MIPS XINU OS, the CFGs generated by Radare were correct.

3.1.1.2 Cutter

Cutter is an open-source GUI for the Radare project [32]. Since Radare is predominantly a text-based interface, visualization of CFGs is difficult so Cutter was chosen to display the graphs in a friendly format. Additionally, as part of this research a custom patch was made for Cutter to allow the user to select basic-blocks of code within the CFG to generate a program slice for dataflow analysis.

3.1.1.3 PyVEX

PyVEX is an open-source Python library that exposes APIs to generate VEX IR [29]. As mentioned earlier, the VEX IR is used for the program analysis of x86 code.

3.2 Process

In order to construct the dataflow graph for a section of binary code, the following steps are taken with the details outlined below.

1. Create a program slice of the binary that is linear in control flow
2. Translate the program slice into the VEX intermediate representation
3. Transform the IR into a graph for each operation and SSA assignment
4. Select a target register to generate the dataflow graph for
5. Build dataflow graph and simplify

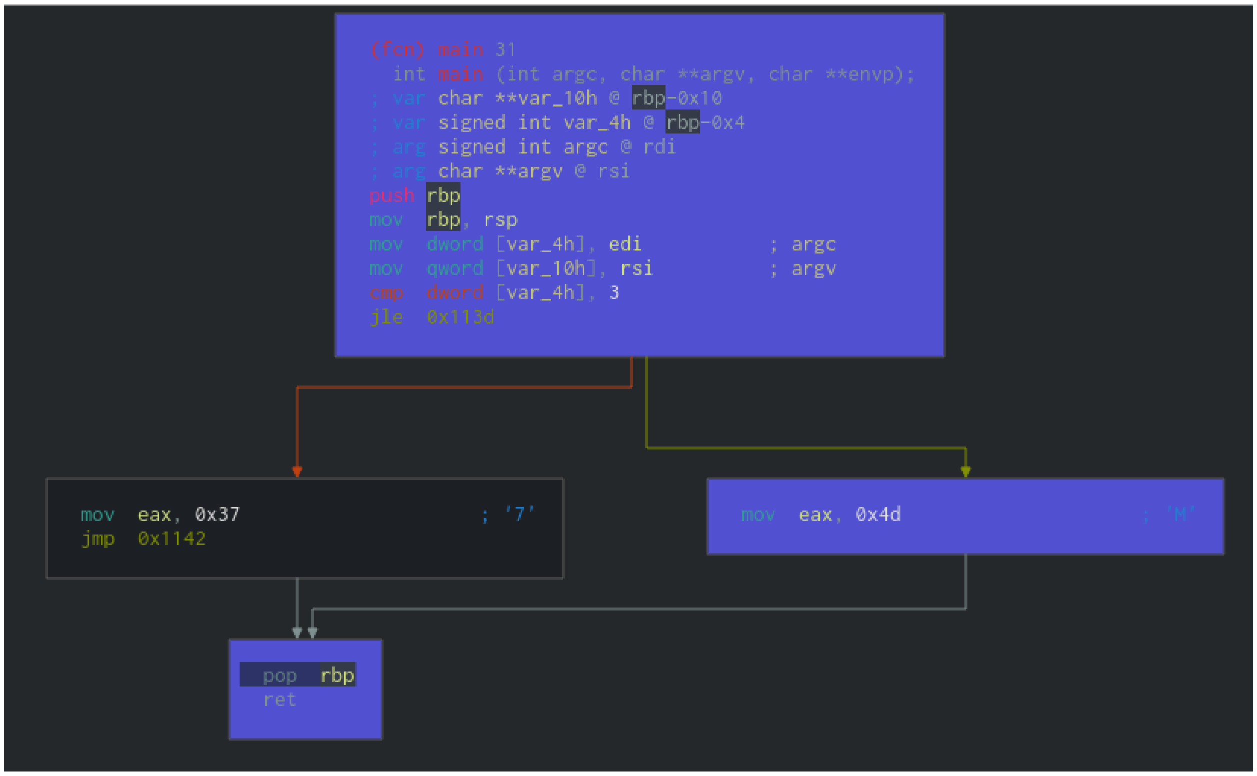
For the dataflow to be computed for a register, the control flow must be linear so that it can be expressed as an equation without having to worry about the added complexity of loops. To achieve this, the user must make a program slice of the binary using the Cutter GUI. A patch was created for Cutter to allow the user to select blocks which are then passed to the analysis engine, piecing the binary bytes together into a slice. In figure 3.1, the program has been sliced to follow the blocks highlighted in blue. This generates a slice containing the following assembly instructions:

```
push rbp
mov rbp, rsp
mov dword [var_4h], edi
mov qword [var_10h], rsi
cmp dword [var_4h], 3
```

```
mov eax, 0x4d

pop rbp

ret
```



```
(fcn) main 31
  int main (int argc, char **argv, char **envp);
; var char **var_10h @ rbp-0x10
; var signed int var_4h @ rbp-0x4
; arg signed int argc @ rdi
; arg char **argv @ rsi
push rbp
mov  rbp, rsp
mov  dword [var_4h], edi           ; argc
mov  qword [var_10h], rsi          ; argv
cmp  dword [var_4h], 3
jle  0x113d
```

```
mov  eax, 0x37                     ; '7'
jmp  0x1142
```

```
mov  eax, 0x4d                     ; 'M'
```

```
pop  rbp
ret
```

Figure 3.1: Cutter UI - Selecting a program slice

The program slice is then translated into the VEX IR which is iterated over to construct a graph for each of the SSA assignments. To demonstrate this, take the following linear control flow program as an example.

```
int foo(int arg1) {

    return arg1 + 0x16;

}
```

Which disassembles to:

```
mov dword [rsp - 4], edi    //Arg1 is in EDI per x86_64 Linux Calling Convention
mov eax, dword [rsp - 4]
add eax, 0x16
ret
```

After lifting the disassembly into the VEX IR and constructing a graph for each of the SSA assignments, we have the following "pool" of graphs as shown in Figure 3.2. Each node represents an operation, constant, or variable, and edges represent assignment or a dependency. For example the graph in the upper right would equate to the following operation: `t21 = Sub64(t7, 0x80)` which is an assignment of the 64-bit subtraction of 0x80 from t7, to t21.

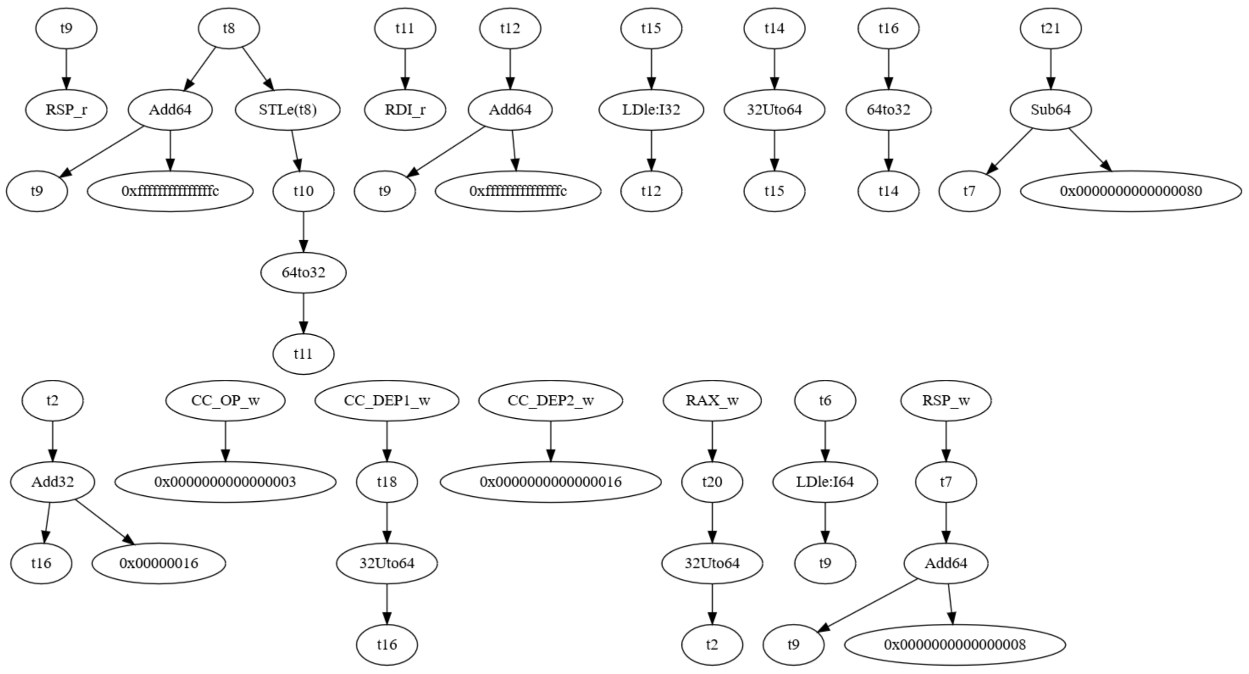


Figure 3.2: Graph "pool"

To construct a dataflow graph for a register, several graph algorithms are applied to recursively chain together the dependency graphs. First, variables that are assigned to equal statements are merged to reduce the size of the resulting graph. Next, a modified depth-first search is run over the graph pool, starting at the register of interest (RAX in this example). When the DFS algorithm encounters a leaf node, it checks if it is a variable and if so, searches the pool for a graph that defines it, connecting the two if found. This continues until all the "dependencies" have been satisfied and the resulting graph contains only the nodes and edges that are related to the target register. Finally, this graph is fed into a simplification pass that merges expressions such as `t1 = t2 = t3` and removes the temporary variables completely where possible. This algorithm is summarized in the pseudocode of Algorithm 1.

The graph in figure 3.3 depicts the dataflow for the RAX register in simplified format after the steps outlined above have been executed. Transforming the graph into equation format, we get the following:

```
t8 = Add64(RSP, 0xfffffffffffffffc)

*t8 = 64to32(RDI)

RAX = 32Uto64(Add32(64to32(32Uto64(*t8))), 0x16)
```

Which, when simplified, equates to `RAX = U32to64(Add32(64to32(32Uto64(RDI))), 0x16)`. This makes sense as, in x86, RAX is the return address register and RDI is the first parameter register. Comparing this to the C code this was derived from, we can observe they are functionally equivalent to adding 0x16 to the first parameter and returning the result.

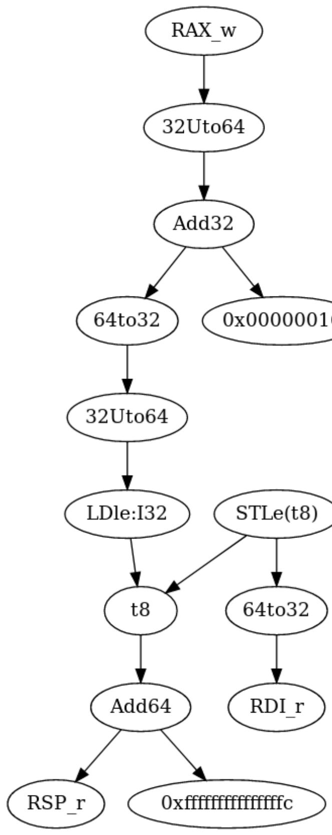


Figure 3.3: RAX Register Dependency Graph

Algorithm 1 Generate DFG for a register

Require:

G is a pool of graphs generated from the VEX IR

target_register is a valid x86 register

```
1:  function GenerateDFGraph($G$, $target\_register$)
2:      $G = MergeRedundantTmpVars(G)$
3:      $connected\_nodes = DFS(G, target\_register, [])$
4:      for each $N \in G$ do
5:          if $N \notin connected\_nodes$ then
6:              RemoveNode(G, N)
7:          end if
8:      end for
9:      $G = MergeRedundantNodes(G)$
10:     return $G$
11: end function
```

Algorithm 2 Support algorithms for GenerateDFGraph()

Require:
G is a pool of graphs generated from the VEX IR
V is a valid vertex in the graph G

```
function DFS($G$, $V$, $visited$)
    $visited = visited + V$
    if IsLeafNode($V$) then
        $defining\_subgraph = FindSubgraphWithRoot(G, V)$
        AddEdge(G, V, defining_subgraph)
        DFS(G, defining_subgraph, visited)
    end if
    for each $N \in Sucessors(G, V)$ do
        if $N \notin visited$ then
            DFS(G, N, visited)
        end if
    end for
    return $visited$
end function
```

```
function MergeRedundantTmpVars($G$)
    $redundant\_graphs = Set\{\}$
    for each $subgraph \in GetSubgraphs(G)$ do
        $labels = PreorderTraversal(subgraph)$
        for each $subgraph\_2 \in GetSubgraphs(G)$ do
            $labels\_2 = PreorderTraversal(subgraph\_2)$
            if $labels == labels\_2$ && $subgraph\_2 \neq subgraph$ then
                $redundant\_graphs = redundant\_graphs + [subgraph, subgraph\_2]$
            end if
        end for
    end for

    for each $redundant\_G \in redundant\_graphs$ do
        $ReplaceReferences(G, redundant\_G[0], redundant\_G[1])$
    end for
    return $G$
end function
```

CHAPTER 4. EVALUATION AND RESULTS

4.1 Introduction

In order to measure the success of the dataflow recovery method outlined in this thesis, it must be evaluated for correctness. This chapter will outline the steps taken to evaluate RegSym and present the results.

4.2 Evaluation

4.2.1 Z3 Theorem Prover

Z3 is a popular theorem prover by Microsoft that has bindings for several programming languages [9]. In order to assess the correctness of the register dataflow equations, we will use Z3 to check if the generated dataflow matches the source code by inputting both the equations and checking them for equality. Since the test cases are not too large, Z3 should be able to check the equality in a reasonable amount of time.

4.2.2 Testing Environment

The x86 dataflow generator code was run on a Ubuntu 19.04 virtual machine with Python 3.7.3 installed. To determine correctness, 20 test cases were created in C that performed arithmetic operations on parameters and ranged from simple addition to more complex equations. An example of one of the test cases is shown in listing 7. The dataflow recovery algorithm was configured to extract the dataflow of the input parameters with respect to the RAX register which holds the return value for the function.

4.2.3 Steps

The following steps were taken to evaluate RegSym:

1. Compile test cases into binary executables
2. Run dataflow graph generation algorithm
3. Input generated dataflow equation to Z3
4. Compare with source code equation

4.3 Results

Shown below in Table 4.1 are the results for the test cases. The "Correct" column indicates whether the dataflow graph was deemed equivalent to the source code equation by Z3. The function nomenclature roughly corresponds to the code in the test case. For example, `and_xor(int a)` is the bitwise *AND* with a constant, and the result *XORed* with another constant.

4.4 Example Vulnerability

In order to demonstrate an applied usage of RegSym, the following vulnerability will be analyzed. Below in Listing 6 is a gambling program that has a very poorly implemented random number generator, `proprietary_rng()`. In an attempt to obfuscate the implementation, garbage code has been added (all the non-highlighted lines) and the only code that is actually used is the last line `return (s/a)`. The generated assembly on the right is rather dense and it is not immediately clear that the rng function only makes use of the first two parameters, returning the division operation as the random number. We will leverage RegSym to reverse engineer the `proprietary_rng()` and crack it as well as exposing a DBZ (divide-by-zero) vulnerability.

Generating the register dataflow graph for the return register in the `proprietary_rng()` function provides us with the output in figure 4.1. From this, we can observe that the return value (RAX) only depends on two of the input parameters (RDI & RSI) which correspond to the first

Table 4.1: Function Dataflow Results

Function	**Correct? (Z3)**
add_const(int c)	y
ab_subtraction(int a, int b)	y
shift_add(int a)	y
unused_param_add(int a, int b, int c)	y
and_xor(int a)	y
div_add(int a)	y
mul_xor(int a, int b, int c)	y
shift_mul_add(int a)	y
or_shift(int a)	y
triple_add(int a, int b, int c)	y
triple_add_shift(int a, int b, int c)	y
triple_add_shift_param(int a, int b, int c, int d)	y
add_mod(int a, int mod)	y
or_sub(int a, int sub)	y
add_ptr(int *a)	y
mul_by_param(uint16_t a, uint8_t *multiplier)	n
float noop_ret_float(float a)	y
float mul_double_const(double a)	n
float add_double_const(double a)	n
float add_double(double a, double b)	n

two parameters. More importantly, we can observe that the `proprietary_rng()` function is only a division operation. With this knowledge we can outsmart the gambling program and win every time by providing dice rolls that match the rng output. In addition, since there is a division operation with two user-controlled parameters, a divide by zero vulnerability can be triggered by passing 0 as the second dice roll. Both of these actions are demonstrated in figure 4.2. While this information could have been deduced by manually inspecting the assembly, RegSym was successful in greatly simplifying the task and removing irrelevant assembly instructions. The usefulness of this tool becomes more apparent with larger and more complex functions.

Listing 6 Vulnerable Program

```c
#include <stdio.h>
#include <stdlib.h>

int proprietary_rng(int s, int a, int b, int c) {
        int d = a + b;
        int l = s / (a^b);
        l &= (d | l) + c;
        int x = (l | d)^(b&a)*37 + s;
        return (s/a);
}

int main(int argc, char **argv) {
        int guess1 = atoi(argv[1]);
        int guess2 = atoi(argv[2]);
        int lotteryDice = proprietary_rng(guess1,
                guess2, 4, 1042341);
        if (guess2 == lotteryDice)
                printf("You won the lottery!\n");
        else
                printf("Better luck next time!\n");
}
```

```
0000000000001145 <proprietary_rng>:
    1145:       push   rbp
    1146:       mov    rbp,rsp
    1149:       mov    DWORD PTR [rbp-0x14],edi
    114c:       mov    DWORD PTR [rbp-0x18],esi
    114f:       mov    DWORD PTR [rbp-0x1c],edx
    1152:       mov    DWORD PTR [rbp-0x20],ecx
    1155:       mov    edx,DWORD PTR [rbp-0x18]
    1158:       mov    eax,DWORD PTR [rbp-0x1c]
    115b:       add    eax,edx
    115d:       mov    DWORD PTR [rbp-0xc],eax
    1160:       mov    eax,DWORD PTR [rbp-0x18]
    1163:       xor    eax,DWORD PTR [rbp-0x1c]
    1166:       mov    esi,eax
    1168:       mov    eax,DWORD PTR [rbp-0x14]
    116b:       cdq
    116c:       idiv   esi
    116e:       mov    DWORD PTR [rbp-0x8],eax
    1171:       mov    eax,DWORD PTR [rbp-0xc]
    1174:       or     eax,DWORD PTR [rbp-0x8]
    1177:       mov    edx,eax
    1179:       mov    eax,DWORD PTR [rbp-0x20]
    117c:       add    eax,edx
    117e:       and    DWORD PTR [rbp-0x8],eax
    1181:       mov    eax,DWORD PTR [rbp-0x8]
    1184:       or     eax,DWORD PTR [rbp-0xc]
    1187:       mov    ecx,eax
    1189:       mov    eax,DWORD PTR [rbp-0x1c]
    118c:       and    eax,DWORD PTR [rbp-0x18]
    118f:       mov    edx,eax
    1191:       mov    eax,edx
    1193:       shl    eax,0x3
    1196:       add    eax,edx
    1198:       shl    eax,0x2
    119b:       add    edx,eax
    119d:       mov    eax,DWORD PTR [rbp-0x14]
    11a0:       add    eax,edx
    11a2:       xor    eax,ecx
    11a4:       mov    DWORD PTR [rbp-0x4],eax
    11a7:       mov    eax,DWORD PTR [rbp-0x14]
    11aa:       cdq
    11ab:       idiv   DWORD PTR [rbp-0x18]
    11ae:       pop    rbp
    11af:       ret
```

4.5 Conclusion

From the data above, it is apparent that the dataflow recovery algorithm has worked quite well, succeeding on 16 out of the 20 test cases. Where RegSym was incorrect was in functions that use floating point operations which was to be somewhat expected. Several of the dataflow equations differed significantly in their composition from the source code equations, but were functionally equivalent. This was interesting and can be described by compiler optimizations that were common during division and modulus operations.

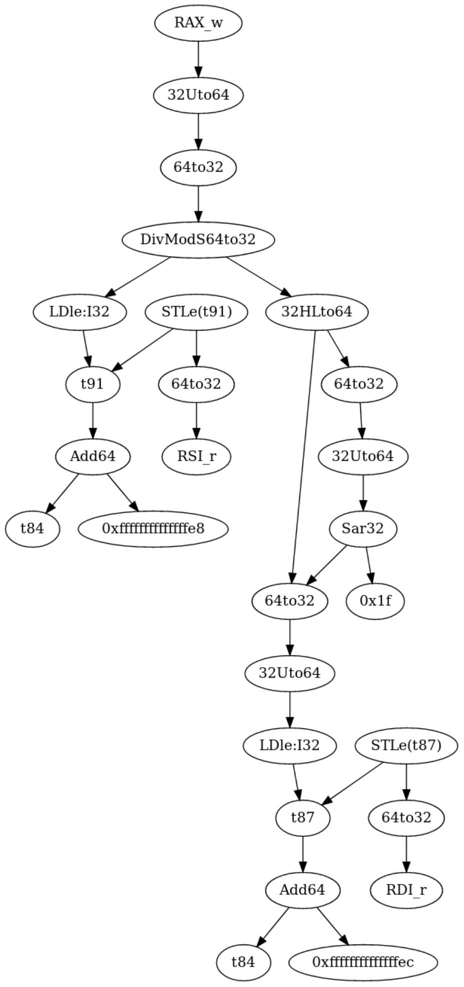
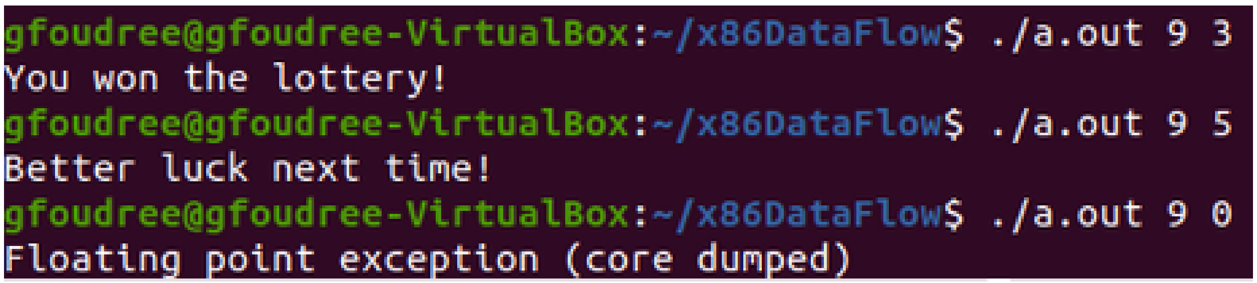


Figure 4.1: Generated Register Flow



```
gfoudree@gfoudree-VirtualBox:~/x86DataFlow$ ./a.out 9 3
You won the lottery!
gfoudree@gfoudree-VirtualBox:~/x86DataFlow$ ./a.out 9 5
Better luck next time!
gfoudree@gfoudree-VirtualBox:~/x86DataFlow$ ./a.out 9 0
Floating point exception (core dumped)
```

Figure 4.2: Reverse Engineered Gambling Program

CHAPTER 5. SUMMARY AND DISCUSSION

5.1 Summary

In conclusion of this research, a new technique to extract dataflow from x86 binaries has been presented and, through evaluation, has been demonstrated to be quite accurate. In addition to this, an overview of binary software analysis has been presented, with several of the common challenges described in detail.

5.2 Limitations

Two noteworthy limitations exist with RegSym. First, even though x86 is a CISC architecture, compiling C code generates significantly more lines of assembly when compared to the original source code. When analyzing a complex function or large chunk of code, many instructions are involved which leads to very verbose dataflow graphs. Even with the simplification passes added to RegSym, the generated graphs can be overwhelming to an analyst. As an example, the test case used below, generates the dataflow graph seen in figure 5.1.

Listing 7 Test Case

```
int triple_add_shift_param(int a, int b, int c, int d) {
        return (a + b + c) >> d;
}
```

In addition, while this is more of a compiler artifact outside of our control, strange things occasionally happen due to compiler optimizations where the generated assembly is quite different from the source code. For example, as mentioned above, during testing it was observed that division operations were changed into multiplication with a strange constant, and the result was fed through various bitwise operations. This might be confusing to an analyst expecting a 1:1 mapping between source and disassembly dataflow equations.

Finally, the last significant limitation is that the current tool only works on linear code and requires loops to be unrolled. This simplifies the overall dataflow graph, but adds complexity when selecting blocks to be analyzed.

5.3 Future Work

Several additions can be made to this research in future work. Improving the graph simplification algorithms would help reduce the size, providing significant value when analyzing large regions of code. To help with this, Z3 has a boolean logic simplification API that can be used to help condense the logic expressions output by Regsym.

In addition to this, porting RegSym to another architecture could be accomplished with reasonable difficulty due to the usage of the underlying VEX IR.

Finally, as mentioned in the limitations section, future work could be done to include control flow statements into the dataflow giving a more traditional dataflow graph, however this will cause a large expansion of the generated graphs and was omitted for this reason.

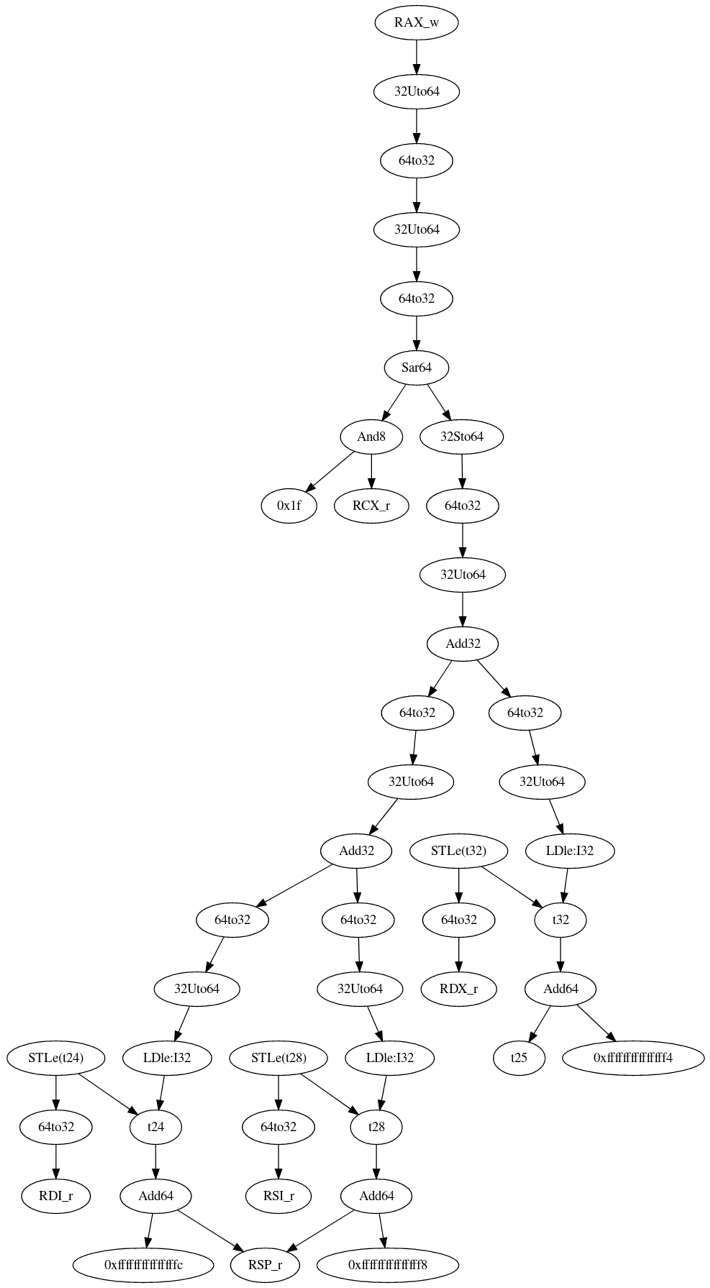


Figure 5.1: triple_add_shift_param test case graph

BIBLIOGRAPHY

[1] Pabsb - packed absolute value, May 2019. URL https://www.felixcloutier.com/x86/pabsb:pabsw:pabsd:pabsq.

[2] N. S. Agency. P-code operation reference, 2019. URL https://ghidra.re/courses/languages/html/pcodedescription.html.

[3] D. Andriesse, A. Slowinska, and H. Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 177–189, April 2017. doi: 10.1109/EuroSP.2017.11.

[4] Avast. Welcome to the retargetable decompiler's home page, 2019. URL https://retdec.com/.

[5] J. Bacon. Mips instruction code formats, 2011. URL http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch05s07.html.

[6] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. BYTEWEIGHT: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 845–860, San Diego, CA, 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bao.

[7] F. Cloutier. x86 and amd64 instruction reference, May 2019. URL https://www.felixcloutier.com/x86/.

[8] F. Cloutier. Vinsertf128/vinsertf32x4/vinsertf64x2/vinsertf32x8/vinsertf64x4 — insert packed floating-point values, May 2019. URL https://www.felixcloutier.com/x86/vinsertf128:vinsertf32x4:vinsertf64x2:vinsertf32x8:vinsertf64x4.

[9] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

[10] T. Deering, S. Kothari, J. Sauceda, and J. Mathews. Atlas: A new way to explore software, build analysis tools. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 588–591, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2768-8. doi: 10.1145/2591062.2591065. URL http://doi.acm.org/10.1145/2591062.2591065.

[11] domas. Breaking the x86 isa, Jul 2017. URL https://www.blackhat.com/docs/us-17/thursday/us-17-Domas-Breaking-The-x86-ISA.pdf.

[12] A. D. Federico, M. Payer, and G. Agosta. rev.ng: a unified binary analysis framework to recover cfgs and function boundaries. In *CC*, 2017.

[13] F. S. Foundation. objdump(1) - linux man page, 2009. URL https://linux.die.net/man/1/objdump.

[14] IBM. Power isa v2.07, May 2013. URL http://fileadmin.cs.lth.se/cs/education/EDAN25/PowerISA_V2.07_PUBLIC.pdf.

[15] Intel. System v application binary interfaceamd64 architecture processor supplement, Jul 2013. URL https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf.

[16] U. Kargén and N. Shahmehri. Inputtracer: A data-flow analysis tool for manual program comprehension of x86 binaries. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 138–143, Sep. 2012. doi: 10.1109/SCAM.2012.16.

[17] J. Kinder. PhD thesis, 2010.

[18] LLVM. Llvm language reference manual, 2019. URL https://llvm.org/docs/LangRef.html.

[19] LLVM. Llvm language reference manual, 2019. URL https://llvm.org/docs/LangRef.html.

[20] LLVM. Llvm's analysis and transform passes, 2019. URL https://llvm.org/docs/Passes.html.

[21] Miasm. Miasm's blog, 2019. URL https://miasm.re/blog/.

[22] M. Mitchell, A. Jaeger, and J. Hubi ̌cka. System v application binary interface, Jan 2005. URL https://refspecs.linuxfoundation.org/elf/x86_64-abi-0.95.pdf.

[23] U. of Wisconsin Madison. Putting the performance in high performance computing, 2019. URL https://dyninst.org/.

[24] pancake. Radare - unix-like reverse engineering framework and commandline toolsradare, 2019. URL https://rada.re/r/index.html.

[25] M. Prasad. Disassembly challenges, Mar 2005. URL https://www.usenix.org/legacy/publications/library/proceedings/usenix03/tech/full_papers/prasad/prasad_html/node5.html.

[26] R. Qiao. *Accurate Recovery of Functions in COTS Binaries*. PhD thesis, 2017.

[27] H. Rays. Ida: About, May 2015. URL https://www.hex-rays.com/products/ida/index.shtml.

[28] J. Salwan. Shell-storm: Taint analysis and pattern matching with pin, Aug 2013. URL http://shell-storm.org/blog/Taint-analysis-and-pattern-matching-with-Pin/.

[29] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. 2015.

[30] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[31] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. 2016.

[32] C. Team and I. Cohen. Cutter - free and open source re platform powered by radare2, Sep 2019. URL https://cutter.re/.

[33] P. University. The xinu page, 2019. URL https://xinu.cs.purdue.edu/.

[34] Valgrind. Valgrind, 2019. URL http://www.valgrind.org/docs/manual/writing-tools.html.