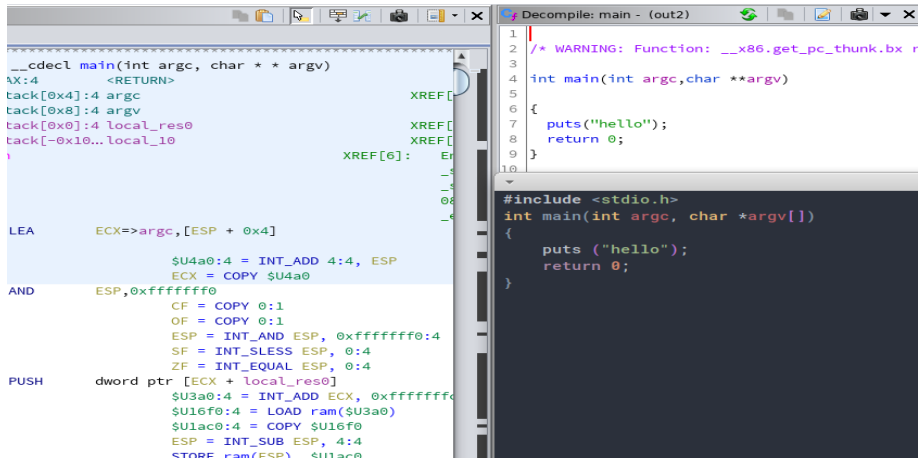


# Ghidra's SLEIGH + disassembly tricks + misc.

Joe Staursky

10-25-2019

# NSA's SLEIGH - Ghidra's disassembler and decompiler



The screenshot displays the Ghidra IDE with the decompiler window open. The left pane shows the assembly code, and the right pane shows the decompiled C code. The decompiled code is a C function named 'main' that takes two arguments, 'argc' and 'argv', and returns an integer. The function body contains a single statement: 'puts("hello");'. The decompiler window title is 'Decompile: main - (out2)'. The assembly code on the left is in SLEIGH format, showing the function signature 'main(int argc, char \* \* argv)' and various instructions like 'LEA ECX=>argc, [ESP + 0x4]', 'CF = COPY 0:1', 'OF = COPY 0:1', 'ESP = INT\_AND ESP, 0xffffffff0:4', 'SF = INT\_SLESS ESP, 0:4', 'ZF = INT\_EQUAL ESP, 0:4', 'PUSH dword ptr [ECX + local\_res0]', '\$U3a0:4 = INT\_ADD ECX, 0xffffffff0', '\$U16f0:4 = LOAD ram(\$U3a0)', '\$U1ac0:4 = COPY \$U16f0', 'ESP = INT\_SUB ESP, 4:4', and 'STORE ram(ESP). \$U1ac0'.

```
Decompile: main - (out2)
1
2 /* WARNING: Function: __x86.get_pc_thunk.bx r
3
4 int main(int argc, char **argv)
5
6 {
7     puts("hello");
8     return 0;
9 }
10

#include <stdio.h>
int main(int argc, char *argv[])
{
    puts ("hello");
    return 0;
}
```

Figure: Accuracy in Ghidra decompiler

# SLEIGH - Modeling Processor Behavior

## What

- SLEIGH is a processor **specification language**. Its primary purpose is to serve as an agnostic intermediary to translate – between machine code and human readable assembly (p-code) – in a platform independent way.

## How

- SLEIGH code describes an abstract machine where p-code executes. It is up to the SLEIGH coder to ensure the p-code model of execution translates accurately with the processor instruction set being modeled.

## Why

- SLEIGH code in and of itself is not all that interesting, its importance derives from its use as a general framework around which binary analysis tools can be made.

# SLEIGH CPU MODEL - Quick Overview

While SLEIGH is simply the **means** used to move to a simpler analysis in p-code; it is by virtue of this fact that since our abstraction is built from sleigh code we need to be familiar with some of its constructs.

You cannot understand p-code if you don't understand the sleigh model under which it runs.

## Consists of 3 components / abstractions

- 1 Address Space  $\approx$  RAM
- 2 Varnodes  $\approx$  registers
- 3 Operations (p-code)  $\approx$  machine instructions

# Address Space

An indexed sequence of memory "words" that can be read and written to by p-code. Offsets within a space are mapped to different aspects of the physical CPU being modeled. e.g., EAX and EBP may both be apart of the REGISTER space, but they are defined at different offsets within this space.

## Size

The size of an address space is given by the number of bytes required to encode an arbitrary index into the address space. e.g., a space of size 4 requires a 32 bit integer to specify all indices and contains  $2^{32}$  bytes.

## CPU relation

Which offset correspond with which CPU definition can be found in the ".sla" file that is compiled from the ".slaspec" file.

## Spaces can be

- RAM SPACE - r/w, addressable
- ROM SPACE - r
- REGISTER SPACE - Non-addressable

# SLEIGH CPU MODEL (Varnodes)

Varnodes are the units of data manipulated by p-code. A varnode is a contiguous sequence of bytes in some address space and are described by the following tuple. (address-space-name, offset, size)

## Properties

- typeless, the contents are interpreted according to the p-code operation being performed on the varnode. Note that interpretations only apply during their respective operations. A different operation can interpret the same varnode in a different way. Consistency in meaning assigned to a varnode is something that must be provided and enforced by the specification designer.
- arbitrary size.
- can overlap with one another.
- numerous, any number of them may be defined.

Intended to emulate target processors instruction set.

## A one to many relation

Often a single machine instruction from the target processor is associated with multiple p-code operations.

## Properties

- p-code operations operate on one or more varnodes as input and can optionally have one varnode as output.
- All data is manipulated explicitly. Instructions have no indirect effects (no side-effects).

I have successfully compiled a disassembler that uses the ghidra's SLEIGH as it's backend. This was a bit of a headache because of the instructions inside ghidra's codebase to do this were often incomplete. I had to infer alot of things and outright guess sometimes, but I eventually was able to build sleigh as a reusable library.

## CAVEATS

Right now I am able to choose disassembly output in either x86 or p-code. However, you need to provide the corresponding `.text` section of your binary if you want meaningful disassembly. Integrating a library such `libbfd` to extract this section for you isn't hard, I just found piping raw binary to the input to be a bit more flexible.



# Anti Disassembly

## Comparing various disassembly libraries

In what follows I compare the results from 3 disassembly libraries: sleigh, capstone and zydis.

## Example

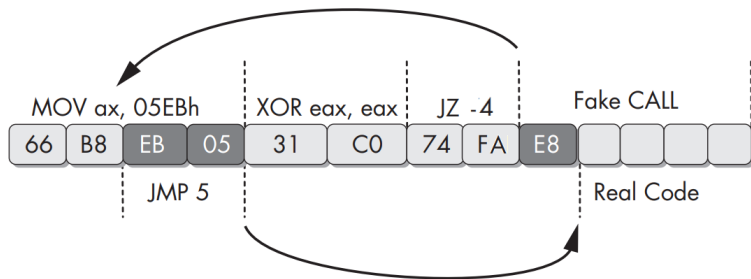


Figure: Thwarting linear disassemblers

# Case study

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

// gcc -m32 trick-disassembler.c -o trick

void message (void)
{
    printf ("\nNow printing from inside message function.\n\n");
    return;
}

int main (int argc, char** argv)
{
    asm(".intel_syntax noprefix");
    asm volatile(".byte 0x66, 0xb8, 0xeb, 0x05, 0x31, 0xc0, 0x74, 0xfa, 0xe8, "
                "0xff, 0xc0, 0x67, 0x48");
    asm(".att_syntax prefix");
    message ();
    return EXIT_SUCCESS;
}
```

# Ghidra's SLEIGH RESULTS

Recall the limitation of the program I built against the SLEIGH library. A commandline trick I use to get disassembly of just the `.text` section is the following.

```
./sleigh_disassembler =(()){objcopy -O binary -j .text tester $1; cat $1} =()) pcode
```

- `=(())` is `zsh`'s form of process substitution. This is a way to store the output of a command in a temporary file that `zsh` will clean up afterwards. Normal process substitution via `<()` is not sufficient when using `objcopy`.
- `(){...$1; cat $1}` `arg` is `zsh`'s anonymous function syntax. This extends the lifetime of the `arg` which in this case is just `=(())` (an empty file).

## sleigh disassembler results

I won't show the full output that `./sleigh_disassembler` yields as it is incorrect. But I will show the p-code at the spot where it was tricked to show some p-code syntax.

# Sleigh disassembler

```
./sleigh_program =(echo -n $'\x66\xb8\xeb\x05\x31\xc0\x74\xfa\xe8\xff\xc0\x67\x48') pcode  
--- 0x00000000: MOV AX,0x5eb  
(register,0x0,2) = COPY (const,0x5eb,2)  
--- 0x00000004: XOR EAX,EAX  
(register,0x200,1) = COPY (const,0x0,1)  
(register,0x20b,1) = COPY (const,0x0,1)  
(register,0x0,4) = INT_XOR (register,0x0,4) (register,0x0,4)  
(register,0x207,1) = INT_SLESS (register,0x0,4) (const,0x0,4)  
(register,0x206,1) = INT_EQUAL (register,0x0,4) (const,0x0,4)  
--- 0x00000006: JZ 0x2  
CBRANCH (ram,0x2,4) (register,0x206,1)  
--- 0x00000008: CALL 0x4867c10c <--- **FAIL**  
(register,0x10,4) = INT_SUB (register,0x10,4) (const,0x4,4)  
STORE (const,0x55c7ce095c40,8) (register,0x10,4) (const,0xd,4)  
CALL (ram,0x4867c10c,4)
```

## Observations

Notice how p-code operates on the varnode expressions which are given by (address-space-name, offset, size). It should be apparent the assignment of values between varnodes using the **const** address-space's **offset** value.

# Other disassembly libraries

## Capstone

I like this one, but for x86 Zydys is better.

```
./cap_dis =(echo -n $'\x66\xB8\xEB\x05\x31\xC0\x74\xFA\xE8\xFF\xC0\x67\x48')
```

```
0x0:  mov     ax,          0x5eb
```

```
0x4:  xor     eax,          eax
```

```
0x6:  je      2
```

```
0x8:  call   0x4867c10c
```

## Zydis

Gives more accurate disassembly. When pitted against the anti-disassembly,

```
./zydis_dis =(echo -n $'\x66\xB8\xEB\x05\x31\xC0\x74\xFA\xE8\xFF\xC0\x67\x48')
```

```
007FFFFFFFF400000  mov ax, 0x5EB
```

```
007FFFFFFFF400004  xor eax, eax
```

```
007FFFFFFFF400006  jz 0xFF400002
```

Still incomplete however.

# Correct execution demo using radare2

The original disassembly trick I found online actually contained a mistake and I was able to figure this out by stepping through an emulation of the assembly using radare2.

```
r2 -a x86 -b 32 =(echo -n $'\x66\x68\xeb\x05\x31\xc0\x74\xfa\xe8\xff\xc0\x67\x48')
```

*queue demo*

Linear disassemblers will always run into trouble against code that uses flow control tricks like the one illustrated in this presentation. The alternative is to implement a recursive disassembler that has some understanding of flow control. A recursive disassembler shouldn't fail against the trick I showed here (unconditional jump to the middle of an instruction).

## A semantics / solver based disassembler?

It might be interesting to implement a disassembler that has inference mechanisms. My main interest in the sleigh library originated from my interest in programming language semantics and as you could see p-code translates assembly in a way that removes side effects. This should make the description of a formal semantics in lambda calculus easier and from there more interesting work will likely follow.

NSA Ghidra's Project Page

<https://www.nsa.gov/resources/everyone/ghidra/>

Ghidra's Github Page

<https://github.com/NationalSecurityAgency/ghidra>

Online Documentation For Sleigh

<https://ghidra-decompiler-docs.netlify.com/sleigh.html>