Program Analysis With Ghidra

Alexei Bulazel

@0xAlexei

github.com/0xAlexei/Publications/tree/master/Ghidra github.com/0xAlexei/Publications/tree/master/Ghidra

Outline

1. Scripting With Ghidra
2. Program Analysis
3. P-Code
4. Example
5. SLEIGH
6. Discussion
7. Conclusion

Scripting With Ghidra

- Available in Java (natively) and Python (via Jython)
- Can be run with interactive GUI or in headless mode
- Ghidra comes with 230+ scripts pre-installed
  - Educational examples
  - Code patching
  - Import / export
  - Analysis enhancements
    - Windows, Mac, Linux, VXWorks
    - PE, ELF, Mach-O, COFF
    - x86, MIPS, ARM/THUMB, 8051, etc…

Ghidra APIs

FlatProgramAPI

- Simple “flattened” API for Ghidra scripting
- Programmatic access to common tasks
    - query / modify / iterate / create / delete - functions / data / instructions / comments
- Mostly doesn’t require the use of Java objects
- Stable

Ghidra Program API

- More complex rich API for deeper scripting
- Object-oriented (Program, Memory, Function, Instruction, etc…)
- Utility functions help with common scripting tasks
- UI scripting / interactivity
- Prone to change between versions

API Highlights

Rich Scripting Interface

- Programmatic access to binary file formats
- P-code interaction
- Decompiler API
- C header parsing
- Interface for graphing (implementation not included)
- Cyclomatic complexity

Common Utilities Included

- UI windows
- Assembly
- Data serialization
- String manipulation
- Hashing
- Search / byte matching
- XML utilities

Outline

1. Scripting With Ghidra
2. Program Analysis
3. P-Code
4. Example
5. SLEIGH
6. Discussion
7. Conclusion

Program Analysis

Wikipedia:

"In computer science, program analysis[1] is the process of **automatically analyzing the behavior of computer programs** regarding a property such as correctness, robustness, safety and liveness. Program analysis focuses on two major areas: program optimization and program correctness. The first focuses on improving the program's performance while reducing the resource usage while the latter focuses on ensuring that the program does what it is supposed to do.

Program analysis can be performed without executing the program (static program analysis), during runtime (dynamic program analysis) or in a combination of both."

Why Do Program Analysis?

Strengths

- Automation
- Complexity
- Scale
- Repeatability

Weaknesses

- Brittleness
- Difficulty to develop
- Computational requirements
- Might be easier to do manually
  - Cost of automating > cost of just doing

Data Flow

```
add eax, ebx
```

Data Flow

```
xor eax, eax

        add eax, ebx
```

Data Flow

```
xor eax, eax                sub ebx, ecx

            add eax, ebx
```

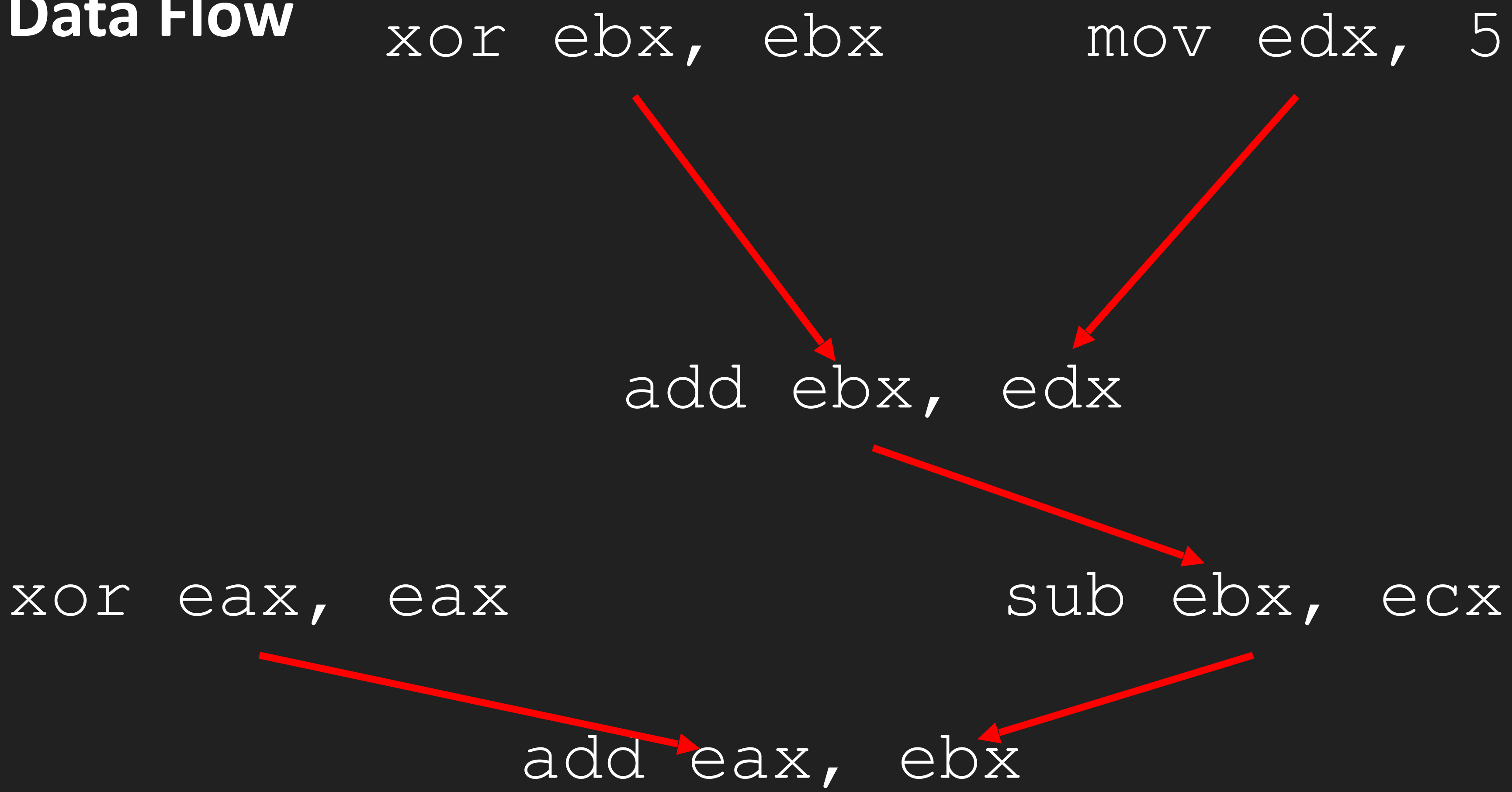
Data Flow

```
                add ebx, edx

xor eax, eax                        sub ebx, ecx

              add eax, ebx
```

Data Flow

```
xor ebx, ebx
add ebx, edx
xor eax, eax
sub ebx, ecx
add eax, ebx
```

Data Flow


xor ebx, ebx
mov edx, 5
add ebx, edx
xor eax, eax
sub ebx, ecx
add eax, ebx

Data Flow

```
xor ebx, ebx          mov edx, 5
mov ecx, dword [esp+12]
add ebx, edx
xor eax, eax          sub ebx, ecx
add eax, ebx
```

SSA Form

- Property of an intermediate representation that means:
  - Every variable assigned once and only once
  - Every variable defined before use
- Code in SSA form is much easier to reason about for compilers and program analysis tools
- *Phi*-nodes represent unification of different potential values at a given point

- Ghidra uses SSA form during decompiler analysis, and allegedly exposes it by API, but I haven’t figured out how to get it

Regular Code vs SSA Form

```
foo(){
	x = 5;
	print(x);
	x = 10;
	print(x);
	x = x + 1;
	print(x)
}
```

```
foo(){
	x1 = 5;
	print(x1);
	x2 = 10;
	print(x2);
	x3 = x2 + 1;
	print(x3)
}
```

Phi Nodes

```
int foo(){
    x = 0;
        if (rand() >
10){
        x = 20;
    } else{
        x = 30;
    }
    return x
}
```

```
int foo(){
    x1 = 0;
        if (rand() >
10){
        x2 = 20;
    } else{
        x3 = 30;
    }
    x4 = ϕ(x2,x3);
    return x4
}
```

Outline

1. Scripting With Ghidra
2. Program Analysis
3. P-Code
4. Example
5. SLEIGH
6. Discussion
7. Conclusion

P-Code

- Ghidra’s intermediate language
- Code for different processors can be lifted into p-code, data-flow analysis and decompilation can then run over the p-code
- Pseudo-assembly, represents lifted instructions as small atomic operations without side-effects
  - Built-in floating point support
- Can be emulated too - there are some blog posts on the topic

```
MOVSXD      RAX,EDX
                    (register, 0x0, 8) = INT_SEXT (register, 0x10, 4)
LEA         RAX,[RAX + RAX*0x4]
                    (unique, 0x660, 8) = INT_MULT (register, 0x0, 8), (const, 0x4, 8)
                    (unique, 0x680, 8) = INT_ADD (register, 0x0, 8), (unique, 0x660, 8)
                    (register, 0x0, 8) = COPY (unique, 0x680, 8)
```

P-Code Design

- The language is machine independent.
- The language is designed to model general purpose processors.
- Instructions operate on user defined registers and address spaces.
- All data is manipulated explicitly. Instructions have no indirect effects.
- Individual p-code operations mirror typical processor tasks and concepts.

Quoted from `docs/languages/html/sleigh.html`

Processor to p-code modeling:

- RAM → *address space*
- Register → *varnode*
- Instruction → *operation*

```
SCASB.REPNE   RDI
                    $U22d0:1 = INT_EQUAL RCX, 0:8
                    CBRANCH *[ram]0x401548:8, $U22d0
                    RCX = INT_SUB RCX, 1:8
                    $U1d90:8 = COPY RDI
                    $U1da0:8 = INT_ADD RDI, 1:8
                    $U1db0:8 = INT_ZEXT DF
                    $U1dc0:8 = INT_MULT 2:8, $U1db0
                    RDI = INT_SUB $U1da0, $U1dc0
                    $U1de0:1 = LOAD ram($U1d90)
                    CF = INT_LESS AL, $U1de0
                    $U1de0:1 = LOAD ram($U1d90)
                    OF = INT_SBORROW AL, $U1de0
                    $U1de0:1 = LOAD ram($U1d90)
                    $Uac60:1 = INT_SUB AL, $U1de0
                    SF = INT_SLESS $Uac60, 0:1
                    ZF = INT_EQUAL $Uac60, 0:1
                    $U22f0:1 = BOOL_NEGATE ZF
                    CBRANCH *[ram]0x401546:8, $U22f0
NOT           RCX
                    RCX = INT_NEGATE RCX
```

P-Code Decompilation / Analysis

- "Raw p-code" = direct translation of one CPU instructions to p-code ops
- During decompilation, p-code is analyzed, and may be modified
  - Insertion of `MULTIEQUAL` instructions (SSA phi-nodes)
  - Association of parameters with `CALL` ops and return values with `RETURN` ops
  - Construction of abstract syntax tree
  - etc… - see linked documents
- The Decompiler is a C++ binary that runs on the host system

- When writing scripts interacting with p-code expect to experiment, read source code, and glean usage from example included scripts

`docs/languages/html/additionalpcode.html`
`Ghidra/Features/Decompiler/src/decompile/cpp/docmain.hh`

Category	P-Code Operations
Data Moving	COPY, LOAD, STORE
Arithmetic	INT_ADD, INT_SUB, INT_CARRY, INT_SCARRY, INT_SBORROW, INT_2COMP, INT_MULT, INT_DIV, INT_SDIV, INT_REM, INT_SREM
Logical	INT_NEGATE, INT_XOR, INT_AND, INT_OR, INT_LEFT, INT_RIGHT, INT_SRIGHT
Int Comparison	INT_EQUAL, INT_NOTEQUAL, INT_SLESS, INT_SLESSEQUAL, INT_LESS, INT_LESSEQUAL
Boolean	BOOL_NEGATE, BOOL_XOR, BOOL_AND, BOOL_OR
Floating Point	FLOAT_ADD, FLOAT_SUB, FLOAT_MULT, FLOAT_DIV, FLOAT_NEG, FLOAT_ABS, FLOAT_SQRT, FLOAT_NAN
FP Compare	FLOAT_EQUAL, FLOAT_NOTEQUAL, FLOAT_LESS, FLOAT_LESSEQUAL
FP Conversion	INT2FLOAT, FLOAT2FLOAT, TRUNC, CEIL, FLOOR, ROUND
Branching	BRANCH, CBRANCH, BRANCHIND, CALL, CALLIND, RETURN
Extension / Truncation	INT_ZEXT, INT_SEXT, PIECE, SUBPIECE

Outline

1. Scripting With Ghidra
2. Program Analysis
3. P-Code
4. Example
5. SLEIGH
6. Discussion
7. Conclusion

What makes calls to `malloc()` interesting for vulnerability research?

- User controlled sizes, particularly when multiplied before use
  - `x = user_input(); malloc(5*x)`
- Size `0` allocations - particularly on old kernels/embedded…
  - See J. Vanegue, WOOT 2010 - “Zero-sized heap allocations vulnerability analysis”
- Known sites for allocations of static sizes
  - Useful for heap grooming / feng shui

`github.com/0xAlexei/Publications/tree/master/Ghidra`

DEMO: Source-Sink Analysis

- This is a very simple *proof-of-concept* script
- Use Ghidra p-code and the decompiler's analysis to identify the sources for values passed to function calls of interest (`malloc`), particularly function calls accepting user input
- *Solving* for the actual arguments requires a *solver*, this is a much simpler analysis that can empower a human analyst to hone in on interesting calls

- Start at the varnode for each argument to `malloc`, then trace back to the p-code operation that it's derived from
  - From there, recursively trace back the p-code operation(s) defining the varnode(s) that define that the inputs to those operations
- At function call sites, trace in, and find how the returned values are derived
- When a parameter is used, trace back to call sites which set the parameter

An Algorithm For Finding Interesting `malloc()`s

For each function…

1. For each `PcodeOp.CALL` to `malloc` in that function…
2. `AnalyzeVariableSource(CALL parameter)`

`AnalyzeVariableSource(variable):`

1. If `variable` is constant
   a. Return `variable`
2. If `variable` is derived from other variables:
   a. For each dependency:
      i. `AnalyzeVariableSource(dependency)`
3. If variable is the return value of a `PcodeOp.CALL`:
   a. For each `PcodeOp.RET` site in called function
      i. `AnalyzeVariableSource(RET value)`
4. If `variable` is a parameter:
   a. For each `PcodeOp.CALL` site of the current function:
      i. `AnalyzeVariableSource(parameter index of variable)`
5. If variable comes from memory:
   a. Unhandled

Future Improvements - Class Project Ideas

- Support a memory model (loads from memory, pointer dereferences)
- Model syscalls
- Make analysis *context sensitive*
- Cache analyzed operations along the way
- Use an SMT solver to crunch numbers and reduce conditions
- Loop support
- Better visualize data flow with a graph rather than text output
- Capture and highlight calls of particular interest

Outline

1. Scripting With Ghidra
2. Program Analysis
3. P-Code
4. Example
5. SLEIGH
6. Discussion
7. Conclusion

SLEIGH

- Ghidra's language for describing instruction sets to facilitate RE
- **Disassembly:** translate bit-encoded machine instructions into human-readable assembly language statements
- **Semantics:** translate machine instructions into p-code instructions (one-to-many) for decompilation, analysis, and emulation
- Based off of SLED (Specification Language for Encoding and Decoding), a 1997 academic IL

SLEIGH Example - x86 `JMP rel8`

Raw bytes: `0xEB 0x03`

x86 instruction: `JMP $+5`

SLEIGH Example - x86 `JMP rel8`

Raw bytes: `0xEB 0x03`

x86 instruction: `JMP $+5`

```
00401f16 eb 03          JMP          LAB_00401f1b
                                          BRANCH *[ram]0x401f1b:8
```

SLEIGH Example - x86 `JMP rel8`

Raw bytes: `0xEB 0x03`

x86 instruction: `JMP $+5`

```
00401f16 eb 03          JMP        LAB_00401f1b
                                       BRANCH *[ram]0x401f1b:8
```

SLEIGH Example - x86 `JMP rel8`

Raw bytes: `0xEB 0x03`

x86 instruction: `JMP $+5`

SLEIGH:

```
:JMP rel8 is vexMode=0 & byte=0xeb; rel8 {
    goto rel8;
}
```

```
00401f16 eb 03          JMP        LAB_00401f1b
                                   BRANCH *[ram]0x401f1b:8
```

SLEIGH Example - x86 `JMP rel8`

Raw bytes: `0xEB 0x03`

x86 instruction: `JMP $+5`

```
rel8: reloc is simm8 [ reloc=inst_next+simm8; ] {
    export *[ram]:$(SIZE) reloc;
}
```

SLEIGH:

```
:JMP rel8 is vexMode=0 & byte=0xeb; rel8 {
    goto rel8;
}
```

```
00401f16 eb 03          JMP        LAB_00401f1b
                                        BRANCH *[ram]0x401f1b:8
```

SLEIGH Example - x86 JMP rel8

Raw bytes: 0xEB 0x03

x86 instruction: JMP $+5

```
rel8: reloc is simm8 [ reloc=inst_next+simm8; ] {
    export *[ram]:$(SIZE) reloc;
}
```

SLEIGH:

```
:JMP rel8 is vexMode=0 & byte=0xeb; rel8 {
    goto rel8;
}
```

```
00401f16 eb 03          JMP          LAB_00401f1b
                                     BRANCH *[ram]0x401f1b:8
```

SLEIGH Example - x86 `JMP rel8`

Raw bytes: `0xEB 0x03`

x86 instruction: `JMP $+5`

```
rel8: reloc is simm8 [ reloc=inst_next+simm8; ] {
    export *[ram]:$(SIZE) reloc;
}
```

SLEIGH:

```
:JMP rel8 is vexMode=0 & byte=0xeb; rel8 {
    goto rel8;
}
```

```
00401f16 eb 03          JMP         LAB_00401f1b
                                    BRANCH *[ram]0x401f1b:8
```

SLEIGH Example - x86 `JMP rel8`

Raw bytes: `0xEB 0x03`

x86 instruction: `JMP $+5`

```
rel8: reloc is simm8 [ reloc=inst_next+simm8; ] {
    export *[ram]:$(SIZE) reloc;
}
```

SLEIGH:

```
:JMP rel8 is vexMode=0 & byte=0xeb; rel8 {
    goto rel8;
}
```

```
00401f16 eb 03          JMP          LAB_00401f1b
                                      BRANCH *[ram]0x401f1b:8
```

SLEIGH Example - x86 `XOR AL, imm8`

Raw bytes: `0x34 0x57`

x86 instruction: `XOR AL, 0x57`

SLEIGH Example - x86 `XOR AL, imm8`

Raw bytes: `0x34 0x57`

x86 instruction: `XOR AL, 0x57`

```
34 57        XOR      AL,0x57
                        CF = COPY 0:1
                        OF = COPY 0:1
                        AL = INT_XOR AL, 0x57:1
                        SF = INT_SLESS AL, 0:1
                        ZF = INT_EQUAL AL, 0:1
```

SLEIGH Example - x86 `XOR AL, imm8`

Raw bytes: `0x34 0x57`

x86 instruction: `XOR AL, 0x57`

```
34 57        XOR      AL,0x57
                        CF = COPY 0:1
                        OF = COPY 0:1
                        AL = INT_XOR AL, 0x57:1
                        SF = INT_SLESS AL, 0:1
                        ZF = INT_EQUAL AL, 0:1
```

SLEIGH Example - x86 `XOR AL, imm8`

Raw bytes: `0x34 0x57`

x86 instruction: `XOR AL, 0x57`

SLEIGH:

```
:XOR AL,imm8 is vexMode=0 & byte=0x34; AL & imm8 {
    logicalflags();
    AL = AL ^ imm8;
    resultflags( AL );
}
```

```
34 57        XOR        AL,0x57
                          CF = COPY 0:1
                          OF = COPY 0:1
                          AL = INT_XOR AL, 0x57:1
                          SF = INT_SLESS AL, 0:1
                          ZF = INT_EQUAL AL, 0:1
```

SLEIGH Example - x86 `XOR AL, imm8`

Raw bytes: `0x34 0x57`

x86 instruction: `XOR AL, 0x57`

```
macro logicalflags() {
 CF = 0;
 OF = 0;
}
```

SLEIGH:

```
:XOR AL,imm8 is vexMode=0 & byte=0x34; AL & imm8 {
    logicalflags();
    AL = AL ^ imm8;
    resultflags( AL );
}
```

```
macro resultflags(result) {
 SF = result s < 0;
 ZF = result == 0;
 # PF, AF not implemented
}
```

```
34 57        XOR      AL,0x57
                      CF = COPY 0:1
                      OF = COPY 0:1
                      AL = INT_XOR AL, 0x57:1
                      SF = INT_SLESS AL, 0:1
                      ZF = INT_EQUAL AL, 0:1
```

SLEIGH Example - x86 `XOR AL, imm8`

Raw bytes: `0x34 0x57`

x86 instruction: `XOR AL, 0x57`

```
macro logicalflags() {
 CF = 0;
 OF = 0;
}
```

SLEIGH:

```
:XOR AL,imm8 is vexMode=0 & byte=0x34; AL & imm8 {
    logicalflags();
    AL = AL ^ imm8;
    resultflags( AL );
}
```

```
macro resultflags(result) {
 SF = result s < 0;
 ZF = result == 0;
 # PF, AF not implemented
}
```

```
34 57        XOR        AL,0x57
                        CF = COPY 0:1
                        OF = COPY 0:1
                        AL = INT_XOR AL, 0x57:1
                        SF = INT_SLESS AL, 0:1
                        ZF = INT_EQUAL AL, 0:1
```

SLEIGH Example - x86 `XOR AL, imm8`

Raw bytes: `0x34 0x57`

x86 instruction: `XOR AL, 0x57`

```
macro logicalflags() {
 CF = 0;
 OF = 0;
}
```

SLEIGH:

```
:XOR AL,imm8 is vexMode=0 & byte=0x34; AL & imm8 {
    logicalflags();
    AL = AL ^ imm8;
    resultflags( AL );
}
```

```
macro resultflags(result) {
 SF = result s < 0;
 ZF = result == 0;
 # PF, AF not implemented
}
```

```
34 57        XOR     AL,0x57
                     CF = COPY 0:1
                     OF = COPY 0:1
                     AL = INT_XOR AL, 0x57:1
                     SF = INT_SLESS AL, 0:1
                     ZF = INT_EQUAL AL, 0:1
```

SLEIGH Example - x86 `XOR AL, imm8`

Raw bytes: `0x34 0x57`

x86 instruction: `XOR AL, 0x57`

```
macro logicalflags() {
 CF = 0;
 OF = 0;
}
```

SLEIGH:

```
:XOR AL,imm8 is vexMode=0 & byte=0x34; AL & imm8 {
    logicalflags();
    AL = AL ^ imm8;
    resultflags( AL );
}
```

```
 macro resultflags(result) {
  SF = result s < 0;
  ZF = result == 0;
  # PF, AF not implemented
 }
```

```
34 57        XOR     AL,0x57
                     CF = COPY 0:1
                     OF = COPY 0:1
                     AL = INT_XOR AL, 0x57:1
                     SF = INT_SLESS AL, 0:1
                     ZF = INT_EQUAL AL, 0:1
```

SLEIGH Example - x86 `XOR AL, imm8`

Raw bytes: `0x34 0x57`

x86 instruction: `XOR AL, 0x57`

```
macro logicalflags() {
 CF = 0;
 OF = 0;
}
```

SLEIGH:

```
:XOR AL,imm8 is vexMode=0 & byte=0x34; AL & imm8 {
    logicalflags();
    AL = AL ^ imm8;
    resultflags( AL );
}
```

```
macro resultflags(result) {
 SF = result s < 0;
 ZF = result == 0;
 # PF, AF not implemented
}
```

```
34 57        XOR      AL,0x57
                      CF = COPY 0:1
                      OF = COPY 0:1
                      AL = INT_XOR AL, 0x57:1
                      SF = INT_SLESS AL, 0:1
                      ZF = INT_EQUAL AL, 0:1
```

SLEIGH Example - x86 `XOR AL, imm8`

Raw bytes: `0x34 0x57`

x86 instruction: `XOR AL, 0x57`

```
macro logicalflags() {
 CF = 0;
 OF = 0;
}
```

SLEIGH:

```
:XOR AL,imm8 is vexMode=0 & byte=0x34; AL & imm8 {
    logicalflags();
    AL = AL ^ imm8;
    resultflags( AL );
}
```

```
macro resultflags(result) {
 SF = result s < 0;
 ZF = result == 0;
 # PF, AF not implemented
}
```

```
34 57        XOR     AL,0x57
                     CF = COPY 0:1
                     OF = COPY 0:1
                     AL = INT_XOR AL, 0x57:1
                     SF = INT_SLESS AL, 0:1
                     ZF = INT_EQUAL AL, 0:1
```

SLEIGH Example - x86 `RDTSC`

Raw bytes: `0x0F 0x31`

x86 instruction: `RDTSC`

SLEIGH Example - x86 `RDTSC`

Raw bytes: `0x0F 0x31`

x86 instruction: `RDTSC`

```
0f 31       RDTSC
                    $U9c60:8 = CALLOTHER "rdtsc"
                    EDX = SUBPIECE $U9c60, 4:4
                    EAX = SUBPIECE $U9c60, 0:4
```

SLEIGH Example - x86 `RDTSC`

Raw bytes: `0x0F 0x31`

x86 instruction: `RDTSC`

```
0f 31        RDTSC
                      $U9c60:8 = CALLOTHER "rdtsc"
                      EDX = SUBPIECE $U9c60, 4:4
                      EAX = SUBPIECE $U9c60, 0:4
```

SLEIGH Example - x86 RDTSC

Raw bytes: 0x0F 0x31

x86 instruction: RDTSC

SLEIGH:

```
:RDTSC is vexMode=0 & byte=0xf; byte=0x31 {
    tmp:8 = rdtsc();
    EDX = tmp(4);
    EAX = tmp(0);
}
```

```
0f 31        RDTSC
             $U9c60:8 = CALLOTHER "rdtsc"
             EDX = SUBPIECE $U9c60, 4:4
             EAX = SUBPIECE $U9c60, 0:4
```

SLEIGH Example - x86 `RDTSC`

Raw bytes: `0x0F 0x31`

x86 instruction: `RDTSC`

SLEIGH:

```
:RDTSC is vexMode=0 & byte=0xf; byte=0x31 {
    tmp:8 = rdtsc();
    EDX = tmp(4);
    EAX = tmp(0);
}
```

```
define pcodeop rdtsc;
```

```
0f 31       RDTSC
            $U9c60:8 = CALLOTHER "rdtsc"
            EDX = SUBPIECE $U9c60, 4:4
            EAX = SUBPIECE $U9c60, 0:4
```

SLEIGH Example - x86 RDTSC

Raw bytes: 0x0F 0x31

x86 instruction: RDTSC

SLEIGH:

```
:RDTSC is vexMode=0 & byte=0xf; byte=0x31 {
    tmp:8 = rdtsc();
    EDX = tmp(4);
    EAX = tmp(0);
}
```

```
define pcodeop rdtsc;
```

```
0f 31       RDTSC
                    $U9c60:8 = CALLOTHER "rdtsc"
                    EDX = SUBPIECE $U9c60, 4:4
                    EAX = SUBPIECE $U9c60, 0:4
```

SLEIGH Example - x86 `RDTSC`

Raw bytes: `0x0F 0x31`

x86 instruction: `RDTSC`

SLEIGH:

```
:RDTSC is vexMode=0 & byte=0xf; byte=0x31 {
    tmp:8 = rdtsc();
    EDX = tmp(4);
    EAX = tmp(0);
}
```

```
define pcodeop rdtsc;
```

```
0f 31        RDTSC
             $U9c60:8 = CALLOTHER "rdtsc"
             EDX = SUBPIECE $U9c60, 4:4
             EAX = SUBPIECE $U9c60, 0:4
```

SLEIGH Example - x86 `RDTSC`

Raw bytes: `0x0F 0x31`

x86 instruction: `RDTSC`

SLEIGH:

```
:RDTSC is vexMode=0 & byte=0xf; byte=0x31 {
    tmp:8 = rdtsc();
    EDX = tmp(4);
    EAX = tmp(0);
}
```

```
define pcodeop rdtsc;
```

```
0f 31        RDTSC
                     $U9c60:8 = CALLOTHER "rdtsc"
                     EDX = SUBPIECE $U9c60, 4:4
                     EAX = SUBPIECE $U9c60, 0:4
```

SLEIGH Example - x86 RDTSC

Raw bytes: 0x0F 0x31

x86 instruction: RDTSC

SLEIGH:

```
:RDTSC is vexMode=0 & byte=0xf; byte=0x31 {
    tmp:8 = rdtsc();
    EDX = tmp(4);
    EAX = tmp(0);
}
```

```
define pcodeop rdtsc;
```

```
0f 31        RDTSC
                       $U9c60:8 = CALLOTHER "rdtsc"
                       EDX = SUBPIECE $U9c60, 4:4
                       EAX = SUBPIECE $U9c60, 0:4
```

Outline

1. Scripting With Ghidra
2. Program Analysis
3. P-Code
4. Example
5. SLEIGH
6. **Discussion**
7. Conclusion

Opinionated thoughts on reverse engineering…

- Ghidra is great for collaboration, embedded architectures, iterative reverse engineering of sequentially released software (1.0, 1.1, 1.2, etc…), running heavy-weight well-developed scripts
- IDA is best for *interactive*, solo reverse engineering, particularly of Windows binaries
- Binary Ninja is amazing for quick, on-the-fly scripting as well as heavy-weight analysis, has an amazing community for support/Q&A/learning, and is constantly evolving and gaining new features

IDA vs Binary Ninja vs Ghidra

<u>IDA</u>

- **Maturity**
- Windows support
- **Decompiler**
- **Existing corpus of powerful plugins**
- Debugger
- Support for paid customers
- Well tested
- Industry standard

<u>Binary Ninja</u>

- **Under active development**
- **Modern**
- **Program analysis features (SSA)**
- Multi-level IL
- Rich API
- Embeddable
- Python-native scripting
- Clean modern UI
- **Community**

<u>Ghidra</u>

- **Maturity**
- Embedded support
- **Decompiler**
- **Massive API**
- Documentation
- Breath of features
- Collaboration
- Version tracking
- **Price and open source extensibility**

Decompiler - IDA Hex-Rays vs Ghidra

IDA Hex-Rays

- Optional add-on for IDA for IDA
- Microcode-based
- Supports limited architectures
- **Better built-in support for Windows**
- **Variables, data, and functions can be xrefed from decompiler**
- Variables can be mapped
- Variable representation can be changed in the decompiler (decimal, hex, char immediate, etc)
- Click to highlight

Ghidra Decompiler Decompiler

- **Deeply integrated with Ghidra**
- P-code based
- **Supports all architectures**
- No way to xref from decompiler
- **Produces fewer `goto` statements and seemingly more idiomatic C**
- Built in program analysis features, e.g., slicing and data flow
- Variables cannot be mapped
- Variable representation cannot be changed in the decompiler
- ***Middle click* to highlight**

ILs - Binary Ninja vs Ghidra

Binary Ninja

- **Multi-level: LLIL, MLIL, HLIL**
- **Machine consumable and human readable**
- **SSA form exposed as a first class feature**
- Designed in light of years of program analysis research
- Feels nicer to work with
- Deferred flag calculations
- Under active development, and shaped by community feedback

Ghidra

- **Single level p-code, but can be enhanced by decompiler analysis**
- **Designed for machine consumption first, not human readability**
- Uses SSA during decompilation, but raw p-code is not SSA
- Design origins based off of program analysis research from 20+ years ago

I Like Ghidra For...

- Scripting reverse engineering
- Firmware / embedded systems analysis
- Analysis of software that Hex-Rays can’t decompile
- Collaborative long-term professional RE
- Professional reversing at a computer workstation with multiple monitors, full keyboard with function keys, mouse with middle click and scroll wheel, etc...

Scripting - Java vs Python

- Java will catch errors at compile time, Ghidra’s API is highly object-oriented and benefits from this

- Complex Python scripts feel like binding together Java API calls with Python control flow and syntax

- **Recommended workflow:** prototype and experiment with APIs / objects in the Python interpreter, write final code in Java

For Reverse Engineers, By Reverse Engineers

- Built for multi-monitor use
- “Moving ants” highlight on control flow graphs
- Configurable “tool” views
- Hotkeys mappable to actions and scripts
- Right click > “extract and import”
- Processor manual integration
- Undo button
- Import directly from zip file
- Snapshot views
- Configurable listings
- Version tracker
- Project-based multi-binary RE
- `F1` to open help on whatever the mouse is pointing at
- File System browser
- Highly configurable assembly code listing
- Data flow analysis built into UI
- Embedded image detection
- Search for matching instructions
- Unique windows
  - Checksum Generator
  - Disassembled View
  - Data Type Preview
  - Function Tags
  - Symbol tree

Contributing to Ghidra

- Ghidra code is available on Github
  - Apache License 2.0

- NSA has been responsive to community questions and bug reports posted on Github

Official site: ghidra-sre.org

Open source: github.com/NationalSecurityAgency/ghidra

github.com/NationalSecurityAgency/ghidra-data

Outline

1. Scripting With Ghidra
2. Program Analysis
3. P-Code
4. Example
5. SLEIGH
6. Discussion
7. Conclusion

Class Project - Program Analysis Ideas

- Find vulnerable / interesting API calls
- Resolve manually imported functions in malware
- Semantically recognize implementations of standard functions (`memcpy`, `strcpy`, etc)
- Use data-flow analysis to find command injection and other logic-style bugs
- Structure recovery
- Find control transfer points not protected by CFI implementations
- Find pointer dereferences not protected properly by PAC

Conclusion

@0xAlexei

Ghidra is a powerful binary reverse engineering tool built by the US National Security Agency

- For reverse engineers, by reverse engineers
- Interactive and headless scripting
- Built for program analysis
  - No better time to explore the domain than today in 2022 with amazing tools available

Official NSA sites:

github.com/NationalSecurityAgency/ghidra
ghidra-sre.org