CS356: Reverse Engineering

W. Michael Petullo

University of Wisconsin–La Crosse

As of November 28, 2021

Reverse Engineering: Introduction

Reverse engineering is the process of analyzing a subject system to

- identify the system's components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction.

> Chikofsky and Cross, "Reverse Engineering and Design Recovery—A Taxonomy"

Binary reverse engineering starts with machine code and aims to recreate some approximation of the original source code and therefore recover how the system works or what it does.

Reverse Engineering: Why?

- Interfacing with an existing system.
- Coming to understand the purpose of a piece of malware.
- Bypassing copyright protections.*
- Analyzing the correctness or security of a system.
- Cost analysis.
- Modifying a component for a new use.

*Might violate the law.

Reverse Engineering: Analysis

Understanding a program starts with understanding two things:

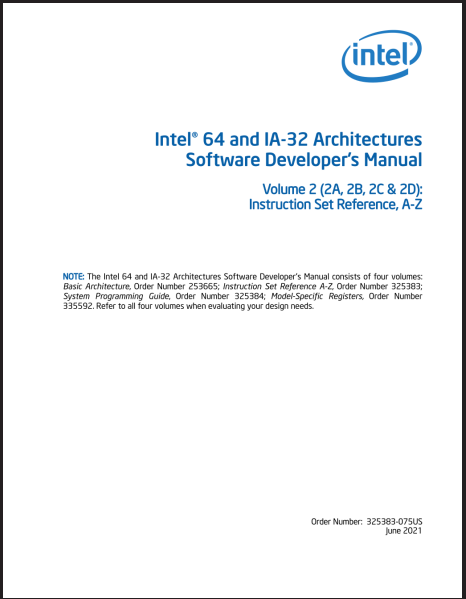
Control Flow Build a graph of execution flows through program; identify loops and conditionals; basic blocks.

Data Flow Understand the flow of information through the program; inputs, variables, and outputs.

Reverse Engineering: Tools

- `objdump -D BINARY`
- GDB
- IDA Pro ($)
- NSA's Ghidra (open source)

Reverse Engineering: Intel Documentation



intel

Intel® 64 and IA-32 Architectures
Software Developer's Manual

Volume 2 (2A, 2B, 2C & 2D):
Instruction Set Reference, A-Z

NOTE: The Intel 64 and IA-32 Architectures Software Developer's Manual consists of four volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-Z*, Order Number 325383; *System Programming Guide*, Order Number 325384; *Model-Specific Registers*, Order Number 335592. Refer to all four volumes when evaluating your design needs.

Order Number: 325383-075US
June 2021

Volume 2 of Intel's architecture manual describes the instructions available to application developers. A link to this document exists on the course resources page.

Objdump's default output uses AT&T syntax; `--disassembler-options=intel` switches to Intel's syntax and ordering. GDB has a similar option.

Reverse Engineering: hello

```
printf("Hello, world!\n");
```

Examples compiled with `-O0`, which turns of compiler optimizations. Optimizations can lead to surprising changes to binaries.

```
401126: push   %rbp
401127: mov    %rsp,%rbp
40112a: mov    $0x402010,%edi
40112f: callq  401030 <puts>
401134: mov    $0x0,%eax
401139: pop    %rbp
40113a: retq
```

Remember: `f(rdi, rsi, rdx, rcx, r8, r9, stack ...)`

Reverse Engineering: if

```
bool b = true;
if (b) {
    printf("Hello, world!\n");
} else {
    printf("Goodbye, ...!\n");
}
```

```
401126: push   %rbp
401127: mov    %rsp,%rbp
40112a: sub    $0x10,%rsp
40112e: movb   $0x1,-0x1(%rbp)
401132: cmpb   $0x0,-0x1(%rbp)
401136: je     401144 <main+0x1e>
401138: mov    $0x402010,%edi
40113d: callq  401030 <puts>
401142: jmp    40114e <main+0x28>
401144: mov    $0x40201e,%edi
401149: callq  401030 <puts>
40114e: mov    $0x0,%eax
401153: leaveq
401154: retq
```

Reverse Engineering: case

```
int i = 1;
switch(i) {
case 0:
    printf("Hello!\n");
    break;
case 1:
    printf("Goodbye!\n");
    break;
case 2:
    printf("Other!\n");
    break;
}
```

```
40112e: movl   $0x1,-0x4(%rbp)
401135: cmpl   $0x2,-0x4(%rbp)
401139: je     401167 <main+0x41>
40113b: cmpl   $0x2,-0x4(%rbp)
40113f: jg     401172 <main+0x4c>
401141: cmpl   $0x0,-0x4(%rbp)
401145: je     40114f <main+0x29>
401147: cmpl   $0x1,-0x4(%rbp)
40114b: je     40115b <main+0x35>
40114d: jmp    401172 <main+0x4c>
40114f: mov    $0x402010,%edi
401154: callq  401030 <puts>
401159: jmp    401172 <main+0x4c>
40115b: mov    $0x40201e,%edi
401160: callq  401030 <puts>
401165: jmp    401172 <main+0x4c>
401167: mov    $0x40202e,%edi
40116c: callq  401030 <puts>
```

Reverse Engineering: for

```c
for (int i = 0; i < 10; i++) {
    printf("Hello, world!\n");
}
```

```
401126: push   %rbp
401127: mov    %rsp,%rbp
40112a: sub    $0x10,%rsp
40112e: movl   $0x0,-0x4(%rbp)
401135: jmp    401145 <main+0x1f>
401137: mov    $0x402010,%edi
40113c: callq  401030 <puts@plt>
401141: addl   $0x1,-0x4(%rbp)
401145: cmpl   $0x9,-0x4(%rbp)
401149: jle    401137 <main+0x11>
40114b: mov    $0x0,%eax
401150: leaveq
401151: retq
```

Reverse Engineering: while

```
int done = false;
while(!done) {
    printf("Hello, world!\n");
    done = true;
}
```

```
401126: push   %rbp
401127: mov    %rsp,%rbp
40112a: sub    $0x10,%rsp
40112e: movl   $0x0,-0x4(%rbp)
401135: jmp    401148 <main+0x22>
401137: mov    $0x402010,%edi
40113c: callq  401030 <puts@plt>
401141: movl   $0x1,-0x4(%rbp)
401148: cmpl   $0x0,-0x4(%rbp)
40114c: je     401137 <main+0x11>
40114e: mov    $0x0,%eax
401153: leaveq
401154: retq
```

Reverse Engineering: do

```
int done = false;
do {
    printf("Hello, world!\n");
    done = true;
} while(!done);
```

```
401126: push   %rbp
401127: mov    %rsp,%rbp
40112a: sub    $0x10,%rsp
40112e: movl   $0x0,-0x4(%rbp)
401135: mov    $0x402010,%edi
40113a: callq  401030 <puts@plt>
40113f: movl   $0x1,-0x4(%rbp)
401146: cmpl   $0x0,-0x4(%rbp)
40114a: je     401135 <main+0xf>
40114c: mov    $0x0,%eax
401151: leaveq
401152: retq
```

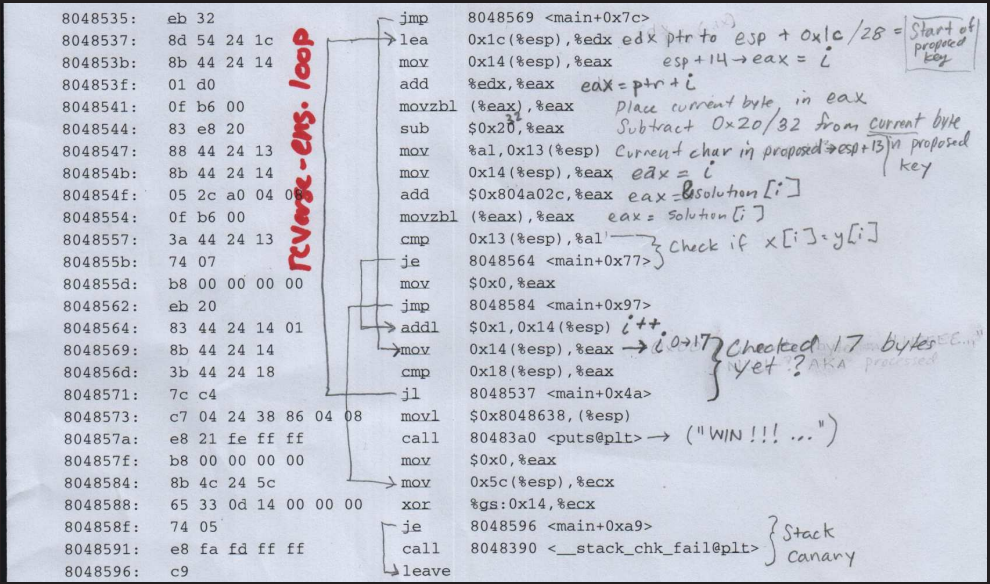
Reverse Engineering: array

```
int buf[] = { 2, 3, 1 };

printf("%d%d%d\n",
        buf[2],
        buf[0],
        buf[1]);
```

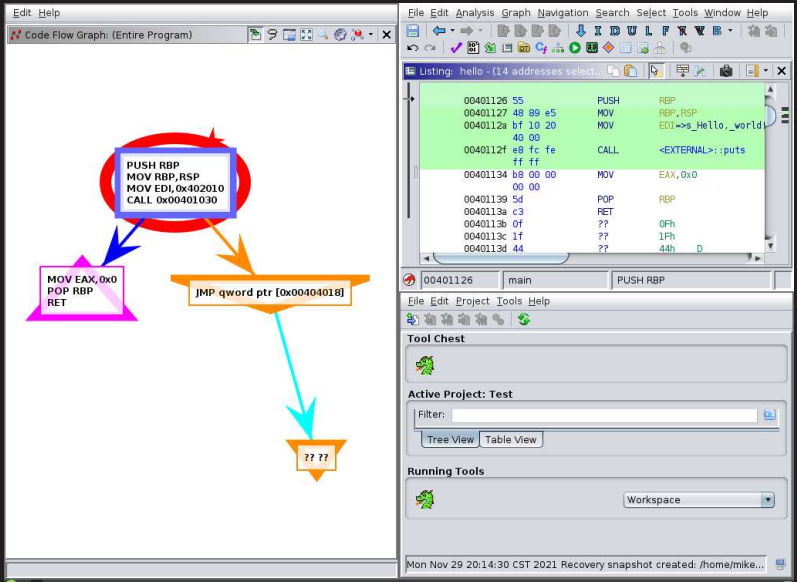
```
401126: push   %rbp
401127: mov    %rsp,%rbp
40112a: sub    $0x10,%rsp
40112e: movl   $0x2,-0xc(%rbp)
401135: movl   $0x3,-0x8(%rbp)
40113c: movl   $0x1,-0x4(%rbp)
401143: mov    -0x8(%rbp),%ecx
401146: mov    -0xc(%rbp),%edx
401149: mov    -0x4(%rbp),%eax
40114c: mov    %eax,%esi
40114e: mov    $0x402010,%edi
401153: mov    $0x0,%eax
401158: callq  401030 <printf>
40115d: mov    $0x0,%eax
401162: leaveq
401163: retq
```

Reverse Engineering: Work



slide-by-hand.tex

Reverse Engineering: Ghidra



Reverse Engineering: Assignments

Graded Homework Aquinas: reveng2, reveng3, and reveng4

`https://www.flyn.org/courses/cs356/schedule`