



# CS451 – Software Analysis

## Lecture 7 **Disassembly and Binary Analysis Fundamentals (part 1)**

Elias Athanasopoulos  
athanasopoulos.elias@ucy.ac.cy

# Disassembly



- Once compiled, binaries have a very specific form
  - They contain several sections with code and data
- It is useful to analyze the binary
  - We have seen some forms of that, already
- Analysis usually involves decomposing the binary and extracting the code
  - **Disassembly**: Extracting the machine code and mapping it to symbolic language (assembly)

# Disassembly vs compilation



- Compilation of a program invokes the assembler to transform the produced assembly to machine code
- The reverse process is not straight-forward
  - Code and data are usually intermixed
  - Some architectures have variable-length instructions
- Types of disassembly
  - Static and dynamic

# Static disassembly



- Extract all code without executing the binary
  - Step 1: load the binary in memory
  - Step 2: find all the machine code of the binary
  - Step 3: disassemble all found machine code
- Step 2 is not trivial, and it is still an open problem
  - Two known techniques in practice: linear and recursive disassembly
- Proprietary disassemblers use additionally common well-known patterns emitted by broadly used compilers

# Linear disassembly



- Simple approach
  - Start from a specific location and treat a stream of characters as a stream of instructions
- Several simple Unix tools incorporate this approach (e.g., objdump)
- It is not uncommon for compilers to inject data (e.g., jump tables) inside code
  - In that case, the linear approach will treat data as machine instructions
  - In dense instruction sets (e.g., x86), any data can be mapped to a potentially valid instruction

# Jump tables



- A compiler may use a jump table to encode a switch-case statement instead of emitting several conditional operations
- The switch-case code contains an indirect jump (e.g., `jmp *%rax`) which uses data (i.e., the table) injected in the code

# Example



```
int foo(char i) {  
    switch (i) {  
        case 'a':  
            return 2;  
            break;  
        case 'b':  
            return 13;  
            break;  
        case 'c':  
            return 24;  
            break;  
        case 'd':  
            return 35;  
            break;  
        case 'e':  
            return 46;  
            break;  
        default:  
            return -1;  
    }  
}
```

# Compile and disassemble



```
$ gcc -Wall -c jump-table.c -o jump-table.o
$ objdump -d jump-table.o | grep jmpq
1f: ff e0                jmpq    *%rax
```



# Linear disassembly - limitations



- Treating data as machine code
  - If data corresponds to valid instructions, the disassembler will treat the data as part of the instruction stream
  - If data corresponds to invalid instructions, the disassembler needs to resolve the next valid instruction
- In both cases, the disassembler is desynchronized

# Desynchronization



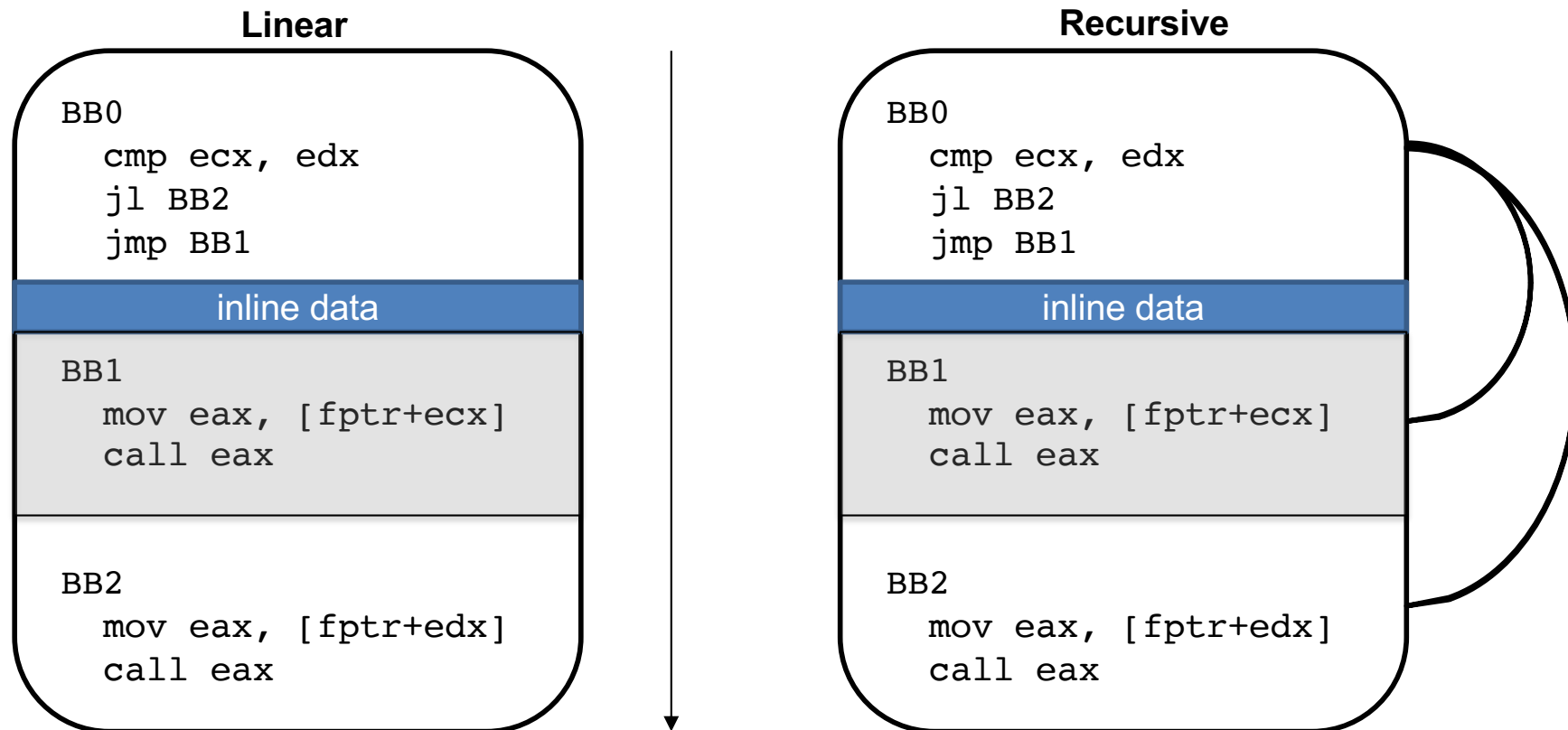
	Inline data				Code										
	8E	20	5C	00	55	48	89	E5	48	83	EC	10	89	7D	FC
Synchronized					push rbp	mov rbp, rsp			sub rsp, 0x10				mov [rbp-0x4], edi		
-4 bytes off	mov fs, [rax]		pop rsp	add [rbp+0x48], dl			mov ebp, esp		sub rsp, 0x10						
-3 bytes off			and [rax+rax*1+0x55], bl			mov rbp, rsp									

# Recursive disassembly



- Start from a specific location, but follow jumps and calls
  - Code does not *execute* linearly but follows a control flow
  - Compared to linear disassembly, recursive disassembly examines each decoded instruction
- Some call/jump targets may be available only at run-time
  - Indirect calls and jumps use target values that depend on the program execution (e.g., call of a function through a function pointer)

# Linear vs recursive



# Comparison



- There is no optimal strategy for disassembling binaries; the problem is still open
- Linear and recursive disassembling have both weaknesses
- Several proprietary tools (e.g., IDAPro) use a combination of strategies
  - Additionally, they have a large database of common patterns (e.g., for jump tables) emitted by known compilers
  - Based on such patterns, disassemblers realize *heuristics* for producing more accurate disassembly

# Dynamic disassembly



- Analyzing the code statically has benefits and weaknesses
- Benefits
  - You can explore all the code of the program
  - In general, static analysis is fast
- Weaknesses
  - Analyzing large programs is hard due to the limitations of linear/recursive disassembling
  - Dead code cannot be avoided
- Another direction is to analyze the code, while it executes

# Analysis of executing code



- In principle the code that is executing is valid
  - It is the code that is processed by the actual machine (e.g., CPU)
- Compared to static analysis, dynamic analysis has all the state of the program
  - Values in memory and hardware registers
- An indirect jump (e.g., `jmp *%rax`) can be tricky for static analysis, but not for dynamic analysis
  - The value of `%rax` is known before the actual jump

# How to perform dynamic disassembly?



- The easiest way to inspect the code while it executes is through debugging
  - For instance, use `ptrace()` and inspect every instruction before executing
  - This can be done in `gdb` using the `'si'` (step instruction) command
- Several other systems have been developed for automating the entire process
  - Intel PIN, DynamoRIO



# Extract all executed code with gdb



```
$ gdb ./test
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-16.el8
[...]
Reading symbols from ./test...(no debugging symbols found)...done.
(gdb) info files
Symbols from "/home/elathan/epl451/src/week4/test".
Local exec file:
    `./home/elathan/epl451/src/week4/test', file type elf64-x86-64.
    Entry point: 0x4004a0
    0x0000000000400238 - 0x0000000000400254 is .interp
    [...]
(gdb) b *0x4004a0
Breakpoint 1 at 0x4004a0
(gdb) set pagination off
(gdb) set logging redirect on
(gdb) set logging on
Redirecting output to gdb.txt.
(gdb) run
(gdb) display/i $pc
(gdb) while 1
    >si
    >end
Hello world.
(gdb)
```

# Dynamic disassembly limitations



- Like static disassembly, dynamic disassembly suffers from weaknesses
- Static disassembly faces accuracy problems
  - Due to indirect branches, and data mixing with code
- Dynamic disassembly suffers from code coverage
  - Only the executed code is captured
  - This code is heavily based on the program's input
  - Large programs can be very complicated with a very large space of valid user inputs (e.g., a PDF viewer may contain a lot of code for parsing uncommon PDF features)
  - UI-based programs need an automatic way to exercise the user-interface

# Hidden code



- Code can be on purpose hidden
  - Consider a malware that starts the malicious activity at a specific time (time bomb)
- Code can be obfuscated
  - Code that is not actually needed in the program may be there just for confusing the analyst (e.g., a loop that computes a value that is never actually used)
- Code can be executed only under specific conditions
  - Malware can try to detect if it is analyzed and hide any malicious activity
  - Detecting if you are traced can be done by observing environmental artifacts
  - Time operations (slow when traced)
  - Create files, open devices (specific naming in VMs)

# Code coverage strategies



- Test suites
  - Unit tests, that exercise a set of features of the main program
- Fuzzing
  - Analyze a program by sending random inputs
- Symbolic execution
  - Replace concrete values in the input stream with variables (*symbols*)

# Test suites



- Small and specific input scenarios
- Many of them, if combined, can trigger a lot of the standard functionality of the program
- However, sometimes programs can be used in non-standard ways
- Not all programs come with a test suite
- Building and maintaining the test suite demands significant human labor

# Fuzzing



- Analyze a program by observing the code executing while processing specific inputs
  - Generation-based: generate inputs from scratch
  - Mutation-based: mutate a known input
- The input space is usually enormous and many of the inputs may be invalid or exercise the same code
  - Gray-box fuzzing: instrument the analyzed program to send feedback to the fuzzer
  - Feedback is usually the code that was exercised due to the last sent input
- The fuzzer based on the feedback can perform mutations and create new inputs
- The fuzzer can observe side-effects
  - A program crash suggests a memory-corruption bug

# Symbolic execution



- Programs process concrete values
  - `x = argv[1]` will eventually set `x` to very concrete value given by the user
- We can emulate the program using variables (or symbols)
  - Treat `x` as a mathematical variable instead of a concrete value
- We can then see the dependencies of other variables of the program and apply constraints
  - `if (x > 5) ...` constrains `x` to be greater than 5
- Treat all variables of the program and their constraints as a system of equations
- Use a SAT solver to calculate which code can be executed
- It can be very demanding to solve the system of equations and constraints, as the analyzed code increases

# Homework



- Create a simple C program that uses a switch statement and observe the disassembly with objdump
  - Spot the indirect jump
- Use gdb to automatically log all executed commands of /bin/lis in a file