Binary Analysis and Disassembly

Petr Ročkai and Lukáš Korenčik

May 7, 2019

Organisation

- theory: 30-50 minutes every week
- remaining time: coding and discussions
- there will be 6 bi-weekly assignments
  - together, they will form a project
  - each assignment is a milestone

Grading

- you need 7 points to pass the subject
- each assignment is worth 1 point
- showing up 10 times is worth 1 point
- up to 2 points for writing code reviews
- up to 2 points for meeting deadlines

Deadlines and Feedback

- the deadline for each assignment is 14 days
- beating the deadline gives you 1/3 of a point
  - the solution must be of sufficient quality
- feedback will be given on the off weeks
  - i.e. 7 days after the deadline

Programming Language

- C or C++ is up to you
- you can use up to C11 and up to C++17
- only the standard library and POSIX
- no boost, no `libelf` or `BFD`

Repositories

- make a repository for your homeworks
  - `git`, `hg` or whatever works for you
  - make it public and email me the URL
- write a simple `Makefile` (inc. dependencies)
  - you will only have a few source files
  - `cmake` is acceptable but discouraged

Assignment Submission

- tag your repository with `hw1`, etc.
  - use `hw1.1` etc. for resubmissions
- tag dates are what counts for deadlines
- we will not look at master head
  - you can break stuff freely there

Semester Plan (part 1)

		date
1.	introduction & preliminaries	19.2.
2.	instruction sets	26.2.
3.	static control flow	5.3.
4.	dynamic control flow	12.3.
5.	executable files, ELF	19.3.
6.	dynamic linking	26.3.

Semester Plan (part 2)

	date
7. debug info	2.4.
8. DWARF	9.4.
9. function calls and frames	16.4.
10. advanced instructions	23.4.
11. debugger basics	30.4.
12. decompilation basics	7.5.

Assignment Schedule

	given	due
1. decoding instructions	26.2.	12.3.
2. basic blocks & branching	12.3.	26.3.
3. making sense of ELF	26.3.	9.4.
4. parsing symbol tables	9.4.	23.4.
5. reconstructing functions	23.4.	7.5.
6. a complete disassembler	7.5.	21.5.

Part 1: Preliminaries

Machine Code

- consists of individual instructions
- encoded in a tightly-packed binary format
  - may be fixed or variable length
- stored program architecture
  - instructions live in addressable memory

Assembly

- symbolic language one level above machine code
  - abstracts away from numeric addresses
  - replaces them with symbolic labels
- instructions are encoded in a text format
- designed for humans (but rarely used nowadays)

C and Compilers

- another layer of abstraction over assembly
- abstracts away the specifics of hardware architecture
  - registers, stack management, opcodes
  - provides structured control flow
- still a low-level language, mostly OS-level programs

Compiled High-Level Languages

- another abstraction rung above C
  - algebraic or class-based type systems
  - abstract data structures
  - extensive standard libraries
  - late dispatch, lexical closures, ...
- e.g. C++, Rust, ML, Haskell, (Java)

Interpreters

- typically the highest rung of the abstraction tower
  - dynamic types, garbage collectors
  - powerful, high-level libraries or APIs
- often realized as JIT compilers / virtual machines
  - usually implemented in C or C++
- e.g. JavaScript, Python, Ruby, bash, R

Disassembly

- going from machine code to assembly
  - decode instruction
  - recover control flow structure
- print the program in human-readable format
- re-assembling should give identical machine code

Decompilation

- attempt to reconstruct high-level code
  - recovery of structured control flow (`if`, `while`)
  - identification of local variables
  - recovery of addresses
- decompile → compile is not idempotent

Exercise 1.1: `objdump`

- read the `objdump` manpage
- try `objdump -x` on some binaries
  - `/usr/bin/gzip`
  - your own test program (hello world style)
  - try `-static`, `-fPIC` &c.
  - try both the `.o` file and the executable
- also try `objdump -x --disassemble`

Exercise 1.2: `gdb`

- compile your test program with `-g`
- `gdb [--args] ./a.out`
- `start`
- `stepi`, `disassemble`, `print $rax`
- `break`
- for more user friendliness: `layout`

Exercise 1.3: reading binary data

```
$ printf "\x03\x12\x01\x00\x00\x00" > file.bin
```

read the above data into the following structure

```
struct __attribute__((packed)) d
{
    short x;
    int y;
}
```

expected result: `x = 4611, y = 1`

Part 2: Instruction Sets

Instruction Anatomy

- opcode: what to do
- operands
  - immediate values (part of instruction)
  - register references
  - memory references (immediate or via register)
- modifiers (e.g. lock)

Opcode Classes

- control flow
- integer arithmetic
- bit operations
- memory access
- floating point arithmetic
- special instructions

Register Classes

- GPR: General Purpose Register
  - hold a single word: integers, addresses
- SIMD (vector) and/or floating point registers
- pointers: stack, instruction, frame (base)
- machine control registers

Control Flow

- conditional & unconditional jumps
  - direct (fixed address)
  - indirect unconditional (computed address)
  - conditional on results of arithmetic
- subroutine calls and returns
  - use the stack for return addresses

Arithmetic

- addition, subtraction
- signed+unsigned division, multiplication
- integer comparison (signed/unsigned)
- standard instructions up to word size (64b)
  - 128b operations are available too

Bit Operations

- bitwise and, or, xor, negate
- shifts and rotations
- bit field packing/unpacking
- bit counting, endianity conversion

Memory Access

- load from and store into memory
- various address computation modes
  - part of the access instruction
  - special-purpose arithmetic (`lea`)
  - general-purpose arithmetic

Addressing Modes

- scalars: base register + offset
  - especially useful for stack variables
  - also globals (relative to program counter)
- arrays: base register + immediate * index register
- 'far' addressing for segmented memory (obsolete)

Floating Point Arithmetic

- separate instruction set
- separate registers (distinct from GPRs)
- variable precision (usually 32b, 64b, 80b)
- governed by IEEE 754

Specials: Synchronisation

- atomic memory access
  - read-modify-write (add, sub, xor, ...)
  - compare + exchange
- memory fences / barriers
- on amd64 encoded using the `lock` prefix

Specials: Vector Instructions

- SIMD: single instruction (opcode), multiple data
- integer and floating-point arithmetic
- 4-8 values packed in 128b or 256b register
- speeds up number crunching considerably
- on top of usual superscalar execution

Specials: User Mode

- checksums (e.g. `crc32`)
- symmetric crypto (`aes-ni`)
- random numbers (`rdrand`, `rdseed`)
- processor capabilities (`cpuid`)
- timers (`rdtsc`)

Specials: Privileged Mode

- CPU management opcodes and registers
- interrupt handling
- system calls
- cache control
- debugging and monitoring
- virtualisation

Exercise 2.1

- learn a bit more about assembly
- use `gcc -S` to produce examples
  - you can also try `-fverbose-asm`
- write a recursive `factorial` (in C)
  - use gdb instruction stepping
  - try an iterative version too

Exercise 2.2

- write a simple assembly program
- borrow the prologue/epilogue from gcc
- sum an arithmetic/geometric sequence
  - use formulas first (just arithmetic)
  - try using a summing loop

Instruction Encoding

- how to encode opcodes and operands into bytes
- fixed-length or variable-length
- fixed: e.g. VLIW (very long instruction word)
    - often employs instruction combining
    - variant: fixed opcodes, trailing immediate operands

Variable-Length Coding

- saves space compared to fixed-width coding
- often much harder to decode
- usually decoded from left to right
- first byte affects what second byte means, &c.
- already-decoded prefix tells you whether to continue

Encoding on AMD64

- programmer's manual in study materials
- variable length (even opcodes)
- very long history of extensions
- different meaning in different CPU modes
- not a very clean encoding of a messy instruction set

Assignment 1

- write an instruction decoder for amd64
- have `make decode` build the binary
- invocation `./decode 74 1a`
    - prints: `je 0x1a(%rip)`
- we will only decode a small subset of instructions
- print `unknown instruction` if that is the case

Assignment 1: Required

- branching: `jmp`, `je`, `jne`, `jb`
  - operands: rel8off, rel32off
- stack: `push`, `pop` (64b only)
- calls: near `call` (rel32off) and `ret`
- `mov` in 64b mem/reg versions (details later)
- a few arithmetic and bitwise ops, `nop`, `int3`

Assignment 1: Arithmetic & Bitwise

- `xor eax` imm32 and `xor rax` imm32
- `add eax` imm32 and `add rax` imm32
- `mul` with 2 64b registers (`rax` – `rdx`)
- `cmp eax` imm32 and `cmp rax` imm32
- `cmp` with 2 64b registers (`rax` – `rdx`)

Assignment 1: `mov`

- only the `89` and `8B` opcodes
  - with 2 64b registers (`rax` – `rdx`)
  - from memory to a 64b register
  - from a 64b register to memory
- memory: address in `rax` or `rbx`
  - `rip` and `rbp` + 32b displacement

Assignment 1: Not Required

- anything in the VEX maps
- memory operands other than
  - `mov` with address in `rax` or `rbx`
  - `mov` with `rip` and `rbp` + disp32
- prefixes other than the REX range

Assignment 1: Hints

- most 64b instructions need a REX prefix (`0x40-0x4F`)
- exceptions: `call`, `ret`, `jmp`, branching
  - some of the `push`/`pop` (those of 'named' GPRs)
- look for complete decoded examples in `objdump`

Part 3: Static Control Flow

Control Flow

- answers the question ‘what to do next?’
- normally, instructions run in a sequence
- just like statements in C
- how about conditionals and loops?

Structured Control Flow

- used in high-level languages
- `if` statements or expressions
- structured loops: `while`, `for`
- not available in machine code

Goto

- also known as unstructured control flow
- `goto` jumps from one place to another
  - the destination is called a label
  - the jump is unconditional (always taken)
- `if` + `goto` $\rightarrow$ any loop

Goto: Example

```
int f( int x )
{
    int i = x;
loop:
    x = x * --i;
    if ( i > 1 )
        goto loop;
    return ~x;
}
```

Machine Code

- goto is basically a jump instruction
- there are no labels in machine code
- assembler computes label offsets
- there is also a conditional jump instruction
  - perform the goto only if a condition holds

Simplified `if`

- in C, `if` can guard arbitrary statements
- what if it could only guard exactly 1 `goto`?
  - and there is no `else` either
- we can still do everything

```
if ( x ) { foo(); bar(); }
else baz();
```

Reinventing if

```
if_begin:
    if ( !x )
        goto if_false;
    foo(); bar();
    goto if_end;
if_false:
    baz();
if_end:
```

Conditional Jump

- recall `if ( x > 0 ) goto loop`
- this is basically 2 instructions
- first is `cmp`, the second is `jg`
- conditional `goto` is conditional jump
- used to encode all control flow in machine code

Basic Blocks

- abstraction used by compilers
- starts with a label
- followed by a sequence of non-jump instructions
  - no labels or jumps in the sequence
- with a single jump/branch at the end

Control Flow Graph

- take instructions as nodes
- control flow as edges
- extremely useful for code analysis
- using basic blocks makes the graph much smaller

Exercise 3.1

- rewrite this program with conditional gotos

```
while ( x < 1000 )
{
    x *= 5;
    if ( x % 7 == 0 )
        break;
    x --;
}
```

```c
int fib( int n ) /* exercise 3.2 */
{
    if ( n <= 2 )
        return 1;
    else
    {
        int a = fib( n - 1 );
        int b = fib( n - 2 );
        return a + b;
    }
}
```

Exercise 3.3

- take the goto version of program from 3.2
- change it to only have one `return` statement
- draw the control flow graph of both versions

Exercise 3.4

- write an iterative version of `fib`
- you can use the argument + 3 variables
- change it into goto form
- draw the control flow graph

Exercise 3.5

- compile all above programs into object files
- disassemble them using `objdump`
- recover control flow from the assembly
  - only add labels that are required
  - identify basic blocks

Exercise 3.6

- rewrite program from 3.4 into assembly by hand
- only use registers for computation
- start from an empty `int fib( int )` skeleton
- check that the program does the right thing

Part 4: Dynamic Control Flow

Last Time

- direct conditional + unconditional jumps
- basic blocks, control flow graph

Today

- direct & indirect function calls, returns
- indirect jumps and jump tables

Function Calls

- `call` is usually static (fixed address)
- but `ret` jumps to a dynamic address
  - also known as return address
- arguments are passed in registers or on stack
- local variables are stored on the stack

Call Frame

- each function uses up a section of the stack
  - known as a frame, holds automatic local variables
  - though some of those might only live in registers
- there's also stuff in-between frames
  - arguments, register spills, return address

Indirect Jump

- jump to a dynamic address (i.e. not constant)
- often arises from `switch` statements (in C)
- either computed or via a jump table
- looks like `jmp *%rax` (if the address is in `rax`)

Ex 4.1

- write a simple C function with a `switch`
- use consecutive integer cases (i.e. 1, 2, 3, ...)
- put different code in each branch (e.g. `return N`)
- compile with `gcc` and `clang` with different `-O`
  - compare the assembly output

Detour: Graphviz

- a simple but powerful tool to draw graphs
- accepts plain-text input that looks like this

```
digraph G {
  1 [ shape=rectangle label="box" ]
  2 [ shape=rectangle label="another\lbox\l" ]
  1 -> 2 [ label="arrow" ]
}
```

Ex 4.2

- draw the CFG from 3.3 or 3.4 using `dot`
- see `https://graphviz.org` for docs
- to look at the result, use `dot -Tx11 < cfg.dot`
  - `dot -Tpdf > cfg.pdf` also works
- put instructions into the boxes

Assignment 2

- extend your decoder to allow multiple instructions
- print each instruction on a separate line
- assume the code starts at address 0
- decompose the code into basic blocks
- use graphviz `dot` to generate a CFG

Assignment 2: Input

- continue to allow ascii/hex bytes in argv[]
- if no args given, read a raw binary from stdin
- you can assume there are only known instructions
- and the input will be at most 2KB (for now)

Assignment 2: Output

- generate ‘maximal’ basic blocks
  - print `# <label>` after jump instructions
- make a separate binary for CFG output (`./cfg`)
  - print the `dot` source to `stdout`
  - use boxes for BB’s in `dot` output
  - put decoded instructions into the boxes as labels

Part 5: Executable Files, ELF

ELF

- Executable and Linkable format
  a. a container for machine code and static data
  b. relocation tables and other linking info
  c. debug information and other metadata
- used on all modern UNIX systems
  - except macOS (which is only half UNIX)

Basic Concepts

- ELF files start with an executable header
  - class: machine word size (32 or 64 bits)
  - endianness: either MSB (big) or LSB (little endian)
  - version number (in case the layout changes)
- program header tables and section header tables follow

Reading ELF Files

- ELF contains a number of data structures
  - those are described as C `struct`
  - `elf.h` contains the definitions
- probably easiest way is to use `mmap`
  - look up how `mmap` works with `man mmap`
  - we will assume the file uses native format

Example: 64b Header

```
typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf64_Quarter   e_type;
        Elf64_Quarter   e_machine;
        Elf64_Half      e_version;
        /* elided */
        Elf64_Quarter   e_shnum;
        Elf64_Quarter   e_shstrndx;
} Elf64_Ehdr;
```

Native ELF Files

- `elf.h` has both `Elf64_*` and `Elf32_*`
- but you can skip the number
  - i.e. use `Elf_Ehdr`
  - this will select the native format at compile time
- if we cared about portability, we'd use `libelf`
  - that would also take care of endianness &c.

Sections

- ELF files are made of sections
- each section has a header in the section table
  - sections contain actual data
  - what data it is depends on the section
- important sections: `.text`, `.data`, `.rodata`

Additional Sections

- `.text`, `.data`, `.rodata` are of type `SHT_PROGBITS`
- `.symtab` the symbol table, type `SHT_SYMTAB`
- `.dynsym` symbols for the dynamic linker, `SHT_DYNSYM`
- `.rel.text` the relocation table for program text, `SHT_REL`
- `.init`, `.fini` 'global' constructors and destructors

Ex 5.1: `mmap`

- open a binary file (e.g. `/usr/bin/gzip`)
- `mmap` it into memory
- print the first 4 bytes of the file

Ex 5.2: `elf.h`

- extend the program from previous exercise
- print out info from the executable header
  - the type of the file (as a human-readable string)
  - machine type (as 4 hexadecimal digits)
  - the address of the entry point
  - the number of section headers present in the file

Program Headers

- represented by `struct Elf_Phdr` (see `man elf`)
- contains information about the entire program
  - dynamic linker path (`PT_INTERP`, ELF 'hashbang')
  - which parts of the file to load (`PT_LOAD`)
  - info for the runtime linker (`PT_DYNAMIC`)

Ex 5.3: Program Headers

- again, extend the program from previous exercise
- read all program headers (`Elf_Phdr`)
- print the interpreter for the program (`PT_INTERP`)
- print the notes (`PT_NOTE`), if any

Part 6: Dynamic Linking

Linking

- putting multiple object files together
- and resolving relocations within them

When?

- static / build time (system linker, `ld`)
- dynamic / run time (runtime linker, `ld.so`)

Build-time Linking

- read in one object at a time
- assign addresses to sections in the file
- merge and update the symbol tables
- resolve all applicable relocations

Relocations

- the compiler leaves space for unknown addresses
  - each gets an entry in the relocation table
  - saying which symbol and where in the file
- `objdump -rd` shows the relocations
- resolving relocations means altering instructions

Aside: Text Sharing

- a program may run in multiple processes
- in that case, the text is loaded only once
  - same goes for `.rodata`
  - of course the code must be read-only
- this is quite important for memory consumption

Relocations vs Sharing

- dynamic relocations could ruin everything
- we want to confine those to a small area
  - this is the global offset table (GOT)
  - holds both data and code relocations
- for calls, PLT stubs are used

Position-Independent Code

- uses `%rip`-relative addressing extensively
  - both for calls and for data
- the GOT is also at a fixed relative address
  - each object file has its own GOT
  - they are merged by the system linker
  - in the end only one GOT per shared object

Procedure Linkage Table

- the caller object has a `foo@plt` stub
  - calls to `foo` go through `foo@plt`
  - such calls are direct (and unconditional)
  - happens for all external functions
- the stub consults the GOT entry for `foo`
  - and performs an indirect jump to that

Lazy Binding

- initially, the GOT points into `ld.so`
- the `ld.so` code patches the GOT entry
- then jumps on to the resolved address
- next call will go directly to the right address

Interface to `ld.so`

- a family of C functions to call into `ld.so`
- most important: `dlopen` and `dlsym`
  - see the manpages for details
  - `dlopen` loads shared libraries
- allows programs to call functions by name
  - including names only known at runtime

Ex 6.1

- create a shared library from C code
  - provide a function and a global variable
  - `cc -fPIC` to build, `cc -shared` to link
- add an executable which uses the lib
  - build as usual, link with `-L. -lmylib`
  - inspect the result with `ldd`

Ex 6.2

- also build the executable with `-fPIC`
  - and link it with `-pie`
- disassemble both and compare the result
- compare to the code in the library

Ex 6.3

- add a second shared library
- use the first library in the second
  - both the variable and the function
  - use `-shared -lmylib` when linking
- inspect the disassembly
  - compare to the executables

Ex 6.4

- write a program that dlopens the second lib
  - use `dlsym` to find and call the function
  - do not link to either of your libs
- check that both libraries got loaded
  - you can use e.g. global constructors
  - and/or attach with `gdb`

Ex 6.5

- use `gdb` to trace the PLT stub
  - call e.g. `puts` 2× in the test program
- reminder: `stepi` steps one instruction
  - `disass` shows the current function
  - `fin` runs until current frame returns
  - `start` gets you to the start of `main`
  - `p /x *(long*)some_address`

Assignment 3

- add `decode.elf` and `cfg.elf`
  - the input file is the first argument
  - if it is ELF, process `.text`
  - otherwise assume raw machine code
- print the address of each instruction
  - only applies to `decode.elf`

Assignment 3 (cont’d)

- try to write a simple test program
  - only use instructions you can decode
  - you will need to write it in assembly
- extend output of `mov` to/from memory
  - add `# section + offset` (akin to `jmp`)
- add labels for `call` targets & print them

Part 7: Debuggers & Debug Info

Debugging

- machine code is a lot simpler than C
- the relationship between them is less than clear
- but machine code is what gets executed
- and what we, ultimately, debug

Debuggers

- originally, a debugger only knew about assembly
- you could step through the instructions
  - like what `stepi` does in `gdb`
  - and set a breakpoint at an address
- you could read register values and memory content

Symbolic Debuggers

- the first thing we can do is work with symbols
    - functions have names and addresses
- from looking at the instruction pointer, we can
    - check in which function the instruction resides
    - print information about it (cf. `objdump`)

Stack Trace

- also known as a backtrace (e.g. in gdb)
- it tells us where in the program we are
- obtained by walking the call frames
- and printing the instruction pointer from each

Stack Frames

- how do we know how big the stack frames are?
- the compiler can embed this info in the binary
  - alternatively, we could use the base pointer
  - but the base pointer must live in an agreed position
  - in our case, this would be `rbp`

Ex 7.1

- write a C program that crashes
  - e.g. division by zero, null dereference
  - in a recursive function
  - hide the crash at least 3 calls deep
- run the program in `gdb`
  - look at `backtrace` and `bt full`
  - build without `-g` (for now)

Line Information

- we would like line information for debugging
  - which instruction belongs to which source line
  - then we can show where in the C code we are
- line-stepping becomes possible with this info
  - for a simple compiler, this is not hard to track

Ex 7.2

- out of the box, compilers do not emit debug info
- build the program from 7.1 with `-g`
  - this tells `gcc` or `clang` to emit debug info
  - it contains (among other things) line information
- compare the info in `bt` and `bt full` with 7.1

Local Variables

- function names and source lines were easy
- (local) variables are actually much harder
  - the value of a given variable moves around
  - mostly a fixed address on the stack
  - but gets loaded into registers and altered there

Global Variables

- global variables are usually easier
- they cannot stay in registers across calls
  - the callee would not know where to look for them
  - their stays in registers are usually shorter
- but in general, they are just as hard as locals

Variable Info

- the first thing we need is just a list of variable names
  - this is not reflected in the program (unlike functions)
  - only available in debug metadata
- for each variable, debuginfo can provide its address
  - absolute for globals, frame-relative for locals

Registers

- variables ‘at rest’ live in memory
- but they can move into registers for considerable time
- some variables only ever appear in registers
- which register holds which variable changes in time

Register Info

- possible to solve in theory
  - to each instruction, attach a variable → register map
- the debugger could then look at this map
  - when we say, e.g., `print foo`
  - and read the correct register to get `foo`
- in practice: `$1 = <optimized out>`

Ex 7.3

```
void foo( int a, double b )
{
    int c = a * b;
    c += a / b;
    printf( "foo: %d, %lf, %d\n", a, b, c );
}
```

- compile with `-g`, run in `gdb`, `break foo`
- try `print c` and `print $rbp - (void*)&c`

Function Arguments

- where arguments live is given by a calling convention
- but the machine code does not tell us their names
  - and on some platforms, even their order
- debug info can (and does) provide this information
  - in C, arguments mostly behave like local variables

```c
void foo( int a, double b )
{
    printf( "foo: %d, %lf\n", a, b );
}

void bar( double a, int b ) { /* analogous */ }

int main()
{
    foo( 14, 3.14 );
    bar( 3.14, 14 );
}
```

Ex 7.4

- load up the previous program in `gdb` (no `-g`)
- break on `foo` and `bar`
- print `$rdi` and `$xmm0.v2_double` in both
- compare with `-g`
  - pay attention to the breakpoint notice

Assignment Time

- you can work on your assignment(s)
- ask questions and/or discuss

Part 8: DWARF

Today's Lecture

- documentation is in study materials
- we won't read/write DWARF in C
- instead we'll look with `readelf`
- and decode/interpret things by hand

Debug Format History

- stabs: text-based, 'symbol tables'
  - many incompatible extensions
- COFF: actually an object file format (like ELF)
  - again a number of semi-compatible variants
  - also used in some versions of Windows
- OMF, IEEE-695: similar story

DWARF History

- created in 1988 for SVR4
  - standardized and adopted as DWARF Version 1
- DWARF 2 was never finished
  - proprietary extensions started to appear
- DWARF 3 released in 2005
- DWARF 4 in 2010, with major extensions
- DWARF 5 is the current version

DWARF and ELF

- DWARF is not particularly tied to ELF
  - but they usually appear together
- DWARF data is spread out across multiple sections
  - `.debug_info` contains DIEs
  - `.debug_loc` contains location data
  - `.debug_line` line number information
  - `.debug_str` strings used in other sections

Basic Structure

- block-structured format (for lexical scoping)
  - arranged in a tree
- tree nodes are called DIE
  - short for Debugging Information Entry
  - describe data, data types, subprograms

DIEs

- different types for different data
  - compilation unit
  - data types
  - subprograms, variables
- a list of attributes and children

Compilation Unit DIEs

- usually one source file / object file
- describes what is contained/used in the CU
  - data types
  - global data
  - subprograms

Data Type DIEs

- basic types (`int`, `short`)
  - describes size and encoding
- derived types (pointers, references)
- aggregate types (`struct`, arrays)
  - children: list of members (fields)

Subprogram DIEs

- represent both procedures and functions
  - in C, this is `void` and ‘normal’ functions
- range(s) of memory addresses occupied
- ‘canonical’ frame address (CFA)
- formal parameters, local variables

Canonical Frame Address

- special section: `.debug_frame` or `.eh_frame`
- tells the debugger how to compute CFA
- abstractly, described by a huge table
  - how to compute CFA for each `%rip` value

CFA Encoding

- a bytecode program that generates the table
- each row can contain another small program
  - called a location expression
  - computes the CFA using current register values
  - can branch, look into memory

Variable DIEs

- gives the name and type of the variable
- and a location expression
  - again a small program that can branch
  - it can use the CFA address
  - it can output an address or a register

Line Number Table

- assigns (file, line, column) to each instruction
- encoded (again) as a bytecode program
  - increment the line number
  - jump to a particular file
  - increment the instruction counter

Data Encoding

- most numbers use LEB128 encoding
  - can be signed or unsigned
- a variable-length, base-128 number
  - least-significant digits first
  - each byte is a digit
  - top bit says whether more digits follow

Ex 8.1

- write an empty `main`
- compile with `cc -g`
- check `readelf -w a.out`
  - find the type DIE for `int`
  - find the subprogram DIE for `main`

Ex 8.2

- add a `struct` with 2 integer fields
- create a local variable of this type in `main`
- find DIE for the user-defined data type
- find the DIE for the variable in `main`

Ex 8.3

- use `objdump -xd a.out` to print the program
- cross-reference with the line number program
  - you can use `readelf -wl a.out` to get it
- try to construct the actual line table

Ex 8.4

- get the CFA program with `readelf -wf a.out`
- cross-reference again with disassembly
- notice the exact instructions where the CFA changes
  - pay specific attention to prolog & epilog
  - that is, `push %rbp; mov %rsp,%rbp`

Ex 8.5

- write a decoder for LEB128
- for both signed and unsigned numbers
- see also `dwarf4.pdf` in study materials

Assignment 4: Symtab

- this is an ELF data structure
  - stored in section `.symtab`
  - see `nm a.out` or `readelf -s a.out`
- write a symtab parser
  - we only care about functions

Assignment 4: Invocation

- add a binary called `symtab`
  - `make symtab` should work
- gets a single file name as an argument
- prints the symbol table in `nm` format
  - see `man nm` for details

Assignment 4: Hints

- actual strings are stored in `.strtab`
- `.symtab` is an array of `Elf_Sym` structures
- the `st_info` field is packed
  - use `ELF64_ST_BIND` and `ELF64_ST_TYPE`
  - remember we only care about functions

Part 9: Function calls and frames

Function in Assembly

- start address
- bunch of basic blocks
  - instructions
- where and what are
  - local variables?
  - arguments?
  - returned value?

ABI

- application binary interface
- per pair of OS and CPU architecture
  - size of alligment, data types
  - exceptions
  - format of object files
  - calling convention

Stack Frame

- each function has a frame on the stack
- stack grows downwards
- stack pointer `%rsp`
  - top of the stack
- frame pointer `%rbp`
  - beginning (lowest address) of the frame

Ex 9.1

- try to call `alloca` in a C program
- look into the binary (`objdump`)
- what happens?

Ex 9.2

- write a C function that prints its return address
  - you may need to look into the binary
- hint: think about addresses of local variables

Ex 9.3

- write a C function that rewrites its own return address
    - use an address of another function in the program
    - also try via a buffer overflow (`strcpy`)
- compile without optimizations
- and with `-fno-stack-protector`

Protecting the stack

- `-fstack-protector`, `-fstack-protector-all`
- changing direction of stack growth is insufficient
- canaries
  - terminator
  - random XOR
- non-executable stack

Ex 9.4

- try to compile some code with -fstack-protector-all
- notice the difference
- try to run program from previous exercise
- check what fails and what works

Preserved Registers

- some registers must be preserved across function calls
  - `%rsp`, `%rbp`, `%rbx`, `%r12` - `%r15`
- saved in function prologue
- the rest must be saved by the caller on its frame

Argument Classes

- integer: bool, char, int, long long, pointers ...
- sse: float, double, ...
- x87: long double, ...
- memory: more than four eightbytes, unaligned fields ...
- aggregate types: split into multiple categories by fields

Arguments

- integer: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
- sse: `%xmm0 - 7`
- x87: stack
- memory: stack
- ellipsis(...): `%al` = upper bound of vector registers used

Return

- integer: `%rax`, `%rdx`
- sse: `%xmm0`, `%xmm1`
- x87: `%st0`
- memory: space provided by the caller
  - passed in a hidden first parameter (`%rdi`)

Ex 9.5

- write assembly which calls external functions
  - standard library
  - printf with floats
- write a C function with a complex type
  - try to call it from assembly
  - pass different structures by value

Part 10: Advanced Instructions

Today

- atomic memory access
- `sysenter` / `syscall`
- floating point, AVX, SIMD
- random numbers, timers
- CRC, AES

Atomic Instructions

- perform complex operation in memory
- must be all in a single instruction
- optionally performed atomically
  - no other CPU core can observe intermediate state
  - atomic instructions are ordered

The `lock` Prefix

- tells the CPU to perform an atomic operation
  - single instruction does not mean atomicity
- originally caused the memory bus to be locked
  - currently much more involved than that
- syntax: `lock; opcode...`

Compare & Exchange

- 2 operands: `addr`, `new`
  - `new` must be a register
- read a value from memory at `addr`
- compare the value to `%RAX`, set `ZF`
  - if equal, write `new` to memory @ `addr`
  - else load memory from `addr` into `%RAX`

Spinlock

```
  mov $1, %rdx
retry: # address in %rbx
  mov $0, %rax
  lock; cmpxchg %rdx, (%rbx)
  jne retry
# locked
```

Ex 10.1: Reminder

- implement a `max` function in assembly
  - takes 2 64b integers, returns one
- write a C program to test it
- link and run the executable

Ex 10.2: Using pthreads

- write a C program with 2 threads
- use `pthread_create`
- and `pthread_join`
- print 2 messages in each thread
  - observe the behaviour
  - maybe add `sleep(1)` between them

Ex 10.3: Spinlocks

- implement `spin_lock` and `spin_unlock`
  - in assembly, using 64b `cmpxchg`
- put a critical section around each thread
  - both messages and the sleep inside 1 section
  - a section starts with `spin_lock`
  - and ends with `spin_unlock`

Arithmetic in Memory

- memory operands in `add`, `sub` &c.
- atomic if a `lock` prefix is specified
- usually much faster than a spinlock

Fetch and Add

- also returns the original value
  - unlike a ‘normal’ addition
- `lock; xadd %eax, 0(%rsp)`
  - mnemonic is for exchange and add
  - available as `xadd` on amd64

System Calls

- instruction `sysenter` or `syscall`
  - very similar semantics
  - one comes from Intel, the other from AMD
- switches the CPU into privileged mode
  - jumps into the kernel (fixed address)

SSE, `xmm` registers

- 8 128b registers
- each can hold (since SSE2):
  - four 32b float values
  - two 64b double-precision values
  - four 32b integers
  - eight 16b integers
  - sixteen 8b integers

SSE Operation

- multiple operations in a single instruction
- always the same operation on all values
- 2 operands, rewrites one of the inputs
- packing mode indicated by the opcode

SSE Scalar Arithmetic

- supersedes `x87` instructions
- uses (parts of) the `xmm` registers
  - `x87` had a register stack
- e.g. `mulsd` (multiply double-precision scalars)

AVX, `ymm` registers

- extends the SSE registers to 256b
- adds 8 more registers (total of 16)
- three-operand format (2 operands, result)
- not entirely compatible with SSE
  - needs to switch between SSE and AVX

AVX Integer Ops

- vector add, multiply
- carry-less multiplication
  - multiply 2 64b numbers
  - obtaining a single 128b result
- vector shifts, bitwise operations
- conditional/masked loads and stores

AVX-512, `zmm`

- further doubles the register file
  - doubles width to 512b
  - doubles count to 32
- fused multiply-add: $a + b \cdot c$
- dot products

AVX-512 Ternary Logic

- 3 vector register operands (`?mmN`)
- bitwise operation on all the bits
- an 8-bit immediate encoding the operation
  - arbitrary bitwise operation
  - encodes the boolean truth table

Randomness

- `rdrand` stores a random number in a register
- `rdseed` obtains entropy (into a register)
  - useful for seeding software PRNGs
- `rdrand` is much faster
  - produces cryptographic-quality numbers

Timers

- each CPU core has a local timer
- those timers are not synchronised
- `rdtsc` stores its current value
  - result in `edx:eax` (clobbers both)
- mainly useful for benchmarking
  - and for timing side-channel attacks

CRC32

- implements cyclic redudancy check
- polynomial division in hardware
  - but only with a fixed divisor
  - much faster than software implementation
- added as part of SSE4

AES

- a fairly complicated block cipher
  - runs in multiple rounds
- each round = 1 `aesenc` (or `aesdec`)
  - 128b operands stored in `xmm` registers
  - last round uses `aesenclast`
- also speeds up round key generation
  - instruction `aeskeygenassist`

Assignment 5: Invocation

- add a `make recfun` target
- the input is an ELF file
  - specified like this: `./recfun a.out`
- start disassembling at the entry point
  - this is part of the file header

Assignment 5: Requirements

- recursively disassemble jump/call targets
  - detect jumps into middle of an instruction
  - print `# [broken]` instead of a label
- identify basic blocks
- identify functions
  - assume each BB belongs to 1 function

Assignment 5: Functions

- assume 1 entry BB per function
- use symtab names if possible
  - look up the address of the entry label
  - use `sub_100f02` otherwise
  - `100f02` is the address of the entry label

Assignment 5: Output

- like `decode.elf` but with function info
- give names to basic blocks (labels)
  - `<fun>_<id>` where
  - `<fun>` is the function name (see previous slide)
  - `<id>` is either `entry` or a number

Part 11: Debuggers

Breakpoints

- stops execution at a given instruction
- can be set manually or automatically
  - to implement e.g. instruction stepping
  - or line stepping

Software Breakpoints

- remember the `int3` instruction?
  - it traps – can divert control
  - conveniently encoded as a single byte
- temporarily rewrite the address with `0xCC`
  - swap it back before executing the address

Hardware Breakpoints

- addresses held in CPU registers
    - `DR0-3` on `x86`
- stop on different access types
    - read, read+write, execute
- virtually no overhead

Software vs Hardware

- SW can only detect execution of an address
  - but not a read or a write
- you can have unlimited SW breakpoints
  - but only 4 hardware (on x86)

Stepping with Breakpoints

- instructions: use a temporary breakpoint
  - when it triggers, move it forward
- how about source lines?
  - either set breakpoints at all exits
  - or just use instruction stepping

`ptrace`

- Process Trace (a system call)
- allows one process to trace another process
  - observe and control execution
  - examine and change the memory and registers

ptrace (cont'd)

- one function to do everything
  - declared in sys/ptrace.h

```
long ptrace( enum __ptrace_request req, pid_t pid,
             void *addr, void *data )
```

- first argument specifies the action
  - remainder is interpreted depending on the action

Tracing

- tracee is always only one thread
- `fork()` with `PTRACE_TRACEME`
  - request is done by tracee
  - special ptrace flag is set
  - control transfered to parent after execve

Tracing

- PTRACE_ATTACH
  - start tracing specified pid
  - ptrace flag is set, SIGSTOP the tracee
- PTRACE_DETACH
  - stop tracing
  - tracer may kill the tracee

When to Notice the Tracer?

- every time syscall is executed
  - `PTRACE_SYSCALL`
- continue the tracee
  - `PTRACE_CONT`
- stepping every instruction
  - `PTRACE_SINGLESTEP`

Examine Registers

- `PTRACE_GETREGS`
  - read registers of the tracee
  - `sys/reg.h` macros with register offsets
  - `struct user_regs_struct` from `sys/user.h`
- `PTRACE_SETREGS`
  - changes the registers of the tracee

Examine Memory

- read the memory of the tracee
  - `PTRACE_PEEK*` family
- set the memory of the tracee
  - `PTRACE_POKE*` family

Exercise 10.1

- write a program that forks
- use `ptrace` to attach to the child
- print every syscall the child performs
  - use `ptrace` to do this
  - the child can e.g. open & read a file

Part 12: Decompilation Basics

Compilation

- source code - C++, Rust, ...
- LLVM IR
- object file
- ELF or other binary format
  - platform dependent

Aside: LLVM IR

- intermediate representation used by compilers
  - partial SSA
- assembly-like, but:
  - virtual registers
  - simple type system
  - functions made of basic blocks

Motivation for Decompiling

- analysis
- LLVM passes, security patches
- sometimes there is no source code available
- binary is actually being executed

Binary

- code sections: `.text`
  - functions
  - blocks
  - instructions
- data sections: `.data`, `.rodata`, ...
- `.eh_frame`, `.debug_*` and others

McSema

- tool to ‘lift’ binaries into LLVM
- two phases
  - recovery of information about binary
  - actual decompilation / lifting

Disassembly

- external disassembler to retrieve information
- functions, sections, externals
- references
  - important and tricky
  - instructions
  - data sections

Lifting

- simulation of the original code
- state structure with all registers
- sections lifted as global variables
  - the data is (almost) preserved
  - references replaced by pointers

Remill

- lifts single instructions
- provides semantic functions
  - must be implemented for every instruction
- cannot lift entire binary alone

External Calls

- how to call externals?
    - they do not share the lifted state
- synchronization with native cpu state (using asm)
- execute the external
- synchronize back (using asm)
- works fine for recompilation

Analysis mode

- everything needs to be done in the IR
- state as global variable
- stack as huge global array of bytes
- external calls
  - call reconstruction using ABI