

T-110.6220

Emulators and disassemblers

Jarkko Turkulainen, F-Secure Corporation

F-SECURE®



BE SURE.

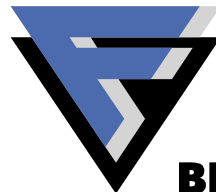
Agenda



- Disassemblers
 - What is disassembly?
 - What makes up an instruction?
 - How disassemblers work
 - Use of disassembly
 - In reverse engineering
 - In anti-virus engine
- Emulators
 - Different types of emulators
 - How emulators work
 - Use of emulators
 - In reverse engineering
 - In anti-virus engine

Disassemblers

F-SECURE[®]



BE SURE.

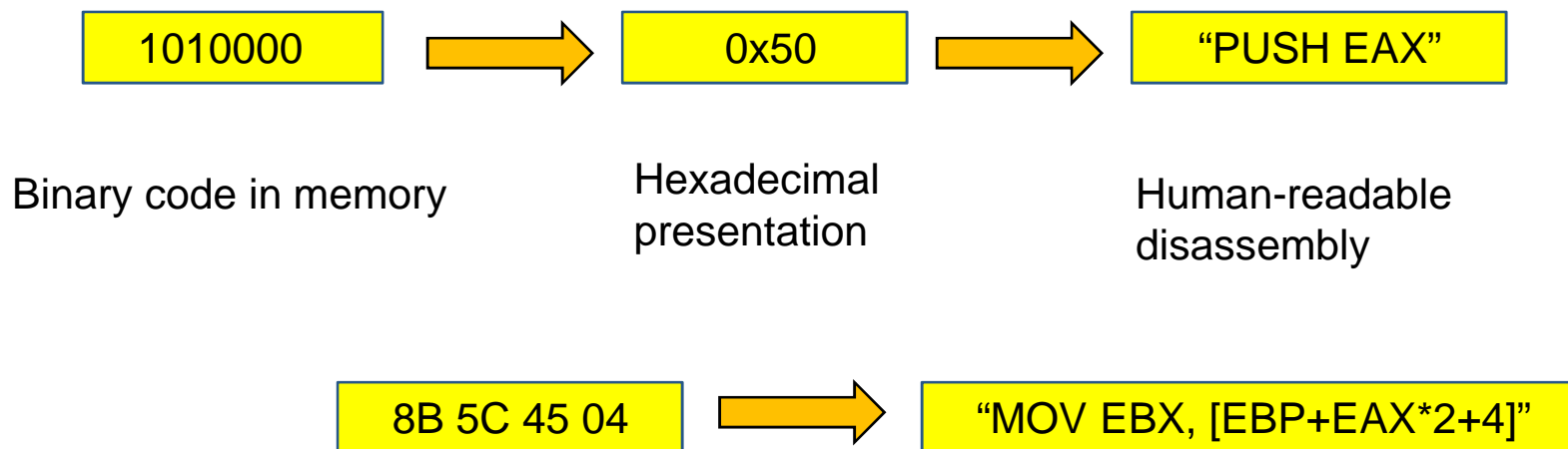
What exactly is disassembly?

- Machine code presented in a human-readable form
- Usually presented in formal language, the assembly language of the target platform
- Wikipedia definition of disassembler: *A disassembler is a computer program that translates machine language into assembly language — the inverse operation to that of an assembler*

Machine code



- Computers handle code in binary format
- Intel x86 example:



Intel x86 instruction format

- Variable-size complex instruction format (CISC)
- Instruction format: OPERATION, [OPERAND 1, OPERAND 2, OPERAND 3]

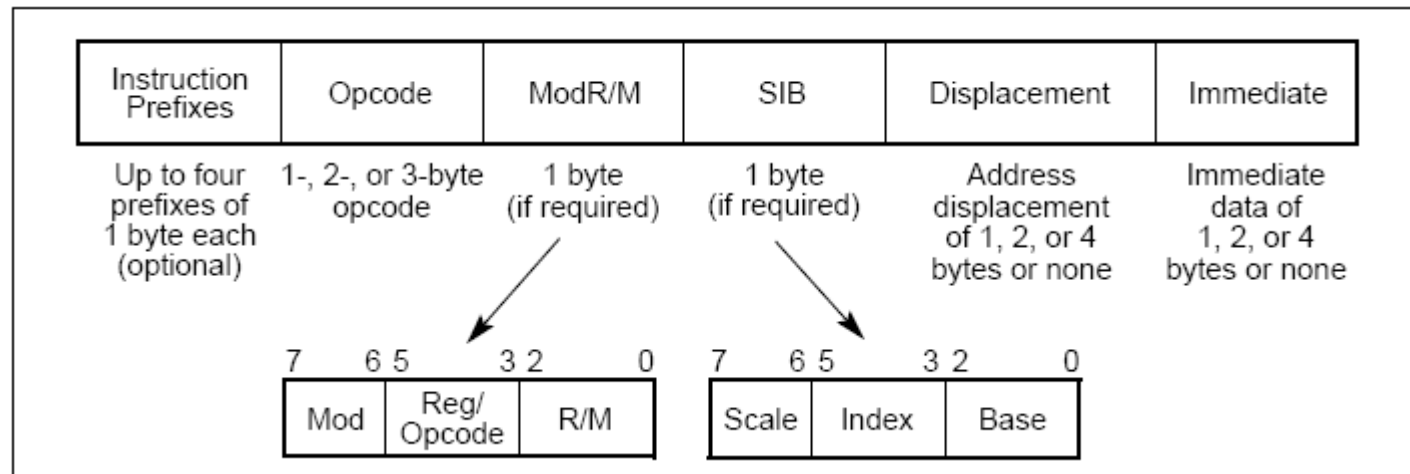


Image Copyright Intel Corporation

- Single instruction can be of any size from 1 byte up to 15 bytes (compare to RISC, where instruction size is constant, for example 4 bytes)!!

Instruction prefix bytes

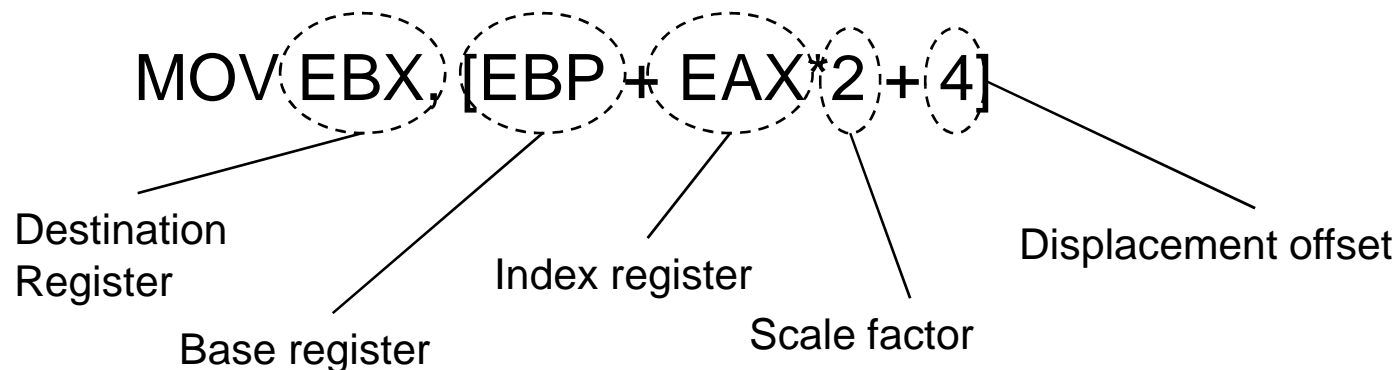
- Group 1: lock and repeat prefixes
 - Lock ensures exclusive use of memory in multiple-processor environments
 - Repeat is used in string operations (sort of a loop)
- Group 2: Segment overrides or branch hint
 - Segment in memory access is implied by the instruction, but can be overridden using Group 2 prefix.
 - Some bytes in this group indicates a branch hint, if the instruction is conditional jump
- Group 3: Operand size override
 - Switch between 16 –and 32-bit operand size
- Group 4: Address size override
 - Switch between 16 –and 32-bit addressing

Instruction opcode byte(s)

- Byte(s) that presents the actual operation (MOV, INC, PUSH, etc.)
- Originally only single byte, but later specifications define 1-3 bytes
- First byte can be any opcode in range 0-256, or escape byte indicating that another opcode byte follows
- Second opcode byte defines an opcode after escape byte
- Third opcode is a prefix to some new multimedia instructions
- Some opcodes imply the registers used, others need more information (MODRM/SIB)
- About 1200 different opcodes (!!!!)

MODRM and SIB

- These bytes define the registers and other data used in the instruction
- MODRM defines the registers used for addressing the memory or register
- Sometimes addressing is more complex than a simple register can offer
- SIB byte is used for complex memory addressing: indexing and scaling
- Example of complex addressing:



32-bit MODRM table

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX MM0 XMM0 0 000	CL CX MM1 XMM1 1 001	DL DX MM2 XMM2 2 010	BL BX MM3 XMM3 3 011	AH SP MM4 XMM4 4 100	CH BP MM5 XMM5 5 101	DH SI MM6 XMM6 6 110	BH DI MM7 XMM7 7 111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[EAX] [ECX] [EDX] [EBX] [-][-] ¹ disp32 ² [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F
[EAX]+disp8 ³ [ECX]+disp8 [EDX]+disp8 [EBX]+disp8 [-][-]+disp8 [EBP]+disp8 [ESI]+disp8 [EDI]+disp8	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F
[EAX]+disp32 [ECX]+disp32 [EDX]+disp32 [EBX]+disp32 [-][-]+disp32 [EBP]+disp32 [ESI]+disp32 [EDI]+disp32	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AH/MM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

- 1) SIB follows the MODRM byte
- 2) 32-bit displacement
- 3) 8-bit displacement

Image Copyright Intel Corporation

32-bit SIB table

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	SS	Index	Value of SIB Byte (in Hexadecimal)							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	99	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

Image Copyright Intel Corporation

Displacement and immediate data



- Some complex addressing forms require offset within the memory reference, called displacement
- Displacement follows immediately the MODRM/SIB bytes (1, 2 or 4 bytes)
- Some instructions use immediate data as operand value (1, 2 or 4 bytes)

Example instructions

- 1-byte instructions (only opcode):
 - 90 – **NOP**
 - 50 – **PUSH EAX**
 - C3 – **INT3**
 - CC – **RET**
- A bit longer instructions:
 - 89 D8 – **MOV EAX, EBX**
 - 31 FE – **XOR ESI, EDI**
 - 80 C1 11 – **ADD CL, 0x11**
- Monster instruction (not exactly valid):
 - F3 36 C7 84 D8 44 33 22 11 88 77 66 55 –
REP MOV [SS:EAX + EBX*8 + 0x11223344], 0x55667788 (13 bytes)

Closer look at the monster instruction

- Assembly presentation: **REP MOV [SS:EAX + EBX*8 + 0x11223344], 0x55667788**
- Data in hex:

F3	36	C7	84	D8	44 33 22 11	88 77 66 55
----	----	----	----	----	-------------	-------------

- F3 – repetition prefix (REP)
- 36 – segment override (SS)
- C7 – opcode (MOV)
- 84, D8 – MORDM, SIB (base register EAX, index register EBX, scale factor 8)
- 44 33 22 11 – displacement data (0x11223344)
- 88 77 66 55 – immediate data (0x55667788)

How disassemblers work?

- Basically, disassembler is a software implementation of the CPU instruction decoder
- Reads a data and decodes it as a stream of instructions, based on specifications
- Prefixes are parsed first
- Instruction opcodes are typically used as an index to the actual instruction tables
- Operands are parsed based on opcode(s), MODRM/SIB, displacement and immediate data

Disassembler pseudo-code



```
// Instruction table (only opcode mnemonics)
char *mnemonics[256] = { "ADD", "ADD", "ADD", "ADD", "ADD", "ADD", "PUSH", "POP", ... };

Disassemble(unsigned char *code, int size)
{
    int i, j;
    unsigned char opcode;

    for (i = 0; i < size;)
    {
        j = get_opcode_index(code + i);          // Parse prefixes, return index to opcode byte
        opcode = *(code + i + j);              // Read the opcode byte
        printf("Mnemonic: %s\n", mnemonics[opcode]);
        i += get_instruction_size(code + i);    // Advance to next instruction
    }
}
```

Use of disassembly in malware analysis



- Makes binary code readable
- Helps in distinguishing code from data (still not very obvious always)
- Good disassembly can also be used for assembling back the code
- Reverse engineering tools offer very detailed disassembly (IDA, OllyDbg, etc.)

Disassembly in anti-virus engines



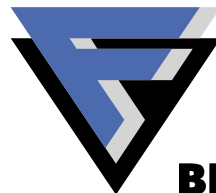
- Can help in determining where to start scanning
 - Follow the code flow statically
- Can also be used for creating signatures
 - Base the detection on abstract presentation of the data (= the disassembly) instead of raw bytes of the data
 - May help in simple forms of code obfuscation

Example:

```
[code starts here]          ; Static code  
  
jmp code_below              ; Jump over the random data  
[random junk]               ; This data is random, it cannot be used for signatures  
code_below:  
[code continues here]       ; Static code continues
```

Emulators

F-SECURE®



BE SURE.

What is emulator?



- From Wikipedia: *An emulator duplicates (provides an emulation of) the functions of one system using a different system, so that the second system behaves like (and appears to be) the first system.*
- In some cases, emulation of the identical platform is used (for example: emulation of x86 on x86)

Emulator types



- Emulation methods
 - Instruction interpretation (example: Bochs)
 - Instruction translation (example: Qemu, Valgrind, Java, ...)
 - Code virtualization (example: VMWare)
- Emulation “depth”
 - Full system emulation (Bochs, Qemu, VMWare)
 - Partial system emulation (Valgrind, Qemu, anti-virus engines)

Instruction interpretation – fetch-decode-execute -loop



- Instructions are decoded from the code stream, usually one at the time
- The instruction is emulated by executing an equivalent operation on the host environment

```
Emulate()
{
    Instruction *i;
    while (true)
    {
        i = decode_instruction(EIP);
        switch (i->opcode)
        {
            case OP_MOV:
                EIP = emulate_mov(i);
                break;
            // Emulate all other opcodes
        }
    }
}
```

Instruction translation

- AKA dynamic code translation (DCT) or just-in-time compilation (JIT)
- Operates usually on basic block-basis, translates the block to host CPU environment
- Translated block is then executed directly on CPU and cached for future use
- Implementation doesn't usually require kernel-level modifications (drivers)
- If the target and host platforms are the same, translation usually involves just translation of memory references and emulating hardware-specific events, like interrupts, exceptions etc.
- Translated code is obviously targeted for host platform, but translator itself can be portable.

Basic block

- **Basic block:** series of instructions that is terminated (usually) by branch instruction. Each instruction in the block modifies the instruction pointer always to the next instruction within the block. Example:

```
push ebp
mov ebp, esp
sub esp, 0x20
cmp [ebp+4], 0x2
jnz next_location
mov ebx, [ebp+8]
xor esi, esi
...
```

Block of code



```
push ebp
mov ebp, esp
sub esp, 0x20
cmp [ebp+4], 0x2
```

Basic block to be translated

Dynamic translation pseudo-code



```
Emulate()
{
    Block *block;

    while (true)
    {
        block = check_block_cache(EIP);
        if (block == NULL)
        {
            // Cache miss, generate new one:
            block = generate_new_block(EIP);
        }
        // Execute the block, proceed to next block
        EIP = block->execute();
    }
}
```

More on code translation



- Code is translated on-demand – only when it is discovered for the first time.
- Translated code can be optimized by translating it in two rounds – first phase is the translation to intermediate format, second time is the optimization of the intermediate presentation (obviously not needed when platforms are the same).
- Branch instructions are handled as control transfers to translated and cached blocks.
- Code translation is more efficient than interpretation because it eliminates the need for instruction fetch-decode-execute –loop, with the price of heavy translation process.
- Code translation works because most code is run several times:
 - Single process instance usually executes blocks several times during its lifetime (loops etc.)
 - Same blocks of code are reused system-wide all the time (libraries, process modules etc.)

Code virtualization



- Virtualization is tied to identical host and target platforms.
- Most (or all) code is run directly on isolated hardware environment
- Hardware resources are virtualized
- Can be done using software solutions (VMWare) or using hardware features (new Intel/AMD processors, IBM z/VM)
- Requires support from the OS, or kernel-level modifications (drivers)

How virtualization works?

- In virtualization, code is not (usually) analyzed or cached, it is just run in isolated environment.
- Isolation usually tries to make the virtual memory transparent to isolated system – thus eliminating the need for dynamic address translation.
- Hardware is usually emulated on I/O-instruction interface.

Emulator uses in malware analysis



- Emulators can be used to run malware in isolation
 - Much more secure than analyzing the malware in production environment
 - Still risky: malware can break out from the emulator via a bug etc.
- Fast restoring of known clean state
 - Example: VMWare snapshots

Emulators in anti-virus engine

- Exact execution paths
 - Static disassembly cannot follow all branch instruction. Examples: “CALL [EAX], JNZ, RET”
- Decryption of malware
 - Sometimes it is faster to allow malware to decrypt itself inside the emulator instead of writing complex decryption routine.
- Generic unpacking
 - Similar idea as malware decryption: let the packer stub to decode itself inside the emulator
- Behavioural analysis
 - Let the malware run inside emulator and see what it is doing (“sandboxing”, see the Resources)
- Further reading: chapters 11.4 and 11.13 from “The Art of Virus Research and Defence “

Attacks on emulators



- Malware can try to detect the presence of emulator
 - Lots of known ways to detect emulators
 - Some emulators might leave visible traces in the system (for example: VMWare control port)
 - Emulators might not be able to emulate everything or the emulation might be incorrect
 - Real system might also have a bug that is not present in the emulator!
 - If emulator is detected, malware can refuse to run or modify its behavior
- Malware can try to break the emulator by exploiting its features or bugs
 - It can execute heavy calculations that are fast enough on real machine
 - It can try to break out from the emulator by exploiting a bug

Resources



- Intel IA-32 Software Developer's Manual - <http://www.intel.com/products/processor/manuals/>
- NASM, the netwide assembler - <http://nasm.sourceforge.net/>
- The Art of Virus Research and Defence (P. Szor)
- Bochs - <http://bochs.sourceforge.net/>
- Qemu - <http://fabrice.bellard.free.fr/qemu/>
- Valgrind - <http://valgrind.org/>
- VMWare – <http://www.vmware.com/>
- "Sandbox Technology Inside AV Scanners" (K. Natvig, Virus Bulletin Conference 2001)

**BE
SURE.**

F-SECURE®

