Being a Better Reverse Engineer

With Pictures

by Kyle Martin

whoami

A Computer Engineer at NYU

A Reverse Engineer at Large

An Intern at Trail of Bits and Vector35

The Vice President and Head of Research at the OSIRIS Lab

Cyber Security Researcher

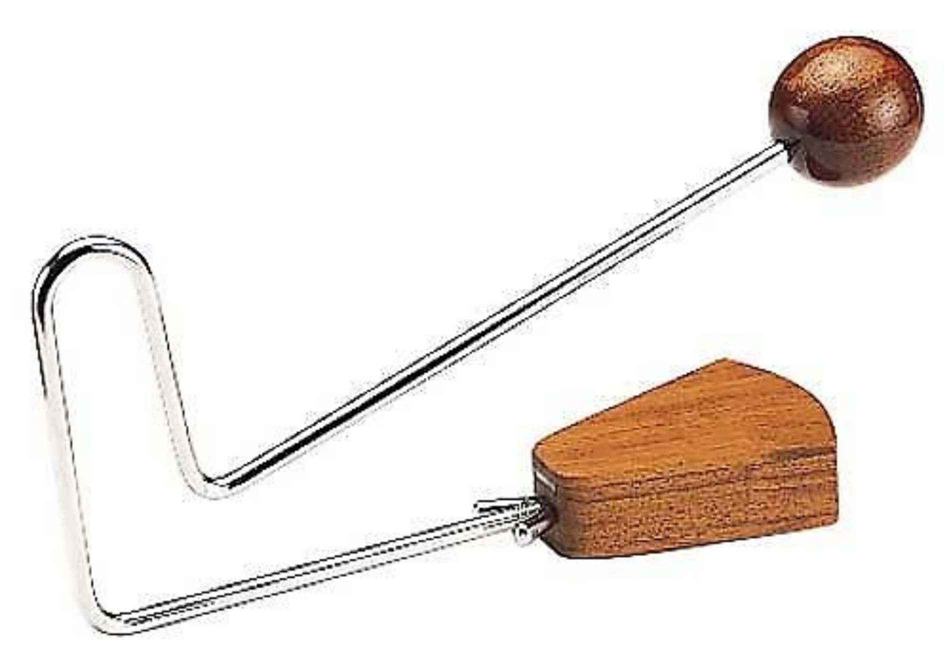
CTF Challenge Writer

Insane

And more

Let’s Start

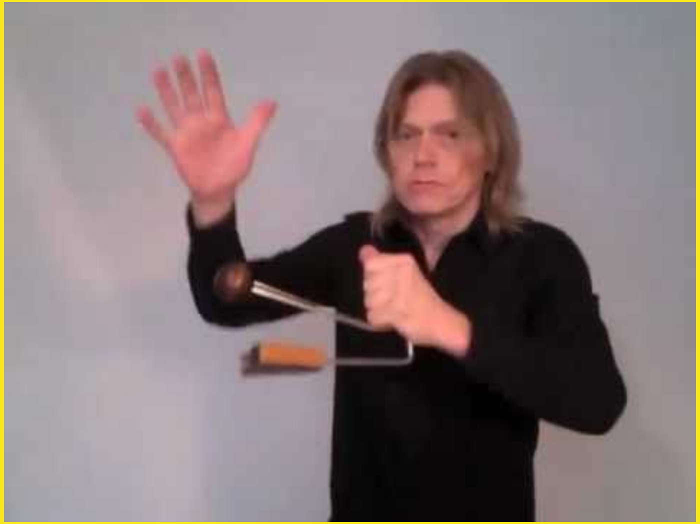
What is this?



Let’s Get Some Hints

Hint One

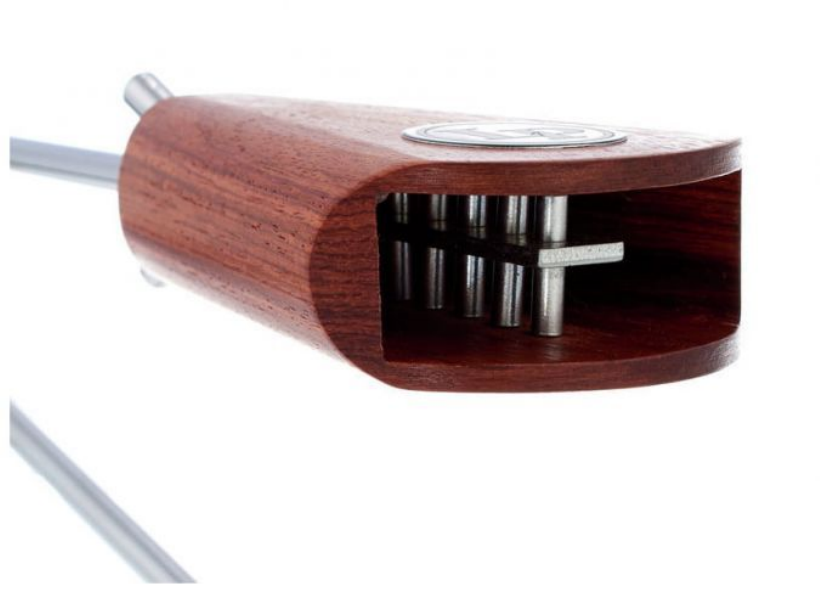
Does this help? (Usage?)



Hint Two

Let’s look inside….

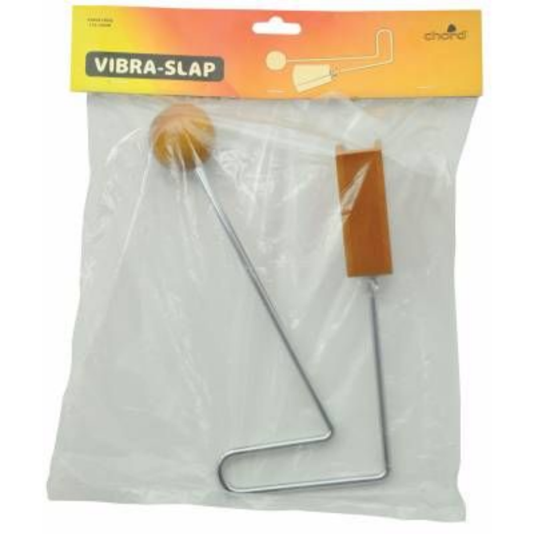
(How it works?)



Hint Three

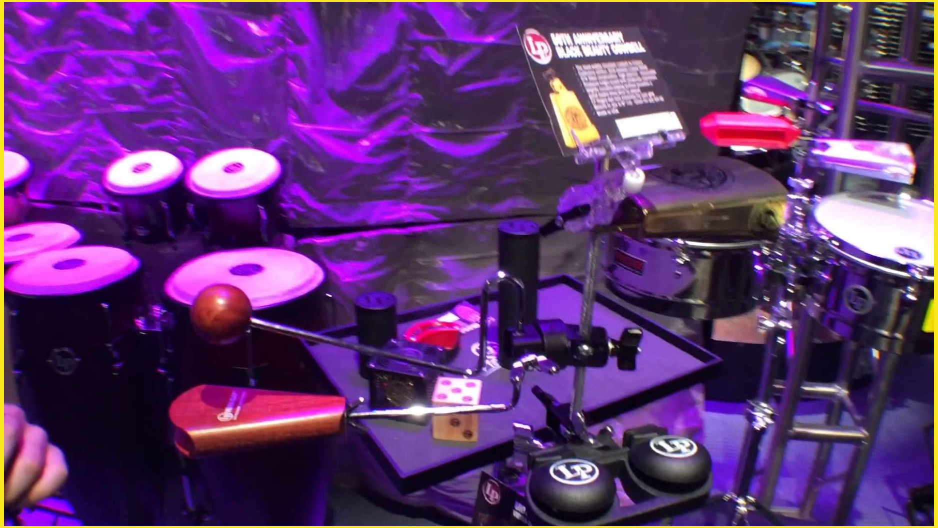
What does it do?

(What it’s called?)

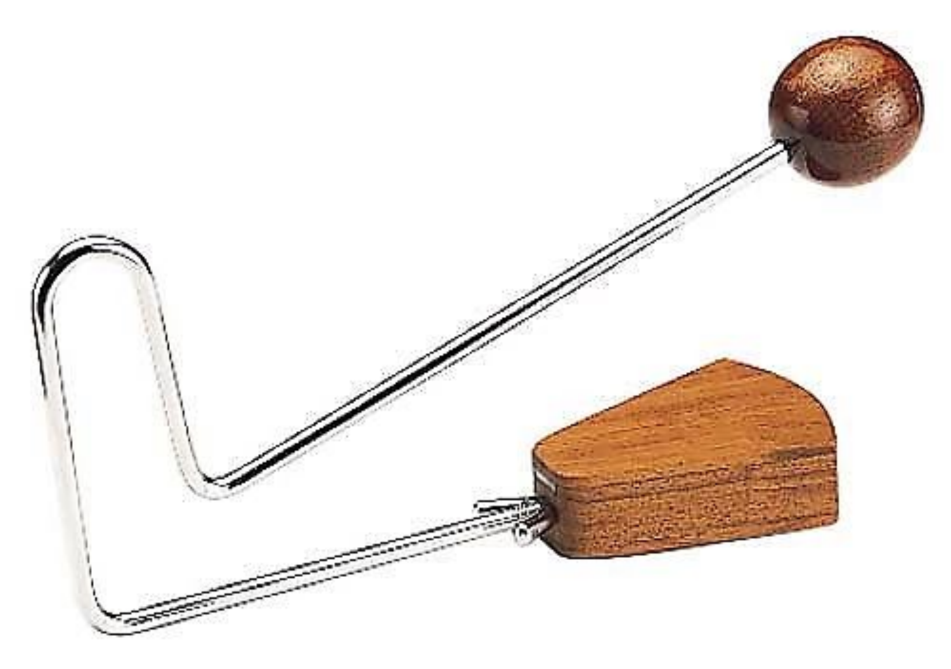


Hint Four

Where does it belong?



What is this?



Here It Is

What Worked Best?

- Hint One
  - Usage
- Hint Two
  - Internals
- Hint Three
  - What it’s called
- Hint Four
  - Context
- Video
  - Putting it all together

What’s The End Goal?

Understand:

- What *it* does?
- How *it* works?
- How *it* is used?
- How *it* was made?
- Why *it* was made?
- ...how *it* can be broken?

But to these ends, we need to know as much as possible about *it*.

How Do Learn About *It* Quickly?

Build Some Tooling For *It*.

Runbooks/Guides (Crime scenes, disaster recovery, forensics)

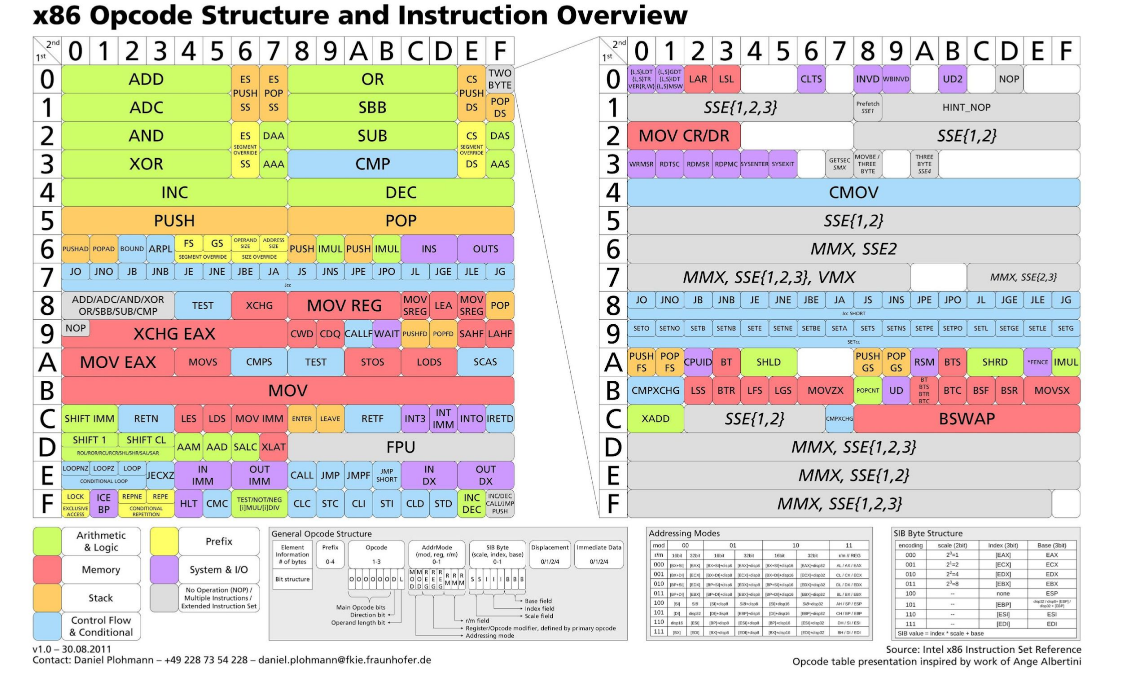
Policy (What to/not to do; Criminal investigation, penetration testing)

Analysis Tools (Crime - DNA, data matching, categorization, last week's talk)

Something Else

Applying This To The *It* We Care About Here

x86



x86 Opcode Structure and Instruction Overview

2nd / 1st	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD						ES PUSH	ES POP	OR						CS PUSH	TWO BYTE
1	ADC						SS PUSH	SS POP	SBB						DS PUSH	POP DS
2	AND						ES SEGMENT OVERRIDE	DAA	SUB						CS SEGMENT OVERRIDE	DAS
3	XOR						SS	AAA	CMP						DS	AAS
4	INC								DEC							
5	PUSH								POP							
6	PUSHAD	POPAD	BOUND	ARPL	FS	GS	OPERAND SIZE	ADDRESS SIZE	PUSH	IMUL	PUSH	IMUL	INS		OUTS	
					SEGMENT OVERRIDE		SIZE OVERRIDE									
7	JO	JNO	JB	JNB	JE	JNE	JBE	JA	JS	JNS	JPE	JPO	JL	JGE	JLE	JG
	Jcc															
8	ADD/ADC/AND/XOR OR/SBB/SUB/CMP				TEST		XCHG		MOV REG				MOV SREG	LEA	MOV SREG	POP
9	NOP	XCHG EAX							CWD	CDQ	CALLF	WAIT	PUSHFD	POPFD	SAHF	LAHF
A	MOV EAX				MOVS		CMPS		TEST		STOS		LODS		SCAS	
B	MOV															
C	SHIFT IMM		RETN		LES	LDS	MOV IMM		ENTER	LEAVE	RETF		INT3	INT IMM	INTO	IRETD
D	SHIFT 1		SHIFT CL		AAM	AAD	SALC	XLAT	FPU							
	ROL/ROR/RCL/RCR/SHL/SHR/SAL/SAR															
E	LOOPNZ	LOOPZ	LOOP	JECXZ	IN IMM		OUT IMM		CALL	JMP	JMPF	JMP SHORT	IN DX		OUT DX	
	CONDITIONAL LOOP															
F	LOCK	ICE BP	REPNE	REPE	HLT	CMC	TEST/NOT/NEG [i]MUL/[i]DIV		CLC	STC	CLI	STI	CLD	STD	INC DEC	INC/DEC CALL/JMP PUSH
	EXCLUSIVE ACCESS		CONDITIONAL REPETITION													

2nd / 1st	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	{L,S}LDT {L,S}TR VER{R,W}	{L,S}GDT {L,S}IDT {L,S}MSW	LAR	LSL			CLTS		INVD	WBINVD		UD2		NOP		
1	*SSE{1,2,3}*								Prefetch *SSE1*	HINT_NOP						
2	MOV CR/DR								*SSE{1,2}*							
3	WRMSR	RDTSC	RDMSR	RDPMC	SYSENTER	SYSEXIT		GETSEC *SMX*	MOVBE / THREE BYTE		THREE BYTE *SSE4*					
4	CMOV															
5	*SSE{1,2}*															
6	*MMX, SSE2*															
7	*MMX, SSE{1,2,3}, VMX*												*MMX, SSE{2,3}*			
8	JO	JNO	JB	JNB	JE	JNE	JBE	JA	JS	JNS	JPE	JPO	JL	JGE	JLE	JG
	Jcc SHORT															
9	SETO	SETNO	SETB	SETNB	SETE	SETNE	SETBE	SETA	SETS	SETNS	SETPE	SETPO	SETL	SETGE	SETLE	SETG
	SETcc															
A	PUSH FS	POP FS	CPUID	BT	SHLD				PUSH GS	POP GS	RSM	BTS	SHRD		*FENCE	IMUL
B	CMPXCHG		LSS	BTR	LFS	LGS	MOVZX		POPCNT	UD	BT BTS BTR BTC	BTC	BSF	BSR	MOVSX	
C	XADD		*SSE{1,2}*					CMPXCHG	BSWAP							
D	*MMX, SSE{1,2,3}*															
E	*MMX, SSE{1,2}*															
F	*MMX, SSE{1,2,3}*															

🟩	Arithmetic & Logic	🟨	Prefix
🟥	Memory	🟪	System & I/O
🟧	Stack	⬜	No Operation (NOP) / Multiple Instructions / Extended Instruction Set
🟦	Control Flow & Conditional		

General Opcode Structure

Element Information # of bytes	Prefix	Opcode	AddrMode (mod, reg, r/m)	SIB Byte (scale, index, base)	Displacement	Immediate Data
	0-4	1-3	0-1	0-1	0/1/2/4	0/1/2/4
Bit structure		O O O O O O D L	MOD MOD REG REG REG RM RM RM	S S I I I B B B		

Main Opcode bits
Direction bit
Operand length bit
r/m field
Register/Opcode modifier, defined by primary opcode
Addressing mode
Base field
Index field
Scale field

Addressing Modes

mod	00		01		10		11
r/m	16bit	32bit	16bit	32bit	16bit	32bit	r/m // REG
000	[BX+SI]	[EAX]	[BX+SI]+disp8	[EAX]+disp8	[BX+SI]+disp16	[EAX]+disp32	AL / AX / EAX
001	[BX+DI]	[ECX]	[BX+DI]+disp8	[ECX]+disp8	[BX+DI]+disp16	[ECX]+disp32	CL / CX / ECX
010	[BP+SI]	[EDX]	[BP+SI]+disp8	[EDX]+disp8	[BP+SI]+disp16	[EDX]+disp32	DL / DX / EDX
011	[BP+DI]	[EBX]	[BP+DI]+disp8	[EBX]+disp8	[BP+DI]+disp16	[EBX]+disp32	BL / BX / EBX
100	[SI]	*SIB*	[SI]+disp8	*SIB*+disp8	[SI]+disp16	*SIB*+disp32	AH / SP / ESP
101	[DI]	disp32	[DI]+disp8	[EBP]+disp8	[DI]+disp16	[EBP]+disp32	CH / BP / EBP
110	disp16	[ESI]	[BP]+disp8	[ESI]+disp8	[BP]+disp16	[ESI]+disp32	DH / SI / ESI
111	[BX]	[EDI]	[BX]+disp8	[EDI]+disp8	[BX]+disp16	[EDI]+disp32	BH / DI / EDI

SIB Byte Structure

encoding	scale (2bit)	Index (3bit)	Base (3bit)
000	$2^0=1$	[EAX]	EAX
001	$2^1=2$	[ECX]	ECX
010	$2^2=4$	[EDX]	EDX
011	$2^3=8$	[EBX]	EBX
100	--	none	ESP
101	--	[EBP]	disp32 / disp8+ [EBP] / disp32 + [EBP]
110	--	[ESI]	ESI
111	--	[EDI]	EDI
SIB value = index * scale + base			

v1.0 – 30.08.2011
Contact: Daniel Plohmann – +49 228 73 54 228 – daniel.plohmann@fkie.fraunhofer.de

Source: Intel x86 Instruction Set Reference
Opcode table presentation inspired by work of Ange Albertini

Reduced Instruction Set Computing

Turing Complete

A system that can be used to solve any computation problem

Simple Turing Completeness?

Break into fundamental operations:

Math : Add

Control-Flow : Jump if ____?

Ultimate Reduced Instruction Set Computing

One instruction:

Subtract and branch if less than or equal to zero

jmp c:

```
subleq Z, Z, c
```

Ultimate Reduced Instruction Set Computing

One instruction:

Subtract and branch if less than or equal to zero (src, dest)

mov a, b:

1. subleq b, b
2. subleq a, Z
3. subleq Z, b
4. subleq Z, Z

The Most Ultimate Reduced Instruction Set Computing

And because x86 is bad and Intel/AMD are monsters:

The ZERO-INSTRUCTION turing complete compiler.

- x86’s fault handling is turing complete. Wonderful.

But is any of this clear?

No.

No *It's* Not.

What’s the problem?

1Sn’7 iT 3Asy???

How do we fix this?

Less Ultimate-RISC

More instructions that perform simple tasks

New Language

14 Operations:

- Add, sub, mul, div
- Xor, And, Or, Not
- Jump If Less Than, Jump If Greater Than, Jump If Equal, Jump
- Mov
- Interrupt - to interact with the rest of the system

Did this help?

14 instructions:

add, sub, mul, div, xor, or, and, not, jl, jg, je, jmp, mov, int; (src, dest)

jmp c:

jmp c

mov a, b:

mov a, b

Back to x86

Over 1700 different instructions

Over 7400 total different forms those instructions can take

- mov reg, reg; mov mem, reg; mov reg, mem; etc

Much undocumented behavior

Intermediate Representations

A Turing Complete system can be used to solve any computation problem

- We can represent anything in x86 in our Turing Complete System

Example: Jump if greater or equal

1. jl end
2. jmp dest
3. end:

Reinventing The Wheel

Tons of Intermediate Representations (IRs) already Exist:

- BinaryNinja’s Low Level and Medium Level Intermediate Representations
- LLVM
- GNU RTL
- CIL
- SIL
- PCODE
- And tons more...

What have we done?

Become faster learners:

- We don’t need to know what every instruction does
  - DO NOT READ EVERY INSTRUCTION (as a human)
- Let the tool do the work, use the IR’s to build most your understanding

Obtained the facts

What Do We Do With The Facts?

Remember, What Is The End Goal?

Understand:

- What *it* does?
- How *it* works?
- How *it* is used?
- How *it* was made?
- Why *it* was made?
- ...how *it* can be broken?

And now we are equipped with the fundamental facts about *it* (the local ‘what’).

What Is Important?

Generally, in software:

- How our data is manipulated
  - (How input is manipulated to generate output)
  - Input => Authentication
  - Input => Goods (Amazon?)
  - Input => Services (Amazon?)

Work To Be Done

You, the reverse engineer, need to

- Stop wasting your time
    - Limit your scope
    - Don’t read everything
    - Don’t reinvent the wheel
    - Recognize the benefit and downfalls of different tools you’re using
    - Use the right tools for the right projects
- Get to what’s most important quickly
    - Determine what’s important
    - Obtain that information
    - There’s still work to be done here (this is the hard part)

Additional Resources/Bibliography

https://en.wikipedia.org/wiki/Turing_completeness

https://softwareengineering.stackexchange.com/questions/132385/what-makes-a-language-turing-complete

http://cs.lmu.edu/~ray/notes/ir/

https://net.cs.uni-bonn.de/fileadmin/user_upload/plohmann/x86_opcode_structure_and_instruction_overview.pdf

Being a Better Reverse Engineer

by Kyle Martin
elyk@nyu.edu
GitHub.com/KyleMiles
KyleMiles.me
@ElykDeer
@Elyk