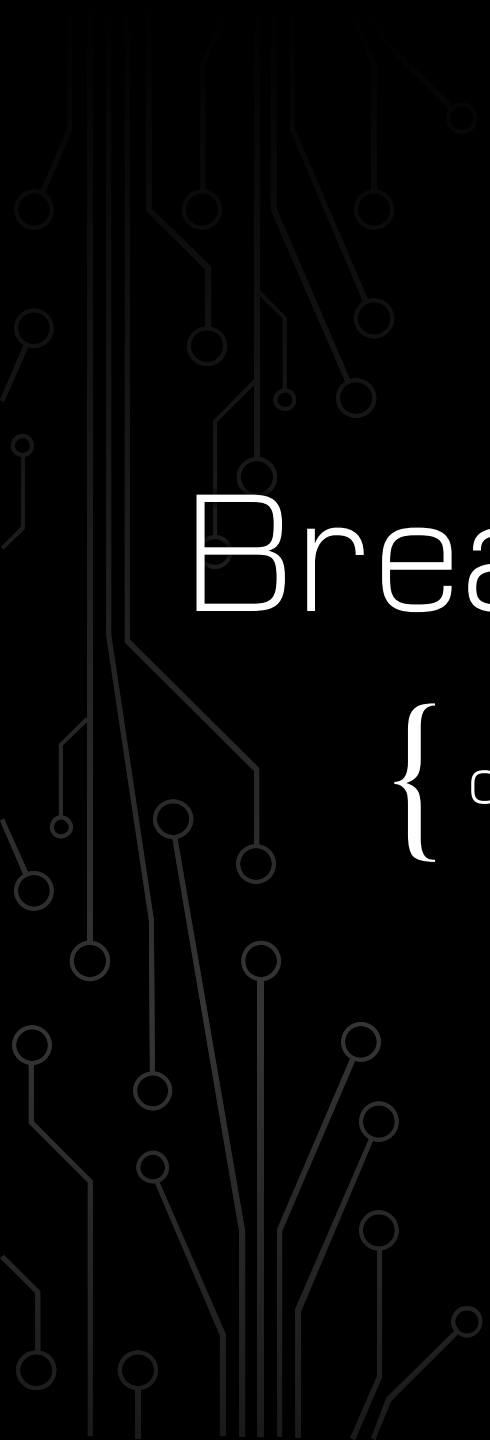


Breaking the x86 ISA

{ domas / @xoreaxeaxeax / Black Hat 2017

./bio

- Christopher Domas
  - Cyber Security Researcher @ Battelle Memorial Institute

The x86 ISA

- 8086: 1978
- A long, tortured history…

x86: evolution

- Modes:
  - Real mode
    - & “unreal” mode
  - Protected mode
  - Virtual 8086
  - System Management Mode
  - Long mode
    - Compatibility mode

x86: evolution

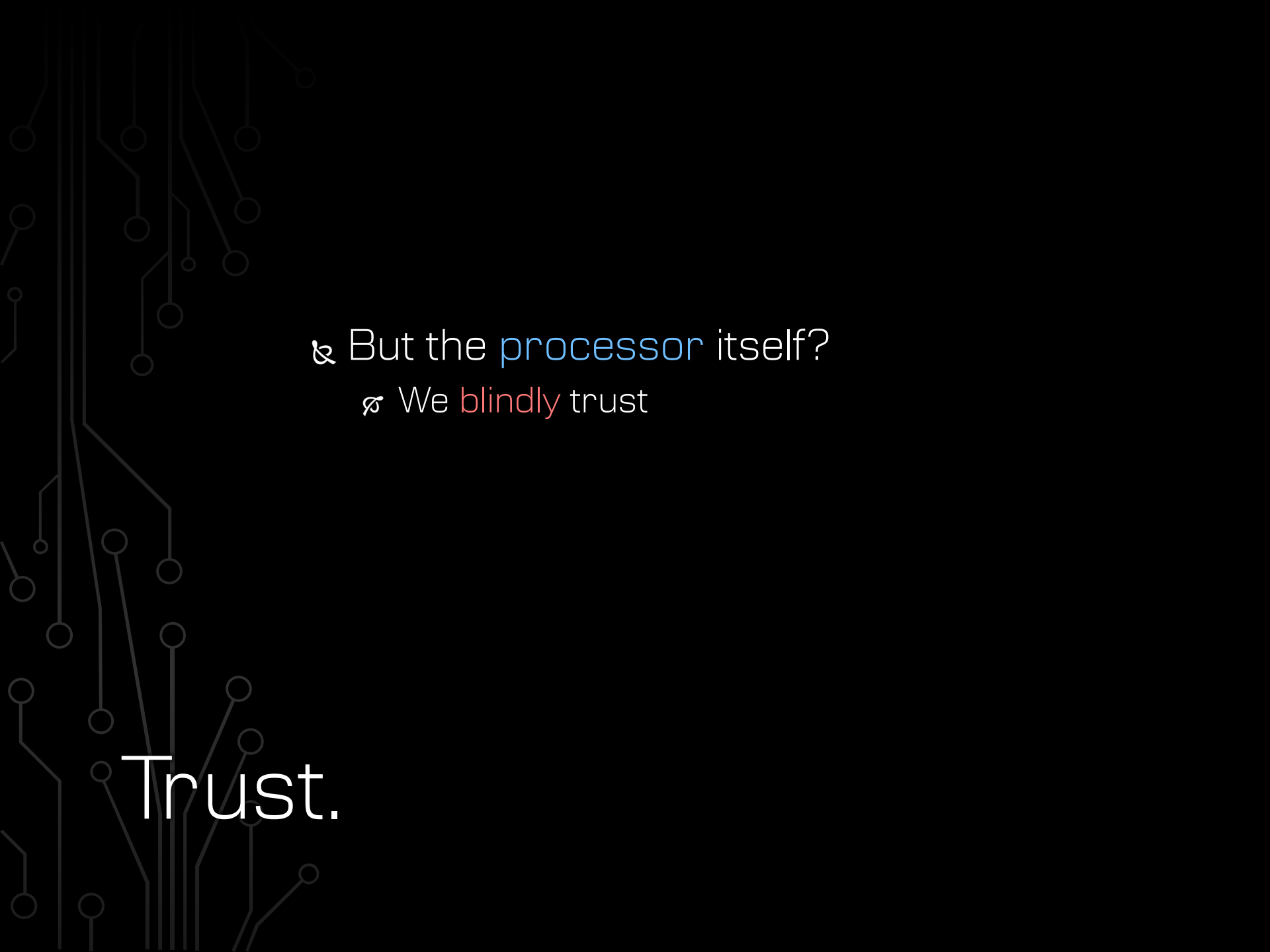
- Instruction set extensions
  - X87
  - IA-32, X86-64
  - MMX, 3DNow!
  - SSE, SSE2, SSE3,SSSE3, SSE4, SSE4.2, SSE5
  - AES-NI, CLMUL, RdRand, SHA
  - MPX, SGX
  - XOP, F16C
  - ADX
  - BMI
  - FMA
  - AVX, AVX2, AVX512
  - VT-x, AMD-V
  - TSX, ASF

x86: evolution

- Modern x86 chips are a complex labyrinth of new and ancient technologies.
  - Things get lost…
- 8086: 29,000 transistors
- Pentium: 3,000,000 transistors
- Broadwell: 3,200,000,000 transistors

Trust.

- We don’t trust software.
  - We audit it
  - We reverse it
  - We break it
  - We sandbox it


But the processor itself?
We blindly trust
Trust.

Trust.

- *Why?*
- Hardware has
  all the same problems as software
- Secret functionality?
  - Appendix H.
- Bugs?
  - F00F, FDIV, TSX, Hyperthreading, Ryzen
- Vulnerabilities?
  - SYSRET, cache poisoning, sinkhole

We should stop
blindly trusting our hardware.

Trust.

- What do we need to worry about?

Backdoors

- Well known from software
  - ProFTPD
    - if (strcmp(target, "ACIDBITCHEZ") == 0) { setuid(0); setgid(0); system("/bin/sh;/sbin/sh"); }
  - Borland Interbase
    - politically:correct
  - Linux kernel
    - if ((options == (__WCLONE|__WALL)) && (current->uid = 0)) retval = -EINVAL;
  - Juniper Netscreen firewalls
    - "<<< %s(un='%s') = %u"

Backdoors

- Hardware
  - FPGAs
    - Actel/Microsemi ProASIC3
  - Hypervisors
    - mov eax, 564D5868h / mov dx, 5658h / out dx, eax
  - Supply chain
    - “A2” - Single gate backdoor added during fabrication
  - Microcode
    - ???

Could a hidden instruction
unlock your CPU?

Hidden instructions

Hidden instructions

- Historical examples
  - ICEBP (f1)
  - LOADALL (0f07)
  - apicall (0ffff0)

Table A-2. One-byte Opcode Map: (00H — F7H) *

	0	1	2	3	4	5	6	7
0	ADD						PUSH ES[64]	POP ES[64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
1	ADC						PUSH SS[64]	POP SS[64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
2	AND						SEG=ES (Prefix)	DAA[64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
3	XOR						SEG=SS (Prefix)	AAA[64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
4	INC[64] general register / REX[64] Prefixes							
	eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	PUSH[64] general register							
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA[64] PUSHAD[64]	POPA[64] POPAD[64]	BOUND[64] Gv, Ma	ARPL[64] Ew, Gw MOVSXD[64] Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc[64], Jb - Short-displacement jump on condition							
	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1[1A]				TEST		XCHG	
	Eb, Ib	Ev, Iz	Eb, Ib[64]	Ev, Ib	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv
9	NOP PAUSE(F3) XCHG r8, rAX	XCHG word, double-word or quad-word register with rAX						
		rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
A	MOV				MOVS/B Yb, Xb	MOVS/W/D/Q Yv, Xv	CMPS/B Xb, Yb	CMPS/W/D Xv, Yv
	AL, Ob	rAX, Ov	Ob, AL	Ov, rAX				
B	MOV immediate byte into byte register							
	AL/R8L, Ib	CL/R9L, Ib	DL/R10L, Ib	BL/R11L, Ib	AH/R12L, Ib	CH/R13L, Ib	DH/R14L, Ib	BH/R15L, Ib
C	Shift Grp 2[1A]		near RET[64] Iw	near RET[64]	LES[64] Gz, Mp VEX+2byte	LDS[64] Gz, Mp VEX+1byte	Grp 11[1A] - MOV	
	Eb, Ib	Ev, Ib					Eb, Ib	Ev, Iz
D	Shift Grp 2[1A]				AAM[64] Ib	AAD[64] Ib		XLAT/ XLATB
	Eb, 1	Ev, 1	Eb, CL	Ev, CL				
E	LOOPNE[64]/ LOOPNZ[64] Jb	LOOPE[64]/ LOOPZ[64] Jb	LOOP[64] Jb	JrCXZ[64]/ Jb	IN		OUT	
					AL, Ib	eAX, Ib	Ib, AL	Ib, eAX
F	LOCK (Prefix)		REPNE XACQUIRE (Prefix)	REP/REPE XRELEASE (Prefix)	HLT	CMC	Unary Grp 3[1A]	
							Eb	Ev

Table A-2. One-byte Opcode Map: (00H — F7H) *

	0	1	2	3	4	5	6	7
0	ADD Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH ES[64]	POP ES[64]
1	ADC Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH SS[64]	POP SS[64]
2	AND Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=ES (Prefix)	DAA[64]
3	XOR Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=SS (Prefix)	AAA[64]
4	INC[64] general register / REX[64] Prefixes eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	PUSH[64] general register rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA[64]/ PUSHAD[64]	POPA[64]/ POPAD[64]	BOUND[64] Gv, Ma	ARPL[64] Ew, Gw MOVSXD[64] Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc[64], Jb - Short-displacement jump on condition O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1[1A] Eb, Ib	Ev, Iz	Eb, Ib[64]	Ev, Ib	TEST Eb, Gb	Ev, Gv	XCHG Eb, Gb	Ev, Gv
9	NOP PAUSE(F3) XCHG r8, rAX	XCHG word, double-word or quad-word register with rAX rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
A	MOV AL, Ob	rAX, Ov	Ob, AL	Ov, rAX	MOVS/B Yb, Xb	MOVS/W/D/Q Yv, Xv	CMPS/B Xb, Yb	CMPS/W/D Xv, Yv
B	MOV immediate byte into byte register AL/R8L, Ib	CL/R9L, Ib	DL/R10L, Ib	BL/R11L, Ib	AH/R12L, Ib	CH/R13L, Ib	DH/R14L, Ib	BH/R15L, Ib
C	Shift Grp 2[1A] Eb, Ib	Ev, Ib	near RET[64] Iw	near RET[64]	LES[64] Gz, Mp VEX+2byte	LDS[64] Gz, Mp VEX+1byte	Grp 11[1A] - MOV Eb, Ib	Ev, Iz
D	Shift Grp 2[1A] Eb, 1	Ev, 1	Eb, CL	Ev, CL	AAM[64] Ib	AAD[64] Ib		XLAT/ XLATB
E	LOOPNE[64]/ LOOPNZ[64] Jb	LOOPE[64]/ LOOPZ[64] Jb	LOOP[64] Jb	JrCXZ[64]/ Jb	IN AL, Ib	eAX, Ib	OUT Ib, AL	Ib, eAX
F	LOCK (Prefix)		REPNE XACQUIRE (Prefix)	REP/REPE XRELEASE (Prefix)	HLT	CMC	Unary Grp 3[1A] Eb	Ev

Table A-2. One-byte Opcode Map: (00H — F7H) *

	0	1	2	3	4	5	6	7
0	ADD Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH ES[64]	POP ES[64]
1	ADC Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH SS[64]	POP SS[64]
2	AND Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=ES (Prefix)	DAA[64]
3	XOR Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=SS (Prefix)	AAA[64]
4	INC[64] general register / REX Prefixes eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	PUSH[64] general register rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA[64]/ PUSHAD[64]	POPA[64]/ POPAD[64]	BOUND[64] Gv, Ma	ARPL[64] Ew, Gw MOVSXD[64] Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc[64], Jb - Short-displacement jump on condition O	NO	B/NAE/C	NB/AE/NC			Jb	
8	Immediate Grp 1[1A] Eb, Ib	Ev, Iz	Eb, Ib[64]		LOCK (Prefix)			REPNE XACQUIRE (Prefix)
9	NOP PAUSE(F3) XCHG r8, rAX		rDX/r10					
A	MOV AL, Ob	rAX, Ov	Ob, AL	Ov, rAX				
B	MOV immediate byte into byte register AL/R8L, Ib	CL/R9L, Ib	DL/R10L, Ib	BL/R11L, Ib	AH/R12L, Ib	CH/R13L, Ib	DH/R14L, Ib	BH/R15L, Ib
C	Shift Grp 2[1A] Eb, Ib	Ev, Ib	near RET[64] Iw	near RET[64]	LES[64] Gz, Mp VEX+2byte	LDS[64] Gz, Mp VEX+1byte	Grp 11[1A] - MOV Eb, Ib	Ev, Iz
D	Shift Grp 2[1A] Eb, 1	Ev, 1	Eb, CL	Ev, CL	AAM[64] Ib	AAD[64] Ib		XLAT/ XLATB
E	LOOPNE[64]/ LOOPNZ[64] Jb	LOOPE[64]/ LOOPZ[64] Jb	LOOP[64] Jb	JrCXZ[64]/ Jb	IN AL, Ib	IN eAX, Ib	OUT Ib, AL	OUT Ib, eAX
F	LOCK (Prefix)		REPNE XACQUIRE (Prefix)	REP/REPE XRELEASE (Prefix)	HLT	CMC	Unary Grp 3[1A] Eb	Ev

So… what's this??

Hidden instructions

- Traditional approaches:
  - Leaked documentation
  - Reverse engineering software
  - NDA
- But what if it's something stealthy?

- Find out what's really there

Goal: Audit the Processor

The challenge

- How to find hidden instructions?

- Instructions can be one byte …
  - inc eax
  - 40
- … or 15 bytes …
  - lock add qword cs:[eax + 4 * eax + 07e06df23h], 0efcdab89h
  - 2e 67 f0 48 818480 23df067e 89abcdef
- Somewhere on the order of
  1,329,227,995,784,915,872,903,807,060,280,344,576
  possible instructions

The challenge

https://code.google.com/archive/p/corkami/wikis/x86oddities.wiki

The challenge

- The obvious approaches don’t work:
  - Try them all?
    - Only works for RISC
  - Try random instructions?
    - Exceptionally poor coverage
  - Guided based on documentation?
    - Documentation can’t be trusted (that’s the point)
    - Poor coverage of gaps in the search space

The challenge

- Goal:
  - Quickly skip over bytes that don’t matter

The challenge

- mov rax, 0x01337C0FFEEC0DE5
- 48 B8 E5 0D EC FE 0F 7C 33 01
- Fuzz the whole thing?
  - Spend majority of time on meaningless constant
  - Will never be able to search the entire space
- Fuzz just the beginning bytes?

The challenge

- lock lock lock lock lock lock lock lock inc dword [rax]
- F0 F0 F0 F0 F0 F0 F0 F0 FF 00
- Fuzz the whole thing?
  - Spend majority of time on superfluous prefixes
  - Will never be able to search the entire space
- How do we proceed?

The challenge

- Observation:
  - The meaningful bytes of an x86 instruction impact either its length or its exception behavior

Tunneling

- A depth-first-search algorithm

- Guess an instruction:

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

- Execute the instruction:

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

Observe its length:

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

Increment the last byte:

00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

- Execute the instruction:

00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

Observe its length:

00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

Increment the last byte:

00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

- Execute the instruction:

00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

Observe its length:

00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

Increment the last byte:

00 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

- Execute the instruction:

00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

Observe its length:

00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

Increment the last byte:

00 04 01 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

- Execute the instruction:

00 04 01 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

Observe its length:

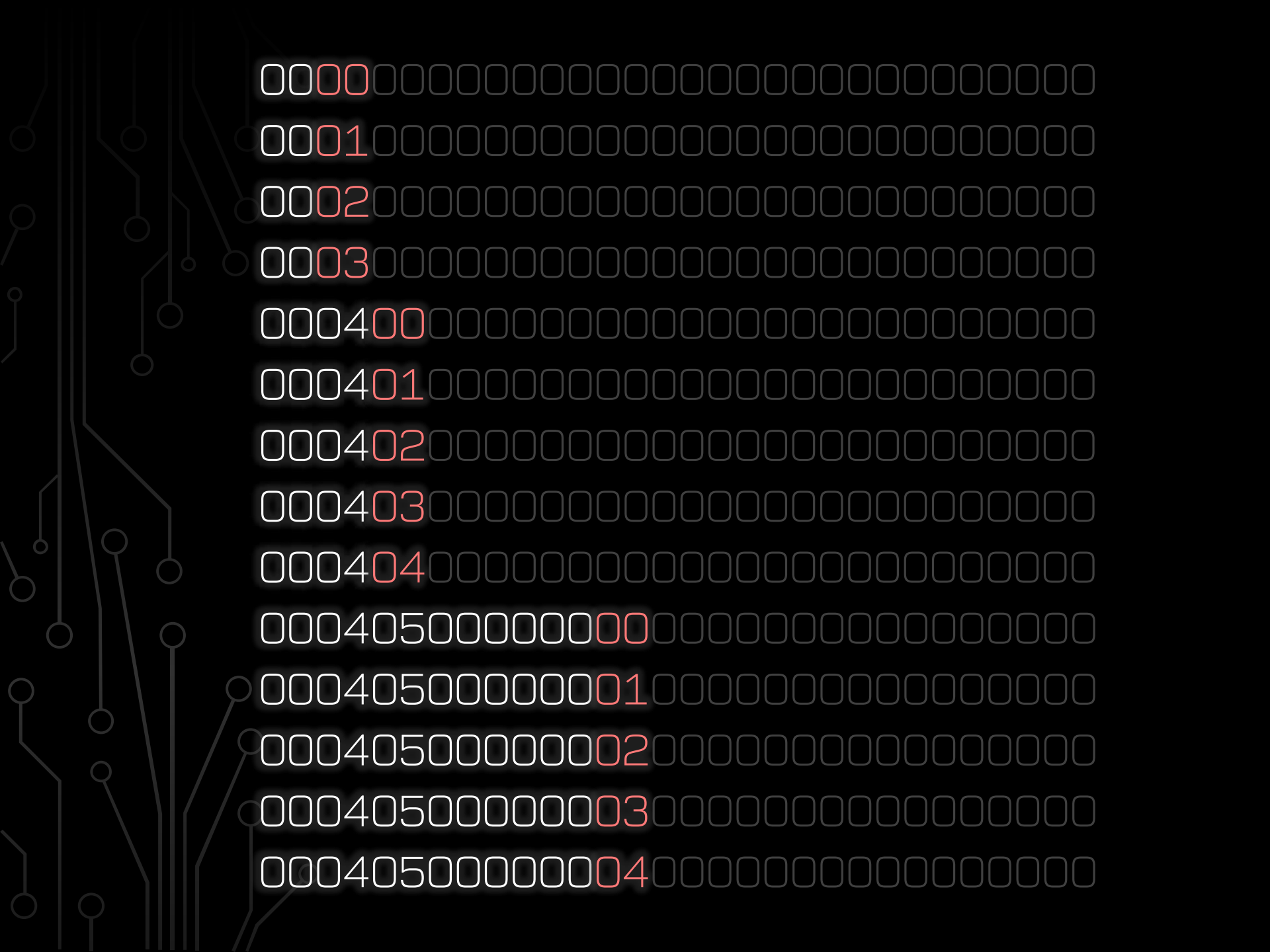
00 04 01 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

Increment the last byte:

00 04 02 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling


0000
0001
0002
0003
000400
000401
000402
000403
000404
00040500000000
00040500000001
00040500000002
00040500000003
00040500000004

- When the last byte is FF…

C7 04 05 00 00 00 00 00 00 00 FF 00 00 00 00

Tunneling

& ... roll over ...

C7 04 05 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

& ... and move to the previous byte

C7 04 05 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

Tunneling

This byte becomes the marker

C7 04 05 00 00 00 00 00 00 00 00 00 00 00 00

Increment the marker

C7 04 05 00 00 00 00 00 00 01 00 00 00 00 00

Tunneling

- Execute the instruction

C7 04 05 00 00 00 00 00 00 01 00 00 00 00 00

Tunneling

- Observe its length

C7 04 05 00 00 00 00 00 00 01 00 00 00 00 00

Tunneling

Tunneling

- If the length has not changed…

C7 04 05 00 00 00 00 00 00 01 00 00 00 00 00

Increment the marker

C7 04 05 00 00 00 00 00 00 02 00 00 00 00 00

Tunneling

- And repeat.

C7 04 05 00 00 00 00 00 00 02 00 00 00 00 00

Tunneling

Continue the process…

C7 04 05 00 00 00 00 00 00 FF 00 00 00 00 00

Tunneling

& ... moving back on each rollover

C7 04 05 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

& ... moving back on each rollover

C7 04 05 00 00 00 00 00 FF 00 00 00 00 00 00

Tunneling

& ... moving back on each rollover

C7 04 05 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

&...

C7 04 05 00 00 00 00 FF 00 00 00 00 00 00 00

Tunneling

C7 04 05 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

&...

C7 04 05 00 00 00 FF 00 00 00 00 00 00 00 00

Tunneling

& . . .

C7 04 05 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

&...

C7 04 05 00 00 FF 00 00 00 00 00 00 00 00 00 00

Tunneling

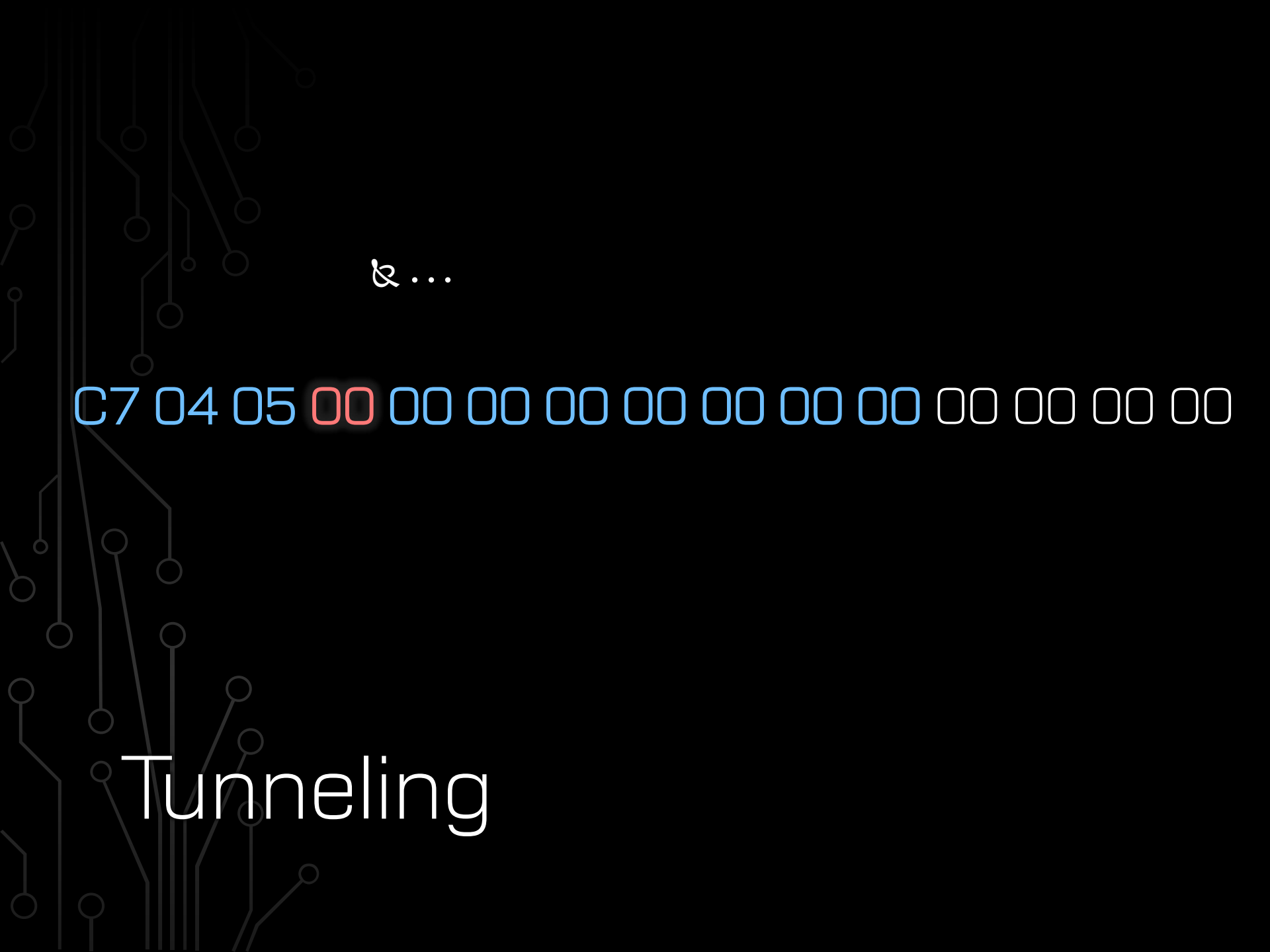
C7 04 05 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

&...

C7 04 05 00 FF 00 00 00 00 00 00 00 00 00 00

Tunneling


&...
C7 04 05 00 00 00 00 00 00 00 00 00 00 00 00
Tunneling

&...

C7 04 05 FF 00 00 00 00 00 00 00 00 00 00 00

Tunneling

& . . .

C7 04 05 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

Tunneling

- When you increment a marker…

C7 04 06 00 00 00 00 00 00 00 00 00 00 00 00

& ... execute the instruction ...

C7 04 06 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

… and the length changes …

C7 04 06 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

& ... move the marker to
the end of the new instruction ...

C7 04 06 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

& ... and resume the process.

C7 04 06 00 00 00 01 00 00 00 00 00 00 00 00 00

Tunneling

Tunneling

- Tunneling through the instruction space
  lets us quickly skip over the bytes
  that *don't* matter,
  and exhaustively search the bytes that do…

Tunneling

- … reducing the search space from 1.3×10^{36} instructions to ~100,000,000 (one day of scanning)

Instruction lengths

- Catch:
  requires knowing the instruction length

Instruction lengths

- Simple approach: trap flag
  - Fails to resolve the length of faulting instructions
  - Necessary to search privileged instructions:
    - ring 0 only: mov cr0, eax
    - ring -1 only: vmenter
    - ring -2 only: rsm
  - It’s hard to even auto-generate a successfully executing ring 3 instruction:
    - mov eax, [random_number]

- Solution: page fault analysis

Instruction lengths

Page fault analysis

- Choose a candidate instruction
  - (we don't know how long this instruction is)

0F 6A 60 6A 79 6D C6 02 6E AA D2 39 0B B7 52

- Configure two consecutive pages in memory
  - The first with read, write, and execute permissions
  - The second with read, write permissions only

Page fault analysis

- Place the candidate instruction in memory
  - Place the first byte at the end of the first page
  - Place the remaining bytes at the start of the second

Page fault analysis

- Execute (jump to) the instruction.

Page fault analysis

- The processor's instruction decoder checks the first byte of the instruction.

Page fault analysis

- If the decoder determines that another byte is necessary, it attempts to fetch it.

Page fault analysis

Page fault analysis

- This byte is on a non-executable page, so the processor generates a page fault.

- The #PF exception provides a fault address in the CR2 register.

Page fault analysis

Page fault analysis

- If we receive a #PF, with CR2 set to the address of the second page, we know the instruction continues.

- Move the instruction back one byte.

Page fault analysis

- Execute the instruction.

Page fault analysis

- The processor's instruction decoder checks the first byte of the instruction.

Page fault analysis

- If the decoder determines that another byte is necessary, it attempts to fetch it.

Page fault analysis

- Since this byte is in an executable page, decoding continues.

Page fault analysis

- If the decoder determines that another byte is necessary, it attempts to fetch it.

Page fault analysis

Page fault analysis

- This byte is on a non-executable page, so the processor generates a page fault.

- Move the instruction back one byte.

Page fault analysis

- Execute the instruction.

Page fault analysis

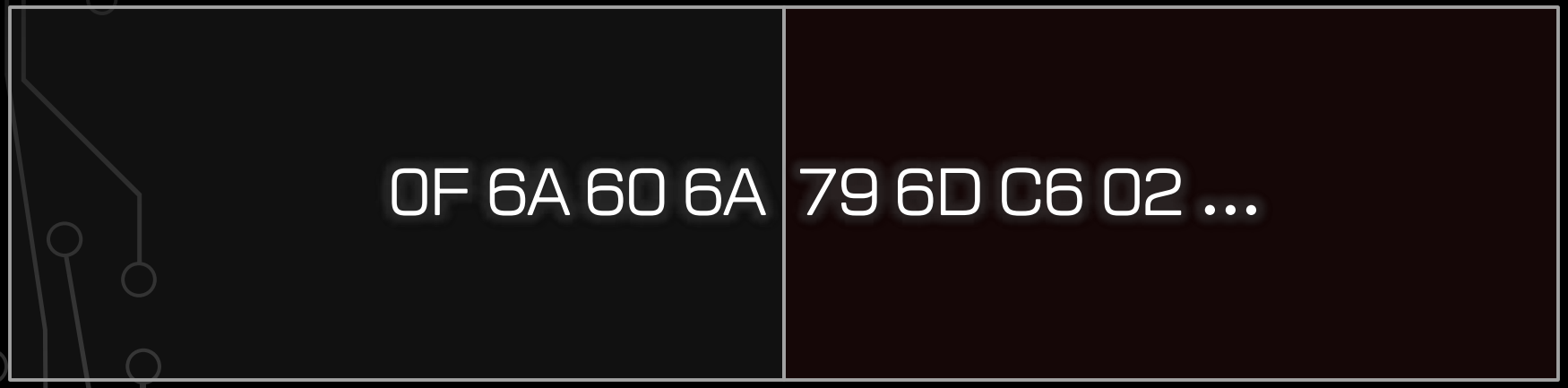
- Continue the process while
  we receive #PF exceptions
  with CR2 = second page address

Page fault analysis

- Move the instruction back one byte.

Page fault analysis

- Execute.



Page fault analysis

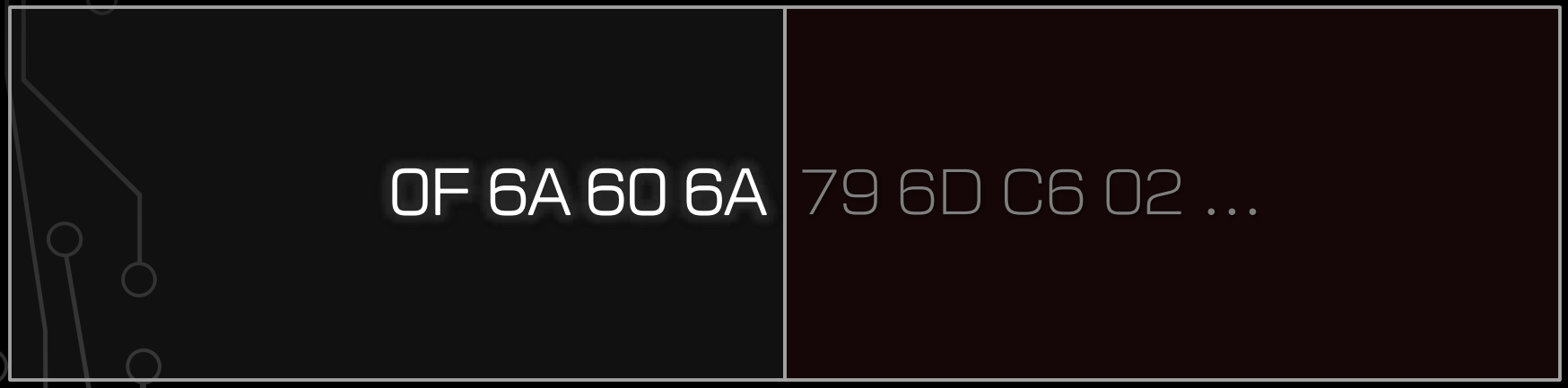
- Eventually, the entire instruction will reside in the executable page.

Page fault analysis

- The instruction could run.
- The instruction could throw a different fault.
- The instruction could throw a #PF,
  but with a different CR2.

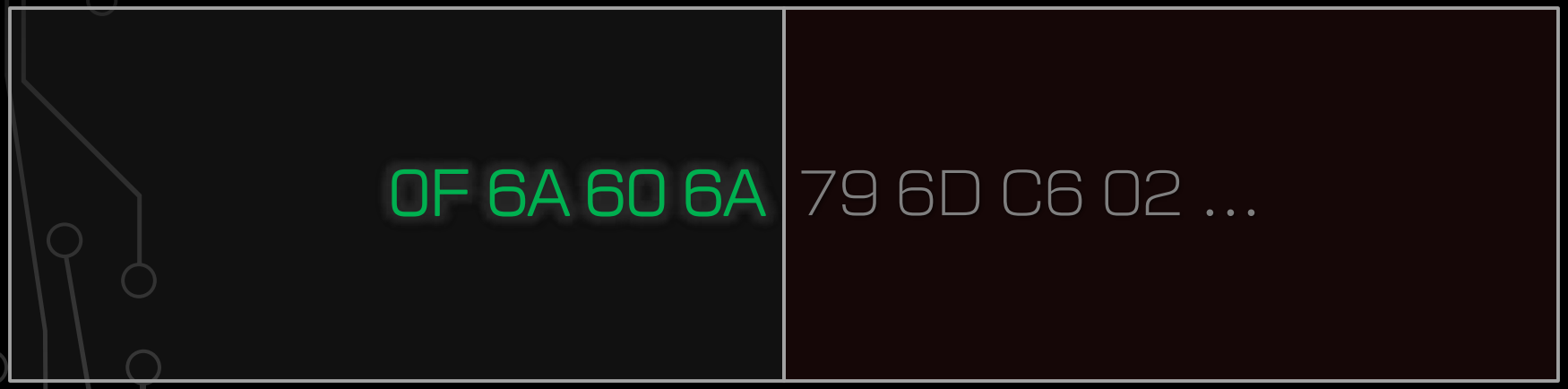
Page fault analysis

- In all cases, we know the instruction has been successfully decoded, so must reside entirely in the executable page.



Page fault analysis

- With this, we know the instruction's length.



Page fault analysis

- We now know how many bytes the instruction decoder consumed
- But just because the bytes were *decoded* does not mean the instruction *exists*
- If the instruction does not exist, the processor generates the #UD exception after the instruction decode (invalid opcode exception)

Page fault analysis

- If we don't receive a #UD, the instruction exists.

Page fault analysis

Page fault analysis

- Resolves lengths for:
  - Successfully executing instructions
  - Faulting instructions
  - Privileged instructions:
    - ring 0 only: mov cr0, eax
    - ring -1 only: vmenter
    - ring -2 only: rsm

Page fault analysis

- We can even resolve
  the lengths of instructions that *don't exist.*

	pfx	8	9	A	
		INVD	WBINVD		
		Vx, Hx, Wx	Vx, Hx, Wx	Vx, Hx, Wx	
	F3				
7		VMREAD Ey, Gy	VMWRITE Gy, Ey		
	66				
	F3				
	F2				

- Example:
  - 0f 7a ac cb 8b 97 e4 4d
  - Group 0f 7a does not exist
    (receive #UD for this entire group)
  - But the decoder fetched 8 bytes to decode it
- Provides insight into the pipelining architecture
- Allows exploration of future instructions

Page fault analysis

The Injector

- The “injector” process performs the page fault analysis and tunneling instruction generation

Surviving

- We're fuzzing the same device that we're running on
- How do we make sure we don't crash?

Surviving

- Step 1:
  - Limit ourselves to ring 3
  - We can still resolve instructions living in deeper rings
  - This prevents accidental total system failure (except in the case of serious processor bugs)

Surviving

- Step 2:
  - Hook all exceptions the instruction might generate
  - In Linux:
    - SIGSEGV
    - SIGILL
    - SIGFPE
    - SIGBUS
    - SIGTRAP
  - Process will clean up after itself when possible

Surviving

- Step 3:
  - Initialize general purpose registers to 0
  - Arbitrary memory write instructions like
    add [eax + 4 * ecx], 0x9102
    will not hit the injecting process's address space
  - Map address 0 into the injector's address space
    - Requires su in Linux
    - Not strictly necessary,
      but allows finer grain fault resolution

Surviving

- Step 3 (continued):
  - Memory calculations using an offset:
    add [eax + 4 * ecx + 0xf98102cd6], 0x9102
    would still result in non-zero accesses
  - Could lead to process corruption
    if the offset falls into the injector's address space

Surviving

- Step 3 (continued):
  - The tunneling approach ensures offsets are constrained
    - 0x0000002F
    - 0x0000A900
    - 0x00420000
    - 0x1E000000
  - The tunneled offsets will not fall into the injector's address space
  - They will seg fault, but seg faults are caught
  - The process still won't corrupt itself

Surviving

- We've handled faulting instructions
- What about non-faulting instructions?
  - The analysis needs to continue after an instruction executes
    - E.g. jmp $-0x39

Surviving

- Set the trap flag prior to executing the candidate instruction
  - Allows us to catch branches, etc.
  - Allows determining execution results
    - If we receive a #DB (trap) exception, we know the instruction successfully executed
- Reload the registers to a known state

Surviving

- With these…
  - Ring 3
  - Exception handling
  - Register initialization
  - Register maintenance
  - Execution trapping
- … the injector survives.

Analysis

- So we now have a way to *search* the instructions space.
  - How do we make *sense* of the instructions we execute?

The Sifter

- The “sifter” process parses the executions from the injector, and pulls out the anomalies

Sifting

- We need a “ground truth”
- Use a disassembler
  - It was written based on the documentation
  - Capstone

Sifting

- Compare:
  - Observed length of instruction vs. disassembled length of instruction
  - Signal generated by instruction vs. expected signal

Sifting

- Undocumented instruction:
  - Disassembler doesn't recognize byte sequence and …
  - Instruction generates anything but a #UD
- Software bug:
  - Disassembler recognizes instruction but …
  - Processor says the length is different
- Hardware bug:
  - ???
  - No consistent heuristic, investigate when something fails

sandsifter - demo

2,259,724
39800/s
112

(sandsifter)

```
VIA Nano U3500@1000MHz
arch: 32 / processor: 0 / vendor: CentaurHauls / family: 6 / model: n/a / stepping: 8 / ucode: n/a
```

```
> .... ............
  > 0f.... ........
    > 0f0d.. ......
    > 0f18..
    > 0f1a..
    > 0f1b..
    > 0f1c..
    > 0f1d..
    > 0f1e..
    > 0f1f..
    > 0fa7..
        0fa7c1
        0fa7c2
        0fa7c3
        0fa7c4
        0fa7c5
        0fa7c6
        0fa7c7
    > 0fae..
  > c4.... ....
  > c5.... ......
  > db..
      dbe0
      dbe1
  > df..
      dfc0
      dfc1
      dfc2
```

```
instruction:
0fa7c2

prefixes:       (__)
valids:         (1)
lengths:        (3)
signums:        (5)
signals:        (sigtrap)
sicodes:        (2)

analysis:
capstone:
  (unk)
  n/a

ndisasm:
  (unknown)
  n/a

objdump:
  (unknown)
  n/a
```

```
j: down,      J: DOWN
k: up,        K: UP
l: expand     L: all
h: collapse   H: all
g: start      G: end
{: previous   }: next
q: quit and print
```

(summarizer)

Scanning

- We now have a way to systematically scan our processor for secrets and bugs

Scanning

- I scanned eight systems in my test library.

Results

- Hidden instructions
- Ubiquitous software bugs
- Hypervisor flaws
- Hardware bugs

Hidden instructions

- Scanned: Intel Core i7-4650U CPU

Intel hidden instructions

Intel hidden instructions

- 0f0dxx
  - Undocumented for non-/1 reg fields
- 0f18xx, 0f{1a-1f}xx
  - Undocumented until December 2016
- 0fae{e9-ef, f1-f7, f9-ff}
  - Undocumented for non-0 r/m fields until June 2014

- dbe0, dbe1
- df{c0-c7}
- f1
- {c0-c1}{30-37, 70-77, b0-b7, f0-f7}
- {d0-d1}{30-37, 70-77, b0-b7, f0-f7}
- {d2-d3}{30-37, 70-77, b0-b7, f0-f7}
- f6 /1, f7 /1

Intel hidden instructions

- Scanned: AMD Athlon (Geode NX1500)

AMD hidden instructions

- 0f0f{40-7f}{80-ff}{xx}
  - Undocumented for range of xx
- dbe0, dbe1
- df{c0-c7}

AMD hidden instructions

- Scanned: VIA Nano U3500, VIA C7-M

VIA hidden instructions

- 0f0dxx
  - Undocumented by Intel for non-/1 reg fields
- 0f18xx, 0f{1a-1f}xx
  - Undocumented by Intel until December 2016
- 0fa7{c1-c7}
- 0fae{e9-ef, f1-f7, f9-ff}
  - Undocumented by Intel for non-0 r/m fields until June 2014
- dbe0, dbe1
- df{c0-c7}

VIA hidden instructions

Hidden instructions

- There are actually more than this
  - But our disassembler recognizes many undocumented instructions
  - If recognized by the disassembler, sandsifter cannot identify them as anomalies

Hidden instructions

- What do these *do?*
  - Some have been reverse engineered
  - Some have no record at all.

Software bugs

Software bugs

- Issue:
  - The sifter is forced to use a disassembler as its “ground truth”
  - Every disassembler we tried as the “ground truth” was littered with bugs.

Software bugs

- Most bugs only appear in a few tools, and are not especially interesting
- Some bugs appeared in *all* tools
  - These can be used to an attacker's advantage.

- 66e9xxxxxxxx (jmp)
- 66e8xxxxxxxx (call)

Software bugs

Software bugs

- 66e9xxxxxxxx (jmp)
- 66e8xxxxxxxx (call)

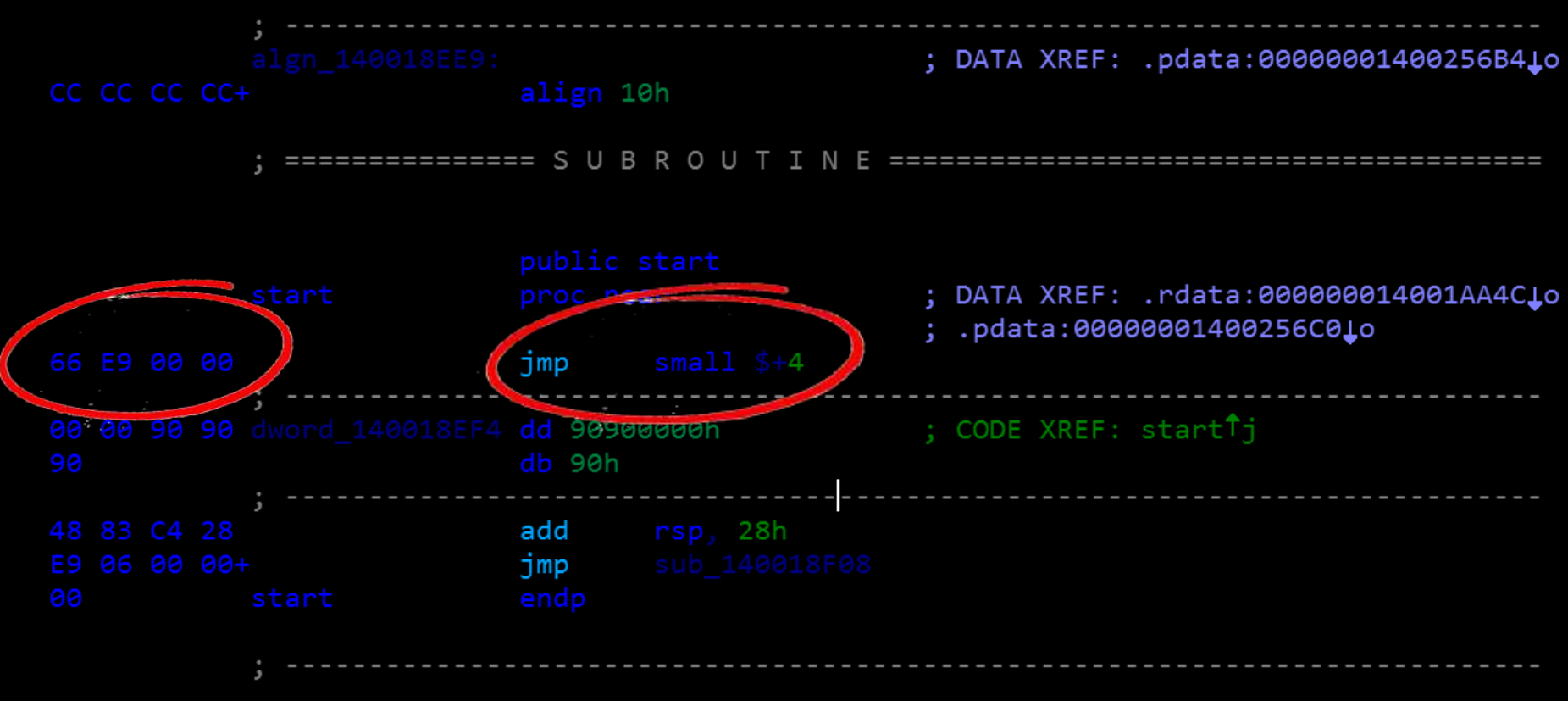
- In x86_64
- Theoretically, a jmp (e9) or call (e8), with a data size override prefix (66)
  - Changes operand size from default of 32
    - Does that mean 16 bit or 64 bit?
    - Neither. 66 is ignored by the processor here.

- Everyone parses this wrong.

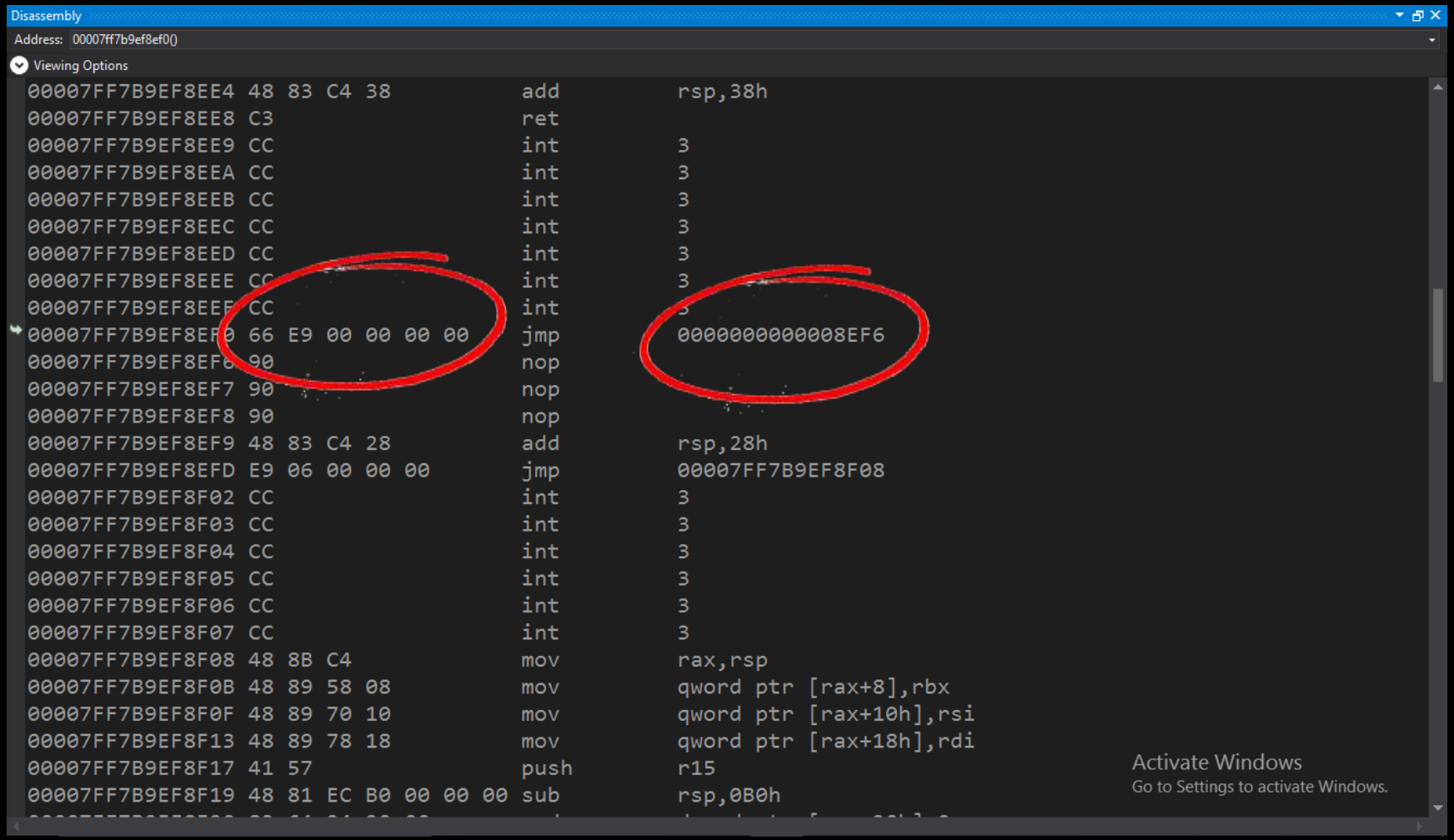
Software bugs

- Demo:
  - IDA
  - Visual Studio

Software bugs



Software bugs (IDA)



Software bugs (VS)

Software bugs

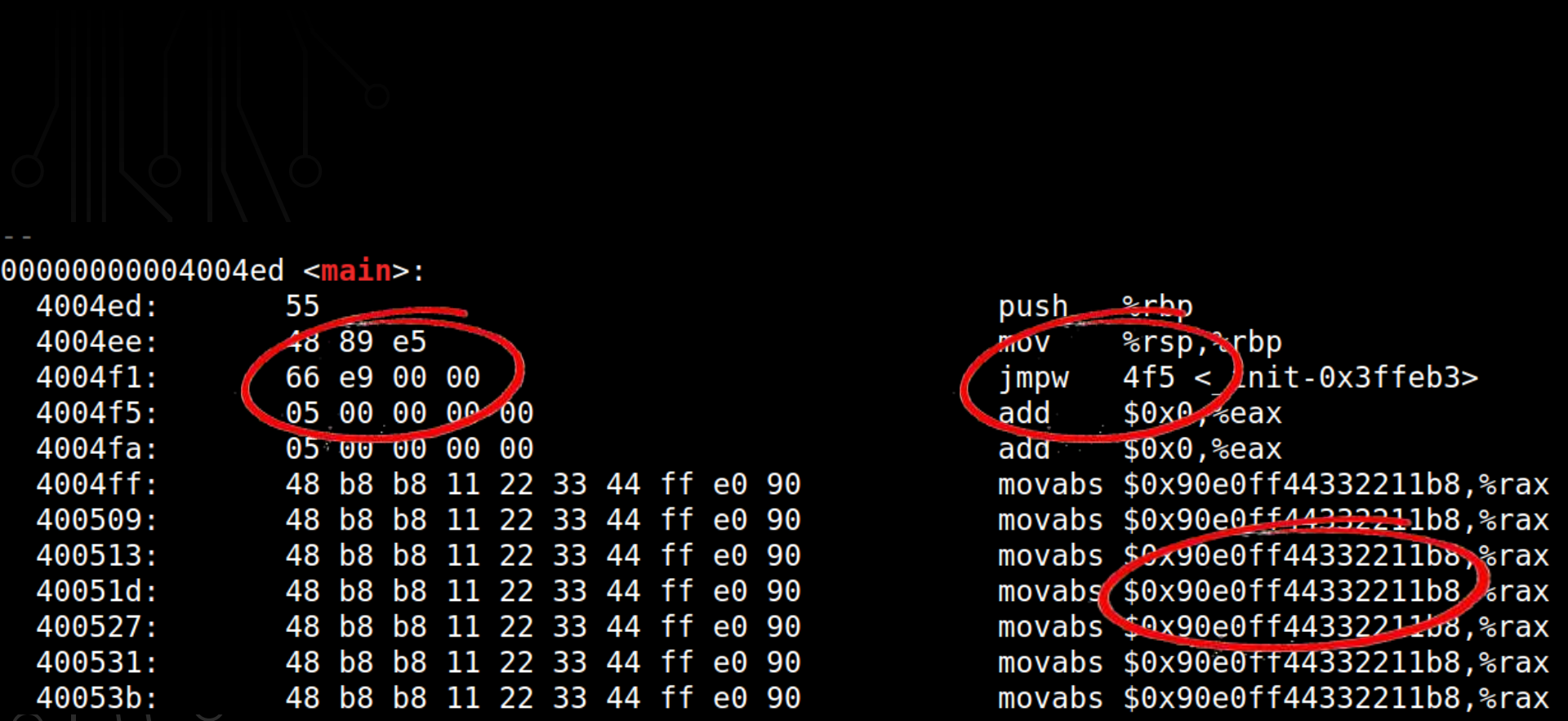
- Everyone misinterprets either:
  - The target of the jump
    - Truncates the instruction pointer to 16 bits
  - The size of the operand
    - Reading the instruction as 4 bytes instead of 6
  - Or (often) both of these

Software bugs

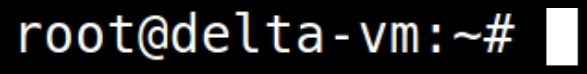
- An attacker can use this to mask malicious behavior
  - Throw off disassembly and jump targets to cause analysis tools to miss the real behavior

Software bugs

- Demo:
  - objdump/gdb
  - QEMU



Software bugs (objdump)



Software bugs (QEMU)

Software bugs

- 66 jmp
- Why does everyone get this wrong?
  - AMD designed the 64 bit architecture,
  - Intel adopted… most of it.
- AMD: override changes operand to 16 bits, instruction pointer truncated
- Intel: override ignored.

Software bugs

- Issues when we can't agree on a standard
  - sysret bugs
- Either Intel or AMD is going to be vulnerable when there is a difference
- Impractically complex architecture
  - Tools cannot parse a jump instruction

Software bugs

- Opcode 82
  - Disassemblers don't recognize this instruction
    - (objdump, Capstone, GDB, ndisasm, etc.)
  - ... even though it's documented by both Intel and AMD
  - Synonym for opcode 80
    - add, or, and, xor, cmp, sub, ...
  - Opcode 82 lets us do computation invisible to a disassembler.

Hypervisor bugs

Azure hypervisor bugs

- In an Azure instance,
  the trap flag is missed
  on the cpuid instruction
  - (cpuid causes a vmexit,
    and the hypervisor forgets
    to emulate the trap)

Azure hypervisor bugs

Hardware bugs

Hardware bugs

- Hardware bugs are troubling
  - A bug in hardware means
    you now have the same bug
    in all of your software.
  - Difficult to find
  - Difficult to fix

- Scanned:
  - Quark, Pentium, Core i7

Intel hardware bugs

- f00f bug on Pentium (anti-climactic)

Intel hardware bugs

- Scanned:
  - Geode NX1500, C-50

AMD hardware bugs

AMD hardware bugs

- f00f20c0 / f00f22c0
  - lock mov cr0, eax / lock mov eax, cr0
  - Looks like a bug
    (lock shouldn't execute on non-mem instruction)
  - Found buried in cpuid enumerations in AMD docs:
    "LOCK MOV CR0 means MOV CR8"

AMD hardware bugs

- On several systems,
  receive a #UD exception
  prior to complete instruction fetch
- Per AMD specifications, this is incorrect.
  - #PF during instruction fetch takes priority
- … until …

Table 8-8. Simultaneous Interrupt Priorities

Interrupt Priority	Interrupt Condition	Interrupt Vector
(High)	Processor Reset	—
0	Machine-Check Exception	18
1	External Processor Initialization (INIT)	—
	SMI Interrupt	
	External Clock Stop (Stpclk)	
2	Data, and I/O Breakpoint (Debug Register)	1
	Single-Step Execution Instruction Trap (RFLAGS.TF=1)	
3	Non-Maskable Interrupt	2
4	Maskable External Interrupt (INTR)	32–255
5	Instruction Breakpoint (Debug Register)	1
	Code-Segment-Limit Violation[1]	13
	Instruction-Fetch Page Fault[1]	14
6	Invalid Opcode Exception[1]	6
	Device-Not-Available Exception	7
	Instruction-Length Violation (> 15 Bytes)	13

Note:

1. This reflects the relative priority for faults encountered when fetching the first byte of an instruction. In the fetching and decoding of subsequent bytes of an instruction, an Invalid Opcode exception may be detected and raised before a fetch-related fault would be seen on a later byte. This behavior is model-dependent.

- Scanned:
  - TM5700

Transmeta hardware bugs

Transmeta hardware bugs

- Instructions: 0f{71,72,73}xxxx
- Can receive #MF exception during fetch
- Example:
  - Pending x87 FPU exception
  - psrad mm4, -0x50 (0f72e4b0)
  - #MF received after 0f72e4 fetched
  - Correct behavior: #PF on fetch,
    last byte is still on invalid page

hardware bugs

- Found on one processor...
- An apparent "halt and catch fire" instruction
  - Single malformed instruction in ring 3
    locks the processor
  - Tested on 2 Windows kernels, 3 Linux kernels
  - Kernel debugging, serial I/O,
    interrupt analysis seem to confirm
- Unfortunately,
  not finished with responsible disclosure
- No detials available
  on chip, vendor, or instructions

ring 3 processor DoS: demo

- First such attack found in 20 years
  (since Pentium f00f)

hardware bugs

- Significant security concern:
  processor DoS from unprivileged user

hardware bugs

- Details (hopefully) released within the next month (stay tuned)

hardware bugs

Conclusions

- Open sourced:
  - The sandsifter scanning tool
  - github.com/xoreaxeaxeax/sandsifter
- Audit your processor,
  break disassemblers/emulators/hypervisors,
  halt and catch fire, etc.

Conclusions

- I've only scanned a few systems
- This is a fraction of what I found on mine
- Who knows what exists on yours

Conclusions

- Check your system
  - Send us results if you can

Conclusions

- Don’t blindly trust the specifications.

Conclusions

- Sandsifter lets us introspect the black box at the heart of our systems.

- github.com/xoreaxeaxeax
  - sandsifter
  - M/o/Vfuscator
  - REpsych
  - x86 0-day PoC
  - Etc.

- Feedback? Ideas?

- domas
  - @xoreaxeaxeax
  - xoreaxeaxeax@gmail.com