Breaking the x86 ISA

domas / @xoreaxeaxeax / Black Hat 2017

./bio

- Christopher Domas
  - Cyber Security Researcher @ Battelle Memorial Institute

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.

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

Trust.

We should stop blindly trusting our hardware.

Trust.

What do we need to worry about?

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[i64]	POP ES[i64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
1	ADC						PUSH SS[i64]	POP SS[i64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
2	AND						SEG=ES (Prefix)	DAA[i64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
3	XOR						SEG=SS (Prefix)	AAA[i64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
4	INC[i64] general register / REX[o64] 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[d64] general register							
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA[i64]/ PUSHAD[i64]	POPA[i64]/ POPAD[i64]	BOUND[i64] Gv, Ma	ARPL[i64] Ew, Gw MOVSXD[o64] Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc[f64], 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[i64]	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[f64] Iw	near RET[f64]	LES[i64] Gz, Mp VEX+2byte	LDS[i64] Gz, Mp VEX+1byte	Grp 11[1A] - MOV	
	Eb, Ib	Ev, Ib					Eb, Ib	Ev, Iz
D	Shift Grp 2[1A]				AAM[i64] Ib	AAD[i64] Ib		XLAT/ XLATB
	Eb, 1	Ev, 1	Eb, CL	Ev, CL				
E	LOOPNE[f64]/ LOOPNZ[f64] Jb	LOOPE[f64]/ LOOPZ[f64] Jb	LOOP[f64] Jb	JrCXZ[f64]/ 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^{i64}	POP ES^{i64}
1	ADC Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH SS^{i64}	POP SS^{i64}
2	AND Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=ES (Prefix)	DAA^{i64}
3	XOR Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=SS (Prefix)	AAA^{i64}
4	INC^{i64} general register / REX^{o64} 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^{d64} general register rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA^{i64}/ PUSHAD^{i64}	POPA^{i64}/ POPAD^{i64}	BOUND^{i64} Gv, Ma	ARPL^{i64} Ew, Gw MOVSXD^{o64} Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc^{f64}, 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^{i64}	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^{f64} Iw	near RET^{f64}	LES^{i64} Gz, Mp VEX+2byte	LDS^{i64} 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^{i64} Ib	AAD^{i64} Ib		XLAT/ XLATB
E	LOOPNE^{f64}/ LOOPNZ^{f64} Jb	LOOPE^{f64}/ LOOPZ^{f64} Jb	LOOP^{f64} Jb	JrCXZ^{f64}/ 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^{i64}	POP ES^{i64}
1	ADC Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH SS^{i64}	POP SS^{i64}
2	AND Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=ES (Prefix)	DAA^{i64}
3	XOR Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=SS (Prefix)	AAA^{i64}
4	INC^{i64} general register / REX^{o64} 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^{d64} general register rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	[illegible]	rSI/r14	rDI/r15
6	PUSHA^{i64}/ PUSHAD^{i64}	POPA^{i64}/ POPAD^{i64}	BOUND^{i64} Gv, Ma	ARPL^{i64} Ew, Gw MOVSXD^{o64} Gv, Ev	SEG=FS (Prefix)	[illegible]	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc^{f64}, Jb - Short-displacement jump on condition O	NO	B/NAE/C	NB/AE/NC	[illegible]	[illegible]	[illegible]	[illegible]
8	Immediate Grp 1^{1A} Eb, Ib	Ev, Iz	Eb, Ib^{i64}	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
9	NOP PAUSE(F3) XCHG r8, rAX	[illegible]	rDX/r10	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
A	MOV AL, Ob	rAX, Ov	Ob, AL	Ov, rAX	[illegible]	[illegible]	[illegible]	[illegible]
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^{f64} Iw	near RET^{f64}	LES^{i64} Gz, Mp VEX+2byte	LDS^{i64} 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^{i64} Ib	AAD^{i64} Ib		XLAT/ XLATB
E	LOOPNE^{f64}/ LOOPNZ^{f64} Jb	LOOPE^{f64}/ LOOPZ^{f64} Jb	LOOP^{f64} Jb	JrCXZ^{f64}/ 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

So… what's this??

	Jb	
LOCK (Prefix)		REPNE XACQUIRE (Prefix)

Goal: Audit the Processor

- Find out what's really there

- How to find hidden instructions?

The challenge

The challenge

- 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

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

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

The challenge

Tunneling

- A depth-first-search algorithm

Tunneling

- Guess an instruction:

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

- Execute the instruction:

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

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

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

Tunneling

- Execute the instruction:

00 02 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 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 03 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling

- Observe its length:

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

Tunneling

Increment the last byte:

00 04 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

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

Tunneling

- Execute the instruction:

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

Observe its length:

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

Tunneling

000000000000000000000000000000
000100000000000000000000000000
000200000000000000000000000000
000300000000000000000000000000
000400000000000000000000000000
000401000000000000000000000000
000402000000000000000000000000
000403000000000000000000000000
000404000000000000000000000000
000405000000000000000000000000
000405000000010000000000000000
000405000000020000000000000000
000405000000030000000000000000
000405000000040000000000000000

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

This byte becomes the marker

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

Tunneling

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

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

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

Tunneling

C7 04 05 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

ঙ...

C7 04 05 00 00 00 00 00 00 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

ঌ When you increment a marker...

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

Tunneling

… 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

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

Tunneling

… and resume the process.

C7 04 06 00 00 00 01 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…

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

Tunneling

- Catch:
  requires knowing the instruction length

Instruction lengths

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

- Solution: page fault analysis

Instruction lengths

- 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

Page fault analysis

Page fault analysis

- 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

0F	6A 60 6A 79 6D C6 02 …

- Execute (jump to) the instruction.

0F | 6A 60 6A 79 6D C6 02 …

Page fault analysis

Page fault analysis

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

0F	6A 60 6A 79 6D C6 02 …

Page fault analysis

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

0F | 6A 60 6A 79 6D C6 02 …

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

0F | 6A 60 6A 79 6D C6 02 ...

Page fault analysis

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

0F	6A 60 6A 79 6D C6 02 ...

Page fault analysis

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

0F	6A 60 6A 79 6D C6 02 ...

Page fault analysis

- Move the instruction back one byte.

0F 6A	60 6A 79 6D C6 02 ...

Page fault analysis

- Execute the instruction.

0F 6A	60 6A 79 6D C6 02 ...

Page fault analysis

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

OF 6A	60 6A 79 6D C6 02 ...

Page fault analysis

Page fault analysis

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

0F 6A	60 6A 79 6D C6 02 …

Page fault analysis

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

0F 6A | 60 6A 79 6D C6 02 …

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

0F 6A	60 6A 79 6D C6 02 ...

Page fault analysis

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

0F 6A	60 6A 79 6D C6 02 …

Page fault analysis

Page fault analysis

- Move the instruction back one byte.

0F 6A 60	6A 79 6D C6 02 …

Page fault analysis

- Execute the instruction.

0F 6A 60	6A 79 6D C6 02 ...

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

0F 6A 60	6A 79 6D C6 02 ...

Page fault analysis

- Move the instruction back one byte.

0F 6A 60 6A	79 6D C6 02 …

Page fault analysis

Execute.

0F 6A 60 6A 79 6D C6 02 ...

Page fault analysis

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

0F 6A 60 6A	79 6D C6 02 ...

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.

0F 6A 60 6A	79 6D C6 02 …

Page fault analysis

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

0F 6A 60 6A	79 6D C6 02 ...

Page fault analysis

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

0F 6A 60 6A | 79 6D C6 02 …

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

- 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

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?

- 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

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

- 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

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

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

- We've handled faulting instructions
- What about non-faulting instructions?
  - The analysis needs to continue after an instruction executes

Surviving

Surviving

- Set the trap flag prior to executing the candidate instruction
- On trap, reload the registers to a known state

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

Surviving

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

- 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

Sifting

sandsifter - demo

```
164  r   shl ebx, 0x6b                                        c1e36b540033859ca18b2158b8ac93f3 0
         (unk)                                                9a8c42843b3e09ee955b8d47d3669fd7 0
         and edx, esi                                         23d6c9de7736d4e05487c87b901e38ee :
         imul edx, dword ptr [rbx], 0x58112d43                6913432d1158e0d5caa58f154f85d650 0
     s   movabs dword ptr [0x82d917b0fbb1eb5b], eax           a35bebb1fbb017d982b12eb7c7f5d833 1
     a   push rsp                                             54be5dbd4c5560fefbbc26fad2ebcf32 :
     n   (unk)                                                1ecf2d0ec243bac1cb16d3116caf847b 1
     d   or eax, 0x13753778                                   0d783775132a0249a46ab9f0182f2d2e 1
         ftst                                                 d9e47e167779df867a13a56b342ebf10 .
  v: 1   jbe 0xfffffffffffffb9                                76b7b83510eeef886efd644375bf4daf 4
  l: 2   jle 0xfffffffffffffdb                                7ed998f203cdbddedb2b165df18fc05f 3
  s: 5   and esi, esp                                         21e6f61038047d0d0d183b9db8855dfb3
  c: 2   and byte ptr [rax], al                               20009eae60ce7f8448f3857ecb9301d0
         push -0x33da2f5b                                     68a5d025cc8fe073716ae07966c82896
     s   in eax, dx                                           ed631809325030733742dffa080c6a50
     i   mov esi, 0xe44908d6                                  bed60849e4abbe0392a277481434afa7
     f   pop rsp                                              26445cfba6a6fad744f67f6d94c9aae7
     t   mov eax, dword ptr [rdi + rax*4 - 0x2f5561f1]        8b84870f9eaad06fd081b5c4470bb590
     e   (unk)                                                d94ae5791c35580523b6f8c694870240
     r   and dword ptr [rdx], edx                             21124b12f1f59d65adff800c0e8162c3

# 2,259,724
  39800/s
# 112

dbe11023eeb94b7a436193c6c73b60be
dbe06ea350976600eb93210563a5f39b
0f1bd311bb6376398c8cc1ab20ccdafd
dfc0a1de21248565a6838e8f5ce435f4
dfc37eff85e9ca82c485c523ba4b201e
dbe1f2552633814af7441c7cfcff0dce
dfc0abd37538a7f3035f10e704311891
0f1ae471537e81fea974e61c20ae0c91
0f0d97a9c3f2542c1047a092b1fdb66f
dfc207323fcb7c7e8b88320fc2587b18
```

(sandsifter)

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

> .... ...........              instruction:
  > 0f.... ........             0fa7c2
    > 0f0d.. ......
    > 0f18..                    prefixes:      (__)
    > 0f1a..                    valids:        (1)
    > 0f1b..                    lengths:       (3)
    > 0f1c..                    signums:       (5)
    > 0f1d..                    signals:       (sigtrap)
    > 0f1e..                    sicodes:       (2)
    > 0f1f..
    > 0fa7..                    analysis:
        0fa7c1                  capstone:
        0fa7c2                    (unk)
        0fa7c3                    n/a
        0fa7c4
        0fa7c5                  ndisasm:
        0fa7c6                    (unknown)
        0fa7c7                    n/a
    > 0fae..
  > c4.... ....                 objdump:
  > c5.... ......                 (unknown)
  > db..                          n/a
      dbe0
      dbe1
  > df..
      dfc0
      dfc1
      dfc2

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

- I scanned eight systems in my test library.

Scanning

Results

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

Hidden instructions

Scanned: Intel Core i7-4650U CPU

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

Intel hidden instructions

- 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

VIA hidden instructions

Scanned: VIA Nano U3500, VIA C7-M

- 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

- 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.

- 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.

Software bugs

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

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.

Software bugs

- Everyone parses this wrong.

Software bugs

```
             ; ---------------------------------------------------------------------------
             algn_140018EE9:                              ; DATA XREF: .pdata:00000001400256B4↓o
CC CC CC CC+                 align 10h

             ; =============== S U B R O U T I N E =======================================


                             public start
             start           proc near                    ; DATA XREF: .rdata:000000014001AA4C↓o
                                                          ; .pdata:00000001400256C0↓o
66 E9 00 00                  jmp     small $+4
             ; ---------------------------------------------------------------------------
00 00 90 90 dword_140018EF4 dd 90900000h                  ; CODE XREF: start↑j
90                           db 90h
             ; ---------------------------------------------------------------------------
48 83 C4 28                  add     rsp, 28h
E9 06 00 00+                 jmp     sub_140018F08
00           start           endp

             ; ---------------------------------------------------------------------------
```

Software bugs (IDA)

Disassembly

Address: 00007ff7b9ef8ef0()

Viewing Options

```
00007FF7B9EF8EE4 48 83 C4 38          add         rsp,38h
00007FF7B9EF8EE8 C3                   ret
00007FF7B9EF8EE9 CC                   int         3
00007FF7B9EF8EEA CC                   int         3
00007FF7B9EF8EEB CC                   int         3
00007FF7B9EF8EEC CC                   int         3
00007FF7B9EF8EED CC                   int         3
00007FF7B9EF8EEE CC                   int         3
00007FF7B9EF8EEF CC                   int         3
00007FF7B9EF8EF0 66 E9 00 00 00 00    jmp         0000000000008EF6
00007FF7B9EF8EF6 90                   nop
00007FF7B9EF8EF7 90                   nop
00007FF7B9EF8EF8 90                   nop
00007FF7B9EF8EF9 48 83 C4 28          add         rsp,28h
00007FF7B9EF8EFD E9 06 00 00 00       jmp         00007FF7B9EF8F08
00007FF7B9EF8F02 CC                   int         3
00007FF7B9EF8F03 CC                   int         3
00007FF7B9EF8F04 CC                   int         3
00007FF7B9EF8F05 CC                   int         3
00007FF7B9EF8F06 CC                   int         3
00007FF7B9EF8F07 CC                   int         3
00007FF7B9EF8F08 48 8B C4             mov         rax,rsp
00007FF7B9EF8F0B 48 89 58 08          mov         qword ptr [rax+8],rbx
00007FF7B9EF8F0F 48 89 70 10          mov         qword ptr [rax+10h],rsi
00007FF7B9EF8F13 48 89 78 18          mov         qword ptr [rax+18h],rdi
00007FF7B9EF8F17 41 57                push        r15
00007FF7B9EF8F19 48 81 EC B0 00 00 00 sub         rsp,0B0h
```


Software bugs (VS)

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

```
--
00000000004004ed <main>:
  4004ed:       55                              push   %rbp
  4004ee:       48 89 e5                        mov    %rsp,%rbp
  4004f1:       66 e9 00 00                     jmpw   4f5 <_init-0x3ffeb3>
  4004f5:       05 00 00 00 00                  add    $0x0,%eax
  4004fa:       05 00 00 00 00                  add    $0x0,%eax
  4004ff:       48 b8 b8 11 22 33 44 ff e0 90   movabs $0x90e0ff44332211b8,%rax
  400509:       48 b8 b8 11 22 33 44 ff e0 90   movabs $0x90e0ff44332211b8,%rax
  400513:       48 b8 b8 11 22 33 44 ff e0 90   movabs $0x90e0ff44332211b8,%rax
  40051d:       48 b8 b8 11 22 33 44 ff e0 90   movabs $0x90e0ff44332211b8,%rax
  400527:       48 b8 b8 11 22 33 44 ff e0 90   movabs $0x90e0ff44332211b8,%rax
  400531:       48 b8 b8 11 22 33 44 ff e0 90   movabs $0x90e0ff44332211b8,%rax
  40053b:       48 b8 b8 11 22 33 44 ff e0 90   movabs $0x90e0ff44332211b8,%rax
```

Software bugs (objdump)

```
root@delta-vm:~#
```

Software bugs (QEMU)

Software bugs

- 66 jmp
- Why does everyone get this wrong?
- AMD: override changes operand to 16 bits, instruction pointer truncated
- Intel: override ignored.

- 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

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

```
deltaop:~/_research/sandsifter$
>>> 
```

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

Intel hardware bugs

- Scanned:
  - Quark, Pentium, Core i7

Intel hardware bugs

- f00f bug on Pentium (anti-climactic)

AMD hardware bugs

- Scanned:
  - Geode NX1500, C-50

- 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 …

AMD hardware bugs

Table 8-8. Simultaneous Interrupt Priorities

Interrupt Priority	Interrupt Condition	Interrupt Vector
(High) 0	Processor Reset	—
	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

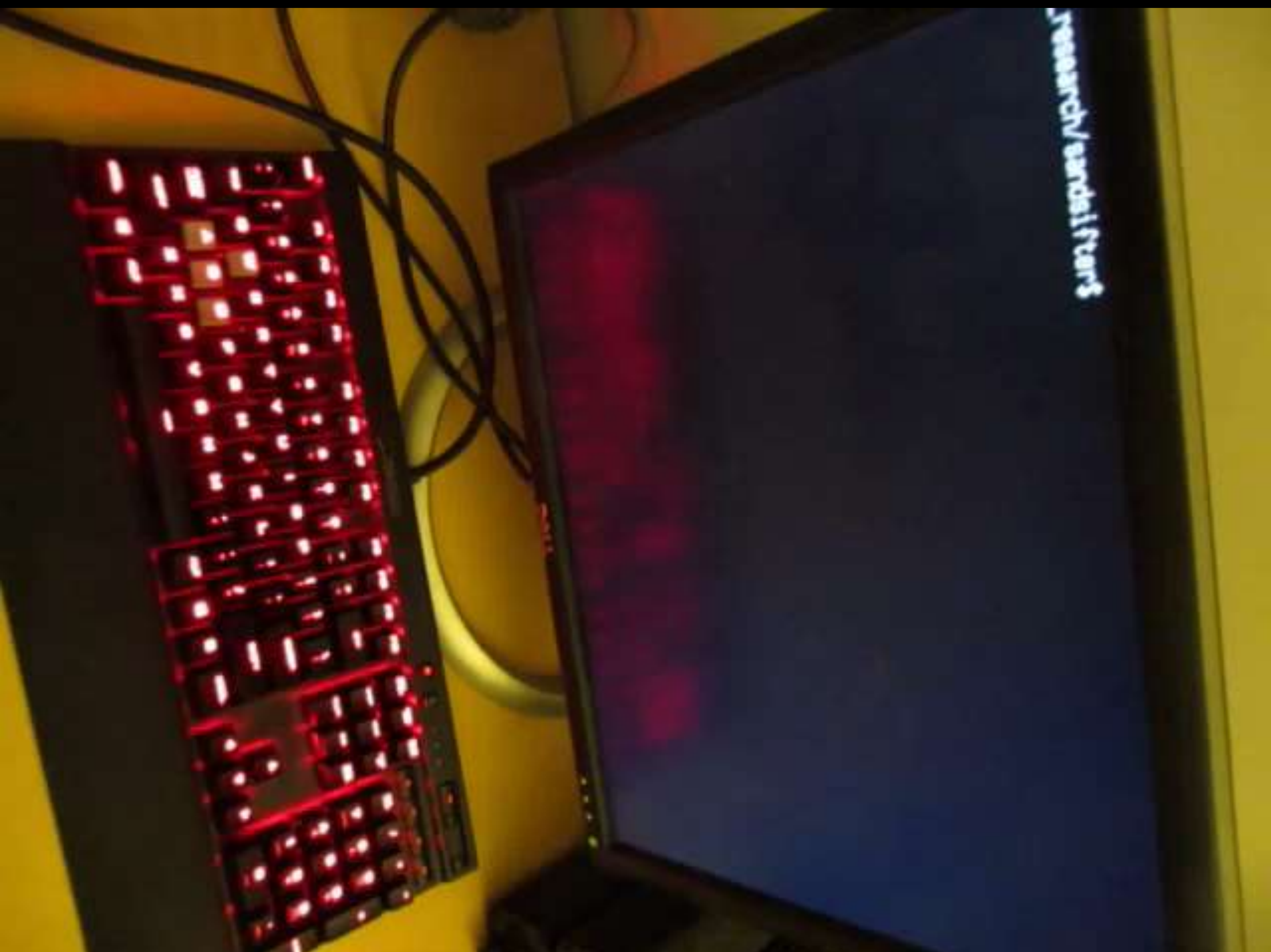
Transmeta hardware bugs

- 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

- 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 details available on chip, vendor, or instructions

hardware bugs

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)

[redacted] 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

Don’t blindly trust the specifications.

Conclusions

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

Conclusions

- github.com/xoreaxeaxeax
  - sandsifter
  - M/o/Vfuscator
  - REpsych
  - x86 0-day PoC
  - Etc.
- Feedback? Ideas?
- domas
  - @xoreaxeaxeax
  - xoreaxeaxeax@gmail.com