Breaking the x86 ISA

domas / @xoreaxeaxeax / DEF CON 2017
Christopher Domas
Cyber Security Researcher @ Battelle Memorial Institute

./bio
8086: 1978
A long, tortured history...

The x86 ISA
Modes:

- Real (Unreal)
- Protected mode (Virtual 8086, SMM)
- Long mode (Compatibility, PAE)

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

x86: evolution
We don’t trust software.

- We audit it
- We reverse it
- We break it
- We sandbox it

Trust.
But the processor itself?
We blindly trust.

Trust.
Why?
Hardware has all the same problems as software

Secret functionality?
- Appendix H.

Bugs?
- F00F, TSX, Hyperthreading.

Vulnerabilities?
- SYSRET, cache poisoning, sinkhole

Trust.
We should stop blindly trusting our hardware.
What do we need to worry about?
Well known from software
Examples

Backdoors
Hardware

- FPGAs
- Hypervisors
- Microcode
- Supply chain

Backdoors
Could a hidden instruction unlock your CPU?
Historical examples

ICEBP

apicall

Hidden instructions
| Table A-2: One-byte Opcode Map: (COH – F7H) *

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gx, Gv</td>
<td>AL, b</td>
<td>bAX, lx</td>
<td>PUSH ESP4</td>
</tr>
<tr>
<td>1</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gx, Gv</td>
<td>AL, b</td>
<td>bAX, lx</td>
<td>PUSH ESP4</td>
</tr>
<tr>
<td>2</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gx, Gv</td>
<td>AL, b</td>
<td>bAX, lx</td>
<td>SEG-ES (Prefix)</td>
</tr>
<tr>
<td>3</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gx, Gv</td>
<td>AL, b</td>
<td>bAX, lx</td>
<td>SEG-SS (Prefix)</td>
</tr>
<tr>
<td>4</td>
<td>aAX REX</td>
<td>aAX REX</td>
<td>aAX REX</td>
<td>aAX REX</td>
<td>aAX REX</td>
<td>aAX REX</td>
<td>aAX REX</td>
</tr>
<tr>
<td>5</td>
<td>aAX REX</td>
<td>aAX REX</td>
<td>aAX REX</td>
<td>aAX REX</td>
<td>aAX REX</td>
<td>aAX REX</td>
<td>aAX REX</td>
</tr>
<tr>
<td>6</td>
<td>PUSH4</td>
<td>PUSH4</td>
<td>POPA4</td>
<td>POPA4</td>
<td>POPA4</td>
<td>POPA4</td>
<td>POPA4</td>
</tr>
<tr>
<td>7</td>
<td>O</td>
<td>NO</td>
<td>BNE/NE</td>
<td>NBNE/NC</td>
<td>ZE</td>
<td>WNE</td>
<td>BE/NA</td>
</tr>
<tr>
<td>8</td>
<td>EB, ib</td>
<td>EB, ib</td>
<td>EB, ib</td>
<td>EB, ib</td>
<td>EB, ib</td>
<td>EB, ib</td>
<td>EB, ib</td>
</tr>
<tr>
<td>9</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
</tr>
<tr>
<td>10</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
</tr>
<tr>
<td>11</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
</tr>
<tr>
<td>12</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
</tr>
<tr>
<td>13</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
</tr>
<tr>
<td>14</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
</tr>
<tr>
<td>15</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
<td>MOV4</td>
</tr>
</tbody>
</table>

Legend:
- **b**: Short-displacement jump on condition
- **ux**: general register / REX4/Preface
- **ib**: Immediate 8-bit, double-word or quad-word register with RAX
- **lx**: XCHD word, double-word or quad-word register with RAX
- **ib**: IMMEDIATE byte into byte-register
- **ib**: IMMEDIATE byte into byte-register
- **ib**: IMMEDIATE byte into byte-register
- **ib**: IMMEDIATE byte into byte-register
- **ib**: IMMEDIATE byte into byte-register
- **ib**: IMMEDIATE byte into byte-register

Hidden instructions

Note: This table represents the one-byte opcode map for certain ranges of values (COH to F7H) in Assembly language, with specific instructions and prefixes highlighted for emphasis.
Traditional approaches:
- Leaked documentation
- Reverse engineering software
- NDA

But what if it’s something stealthy?

Hidden instructions
Goal: Audit the Processor

Find out what’s really there
How to find hidden instructions?

Approach
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

Approach
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

Approach
A depth-first-search algorithm

[Overview]
- Catch: requires knowing the instruction length
- 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

(Overview)
 Trap flag
   - Catch branching instructions
   - Differentiate between fault types

Cleanup
Reduces search space from $1.3 \times 10^{36}$ instructions to $\sim 100,000,000$ (one day of scanning)

This gives us a way to *search* the instructions space.

How do we make sense of the instructions we execute?

Tunneling
We need a “ground truth”

- Use a disassembler
  - It was written based on the documentation
  - Capstone
Compare:

- Observed length of instruction vs. disassembled length of instruction
- Signal generated by instruction vs. expected signal
sandsifter
Hidden instructions
Ubiquitous software bugs
Hypervisor flaws
Hardware bugs

Results
Undocumented for non-/1 reg fields

Undocumented until December 2016

Undocumented for non-0 r/m fields until June 2014

Hidden instructions
Catch:
- Undocumented instructions recognized by the disassembler are not found

Hidden instructions
ISSUE:

Our “ground truth” (the disassembler) is also prone to errors.
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
Software bugs

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

- 66 jmp
- Demo:
  - IDA
  - Visual Studio
  - objdump
  - QEMU
66 jmp

Why does everyone get this wrong?

- AMD designed the 64 bit architecture
- Intel adopted... most of it.
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

Complex architecture
- Tools cannot parse a jump instruction

Software bugs
Hypervisor bugs

Azure

CPUID / Trap flag bug
Hardware bugs

- Intel:
  - f00f bug on Pentium

- AMD:
  - Incorrect signals during decode

- Transmeta:
  - Of\{71,72,73\}xxxx
  - Premature #GP0 signal during decode
Our processors are not doing what we think they are

- We need formal specifications
- We need auditing tools
- This is a start.

Conclusions
Sandsifter lets us introspect what is otherwise a black box
Open sourced:

- The sandsifter scanning tool
- github.com/xoreaxeaxeax/sandsifter

Conclusions
Use sandsifter to audit your processor
Reveal the instructions it really supports
Search for hardware errata
Break disassemblers, emulators, and hypervisors
Send us the results

Conclusions
github.com/xoreaxeaxeaxe

{sandsifter
M/o/Vfuscator
REpsych
x86 0-day PoC
Etc.

Feedback? Ideas?

domas

@xoreaxeaxeaxe
xoreaxeaxeaxe@gmail.com