The ring 0 façade: awakening the processor's inner demons

{ domas / @xoreaxeaxeax / DEF CON 2018

- Christopher Domas
  - Cyber Security Researcher

./bio

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.

- General-purpose-registers
- Special-purpose-registers
- FPU, MMX, XMM, YMM, ZMM
- Control registers
- *Model-specific-registers*

Processor registers

- Debugging
- Execution tracing
- Performance monitoring
- Clocking
- Thermal controls
- Cache behavior

Model-specific-registers

- Accessing MSRs:
  - Ring 0
  - Accessed by *address*, not *name*
    - 0x00000000 – 0xFFFFFFFF
  - Only a small fraction are implemented
    - 10s – few 100s
  - 64 bits
  - Read with rdmsr
    - Read to edx:eax
  - Written with wrmsr
    - Written from edx:eax

Model-specific-registers

```
movl $0x1a0, %%ecx /* load msr address */

rdmsr /* read msr 0x1a0 */

/* configure new values for msr */
orl $1, %%eax
orl $4, %%edx

wrmsr /* write msr 0x1a0 */
```

Model-specific-registers

- Undocumented debug features
- Unlock disabled cores
- Hardware backdoors

Secrets…

"

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 .

"

- US 8341419

Secrets…

"

Nevertheless, the existence and location of the undocumented control MSRs are easily found by programmers, who typically then publish their findings for all to use.

Furthermore, a processor manufacturer may need to disclose the addresses and description of the control MSRs to its customers for their testing and debugging purposes.

The disclosure to the customer may result in the secret of the control MSRs becoming widely known, and thus usable by anyone on any processor.

"

- US 8341419

Secrets…

"

The microprocessor also includes a secret key, manufactured internally within the microprocessor and externally invisible.

The microprocessor also includes an encryption engine, coupled to the secret key,

configured to decrypt a user-supplied password using the secret key to generate a decrypted result

in response to a user instruction instructing the microprocessor to access the control register.

"

- US 8341419

Secrets...

Secrets…

- Could my processor have…

secret,

undocumented,

password protected

…registers in it, right now?

- AMD K7, K8
  - Known password protected MSRs
  - Discovered through firmware RE
  - 32 bit password loaded into a GPR
- Let's start here, as a case study

Password protections

```
movl $0x12345678, %%edi  /* password */
movl $0x1337c0de, %%ecx  /* msr */
rdmsr

/* if MSR 0x1337c0de does not exist,
        CPU generates #GP(0) exception */

/* if password 0x12345678 is incorrect,
        CPU generates #GP(0) exception */
```

Password protections

- Challenge:
  - To detect a password protected MSR, must guess the MSR address *and* the MSR password
  - Guessing either one wrong gives the same result: #GP(0) exception
  - 32 bit address + 32 bit password = 64 bits

Password protections

```
// naive password protected MSR identification

for msr in 0 to 0xffffffff:
    for p in 0 to 0xffffffff:
        p -> eax, ebx, edx, esi, edi, esp, ebp
        msr -> ecx
        try:
            rdmsr
        catch:
            // fault: incorrect password, or msr does not exist
            continue
        // no fault: correct password, and msr exists
        return (msr, p)
```

Password protections

- Even in the simple embodiment (32 bit passwords)
  - At 1,000,000,000 guesses per second
  - Finding all password-protected MSRs takes 600 years

Password protections

How might we detect whether our processor is hiding password protected registers, without needing to know the password first?

Password protections

- Assembly is a high level abstraction
  - CPU micro-ops execute assembly instructions

Speculation

- Possible pseudocode for microcoded MSR accesses (rdmsr, wrmsr):

```
if msr == 0x1:
   ... // (service msr 0x1)
elif msr == 0x6:
   ... // (service msr 0x6)
elif msr == 0x1000:
   ... // (service msr 0x1000)
else:
   // msr does not exist
   // raise general protection exception
   #gp(0)
```

Speculation

- Possible pseudocode for password-protected microcoded MSR accesses (rdmsr, wrmsr):

```
if msr == 0x1:
    ... // (service msr 0x1)
elif msr == 0x6:
    ... // (service msr 0x6)
elif msr == 0x1337c0de:
    // password protected register – verify password
    if ebx == 0xfeedface:
            ... // (service msr 0x1337c0de)
    else:
            // wrong password
            // raise general protection exception
            #gp(0)
else:
    // msr does not exist
    // raise general protection exception
    #gp(0)
```

Speculation

- Microcode:
  - Checks if the user is accessing a password-protected register
  - Then checks if supplied password is correct
- Same visible result to the user
- But…
  - Accessing a password-protected MSR takes a *slightly different* amount of time than accessing a non-password-protected MSR

Speculation

- Non-password-protected path:

```
if msr == 0x1:
   … // (service msr 0x1)
elif msr == 0x6:
   … // (service msr 0x6)
elif msr == 0x1337c0de:
   // password protected register – verify password
   if ebx == 0xfeedface:
                 … // (service msr 0x1337c0de)
   else:
                 // wrong password
                 // raise general protection exception
                 #gp(0)
else:
   // msr does not exist
   // raise general protection exception
   #gp(0)
```

Speculation

- Password-protected path:

```
if msr == 0x1:
   ... // (service msr 0x1)
elif msr == 0x6:
   ... // (service msr 0x6)
elif msr == 0x1337c0de:
   // password protected register – verify password
   if ebx == 0xfeedface:
                 ... // (service msr 0x1337c0de)
   else:
                 // wrong password
                 // raise general protection exception
                 #gp(0)
else:
   // msr does not exist
   // raise general protection exception
   #gp(0)
```

Speculation

- Depending on exact implementation
  - Password-protected path may be longer or shorter
  - But should be *different*
- Possible to craft each path to have identical execution time
  - Complexities of microcode + no public research attacking MSRs = seems unlikely

Speculation

```
mov %[_msr], %%ecx            /* load msr */

mov %%eax, %%dr0              /* serialize */
rdtsc                         /* start time */
movl %%eax, %%ebx

rdmsr                         /* access msr */

rdmsr_handler:                /* exception handler */

mov %%eax, %%dr0              /* serialize */
rdtsc                         /* end time */

subl %%ebx, %%eax             /* calculate access time */
```

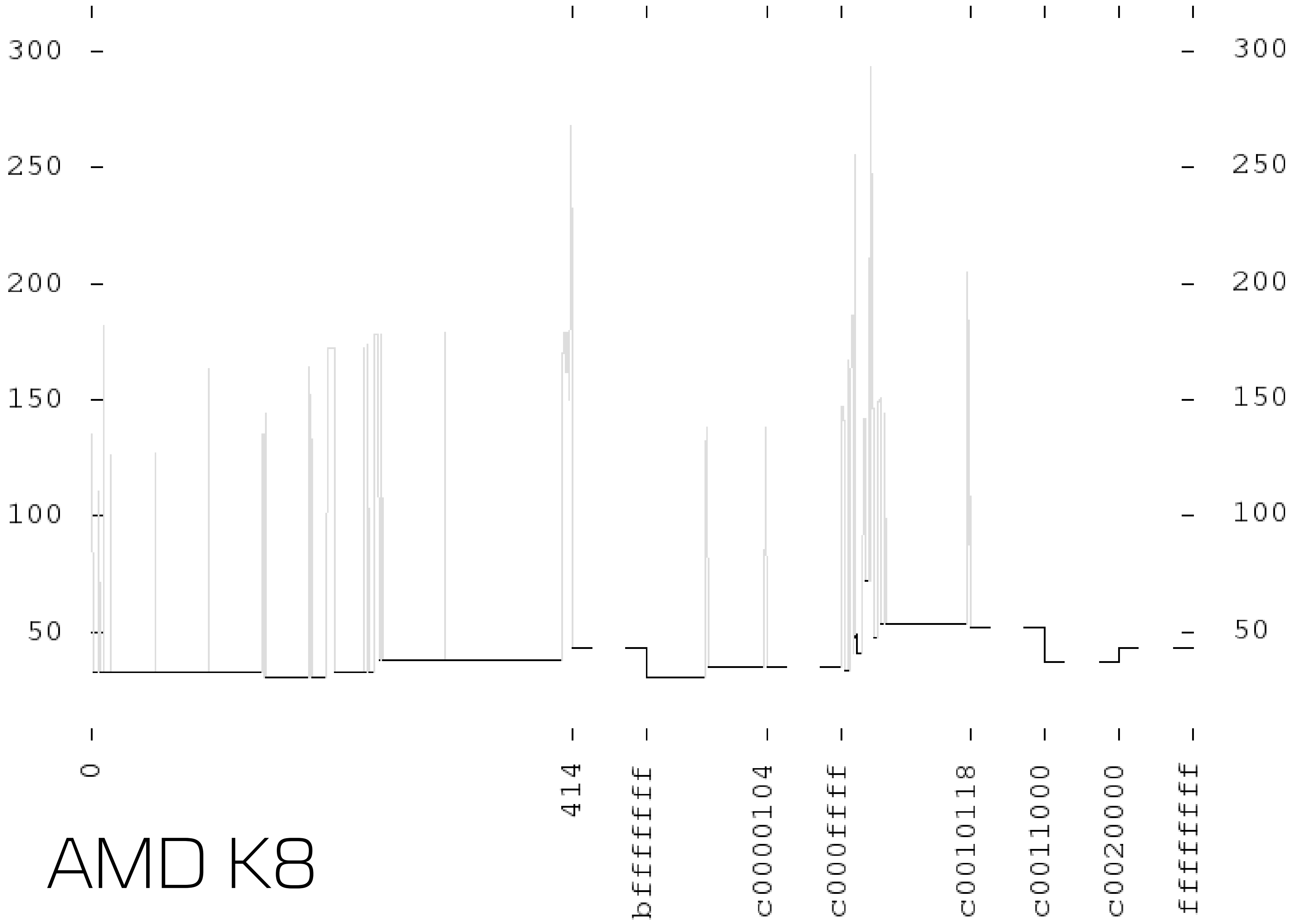
A timing side-channel

- Attack executed in ring 0 kernel module
- #GP(0) exception is redirected
  to instruction following rdmsr
- System stack reset after
  each measurement
  to avoid specific fault handling logic

A timing side-channel

- Initial configuration routine measures execution time of a #GP(0) exception (by executing a ud2 instruction)
  - Subtracted out of faulting MSR measurements
- Serialization handles out-of-order execution
- Simplicity: only track low 32 bits of timer
- Repeat sample, select lowest measurement

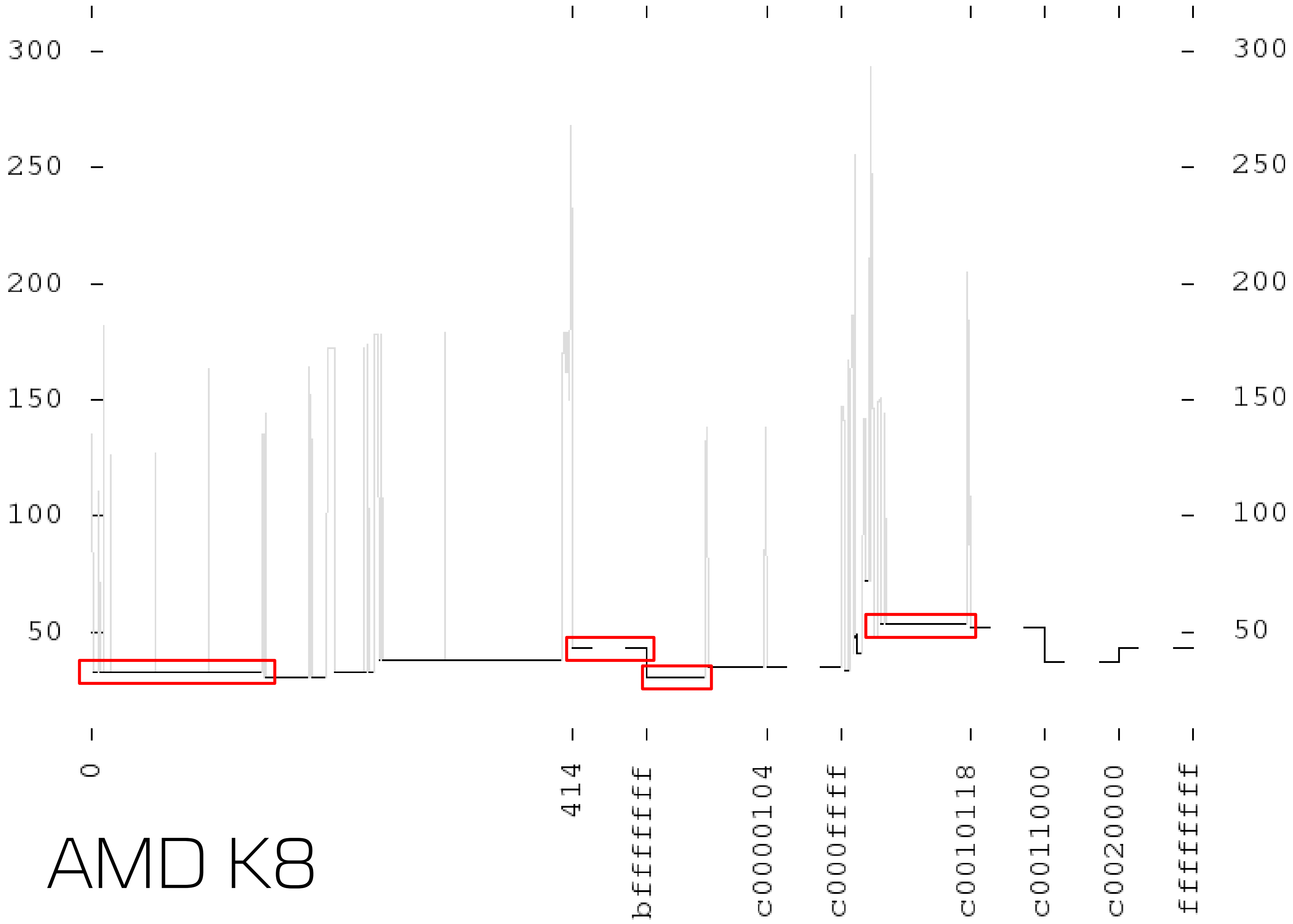
A timing side-channel


300
250
200
150
100
50
0
414
bfffffff
c0000104
c000ffff
c0010118
c0011000
c0020000
ffffffff

AMD K8

- Timing measurements let us speculate on a rough model of the underlying microcode
- Specifically, focused on variations in observed fault times.

A timing side-channel


300
250
200
150
100
50
0
414
bfffffff
c0000104
c000ffff
c0010118
c0011000
c0020000
ffffffff

AMD K8

- Speculate that separate ucode paths exist for processing each fault group.
  - Practically speaking, breaking ucode MSR checks into groups improves execution time

A timing side-channel

// possible k8 ucode model

```
if msr < 0x174:
        if msr == 0x0: ...
        elif msr == 0x1: ...
        elif msr == 0x10: ...
        ...
        else: #gp(0)
elif msr < 0x200:
        if msr == 0x174: ...
        ...
        else: #gp(0)
elif msr < 0x270:
        if msr == 0x200: ...
        ...
        else: #gp(0)
elif msr < 0x400:
        if msr == 0x277: ...
        ...
        else: #gp(0)
elif msr < 0xc0000000:
        if msr == 0x400: ...
        ...
        else: #gp(0)
elif msr < 0xc0000080:
        #gp(0)
elif msr < 0xc0010000:
        if msr == 0xc0000080: ...
        ...
        else: #gp(0)
elif msr < 0xc0011000:
        if msr == 0xc0010000: ...
        ...
        else: #gp(0)
elif msr < 0xc0020000:
        #gp(0)
else:
        #gp(0)
```

- Find the bounds checks that
  appear to *exist for no purpose*
  - i.e. regions explicitly checked by ucode,
    even though there are no visible MSRs within them

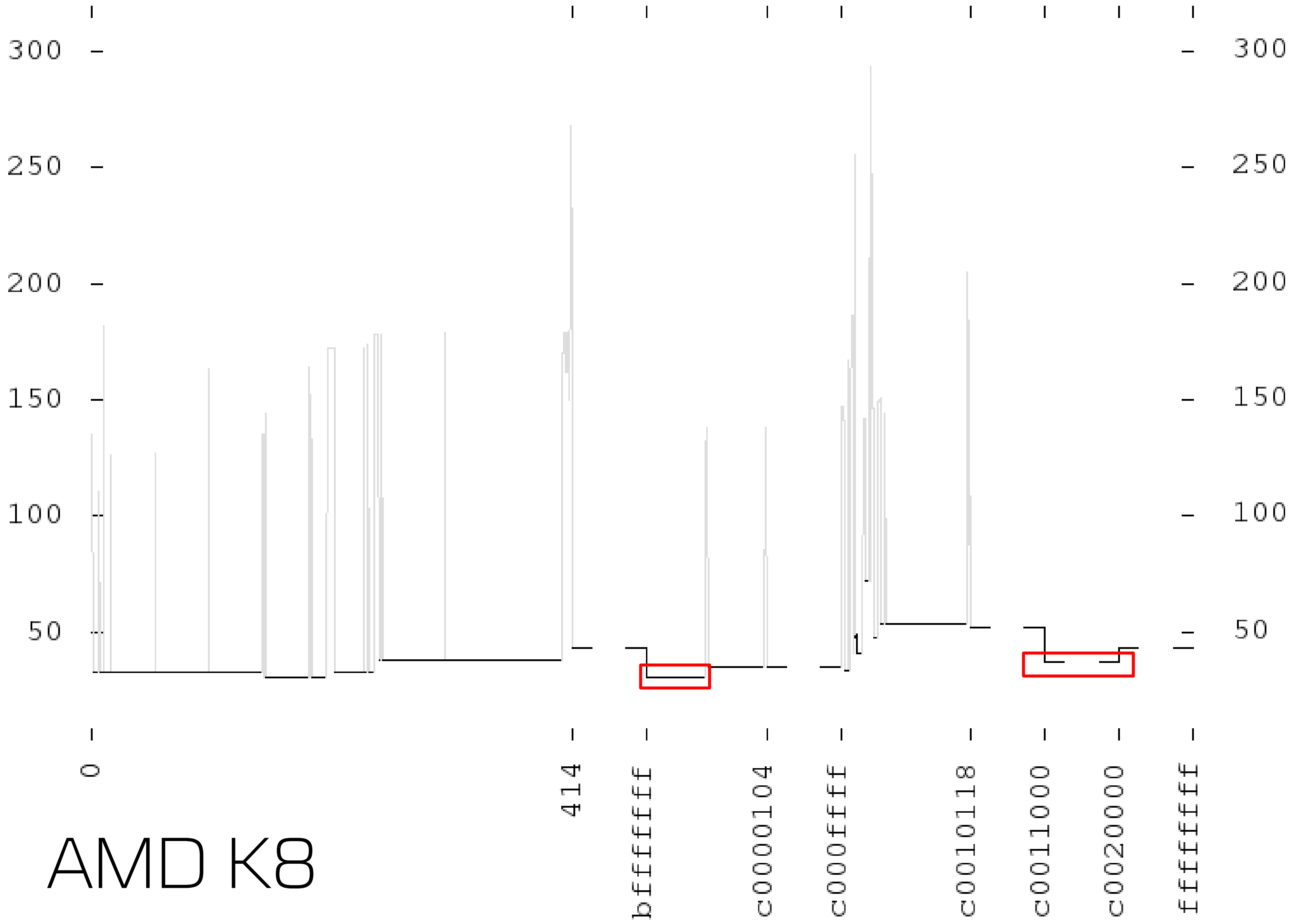
A timing side-channel

// possible k8 ucode model

```
if msr < 0x174:
        if msr == 0x0: ...
        elif msr == 0x1: ...
        elif msr == 0x10: ...
        ...
        else: #gp(0)
elif msr < 0x200:
        if msr == 0x174: ...
        ...
        else: #gp(0)
elif msr < 0x270:
        if msr == 0x200: ...
        ...
        else: #gp(0)
elif msr < 0x400:
        if msr == 0x277: ...
        ...
        else: #gp(0)
elif msr < 0xc0000000:
        if msr == 0x400: ...
        ...
        else: #gp(0)
elif msr < 0xc0000080:
        #gp(0)
elif msr < 0xc0010000:
        if msr == 0xc0000080: ...
        ...
        else: #gp(0)
elif msr < 0xc0011000:
        if msr == 0xc0010000: ...
        ...
        else: #gp(0)
elif msr < 0xc0020000:
        #gp(0)
else:
        #gp(0)
```

- Timings show that there are explicit ucode checks on the regions:
  - 0xC0000000 – 0xC000007F
  - 0xC0011000 – 0xC001FFFF
  - … even though there are no visible MSRs in those regions

A timing side-channel


300
250
200
150
100
50
0
414
bfffffff
c0000104
c000ffff
c0010118
c0011000
c0020000
ffffffff

AMD K8

- Speculate that anomalies *must* be due to password checks
  - Reduces MSR search space by 99.999%
  - Cracking passwords is now feasible

Cracking protected registers

- Simple embodiment:
  - 32 bit password
  - Loaded into GPR or XMM register
  - Use list of side-channel derived MSRs
  - Continue until MSR is unlocked, or all passwords are tried

Cracking protected registers

```
// side-channel assisted password identification

for msr in [0xC0000000:0xC000007F, 0xC0011000: 0xC001FFFF]:
  for p in 0 to 0xffffffff:
    p -> eax, ebx, edx, esi, edi, esp, ebp
    msr -> ecx
    try:
      rdmsr
    catch:
      // fault: incorrect password, or msr does not exist
      continue
    // no fault: correct password, and msr exists
    return (msr, p)
```

Cracking protected registers

- Cracked the AMD K8
  - Password 0x9c5a203a loaded into edi
  - MSRs: 0xc0011000 – 0xc001ffff
  - Check on 0xc0000000 – 0xc000007f remains unexplained

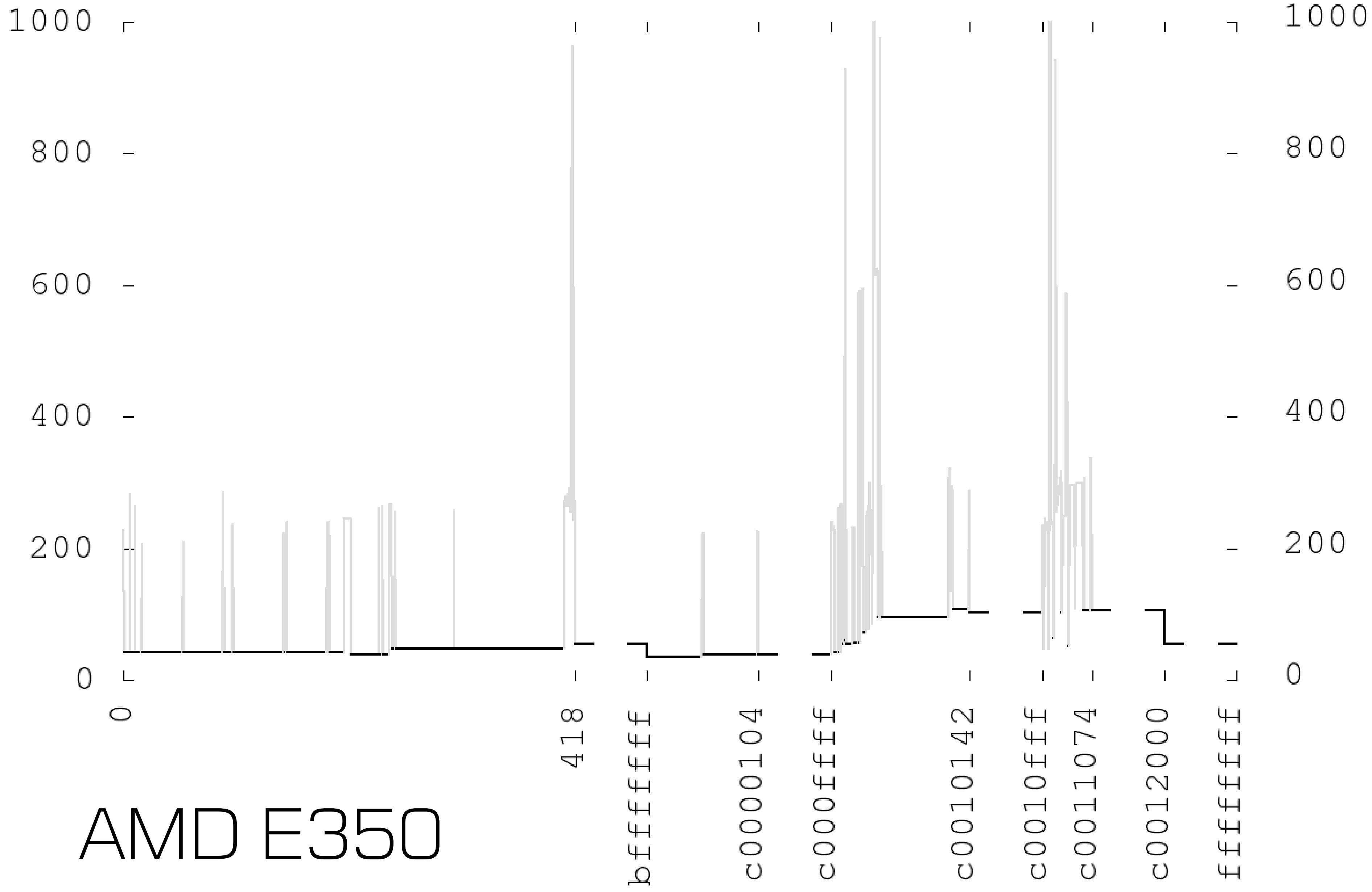
Cracking protected registers

- This region and password were already known through firmware reverse engineering
- But this is the first approach to uncovering these MSRs without first observing them in use

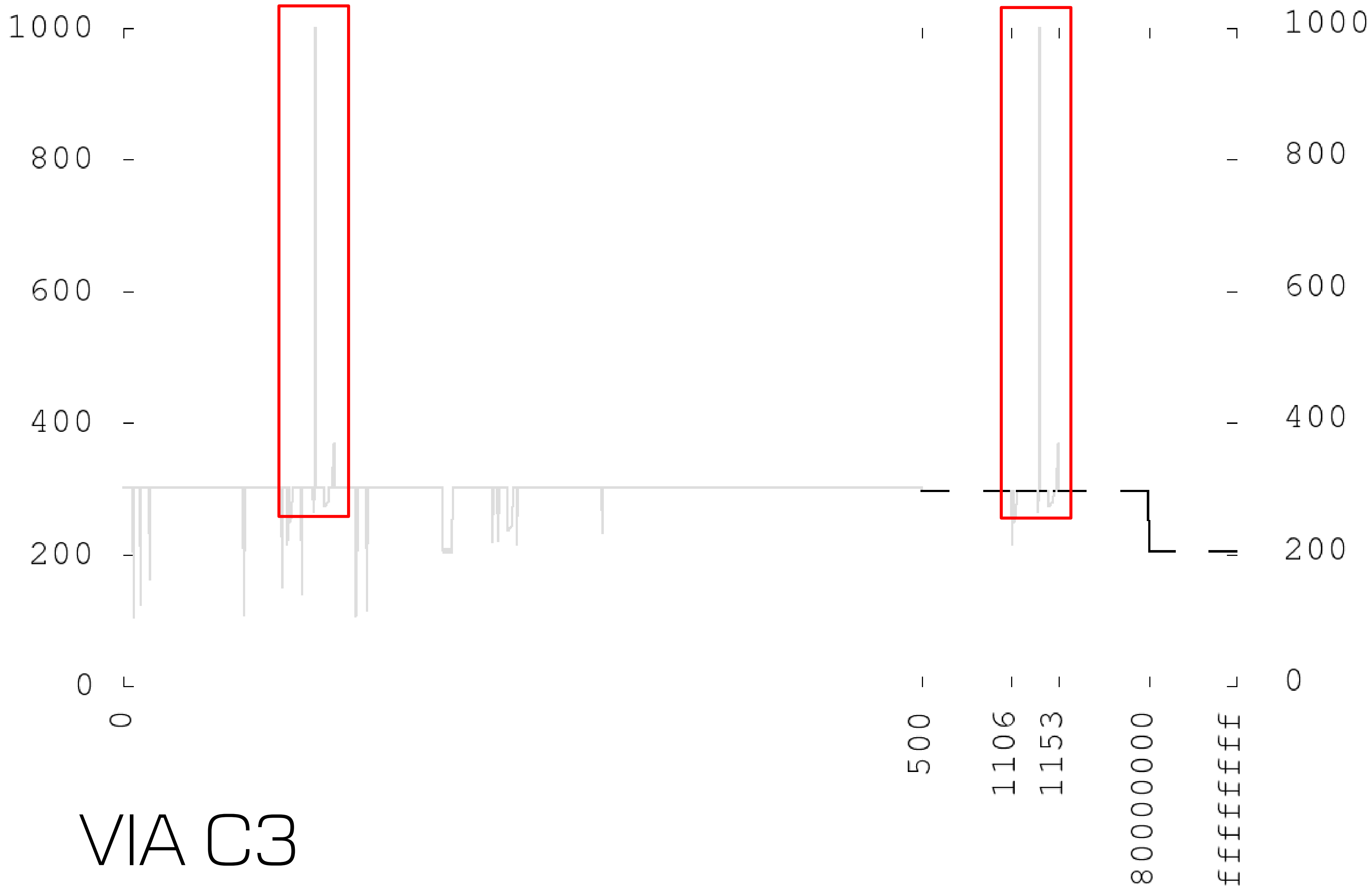
Cracking protected registers

- Side-channel attacks into the microcode offer a powerful opportunity
- … so what else can we find?

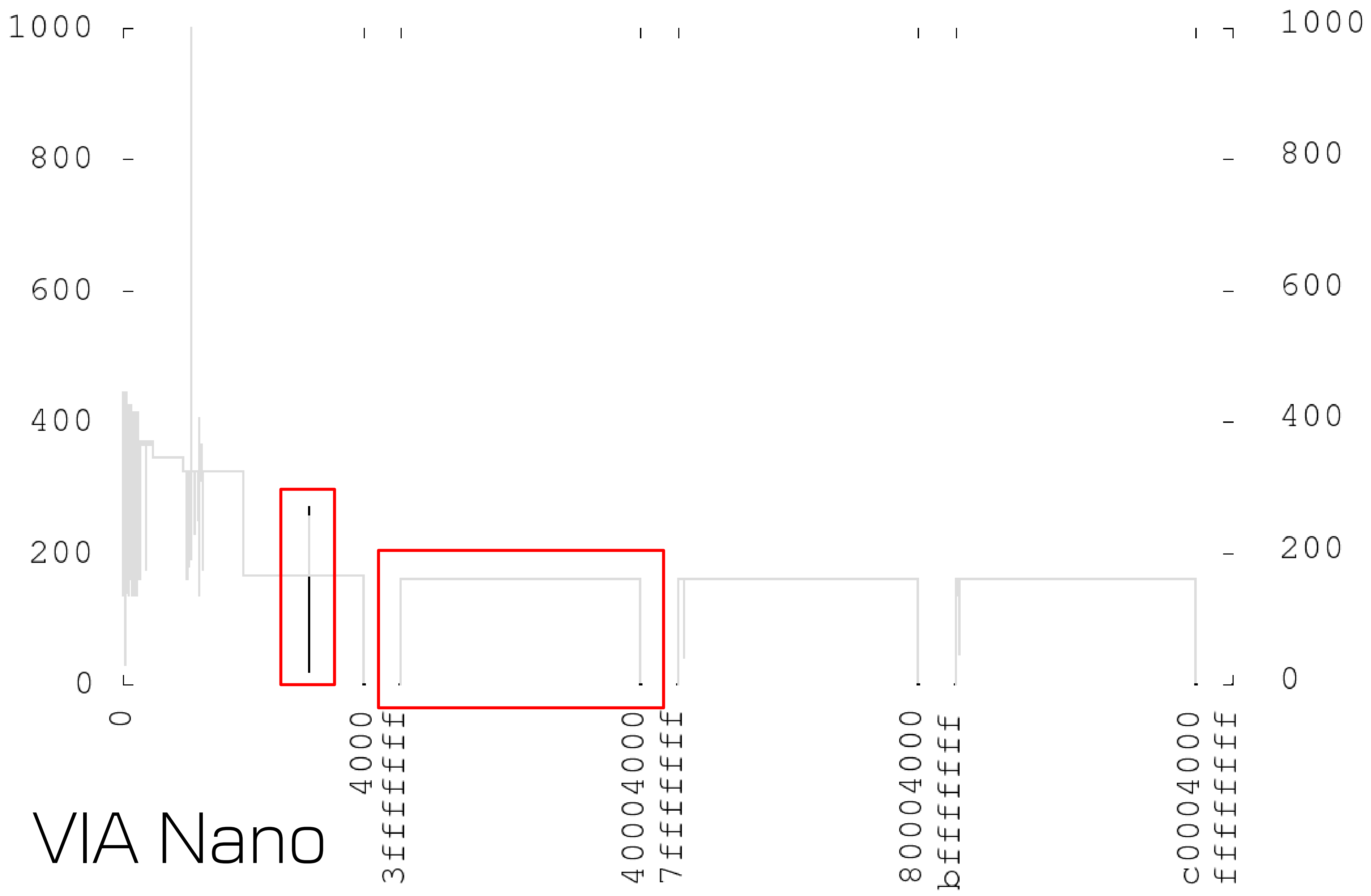
Digging deeper…


1000
800
600
400
200
0
0
418
bfffffff
c0000104
c000ffff
c0010142
c0010fff
c0011074
c0012000
ffffffff

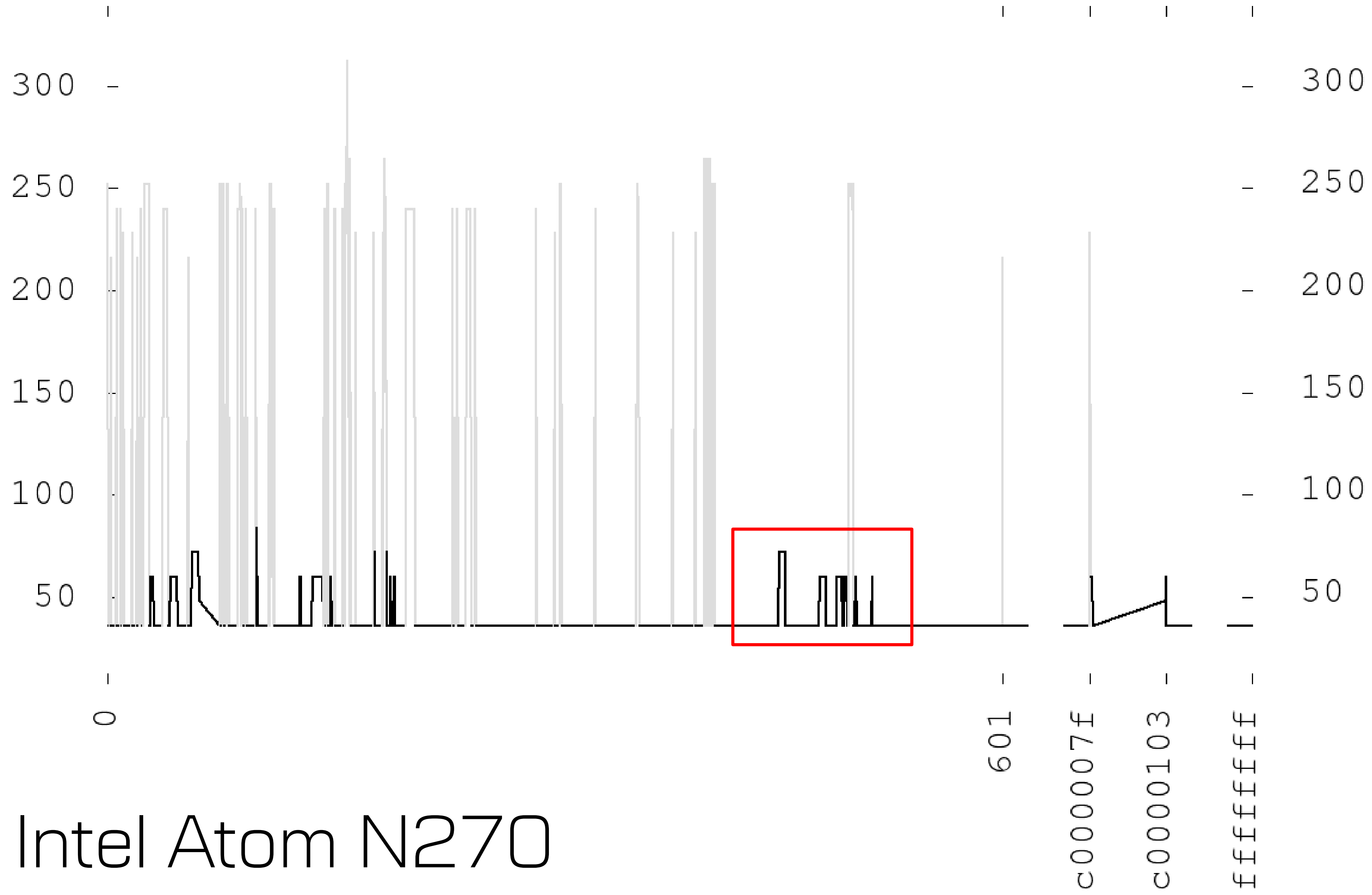
AMD E350


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

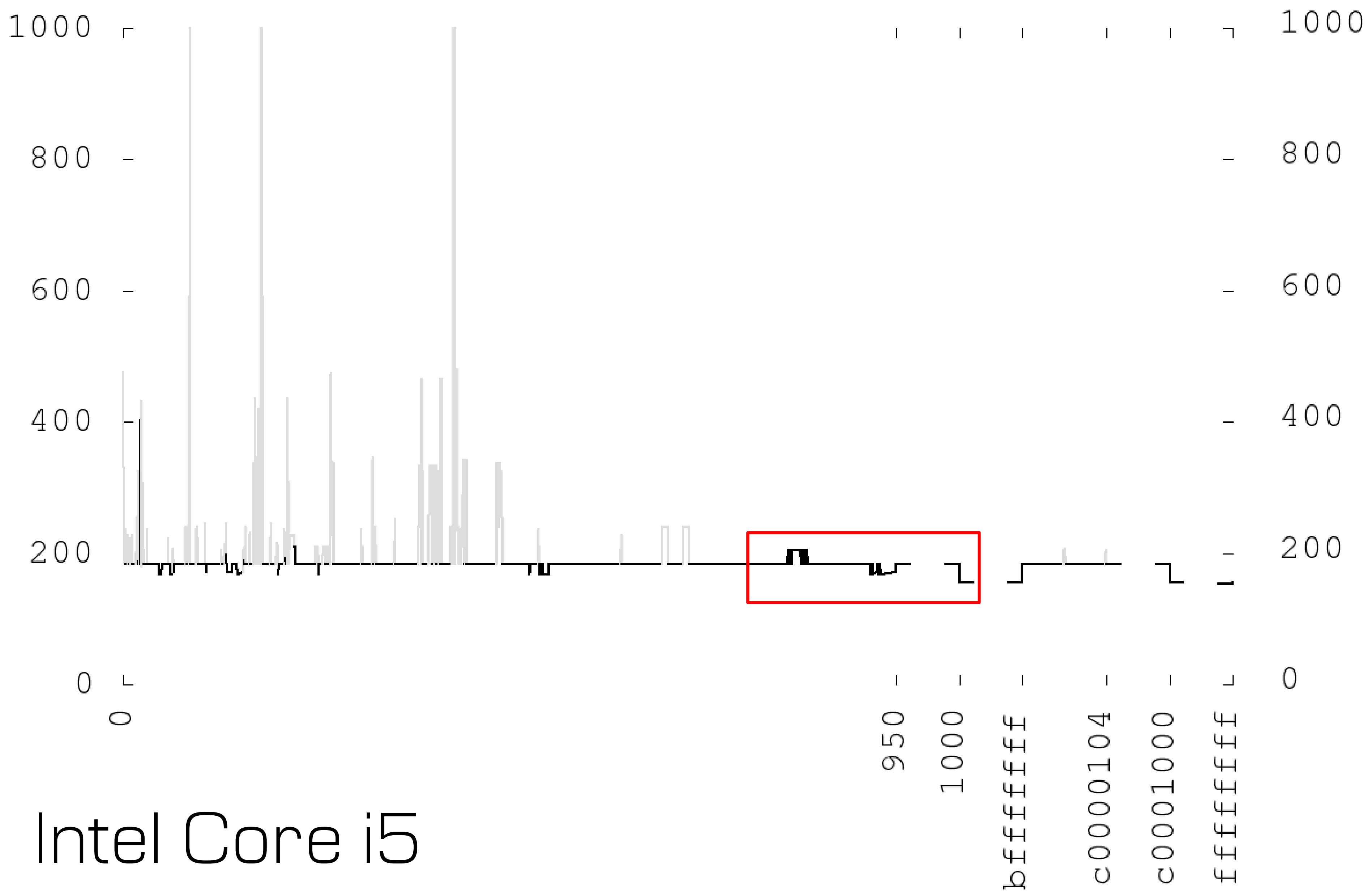
VIA C3


1000
800
600
400
200
0
0
4000
3fffffff
40004000
7fffffff
80004000
bfffffff
c0004000
ffffffff

VIA Nano



Intel Atom N270



Intel Core i5

Advanced cracking

- Cracking extensions:
  - Write protected MSRs
  - 64 bit password in 2 32 bit registers
    - Accessible from real mode

- And…
  - Failed.
  - No new passwords uncovered.

Advanced cracking

- Sometimes, that’s research.

Conclusions

Conclusions

- How to explain the timing anomalies?
  - 64/128/256 bit passwords in XMM etc. registers
  - More advanced password checks,
    as described in patent literature
  - MSRs only accessible in
    ultra-privileged modes beyond ring 0

- … or, something totally benign:
  - Microcode checks on processor family, model, stepping
    - Allow one ucode update to be used on many processors
  - Timing anomalies in MSR faults on Intel processors
    seemed to accurately align with
    specific documented MSRs on related families

Conclusions

Conclusions

- So, we're in the clear?
  - Sadly, no.
  - Instruction grep through firmware databases reveals previously unknown passwords:
    - 0x380dcb0f in esi register
    - Hundreds of firmwares, variety of vendors
    - Windows kernel

- We've raised more questions than we've answered
- But the stakes are high:
  - MSRs control *everything* on the processor
- Research is promising
  - Entirely new approach to detecting processor secrets

The truth is out there…

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

- Feedback? Ideas?

- domas
  - @xoreaxeaxeax
  - xoreaxeaxeax@gmail.com