

Custom Processing Unit: Tracing and Patching Intel Atom Microcode

Black Hat USA 2022

Pietro Borrello

Sapienza University of Rome

Michael Schwarz

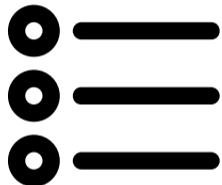
CISPA Helmholtz Center for Information Security

Martin Schwarzl

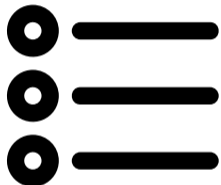
Graz University of Technology

Daniel Gruss

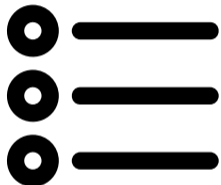
Graz University of Technology



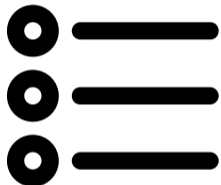
1. Deep dive on CPU μ code



1. Deep dive on CPU μ code
2. μ code Software Framework



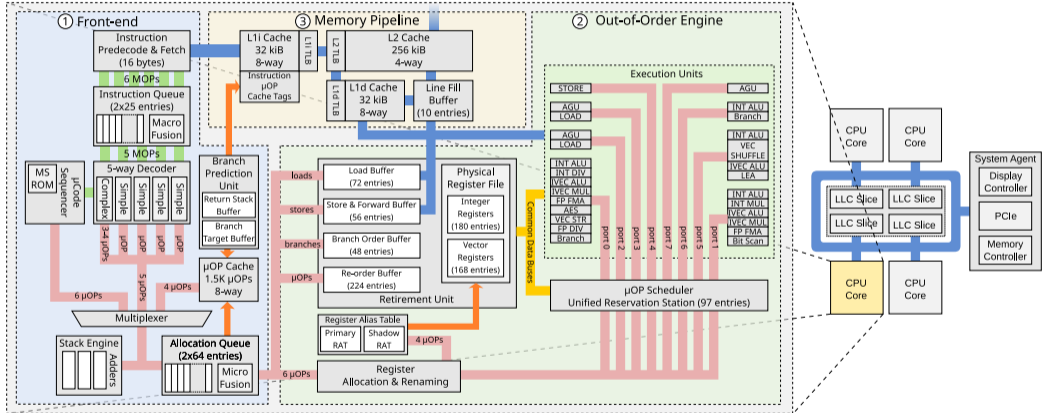
1. Deep dive on CPU μ code
2. μ code Software Framework
3. Reverse Engineering of the secret μ code update algorithm



1. Deep dive on CPU μ code
2. μ code Software Framework
3. Reverse Engineering of the secret μ code update algorithm
4. Some bonus content ;)



- This is based on **our understanding** of CPU Microarchitecture.
- In theory, it may be **all wrong**.
- In practice, a lot **seems right**.





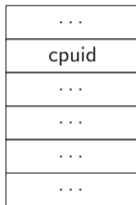
- Red Unlock of Atom Goldmont (GLM) CPUs



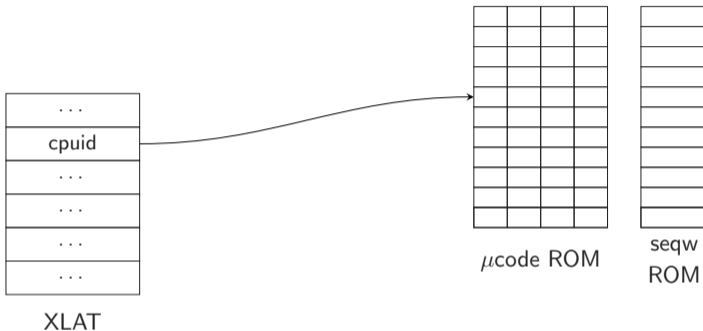
- **Red Unlock** of Atom Goldmont (GLM) CPUs
- **Extraction** and **reverse engineering** of GLM μ code format

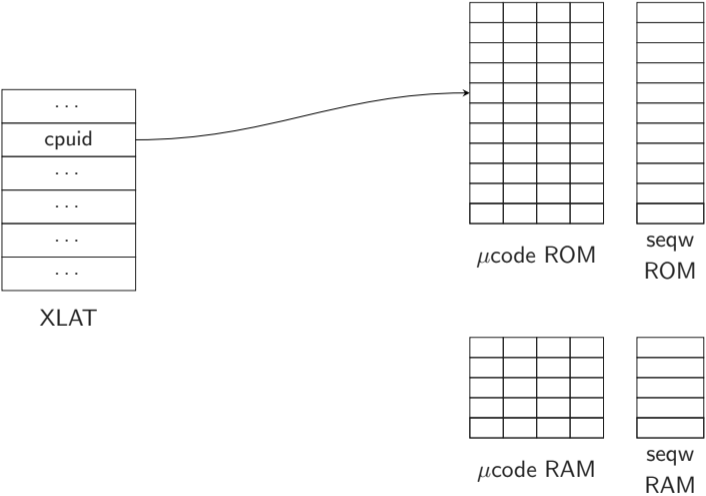


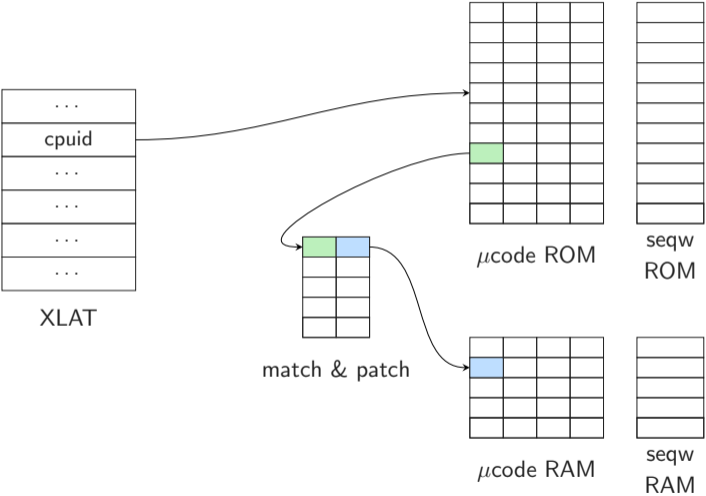
- **Red Unlock** of Atom Goldmont (GLM) CPUs
- **Extraction** and **reverse engineering** of GLM μ code format
- Discovery of undocumented control instructions to access **internal** buffers



XLAT







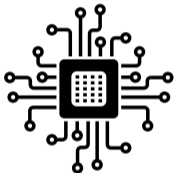
OP1	OP2	OP3	SEQW
09282eb80236	0008890f8009	092830f80236	0903e480

```
U1a54: 09282eb80236          CMPUJZ_DIRECT_NOTTAKEN(tmp6, 0x2, U0e2e)
U1a55: 0008890f8009          tmp8:= ZEROEXT_DSZ32(0x2389)
U1a56: 092830f80236          SYNC-> CMPUJZ_DIRECT_NOTTAKEN(tmp6, 0x3, U0e30)
U1a57: 000000000000          NOP
SEQW:      0903e480          SEQW GOTO U03e4
```

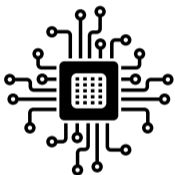


```
U32f0: 002165071408      tmp1:= CONCAT_DSZ32(0x04040404)
U32f1: 004700031c75      tmp1:= NOTAND_DSZ64(tmp5, tmp1)
U32f2: 006501031231      tmp1:= SHR_DSZ64(tmp1, 0x00000001)
      |   |   |   01c4c980      SEQW GOTO U44c9
-----
U32f4: 0251f25c0278      UJMPC DIRECT_NOTTAKEN_CONDNS(tmp8, U37f2)
U32f5: 006275171200      tmp1:= MOVEFROMCREG_DSZ64( , PMH_CR_EMRR_MASK)
U32f6: 186a11dc02b1      BTUJB DIRECT_NOTTAKEN(tmp1, 0x0000000b, generate_#GP) !m0,m1
      |   |   |   01e15080      SEQW GOTO U6150
-----
U32f8: 000c85e80280      SAVEUIP( , 0x01, U5a85) !m0
U32f9: 000406031d48      tmp1:= AND_DSZ32(0x00000006, tmp5)
U32fa: 1928119c0231      CMPUJZ DIRECT_NOTTAKEN(tmp1, 0x00000002, generate_#GP) !m0,m1
      |   |   |   0187bd80      SEQW GOTO U07bd
-----
U32fc: 00251a032235      tmp2:= SHR_DSZ32(tmp5, 0x0000001a)
U32fd: 0062c31b1200      tmp1:= MOVEFROMCREG_DSZ64( , 0x6c3)
U32fe: 000720031c48      tmp1:= NOTAND_DSZ32(0x00000020, tmp1)
      |   |   |   01c4d580      SEQW GOTO U44d5
-----
```

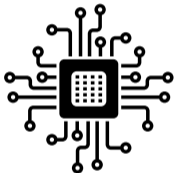
```
1 |
2 | void rc4_decrypt(ulong tmp0_i,ulong tmp1_j,byte *ucode_patch_tmp5,int len_tmp6,byte *S_tmp7,
3 |                 long callback_tmp8)
4 |
5 | {
6 |     byte bVar1;
7 |     byte bVar2;
8 |
9 |     do {
10 |         tmp0_i = (ulong)(byte)((char)tmp0_i + 1);
11 |         bVar1 = S_tmp7[tmp0_i];
12 |         tmp1_j = (ulong)(byte)(bVar1 + (char)tmp1_j);
13 |             /* swap S[i] and S[j] */
14 |         bVar2 = S_tmp7[tmp1_j];
15 |         S_tmp7[tmp0_i] = bVar2;
16 |         S_tmp7[tmp1_j] = bVar1;
17 |         *ucode_patch_tmp5 = S_tmp7[(byte)(bVar2 + bVar1)] ^ *ucode_patch_tmp5;
18 |         ucode_patch_tmp5 = ucode_patch_tmp5 + 1;
19 |         len_tmp6 += -1;
20 |     } while (len_tmp6 != 0);
21 |     (*(code *) (callback_tmp8 * 0x10))();
22 |     return;
23 | }
24 |
```



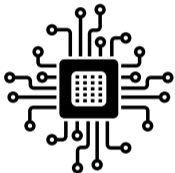
- CPU interacts with its internal components through the CRBUS



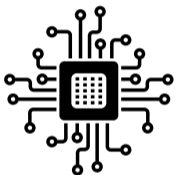
- CPU interacts with its internal components through the CRBUS
- MRSs → CRBUS addr



- CPU interacts with its internal components through the CRBUS
- MRSs → CRBUS addr
- **Control** and **Status** registers



- CPU interacts with its internal components through the CRBUS
- MRSs → CRBUS addr
- **Control** and **Status** registers
- **SMM** configuration



- CPU interacts with its internal components through the CRBUS
- MRSs → CRBUS addr
- **Control** and **Status** registers
- **SMM** configuration
- Local Direct Access Test (**LDAT**) access



- The μ code Sequencer manages the access to μ code ROM and RAM



- The μ code Sequencer manages the access to μ code ROM and RAM
- The LDAT has access to the μ code Sequencer



- The μ code Sequencer manages the access to μ code ROM and RAM
- The LDAT has access to the μ code Sequencer
- We can access the LDAT through the CRBUS



- The μ code Sequencer manages the access to μ code ROM and RAM
- The LDAT has access to the μ code Sequencer
- We can access the LDAT through the CRBUS
- If we can access the CRBUS we can control μ code!

Positive Technologies discovered the existence of two secret instructions that can access (RW):



- System agent
- URAM
- Staging buffer
- I/O ports
- Power supply unit

Positive Technologies discovered the existence of two secret instructions that can access (RW):



- System agent
- URAM
- Staging buffer
- I/O ports
- Power supply unit
- CRBUS

```
def CRBUS_WRITE(ADDR, VAL):  
    udbgwr(  
        rax: ADDR,  
        rbx|rdx: VAL,  
        rcx: 0,  
    )
```

```
//Decompile of: U2782 - part of ucode update routine  
write_8 (crbus_06a0, (ucode_address - 0x7c00));  
MSLOOPCTR = (*(ushort *)((long)ucode_update_ptr + 3) - 1);  
syncmark();  
if ((in_ucode_ustate & 8) != 0) {  
    syncfull();  
    write_8 (crbus_06a1, 0x30400);  
    ucode_ptr = (ulong *)((long)ucode_update_ptr + 5);  
    do {  
        ucode_qword = *ucode_ptr;  
        ucode_ptr = ucode_ptr + 1;  
        write_8 (crbus_06a4, ucode_qword);  
        write_8 (crbus_06a5, ucode_qword >> 0x20);  
        syncwait();  
        MSLOOPCTR --; 1;  
    } while (-1 < MSLOOPCTR);  
    syncfull();  
}
```

```
def ucode_sequencer_write(SELECTOR, ADDR, VAL):
    CRBUS[0x6a1] = 0x30000 | (SELECTOR << 8)
    CRBUS[0x6a0] = ADDR
    CRBUS[0x6a4] = VAL & 0xffffffff
    CRBUS[0x6a5] = VAL >> 32
    CRBUS[0x6a1] = 0

with SELECTOR:
    2 -> SEQW  PATCH RAM
    3 -> MATCH & PATCH
    4 -> UCODE PATCH RAM
```


Redirects execution from μ code ROM to μ code RAM to execute patches.

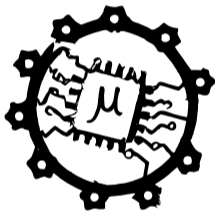
```
patch_off = (patch_addr - 0x7c00) / 2;
```

```
entry:
```

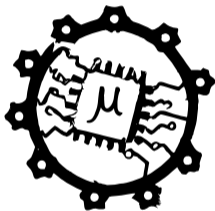
```

+---+-----+-----+-----+-----+
|3e| patch_off |      match_addr      |enb|
+---+-----+-----+-----+-----+
 24         16                1    0

```

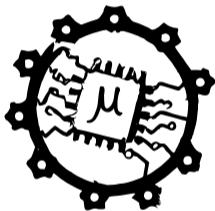


Leveraging `udbgrd/wr` we can patch μ code via software



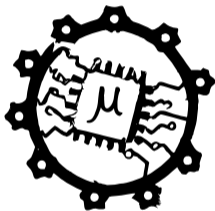
Leveraging `udbgrd/wr` we can patch μ code via software

- Completely `observe` CPU behavior



Leveraging `udbgrd/wr` we can patch μ code via software

- Completely **observe** CPU behavior
- Completely **control** CPU behavior



Leveraging `udbgrd/wr` we can patch μ code via software

- Completely **observe** CPU behavior
- Completely **control** CPU behavior
- All within a **BIOS** or **kernel** module



Patch μcode



Patch μcode



Hook μcode



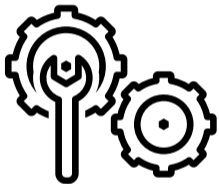
Patch μcode



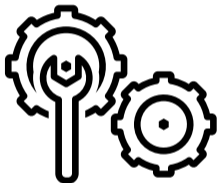
Hook μcode



Trace μcode

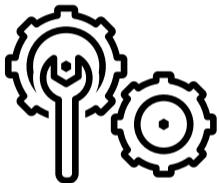


We can change the CPU's behavior.



We can change the CPU's behavior.

- Change microcoded instructions



We can change the CPU's behavior.

- Change microcoded instructions
- Add functionalities to the CPU

```
.patch 0x0428 # RDRAND ENTRY POINT
.org 0x7c00
rax:= ZEROEXT_DSZ64(0x6f57206f6c6c6548) # 'Hello Wo'
rbx:= ZEROEXT_DSZ64(0x21646c72) # 'rld!\x00'
UEND
```

```
.patch 0x0428 # RDRAND ENTRY POINT
.org 0x7c00
rax:= ZEROEXT_DSZ64(0x6f57206f6c6c6548) # 'Hello Wo'
rbx:= ZEROEXT_DSZ64(0x21646c72) # 'rld!\x00'
UEND
```

1. Assemble µcode
2. Write µcode at 0x7c00
3. Setup Match & Patch: 0x0428 → 0x7c00
4. rdrand → "Hello World!"

rdrand returns random data, what if we make it return SMM memory?

```
.patch 0x0428 # RDRAND ENTRY POINT
.org 0x7c00
tmp1:= MOVEFROMCREG_DSZ64(CR_SMRR_MASK)
tmp2:= ZEROEXT_DSZ64(0x0)
MOVETOCREG_DSZ64(tmp2, CR_SMRR_MASK) # DISABLE SMM MEMORY RANGE

rax:= LDPPHYS_DSZ64(0x7b000000) # SMROM ADDR

MOVETOCREG_DSZ64(tmp1, CR_SMRR_MASK)
UEND
```

rdrand returns random data, what if we make it return SMM memory?

```
.patch 0x0428 # RDRAND ENTRY POINT
.org 0x7c00
tmp1:= MOVEFROMCREG_DSZ64(CR_SMRR_MASK)
tmp2:= ZEROEXT_DSZ64(0x0)
MOVETOCREG_DSZ64(tmp2, CR_SMRR_MASK) # DISABLE SMM MEMORY RANGE

rax:= LDPPHYS_DSZ64(0x7b000000) # SMROM ADDR

MOVETOCREG_DSZ64(tmp1, CR_SMRR_MASK)
UEND
```

rdrand returns random data, what if we make it return SMM memory?

```
.patch 0x0428 # RDRAND ENTRY POINT
.org 0x7c00
tmp1:= MOVEFROMCREG_DSZ64(CR_SMRR_MASK)
tmp2:= ZEROEXT_DSZ64(0x0)
MOVETOCREG_DSZ64(tmp2, CR_SMRR_MASK) # DISABLE SMM MEMORY RANGE

rax:= LDPPHYS_DSZ64(0x7b000000) # SMROM ADDR

MOVETOCREG_DSZ64(tmp1, CR_SMRR_MASK)
UEND
```


rdrand returns random data, what if we make it return SMM memory?

```
.patch 0x0428 # RDRAND ENTRY POINT
.org 0x7c00
tmp1:= MOVEFROMCREG_DSZ64(CR_SMRR_MASK)
tmp2:= ZEROEXT_DSZ64(0x0)
MOVETOCREG_DSZ64(tmp2, CR_SMRR_MASK) # DISABLE SMM MEMORY RANGE

rax:= LDPPHYS_DSZ64(0x7b000000) # SMROM ADDR

MOVETOCREG_DSZ64(tmp1, CR_SMRR_MASK)
UEND
```

DEMO

BH DEMO

fs0:\EFI> █

BH DEMO

fs0:\EFI> █



Install μcode hooks to observe events.

- Setup Match & Patch to execute custom μcode at certain events
- Resume execution

We can make the CPU to react to certain μ code events, e.g., `verw` executed

```
.patch 0xXXXX # INSTRUCTION ENTRY POINT
.org 0x7da0

tmp0:= ZEROEXT_DSZ64(<counter_address>)
tmp1:= LDPPHYSTICKLE_DSZ64_ASZ64_SC1(tmp0)
tmp1:= ADD_DSZ64(tmp1, 0x1) # INCREMENT COUNTER
STADPPHYSTICKLE_DSZ64_ASZ64_SC1(tmp0, tmp1)

UJMP(0xXXXX + 1) # JUMP TO NEXT UOP
```

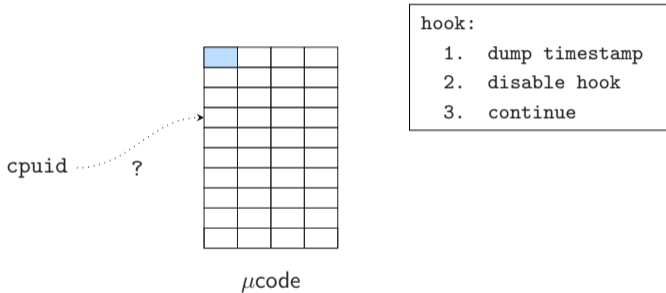
We can make the CPU to react to certain μ code events, e.g., `verw` executed

```
.patch 0xXXXX # INSTRUCTION ENTRY POINT
.org 0x7da0

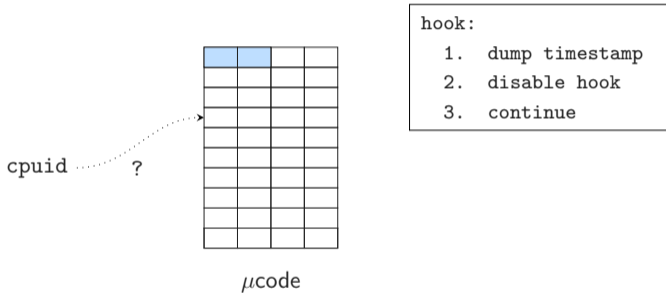
tmp0:= ZEROEXT_DSZ64(<counter_address>)
tmp1:= LDPPHYSTICKLE_DSZ64_ASZ64_SC1(tmp0)
tmp1:= ADD_DSZ64(tmp1, 0x1) # INCREMENT COUNTER
STADPPHYSTICKLE_DSZ64_ASZ64_SC1(tmp0, tmp1)

UJMP(0xXXXX + 1) # JUMP TO NEXT UOP
```

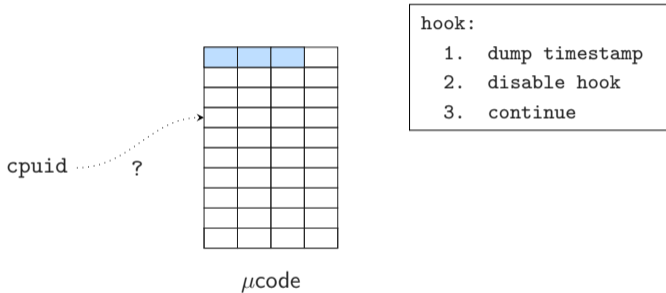
Trace μcode execution leveraging hooks.



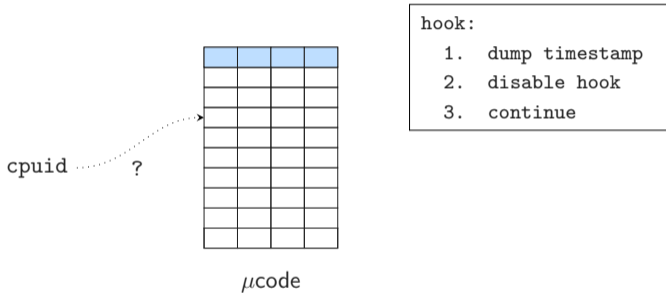
Trace μcode execution leveraging hooks.



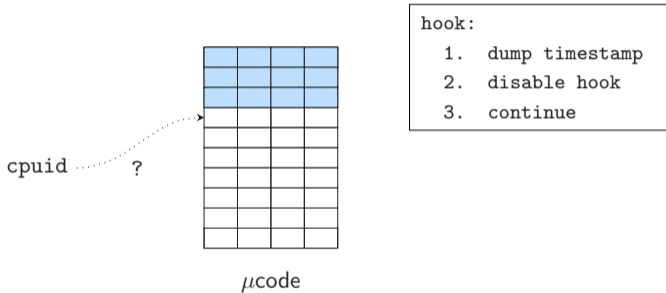
Trace μcode execution leveraging hooks.



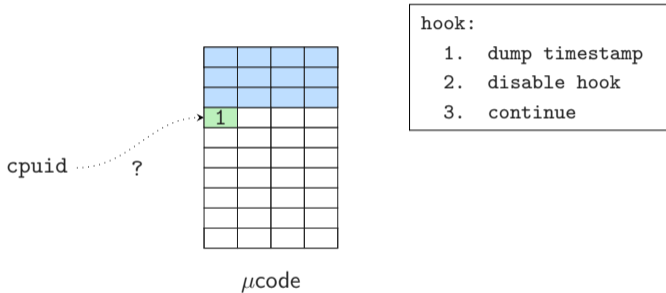
Trace μcode execution leveraging hooks.



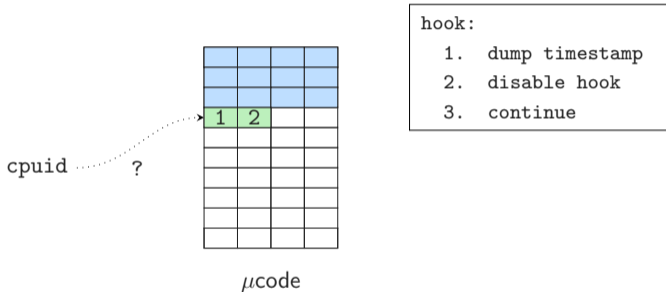
Trace μcode execution leveraging hooks.



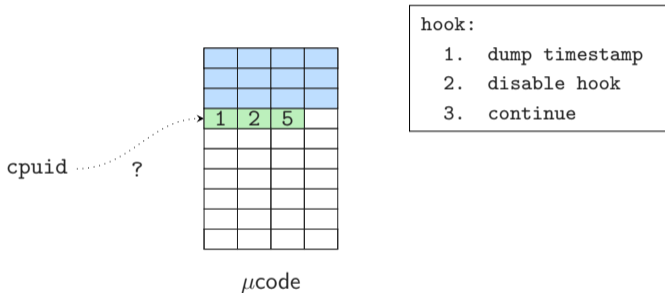
Trace μcode execution leveraging hooks.



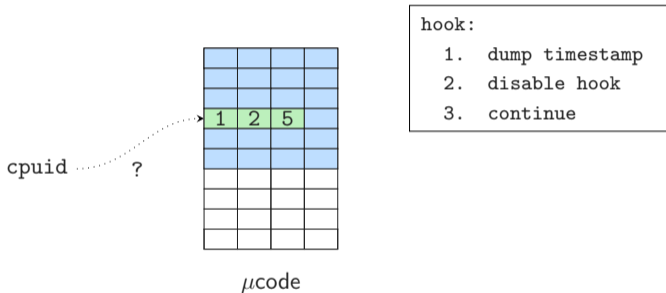
Trace μcode execution leveraging hooks.



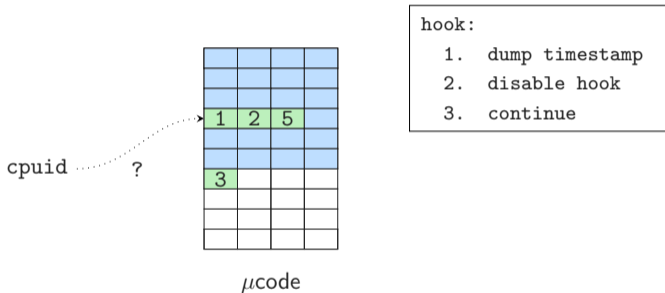
Trace μcode execution leveraging hooks.



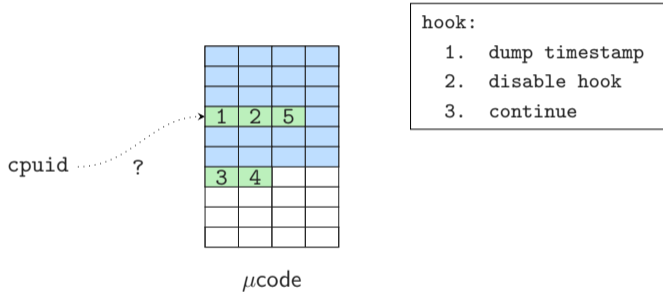
Trace μcode execution leveraging hooks.



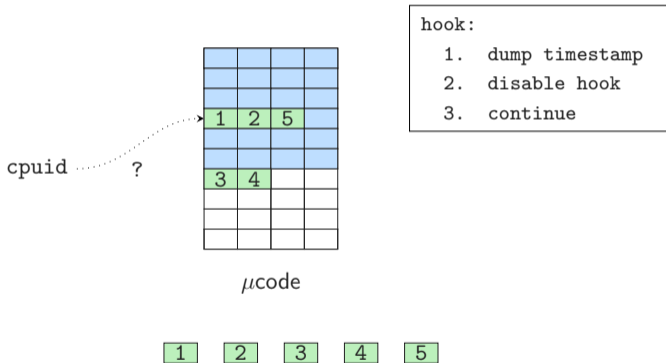
Trace μcode execution leveraging hooks.



Trace μcode execution leveraging hooks.



Trace μcode execution leveraging hooks.





μ code update algorithm has always been kept secret by Intel
Let's trace the execution of a μ code update!



μ code update algorithm has always been kept secret by Intel
Let's trace the execution of a μ code update!

- Trigger a μ code update



μ code update algorithm has always been kept secret by Intel
Let's trace the execution of a μ code update!

- Trigger a μ code update
- Trace if a microinstruction is executed



μ code update algorithm has always been kept secret by Intel
Let's trace the execution of a μ code update!

- Trigger a μ code update
- Trace if a microinstruction is executed
- Repeat for all the possible μ code instructions

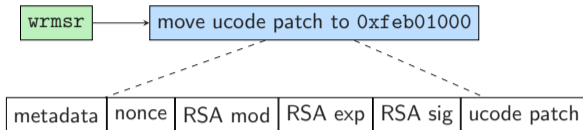


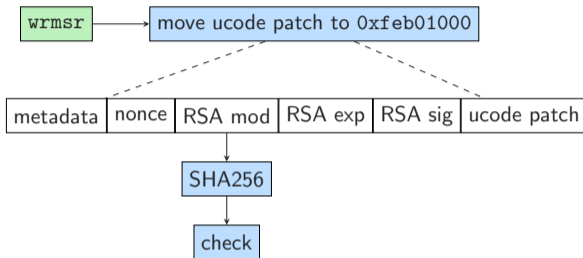
μ code update algorithm has always been kept secret by Intel
Let's trace the execution of a μ code update!

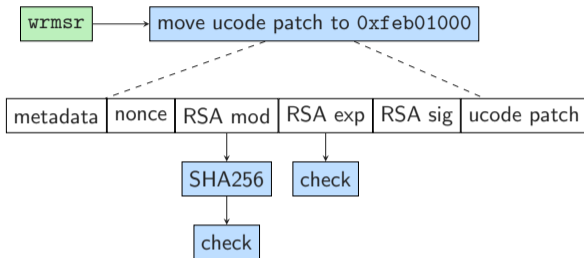
- Trigger a μ code update
- Trace if a microinstruction is executed
- Repeat for all the possible μ code instructions
- Restore order

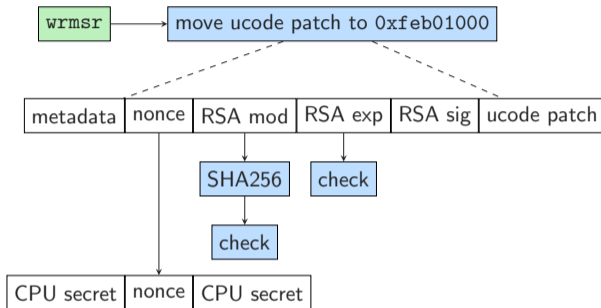
wrmsr

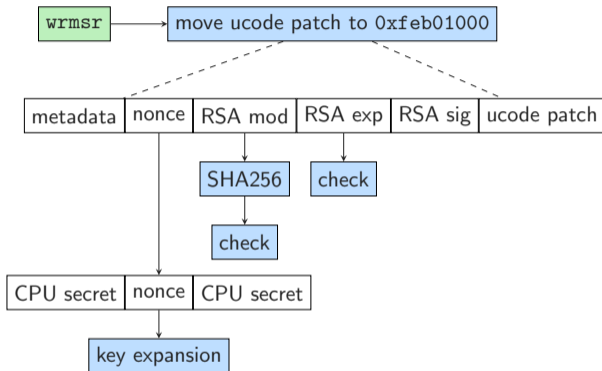


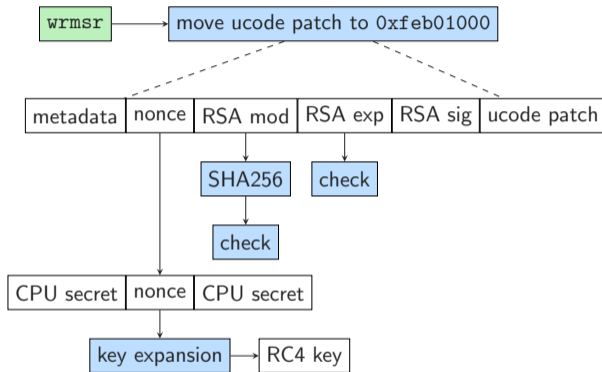


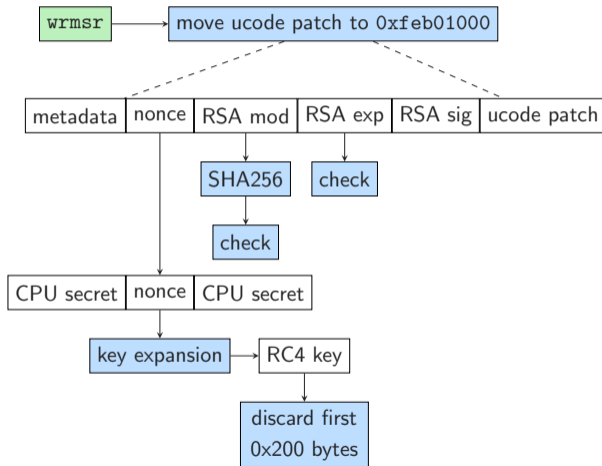


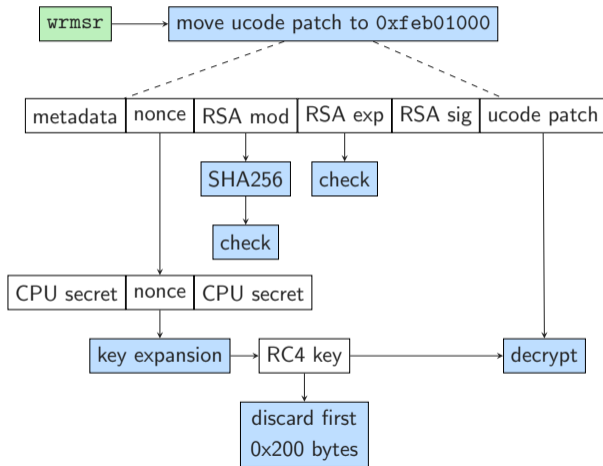


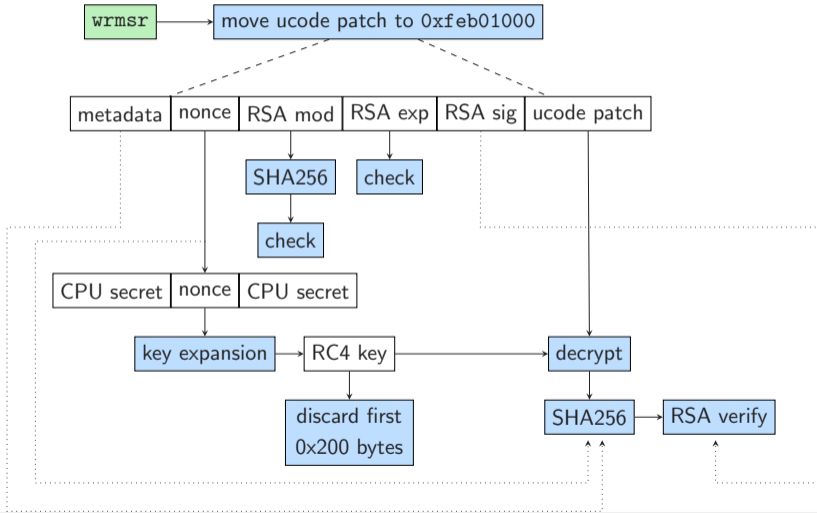


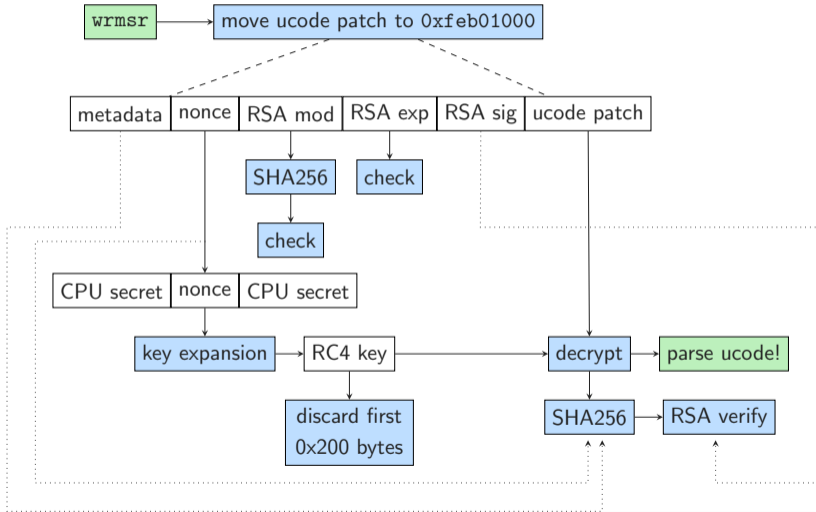












The temporary physical address where μ code is decrypted.

The temporary physical address where μ code is decrypted.

```
> sudo cat /proc/iomem | grep feb00000  
:(
```

The temporary physical address where μ code is decrypted.

```
> sudo cat /proc/iomem | grep feb00000
```

```
:(
```

```
> read_physical_address 0xfeb01000
```

```
00000000: ffff ffff ffff ffff ffff ffff ffff ffff
```

```
00000010: ffff ffff ffff ffff ffff ffff ffff ffff
```

```
00000020: ffff ffff ffff ffff ffff ffff ffff ffff
```

```
00000030: ffff ffff ffff ffff ffff ffff ffff ffff
```




- **Dynamically** enabled by the CPU



- Dynamically enabled by the CPU
- Access time: about 20 cycles



- **Dynamically** enabled by the CPU
- Access time: about **20 cycles**
- Content **not shared** between cores



- **Dynamically** enabled by the CPU
- Access time: about **20 cycles**
- Content **not shared** between cores
- Can fit 64-256Kb of valid data



- **Dynamically** enabled by the CPU
- Access time: about **20 cycles**
- Content **not shared** between cores
- Can fit 64-256Kb of valid data
- **Replacement policy** on the content?!

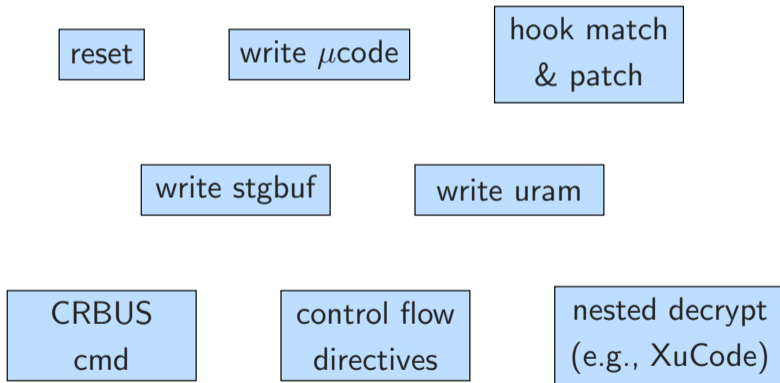


- **Dynamically** enabled by the CPU
- Access time: about **20 cycles**
- Content **not shared** between cores
- Can fit 64-256Kb of valid data
- **Replacement policy** on the content?!
- It's a special CPU view on the **L2 cache!**

```

00000000: 0102 007c 3900 0a00 3f88 4bed c000 080c  ...|9...?.K.....
00000010: 0b01 4780 0000 0a00 3f88 4fad 0003 0a00  ..G.....?.0.....
00000020: 2f20 4b2d 8002 080c 0322 4740 a903 0a00  / K-....."G@....
00000030: 2f20 4f6d 1902 0002 0353 6380 c000 3002  / Om.....Sc...0.
00000040: b8a6 6be8 0000 0002 0320 63c0 0003 f003  ..k..... c.....
00000050: f8a6 6b28 c000 0800 03c0 0bed 0000 0b10  ..k(.....
00000060: 7f00 0800 8001 3110 0300 a140 c000 310c  .....1....@..1.
00000070: 0300 0700 0000 4012 0b30 6210 0003 4b1c  .....@..0b...K.
00000080: 7f00 0440 c000 3112 0310 2400 0000 310c  ...@..1...$...1.
00000090: 0300 01c0 0003 0800 03c0 0fad 0002 00d2  .....
    
```

A μ code update is bytecode: the CPU interprets commands from the μ code update





- Create a **parser** for μcode updates



- Create a **parser** for μcode updates
- Automatically collect existing μcode (s) for GLM

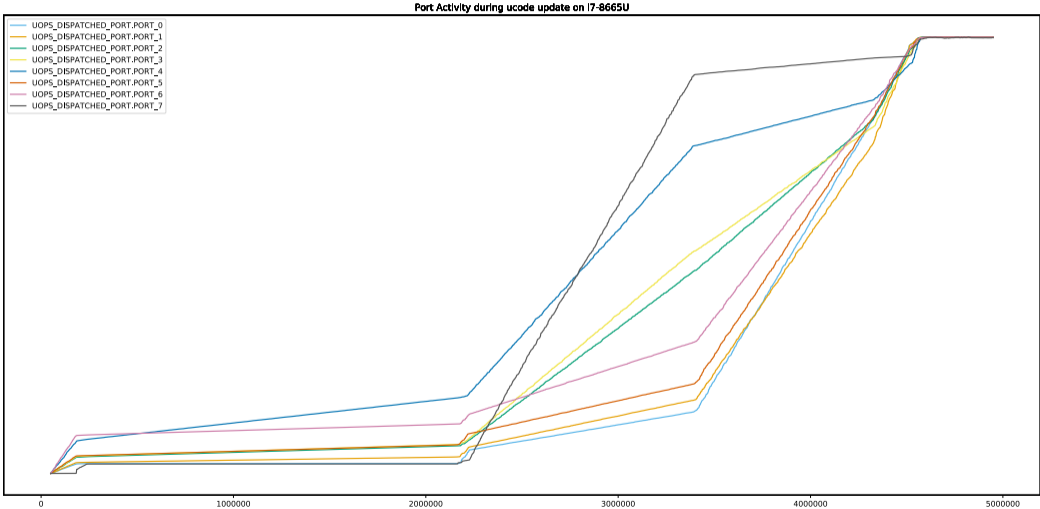


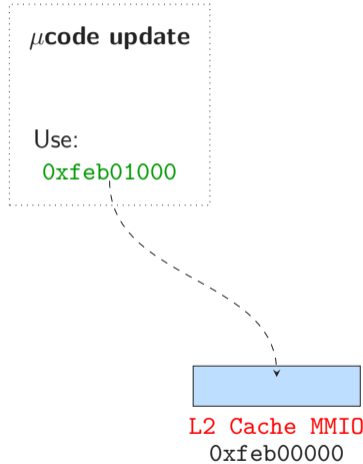
- Create a **parser** for μcode updates
- Automatically collect existing μcode (s) for GLM
- **Decrypt** all GLM updates

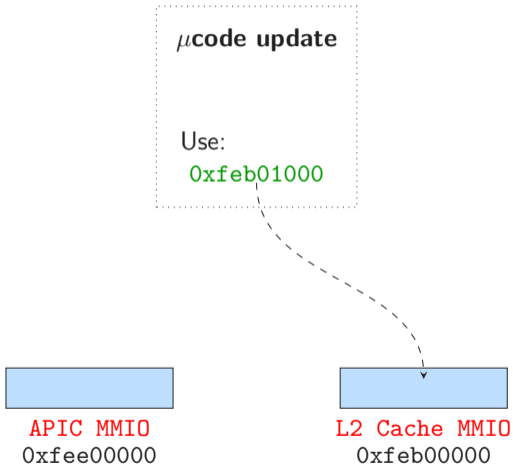


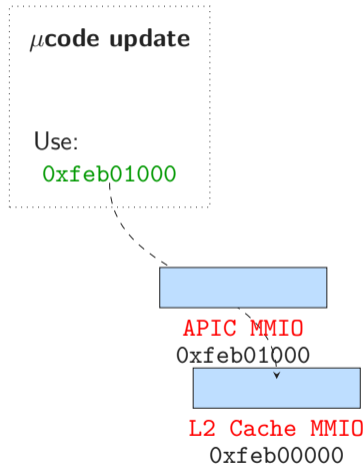
- Create a **parser** for μcode updates
- Automatically collect existing μcode (s) for GLM
- **Decrypt** all GLM updates

`github.com/pietroborrello/CustomProcessingUnit/ucode_collection`

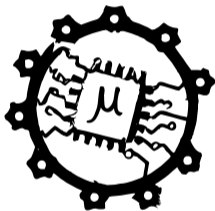


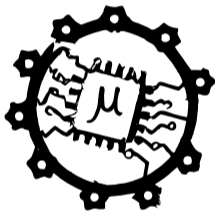




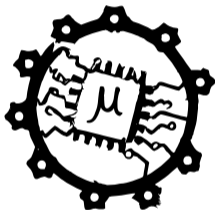


- Deepen understanding of modern CPUs with μ code access

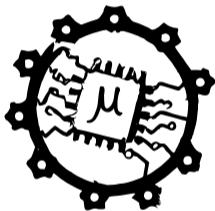




- Deepen understanding of modern CPUs with μ code access
- Develop a static and dynamic analysis framework for μ code:



- Deepen understanding of modern CPUs with μ code access
- Develop a static and dynamic analysis framework for μ code:
 - μ code decompiler
 - μ code assembler
 - μ code patcher
 - μ code tracer



- Deepen understanding of modern CPUs with **μcode** access
- Develop a static and dynamic analysis framework for **μcode**:
 - **μcode** decompiler
 - **μcode** assembler
 - **μcode** patcher
 - **μcode** tracer
- Let's **control** our CPUs!

`github.com/pietroborrello/CustomProcessingUnit`