CustomProcessingUnit: Reverse Engineering and Customization of Intel Microcode

OffensiveCon 2023

Pietro Borrello
Sapienza University of Rome

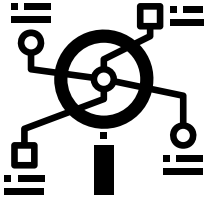
Catherine Easdon
Dynatrace Research & Graz University of Technology

Martin Schwarzl
Graz University of Technology

Roland Czerny
Graz University of Technology

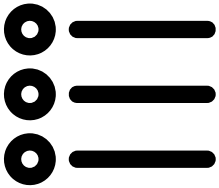
Michael Schwarz
CISPA Helmholtz Center for Information Security

About Me



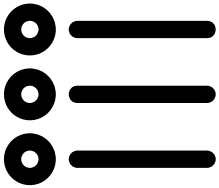
- PhD from Sapienza University of Rome (defended two days ago)
- Interested in (very) low-level research
- 3x BlackHat speaker
- 2x Pwnie Award recipient

Outline



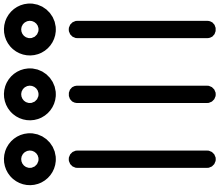
1. Deep dive on CPU µcode

Outline



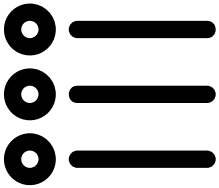
1. Deep dive on CPU µcode
2. µcode Software Framework

Outline



1. Deep dive on CPU µcode
2. µcode Software Framework
3. Case Studies: x86 PAC, µsoftware bp, constant-time div

Outline

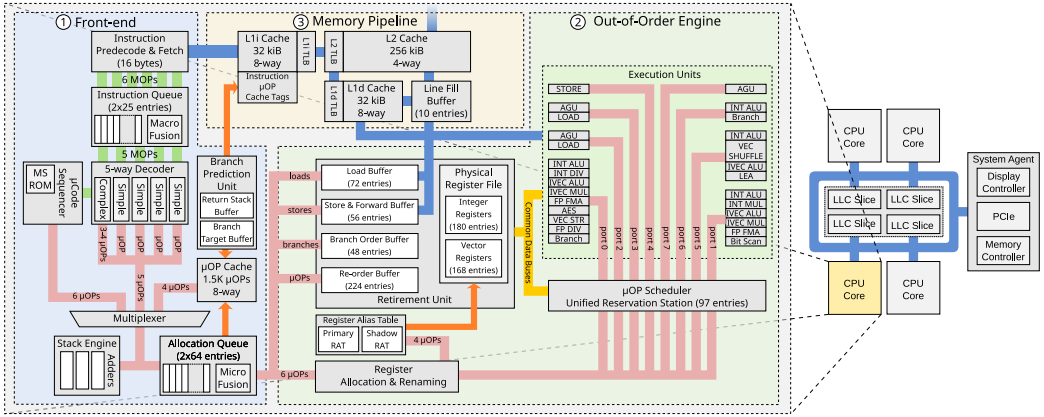


1. Deep dive on CPU µcode
2. µcode Software Framework
3. Case Studies: x86 PAC, µsoftware bp, constant-time div
4. Reverse Engineering of the secret µcode update algorithm

Disclaimer

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

How do CPUs work?



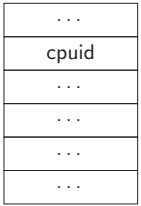
- Red Unlock of Atom Goldmont (GLM) CPUs

- Red Unlock of Atom Goldmont (GLM) CPUs
- Extraction and reverse engineering of GLM μcode format

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

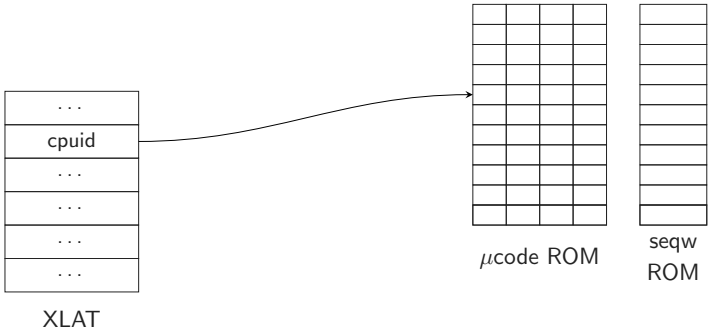
What can you do with access to microarchitectural buffers?

Microcoded Instuctions 101

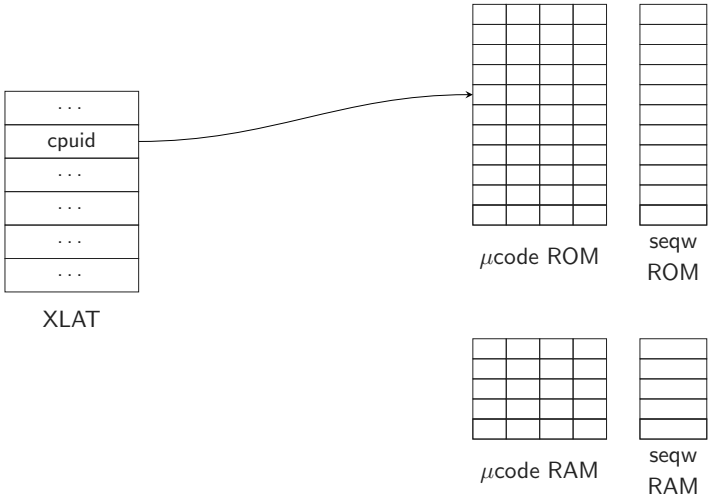


XLAT

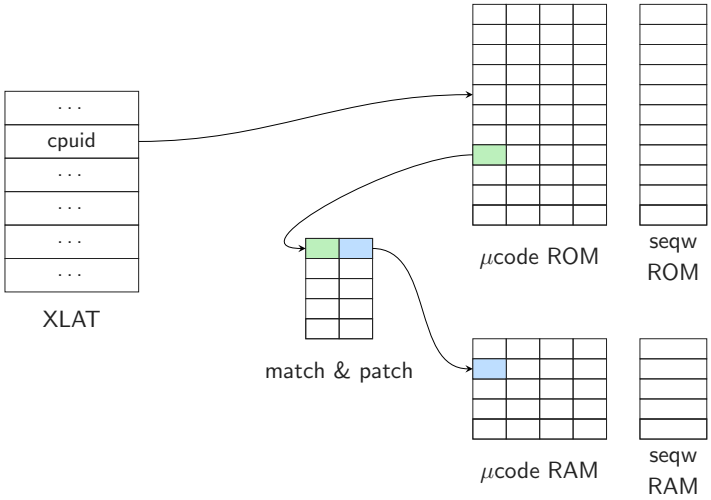
Microcoded Instuctions 101



Microcoded Instuctions 101



Microcoded Instuctions 101



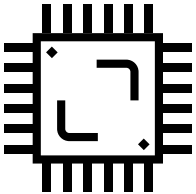
µcode instruction

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

Deep Dive into the μcode

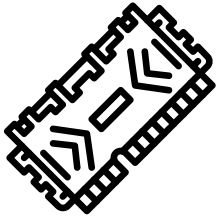
```
U1a54: 09282eb80236          CMPUJZ_DIRECT_NOTTAKEN(tmp6, 0x2, U0e2e)
U1a55: 0008890f8009          tmp8:= ZEROEXT_DSZ32(0x2389)
U1a56: 092830f80236    SYNC-> CMPUJZ_DIRECT_NOTTAKEN(tmp6, 0x3, U0e30)
U1a57: 000000000000          NOP
SEQW:      0903e480          SEQW GOTO U03e4
```

General Purpose µcode registers



- Architectural: `rax`, `rbx`, `rcx`, ..., `r8-r15`
- Temporary: `tmp0-tmp15` + `flag0-flag15`
- Remappable Temporary: `tmpv0-tmpv3`
- Instruction Dependent: `r64dst`, `r64src`, `r64base`, `r64idx`
- µcode IP: `uip0`, `uip1`

µcode Memory



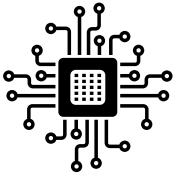
- Physical Memory (virtually or physically addressable)
- URAM
- Staging Buffer
- Constants ROM

Building a Ghidra μcode Decompiler

```
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                UJMPCC_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
-------------------------------------------------------------------------------
```

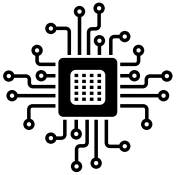
```
1
2  void rc4_decrypt(ulong tmp0_i,ulong tmp1_j,byte *ucode_patch_tmp5,int len_tmp6,byte *S_tmp7,
3                   long callback_tmp8)
4
5  {
6    byte bVar1;
7    byte bVar2;
8
9    do {
10     tmp0_i = (ulong)(byte)((char)tmp0_i + 1);
11     bVar1 = S_tmp7[tmp0_i];
12     tmp1_j = (ulong)(byte)(bVar1 + (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
```

Control Registers Bus



CPU controls its internal units through the `CRBUS`

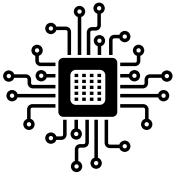
Control Registers Bus



CPU controls its internal units through the `CRBUS`

- MSRs $\rightarrow$ CRBUS addr

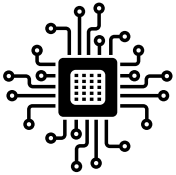
Control Registers Bus



CPU controls its internal units through the `CRBUS`

- MSRs $\rightarrow$ CRBUS addr
- Control and Status registers

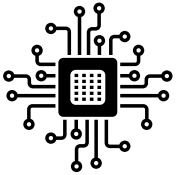
Control Registers Bus



CPU controls its internal units through the `CRBUS`

- MSRs $\rightarrow$ CRBUS addr
- Control and Status registers
- SMM configuration

Control Registers Bus



CPU controls its internal units through the `CRBUS`

- MSRs $\rightarrow$ CRBUS addr
- Control and Status registers
- SMM configuration
- Post Silicon Validation features (LDAT)

Accessing the µcode Sequencer

- The µcode Sequencer manages the access to µcode ROM and RAM

Accessing the µcode Sequencer

- The µcode Sequencer manages the access to µcode ROM and RAM

→ The LDAT has access to the µcode Sequencer

Accessing the µcode Sequencer

- The µcode Sequencer manages the access to µcode ROM and RAM

→ The LDAT has access to the µcode Sequencer

→ We can access the LDAT through the CRBUS

Accessing the µcode Sequencer

- The µcode Sequencer manages the access to µcode ROM and RAM

→ The LDAT has access to the µcode Sequencer

→ We can access the LDAT through the CRBUS

→ If we can access the CRBUS we can control µcode!

Mark Ermolov, Maxim Goryachy & Dmitry Sklyarov discovered the existance of two secret instructions that can access (RW):

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

Mark Ermolov, Maxim Goryachy & Dmitry Sklyarov discovered the existance of two secret instructions that can access (RW):

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

e.g., Writing to the CRBUS

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

Reverse engineer patterns of how the CPU itself accesses the CRBUS

```
//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);
```

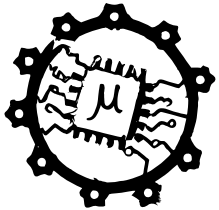
Writing to the μcode Sequencer

```
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 µcode ROM to µcode RAM to execute patches.

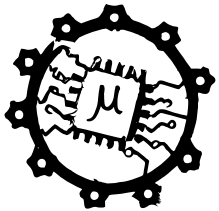
```
patch_off = (patch_addr - 0x7c00) / 2;
entry:
 +--+-----------+-----------------------+----+
 |3e| patch_off |       match_addr      |enbl|
 +--+-----------+-----------------------+----+
  24          16                        1    0
```

The first μcode Framework



Leveraging udbgrd/wr we can patch μcode via software

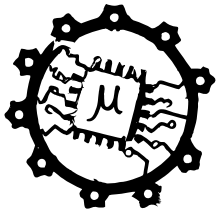
The first µcode Framework



Leveraging udbgrd/wr we can patch µcode via software

- Completely observe CPU behavior

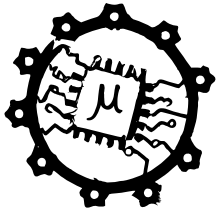
The first µcode Framework



Leveraging udbgrd/wr we can patch µcode via software

- Completely observe CPU behavior
- Completely control CPU behavior

The first µcode Framework



Leveraging udbgrd/wr we can patch µcode via software

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

Patch µcode

µcode Framework

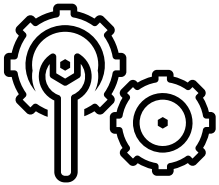
Patch µcode

Hook µcode

Patch µcode

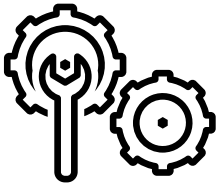
Hook µcode

Trace µcode



We can change the CPU's behavior.

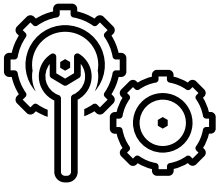
µcode patches



We can change the CPU's behavior.

- Change microcoded instructions

µcode patches



We can change the CPU's behavior.

- Change microcoded instructions
- Add functionalities to the CPU

μcode patch Hello World!

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

µcode patch Hello World!

```
.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!"


DEMO - HELLO WORLD
fs0:\EFI>

Make `rdrand` less boring

`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
```

Make `rdrand` less boring

`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
```

Make `rdrand` less boring

`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
```

Make `rdrand` less boring

`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 - SMM DUMP
fs0:\EFI>

x86 PAC

Pointer Authentication Codes

ARM mitigation:

protect pointers from corruption using a cryptographic signature

Pointer Authentication Codes

ARM mitigation:
protect pointers from corruption using a cryptographic signature

- `pacia ptr, ctx` → **sign** pointer `ptr` using salt `ctx`
  `0x1337 -> 0xaced000000001337`

Pointer Authentication Codes

ARM mitigation:
protect pointers from corruption using a cryptographic signature

- `pacia ptr, ctx` → **sign** pointer `ptr` using salt `ctx`
  `0x1337 -> 0xaced000000001337`
- `autia ptr, ctx` → **authenticate** pointer `ptr` using salt `ctx`
  `0xaced000000001337 -> 0x1337`
  `0xaced0000000013ff -> 0xdead0000000013ff`

x86 Pointer Authentication Codes

~~ARM~~ x86 mitigation:
protect pointers from corruption using a cryptographic signature

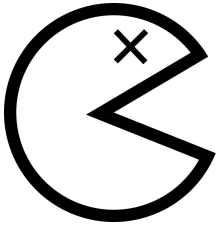
- ~~`pacia`~~ `verw (ptr, ctx, 1)` → **sign** pointer `ptr` using salt `ctx`
  `0x1337 -> 0xaced000000001337`
- ~~`autia`~~ `verr (ptr, ctx, 1)` → **authenticate** pointer `ptr` using salt `ctx`
  `0xaced000000001337 -> 0x1337`
  `0xaced0000000013ff -> 0xdead0000000013ff`

x86 Pointer Authentication Codes - implementation

```
.seg 0x7c00

# declare variables
int [ptr] := rt4:dst;       int [t0] := tmp1
int [ctx] := rt4:src;       int [t1] := tmp2
int [key] := tmp5;          int [t2] := tmp3
int [key_addr] := 0xbe48;   int [t3] := tmp4
int [pac] := tmp6

# ---- initialize ----
[key] := LEUTHER_HASH_ASCII6_SCI([key_addr])
# v0 = 0x736f6d6570736570 ^ key0
[v0] := DERGET_MACRO(0x7366656570736570)
[v0] := XOR_DSI64([v0], [key])
# v1 = 0x646f72616e646f6d ^ ctx2
[v1] := DERGET_MACRO(0x646f72616e646f6d)
[v1] := XOR_DSI64([v1], [ctx])
# v2 = 0x6c7967656e657261 ^ key1
[v2] := DERGET_MACRO(0x6c7967656e657261)
[v2] := XOR_DSI64([v2], [key])
# v3 = 0x7465646279746573 ^ ctx3
[v3] := DERGET_MACRO(0x7465646579746573)
[v3] := XOR_DSI64([v3], [ctx])

# --- update ---
[v3] := XOR_DSI64([v3], [ptr]) # v3 ^= ptr;
[v0] := ADD_DSI64([v0], [v1]) # v0 += v1;
[v2] := ADD_DSI64([v2], [v3]) # v2 += v3;
[v1] := ROL_DSI64([v1], 0x0d) # v1 = RotateLeft<13>(v1);
[v3] := ROL_DSI64([v3], 0x10) # v3 = RotateLeft<16>(v3);
[v1] := XOR_DSI64([v1], [v0]) # v1 ^= v0;
[v3] := XOR_DSI64([v3], [v2]) # v3 ^= v2;
[v0] := ROL_DSI64([v0], 0x20) # v0 = RotateLeft<32>(v0);
[v2] := ADD_DSI64([v2], [v1]) # v2 += v1;
[v0] := ADD_DSI64([v0], [v3]) # v0 += v3;
[v1] := ROL_DSI64([v1], 0x11) # v1 = RotateLeft<17>(v1);
[v3] := ROL_DSI64([v3], 0x15) # v3 = RotateLeft<21>(v3);
[v1] := XOR_DSI64([v1], [v2]) # v1 ^= v2;
[v3] := XOR_DSI64([v3], [v0]) # v3 ^= v0;
[v2] := ROL_DSI64([v2], 0x20) # v2 = RotateLeft<32>(v2);
[v0] := XOR_DSI64([v0], [ptr]) # v0 ^= ptr;

# --- finalize ---
[v2] := XOR_DSI64([v2], 0xff) # v2 ^= 0xff;
[v0] := ADD_DSI64([v0], [v1]) # v0 += v1;
[v2] := ADD_DSI64([v2], [v3]) # v2 += v3;
[v1] := ROL_DSI64([v1], 0x0d) # v1 = RotateLeft<13>(v1);
[v3] := ROL_DSI64([v3], 0x10) # v3 = RotateLeft<16>(v3);
[v1] := XOR_DSI64([v1], [v0]) # v1 ^= v0;
[v3] := XOR_DSI64([v3], [v2]) # v3 ^= v2;
[v0] := ROL_DSI64([v0], 0x20) # v0 = RotateLeft<32>(v0);
[v1] := ADD_DSI64([v1], [v2]) # v2 += v1;
[v0] := ADD_DSI64([v0], [v3]) # v0 += v3;
[v1] := ROL_DSI64([v1], 0x11) # v1 = RotateLeft<17>(v1);
[v3] := ROL_DSI64([v3], 0x15) # v3 = RotateLeft<21>(v3);
[v1] := XOR_DSI64([v1], [v2]) # v1 ^= v2;
[v3] := XOR_DSI64([v3], [v0]) # v3 ^= v0;
[v2] := ROL_DSI64([v2], 0x20) # v2 = RotateLeft<32>(v2);

# pac = (v0 ^ v1) ^ (v2 ^ v3)) << 48;
[pac] := XOR_DSI64([v0], [v1])
[pac] := XOR_DSI64([pac], [v2])
[pac] := XOR_DSI64([pac], [v3])
[pac] := REG_DSI64([pac], 0x30)

# sign ptr
[ptr] := XOR_DSI64([pac], [ptr])
```

- Single-round SipHash for fast keyed-hashing
- PAC key stored in an internal CPU buffer
- 54 µops → ∼25 cycles PAC operation

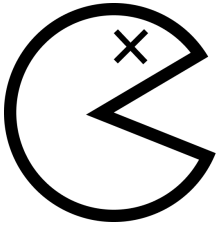
PACMAN Attack



Bruteforce PAC using speculative execution

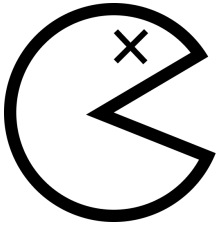
```
if (condition):
 auth_ptr <- aut(ptr) // speculatively executed
 load(auth_ptr)       // if correct -> address loaded
                      // if wrong -> address not loaded
```

x86 PACMAN Attack



Bruteforce **x86** PAC using speculative execution

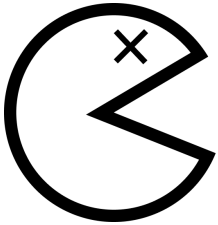
```
if (condition):
 auth_ptr <- aut(ptr) // speculatively executed
 load(auth_ptr)       // never loaded
```



Bruteforce **x86** PAC using speculative execution

```
if (condition):
 auth_ptr <- aut(ptr) // speculatively executed
 load(auth_ptr)       // never loaded
```

our 54 µops fill the speculative window!

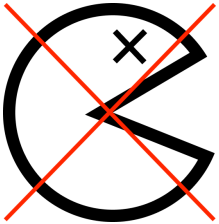


Attack a weaker version of **x86** PAC (1/2 round SipHash)

```
if (condition):
 auth_ptr <- aut(ptr) // speculatively executed
 load(auth_ptr)       // if correct -> address loaded
                      // if wrong -> address not loaded
```

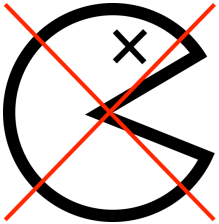

DEMO - PACMAN
fs0:\EFI>

x86 PACMAN Mitigation



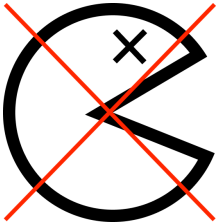
- Speculation Barrier

x86 PACMAN Mitigation



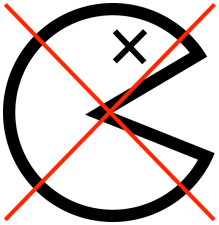
- ~~Speculation Barrier~~ → high overhead + partial protection

x86 PACMAN Mitigation



- ~~Speculation Barrier~~ → high overhead + partial protection
- Make `aut` fault when invalid PAC

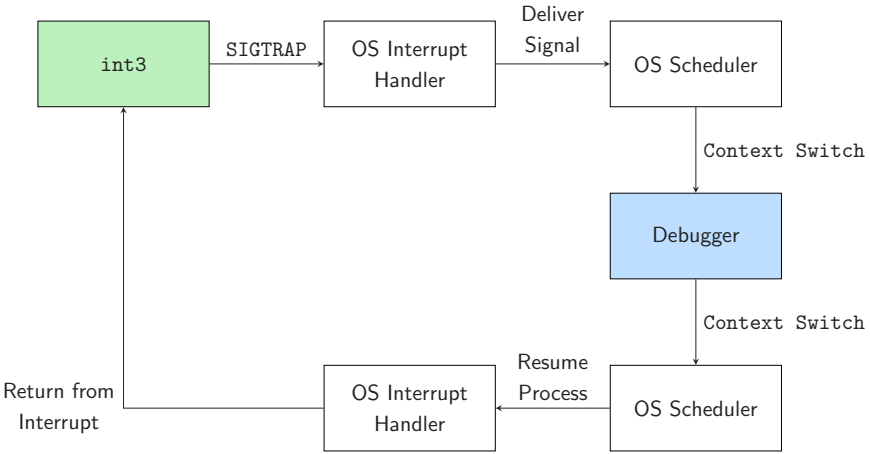
x86 PACMAN Mitigation



- ~~Speculation Barrier~~ → high overhead + partial protection
- Make `aut` fault when invalid PAC
- Always remove signature from pointer
  → pointer access always valid on speculative paths

μsoftware breakpoints

Breakpoint Flow



µsoftware breakpoints

Patch `int3` to implement breakpointing logic directly in µcode

- Avoid interrupt overhead (∼50x faster)
- Avoid context switch overhead (∼1000x faster)
- → 10 cycles for a breakpoint

Implement fuzzing coverage collection in μcode

```
.patch 0xa78 # int3 entry point
let [cov_map] := tmp1
let [rip] := tmp0

# load address of coverage map from staging buffer
[cov_map]:= LDSTGBUF_DSZ64_ASZ16_SC1(0xba00)
# get instruction pointer low bits
[rip]:= ZEROEXT_DSZ64(IMM_MACRO_ALIAS_RIP) !m0
[rip]:= AND_DSZ64(0xffff, [rip])
# set coverage for basic block
STADPPHYS_DSZ8_ASZ64_SC1([cov_map], [rip], 0x01)
```

Constant-Time division

Constant-Time division

The `div` instruction latency depends on the input data

Constant-Time division

The `div` instruction latency depends on the input data

- Side channel attacks can infer input data

Constant-Time division

The `div` instruction latency depends on the input data

- Side channel attacks can infer input data
- $2/1 \rightarrow 22$ cycles

Constant-Time division

The `div` instruction latency depends on the input data

- Side channel attacks can infer input data
- $2/1 \rightarrow 22$ cycles
- $0x113371337/1 \rightarrow 40$ cycles

Software Constant-Time division

Software `div` implementation mitigates the issue

Software Constant-Time division

Software `div` implementation mitigates the issue

- High overhead $\rightarrow \sim$ 700 cycles

Software Constant-Time division

Software `div` implementation mitigates the issue

- High overhead $\rightarrow \sim$ 700 cycles
- Need compilation pass or binary patching

μcode Constant-Time division

```
.org 0x7c00
.patch 0x6c8 # div entry point
.entry 0

let [dividend] := rax;              let [temp1]    := tmp3
let [divisor]  := rcx;              let [temp2]    := tmp4
let [size]     := 0x3f;             let [temp3]    := tmp5
let [quotient] := tmp0;             let [temp4]    := tmp7
let [temp]     := tmp1;             let [temp5]    := tmp8
let [i]        := tmp2;             let [comp]     := tmp6
[temp] := ZEROEXT_DSZ64(0x0);       [i] := ZEROEXT_DSZ64([size])
[quotient] := ZEROEXT_DSZ64(0x0)

<loop>
# if (i < 0 ) goto end;
UJMPCC_DIRECT_NOTTAKEN_CONDB([i], <end>)

# temp = (temp << 1uLL) | ((dividend >> i) & 1);
[temp1]:= SHL_DSZ64([temp], 0x1)
[temp2]:= SHR_DSZ64([dividend], [i])
[temp2]:= AND_DSZ64([temp2], 0x1)
[temp] := OR_DSZ64([temp1], [temp2])

# comp = (temp >= divisor);
[comp]  := SUB_DSZ64([divisor], [temp])

# temp -= comp? divisor : 0;
[temp3]:= SELECTCC_DSZ64_CONDB([comp], [divisor])
[temp] := SUB_DSZ64([temp3], [temp])

# quotient |= comp ? 1uLL << i : 0;
[temp4]:= SHL_DSZ64(0x1, [i])
[temp5]:= SELECTCC_DSZ64_CONDB([comp], [temp4])
[quotient] := OR_DSZ64([quotient], [temp5])

# i--; goto loop
[i] := SUB_DSZ64(0x1, [i]) SEQW GOTO <loop>

<end>
# return quotient, ignore the remainder for simplicity
rax := ZEROEXT_DSZ64([quotient])
rdx := ZEROEXT_DSZ64(0x0)
```

We can patch the `div` instruction to be constant time

μcode Constant-Time division

```
.org 0x7c00
.patch 0x6c8 # div entry point
.entry 0

let [dividend] := rax;              let [temp1]    := tmp3
let [divisor]  := rcx;              let [temp2]    := tmp4
let [size]     := 0x3f;             let [temp3]    := tmp5
let [quotient] := tmp0;             let [temp4]    := tmp7
let [temp]     := tmp1;             let [temp5]    := tmp8
let [i]        := tmp2;             let [comp]     := tmp6
[temp] := ZEROEXT_DSZ64(0x0);       [i] := ZEROEXT_DSZ64([size])
[quotient] := ZEROEXT_DSZ64(0x0)

<loop>
# if (i < 0 ) goto end;
UJMPCC_DIRECT_NOTTAKEN_CONDB([i], <end>)

# temp = (temp << 1uLL) | ((dividend >> i) & 1);
[temp1]:= SHL_DSZ64([temp], 0x1)
[temp2]:= SHR_DSZ64([dividend], [i])
[temp2]:= AND_DSZ64([temp2], 0x1)
[temp] := OR_DSZ64([temp1], [temp2])

# comp = (temp >= divisor);
[comp]  := SUB_DSZ64([divisor], [temp])

# temp -= comp? divisor : 0;
[temp3]:= SELECTCC_DSZ64_CONDB([comp], [divisor])
[temp] := SUB_DSZ64([temp3], [temp])

# quotient |= comp ? 1uLL << i : 0;
[temp4]:= SHL_DSZ64(0x1, [i])
[temp5]:= SELECTCC_DSZ64_CONDB([comp], [temp4])
[quotient] := OR_DSZ64([quotient], [temp5])

# i--; goto loop
[i] := SUB_DSZ64(0x1, [i]) SEQW GOTO <loop>

<end>
# return quotient, ignore the remainder for simplicity
rax := ZEROEXT_DSZ64([quotient])
rdx := ZEROEXT_DSZ64(0x0)
```

We can patch the `div` instruction to be constant time

- Reduced overhead $\rightarrow \sim$ 400 cycles

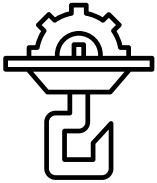
μcode Constant-Time division

```
.org 0x7c00
.patch 0x6c8 # div entry point
.entry 0

let [dividend] := rax;              let [temp1]    := tmp3
let [divisor]  := rcx;              let [temp2]    := tmp4
let [size]     := 0x3f;             let [temp3]    := tmp5
let [quotient] := tmp0;             let [temp4]    := tmp7
let [temp]     := tmp1;             let [temp5]    := tmp8
let [i]        := tmp2;             let [comp]     := tmp6
[temp] := ZEROEXT_DSZ64(0x0);       [i] := ZEROEXT_DSZ64([size])
[quotient] := ZEROEXT_DSZ64(0x0)

<loop>
# if (i < 0 ) goto end;
UJMPCC_DIRECT_NOTTAKEN_CONDB([i], <end>)

# temp = (temp << 1uLL) | ((dividend >> i) & 1);
[temp1]:= SHL_DSZ64([temp], 0x1)
[temp2]:= SHR_DSZ64([dividend], [i])
[temp2]:= AND_DSZ64([temp2], 0x1)
[temp] := OR_DSZ64([temp1], [temp2])

# comp = (temp >= divisor);
[comp]  := SUB_DSZ64([divisor], [temp])

# temp -= comp? divisor : 0;
[temp3]:= SELECTCC_DSZ64_CONDB([comp], [divisor])
[temp] := SUB_DSZ64([temp3], [temp])

# quotient |= comp ? 1uLL << i : 0;
[temp4]:= SHL_DSZ64(0x1, [i])
[temp5]:= SELECTCC_DSZ64_CONDB([comp], [temp4])
[quotient] := OR_DSZ64([quotient], [temp5])

# i--; goto loop
[i] := SUB_DSZ64(0x1, [i]) SEQW GOTO <loop>

<end>
# return quotient, ignore the remainder for simplicity
rax := ZEROEXT_DSZ64([quotient])
rdx := ZEROEXT_DSZ64(0x0)
```

We can patch the `div` instruction to be constant time

- Reduced overhead $\rightarrow \sim$ 400 cycles
- Transparent to the running program

µcode hooks



Install µcode hooks to observe events.

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

Make your own performance counter

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

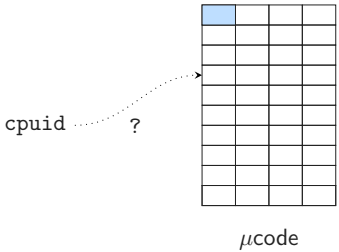
```
.patch 0xXXXX # INSTRUCTION ENTRY POINT
.org 0x7da0

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
```

Make your own performance counter

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

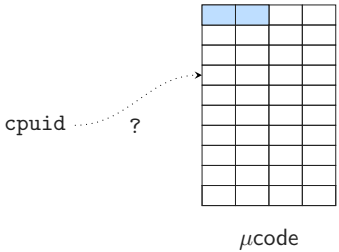
```
.patch 0xXXXX # INSTRUCTION ENTRY POINT
.org 0x7da0

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.



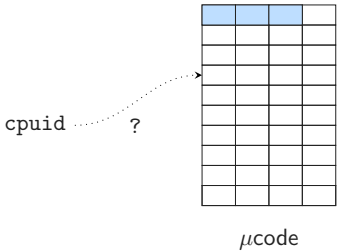

```
hook:
  1.  dump timestamp
  2.  disable hook
  3.  continue
```

Trace µcode execution leveraging hooks.




```
hook:
  1.  dump timestamp
  2.  disable hook
  3.  continue
```

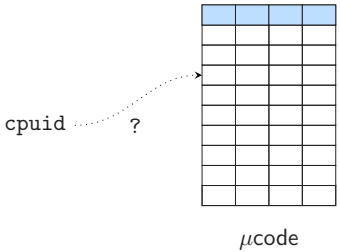
Trace µcode execution leveraging hooks.




```
hook:
  1.  dump timestamp
  2.  disable hook
  3.  continue
```

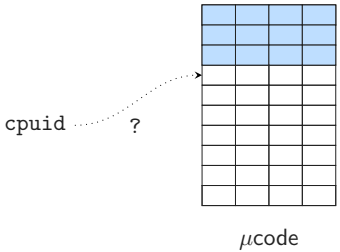
µcode traces

Trace µcode execution leveraging hooks.



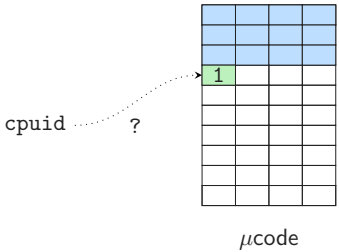

```
hook:
  1.  dump timestamp
  2.  disable hook
  3.  continue
```

Trace µcode execution leveraging hooks.



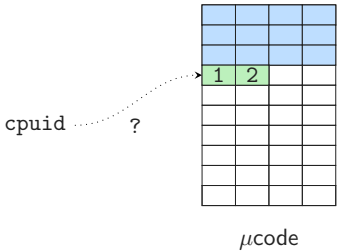

```
hook:
  1.  dump timestamp
  2.  disable hook
  3.  continue
```

Trace μcode execution leveraging hooks.



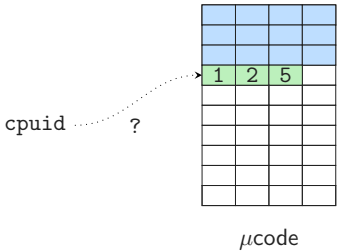

```
hook:
  1.  dump timestamp
  2.  disable hook
  3.  continue
```

Trace µcode execution leveraging hooks.



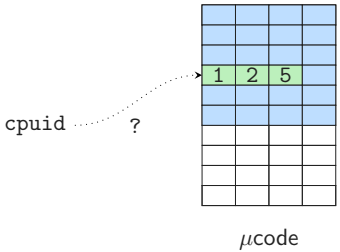

```
hook:
  1.  dump timestamp
  2.  disable hook
  3.  continue
```

Trace μcode execution leveraging hooks.



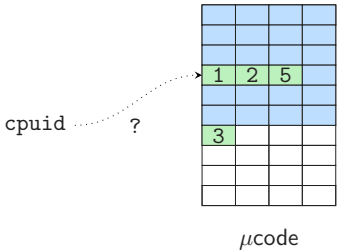

```
hook:
  1.  dump timestamp
  2.  disable hook
  3.  continue
```

Trace μcode execution leveraging hooks.



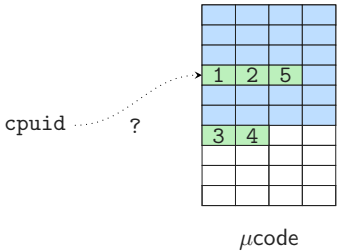

```
hook:
  1.  dump timestamp
  2.  disable hook
  3.  continue
```

Trace μcode execution leveraging hooks.



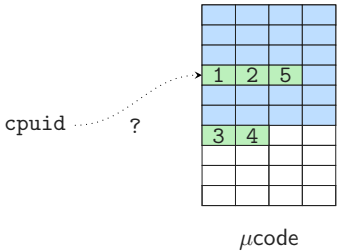

```
hook:
  1.  dump timestamp
  2.  disable hook
  3.  continue
```

Trace μcode execution leveraging hooks.




```
hook:
  1.  dump timestamp
  2.  disable hook
  3.  continue
```

Trace µcode execution leveraging hooks.




```
hook:
  1.  dump timestamp
  2.  disable hook
  3.  continue
```

1 2 3 4 5

Reversing µcode updates

µcode update algorithm has always been kept secret by Intel

Let's trace the execution of a µcode update!

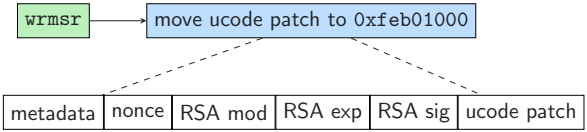
- Trigger a µcode update
- Trace if a microinstruction is executed
- Repeat for all the possible µcode instructions
- Restore order

GLM μcode update algorithm

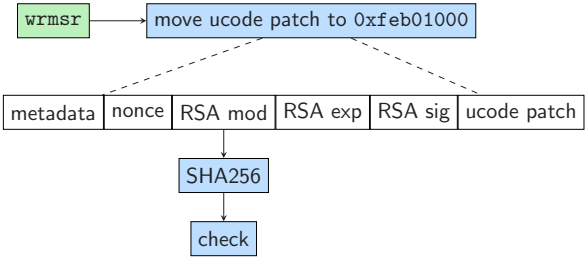
`wrmsr`

GLM μcode update algorithm

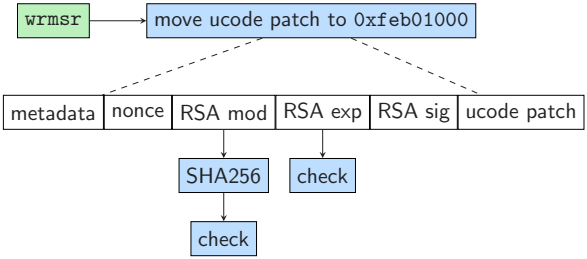
GLM μcode update algorithm



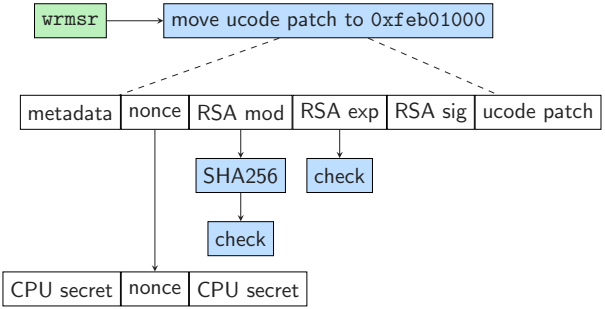
GLM μcode update algorithm



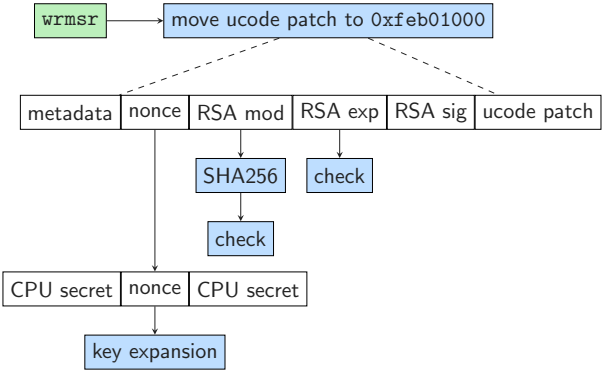
GLM μcode update algorithm



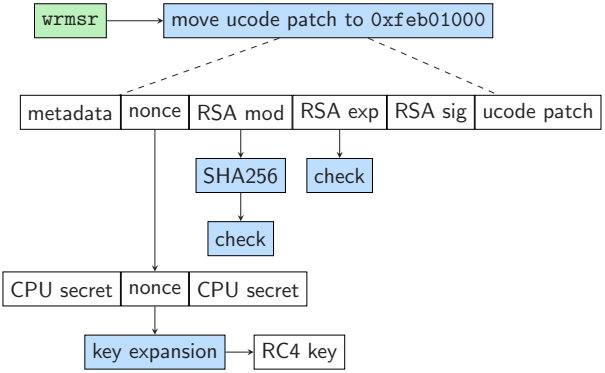
GLM μcode update algorithm



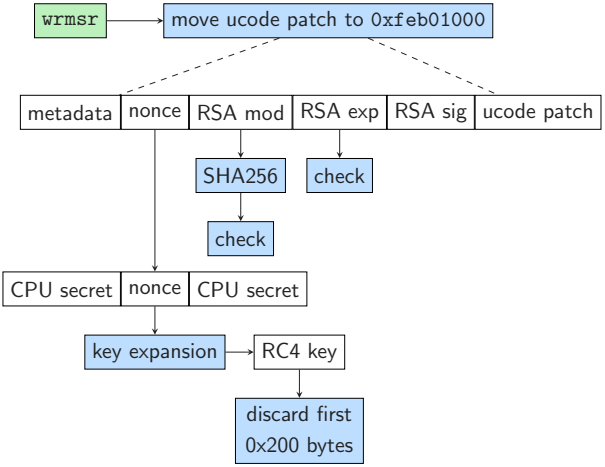
GLM μcode update algorithm



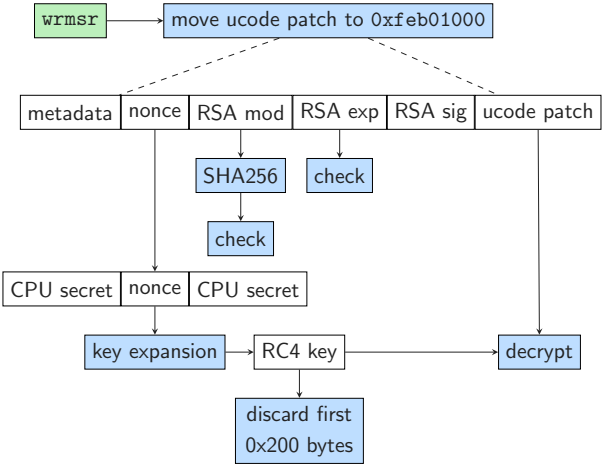
GLM μcode update algorithm



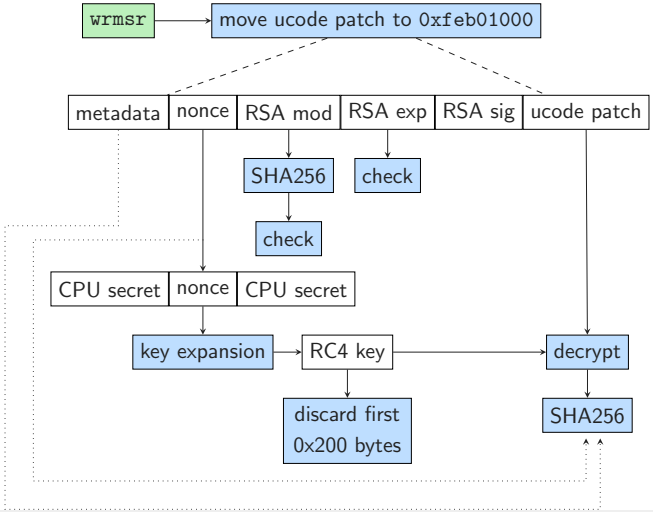
GLM μcode update algorithm



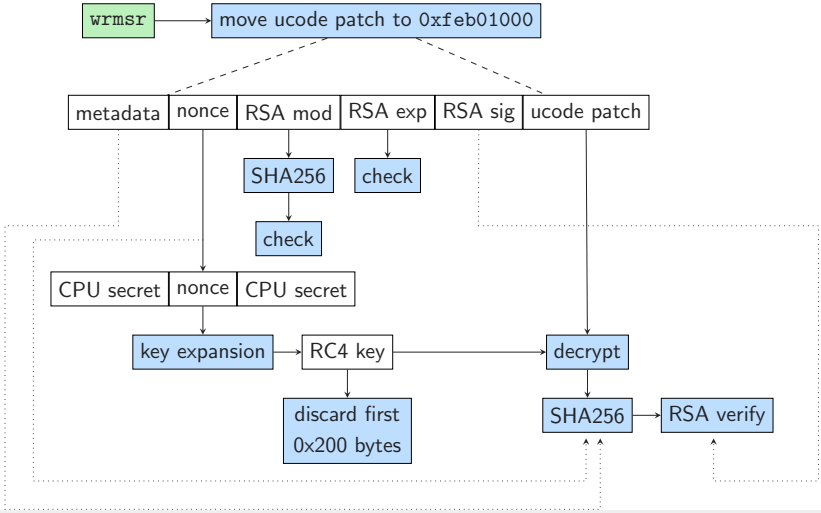
GLM μcode update algorithm



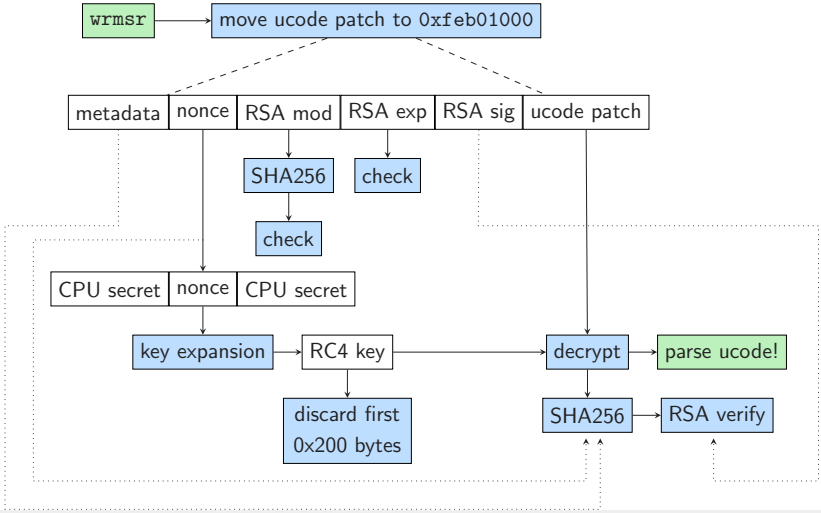
GLM μcode update algorithm



GLM μcode update algorithm



GLM μcode update algorithm



Used to decrypt updates

- 16B per-CPU-model key, 32B per-update nonce → expanded to 256B
- Discard first 512 bytes to reduce bias
- Attacks on RC4 use $> 1\_000\_000$ ciphertexts → only 453 public updates :(

µcode update - RSA

Used to check update signature

- 2048 bits, PKCS#1 v1.5 padding
- RSA modulus and exponent hardcoded
- Signature checks:
  - security revision
  - `cpuid` values
  - nonce
  - decrypted update

0xfeb01000 Secure Memory (1/2)

The temporary physical address where µcode is decrypted.

→ Used as secure memory

0xfeb01000 Secure Memory (1/2)

The temporary physical address where µcode is decrypted.

→ Used as secure memory

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

0xfeb01000 Secure Memory (1/2)

The temporary physical address where μcode is decrypted.

→ Used as secure memory

```
> 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!

Can we corrupt the µcode after decryption but before being applied

- Each core has a private Secure Memory area

µcode update - Secure Memory Race Conditions

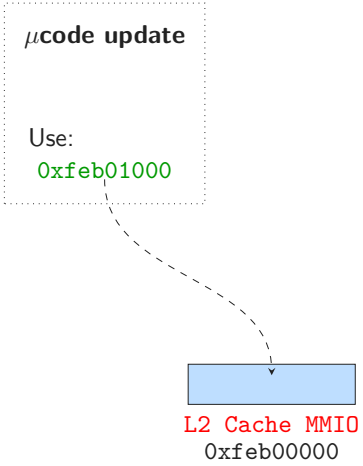
Can we corrupt the µcode after decryption but before being applied

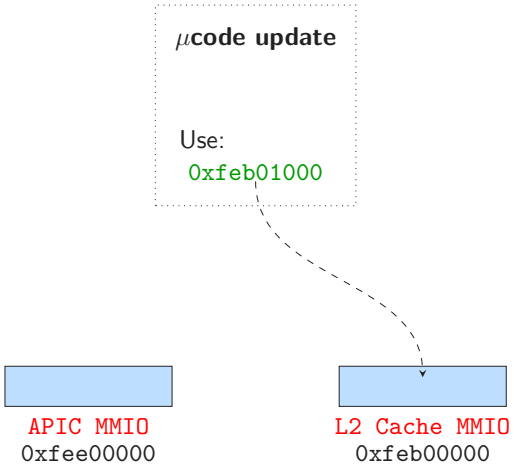
- Each core has a private Secure Memory area
- `wrmsr` → Hyperthreading disabled during update

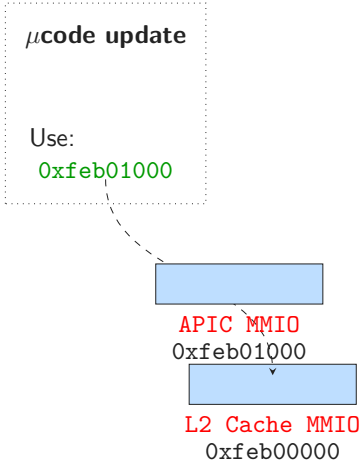
µcode update - Secure Memory Race Conditions

Can we corrupt the µcode after decryption but before being applied

- Each core has a private Secure Memory area
- `wrmsr` → Hyperthreading disabled during update
- Secure Memory only enabled during updates → read `0xff`


µcode update
Use:
0xfeb01000
L2 Cache MMIO
0xfeb00000


µcode update
Use:
0xfeb01000
APIC MMIO
0xfee00000
L2 Cache MMIO
0xfeb00000


µcode update
Use:
0xfeb01000
APIC MMIO
0xfeb01000
L2 Cache MMIO
0xfeb00000

Can we transiently leak µcode updates?

- Fallout/MDS → No hyperthreading

Can we transiently leak µcode updates?

- Fallout/MDS → No hyperthreading
- L1TF → Only leaks of internal buffers (L1)

µcode update - Microarchitectural Attacks

Can we transiently leak µcode updates?

- Fallout/MDS → No hyperthreading
- L1TF → Only leaks of internal buffers (L1)
- Inverse ÆPIC Leak → Internal buffers flushed

µcode update - Fault Injection Attacks (*preliminary idea*)

Can we induce faults and skip checks?

- We found no real fault-injection protection

µcode update - Fault Injection Attacks (*preliminary idea*)

Can we induce faults and skip checks?

- We found no real fault-injection protection
- Signature check operations seems the most profitable

µcode update - Fault Injection Attacks (*preliminary idea*)

Can we induce faults and skip checks?

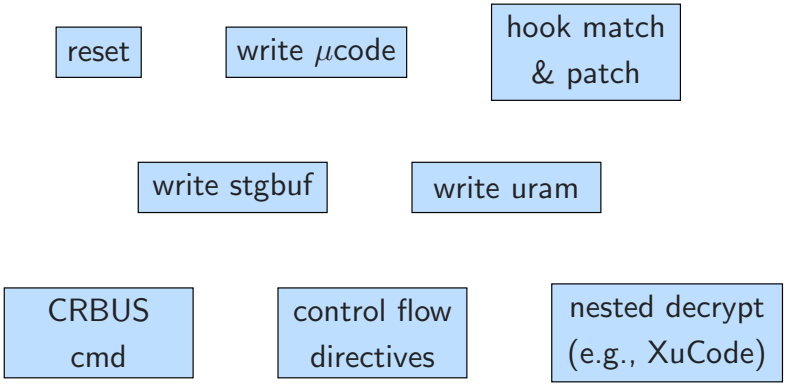
- We found no real fault-injection protection
- Signature check operations seems the most profitable
- The update is not persistent → repeat attack at each boot

Parsing µcode updates

```
00000000: 0102 007c 3900 0a00 3f88 4bed c000 080c  ...|9...?.K.....
00000010: 0b01 4780 0000 0a00 3f88 4fad 0003 0a00  ..G.....?.O.....
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  ................
```

Parsing μcode updates

A μcode update is bytecode: the CPU interprets commands from the μcode update



- Create a parser for µcode updates

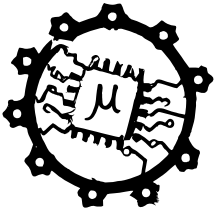
- Create a parser for µcode updates
- Automatically collect existing µcode (s) for GLM

- 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

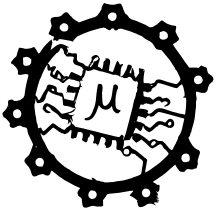
`github.com/pietroborrello/CustomProcessingUnit/ucode_collection`

Conclusion



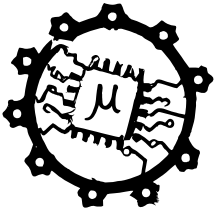
- Deepen understanding of modern CPUs with μcode access

Conclusion



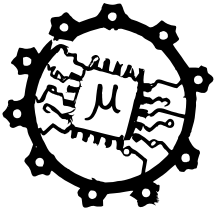
- Deepen understanding of modern CPUs with μcode access
- Develop a static and dynamic analysis framework for μcode:

Conclusion



- 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

Conclusion



- 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`